Randomized Meldable Priority Queues

Article · April 2000

Source: CiteSeer

CITATIONS
2

READS
282

4 authors, including:

Anna Gambin
University of Warsaw
181 PUBLICATIONS **2,090** CITATIONS
SEE PROFILE

Adam Malinowski
University of Warsaw
12 PUBLICATIONS **197** CITATIONS
SEE PROFILE

Randomized Meldable Priority Queues

Anna Gambin and Adam Malinowski

Instytut Informatyki, Uniwersytet Warszawski,
Banacha 2, Warszawa 02-097, Poland,
{aniag,amal}@mimuw.edu.pl

Abstract. We present a practical meldable priority queue implementation. All priority queue operations are very simple and their logarithmic time bound holds with high probability, which makes this data structure more suitable for real-time applications than those with only amortized performance guarantees. Our solution is also space-efficient, since it does not require storing any auxiliary information within the queue nodes.

1 Introduction

In this paper we present a randomized approach to the problem of efficient meldable priority queue implementation. The operations supported by this data structure are the following [10]:

MakeQueue returns an empty priority queue.
FindMin(Q) returns the minimum item from priority queue Q.
DeleteMin(Q) deletes and returns the minimum item from priority queue Q.
Insert(Q, e) inserts item e into priority queue Q.
Meld(Q_1, Q_2) returns the priority queue formed by combining disjoint priority queues Q_1 and Q_2.
DecreaseKey(Q, e, e') replaces item e by e' in priority queue Q provided $e' \leq e$ and the location of e in Q is known.
Delete(Q, e) deletes item e from priority queue Q provided the location of e in Q is known.

(The last two operations are sometimes considered optional.)

In existing priority queue implementations the approach is two-fold. Most data structures require storing additional balance information associated with queue nodes in order to guarantee the worst-case efficiency of individual operations (e.g. leftist trees [8], relaxed heaps [5], Brodal queues [3, 4]). Others achieve good amortized performance by adjusting the structure during some operations rather than struggling to maintain balance constantly (skew heaps [12, 13], pairing heaps [6]). Experiments indicate that the latter approach is more promising in practice [1, 7, 9]. This is due to the fact that the worst-case efficient structures tend to be complex and hard to implement therefore big constant factors hidden in their complexity estimates prevail their theoretically superior performance.

On the other hand, the main disadvantage of the amortized approach is that it cannot be applied in real-time programs, where the worst-case bound on the running time of each individual operation is crucial.

Our solution, both simple and worst-case efficient (in the probabilistic sense), avoids these drawbacks by adopting the randomized approach, earlier applied to construct abstract data structures mainly in the context of dictionaries (e.g. [2, 11]). The idea is loosely based on leftist trees and skew heaps. All other operations are defined in terms of MELD which in both structures is performed along right paths in melded trees. The subtrees of a node on the right path are exchanged in order to keep the path short: in leftist trees – sometimes (depending on their ranks); in skew heaps – always. In our data structure MELD operation is performed along *random* paths in melded trees. This approach has the following advantages:

Simplicity. All operations are easy to implement and the constant factors in their complexity bounds are small, thus, given a fast random number generator, the heaps should perform well in practice.

Space economy. Since we do not need to preserve any balance conditions, no satellite information within nodes is necessary.

Applicability to parallel computations. A single-pass top-down scheme of each operation allows to perform a sequence of operations in a pipelined fashion. Moreover, the loose structure of a heap allows to process disjoint sets of nodes independently.

Worst-case efficiency. The execution time of each individual operation is at most logarithmic with high probability. The expected time behaviour depends on the random choices made by the algorithm rather than the distribution of an input sequence, which allows using this data structure in real-time applications.

The rest of this paper is organized as follows. In Section 2 we describe the data structure and the implementation of meldable priority queue operations. Section 3 is devoted to the efficiency analysis of these algorithms. Section 4 presents some experimental results. Finally, Section 5 contains discussion of some extensions of the data structure and the conclusions.

2 The Randomized Heap

The underlying data structure of the *randomized heap* is a binary tree with one item per node, satisfying *heap property*: if x and y are nodes and x is the parent of y then $item(x) \leq item(y)$. The heap is accessed by the root of the tree.

Let us now describe the implementation of meldable priority queue operations for randomized heap. MAKEQUEUE returns an empty tree and FINDMIN returns an item held in the root. In order to MELD two nonempty trees with roots Q_1 and Q_2, respectively, we compare the items held in the roots. The root with the smaller key, say Q_1, becomes the root of the resulting tree and Q_2, the remaining one, is recursively melded with either left or right child of Q_1, depending on

the outcome of a coin toss. More formal definition is given by the following pseudocode:

heap function MELD(**heap** Q_1, Q_2)
    if $Q_1 =$ NULL $\Rightarrow$ **return** Q_2
    if $Q_2 =$ NULL $\Rightarrow$ **return** Q_1
    if $item(Q_1) > item(Q_2) \Rightarrow Q_1 \leftrightarrow Q_2$
    if *toss_coin* = HEADS $\Rightarrow left(Q_1) :=$ MELD$(left(Q_1), Q_2)$
    else $right(Q_1) :=$ MELD$(right(Q_1), Q_2)$
    return Q_1

(The results of Section 3 imply that the recursion depth is at most logarithmic with high probability. Moreover, this tail-recursion is easily removable and serves the purpose of increasing readability only.)

The simplest way to describe all remaining priority queue operations is to define them in terms of MELD. In order to INSERT item e into heap Q we create a single node containing item e and meld it with Q. DELETEMIN melds the left and right subtrees of the root and returns the item held in the (old) root.

For DECREASEKEY and DELETE we need the parent pointer in each node. In order to decrease the value of node x in heap Q we detach the tree rooted at x from Q, adjust the item at x accordingly and then meld Q with the heap rooted at x. Operation DELETE also detaches the tree rooted at x from heap Q, and then performs DELETEMIN on heap rooted at x and finally MELD the resulting heap and Q.

3 The Efficiency Analysis

Since all non-constant-time operations are defined in terms of MELD, it is enough to analyze the complexity of melding two randomized heaps.

Let us fix an arbitrary binary tree Q with n *interior* nodes containing keys and $n+1$ *exterior* null nodes – the leaves of the tree. Define a random variable h_Q to be the length (the number of edges) of a random path from the root down to an exterior node (the child following each interior node on a path is chosen randomly and independently). In other words, the probability space is the set of all exterior nodes in Q with probability of a node at depth t equal to 2^{-t}, and h_Q is the depth of an exterior node chosen randomly with respect to this distribution.

Lemma 1. *Melding two randomized heaps Q_1 and Q_2 requires time $O(h_{Q_1} + h_{Q_2})$.*

Proof. The melding procedure traverses a random path in each tree until an exterior node in one of them is reached. □

It follows from Lemma 1 that in order to bound the complexity of melding randomized heaps it is enough to estimate h_Q for an arbitrary binary tree Q.

Theorem 1. *Let Q be an arbitrary binary tree with n interior nodes.*

(a) *The expected value $Eh_Q \leq \log(n+1)$.*
(b) *$Pr[h_Q > (c+1)\log n] < \frac{1}{n^c}$, for any constant $c > 0$.*

Proof. (a) The proof follows by induction on n. Assume $n > 0$ and let n_L and n_R be the number of interior nodes in the left (Q_L) and right (Q_R) subtree of Q, respectively (thus $n = n_L + n_R + 1$). We have

$$\begin{aligned} Eh_Q &= \frac{1}{2}((1 + Eh_{Q_L}) + (1 + Eh_{Q_R})) \leq 1 + \frac{1}{2}(\log(n_L + 1) + \log(n_R + 1)) \\ &= \log 2\sqrt{(n_L+1)(n_R+1)} \leq \log 2\frac{(n_L+1)+(n_R+1)}{2} \\ &= \log(n_L + n_R + 2) = \log(n+1) \end{aligned}$$

(b) Note that for any fixed path γ from the root to an exterior node the probability that γ is the outcome of a random walk down the tree equals $2^{-|\gamma|}$, where $|\gamma|$ is the length of γ.

Let Γ be the set of all paths from the root to an exterior node in Q with length exceeding $(c+1)\log n$. We have

$$\Pr[h_Q > (c+1)\log n] = \sum_{\gamma \in \Gamma} 2^{-|\gamma|} < \sum_{\gamma \in \Gamma} 2^{-(c+1)\log n} = |\Gamma|\, n^{-(c+1)} \leq n^{-c}$$

□

Corollary 1. *The expected time of any meldable priority queue operation on a n-node randomized heap is $O(\log n)$. Moreover, for each constant $\epsilon > 0$ there exists a constant $c > 0$ such that the probability that the time of each operation is at most $c \log n$ exceeds $1 - n^{-\epsilon}$.*

Proof. Immediate by Lemma 1 and Theorem 1. □

4 Experiments

We have carried out some tests to measure the behaviour of the randomized heap in practice. It is not hard to see that the value h_Q is bigger for more balanced trees and smaller for "thinner" ones. When we create a tree by inserting the keys $1, \ldots, n$ in the order of some permutation π then the tree is more balanced if π is closer to the sorted sequence $< 1, \ldots, n >$, and "thinner" if π is closer to the inverted sequence $< n, \ldots, 1 >$. Thus our methodology was the following: for a fixed n subsequently we created a tree

- from an almost sorted permutation ($\frac{n}{2}$ transpositions away from $< 1, \ldots, n >$),
- from a random permutation,

– from an almost inversely sorted permutation ($\frac{n}{2}$ transpositions away from $< n, \ldots, 1 >$),

then we computed the value of h and the total length of paths traversed while melding two such trees (both consisting of keys $1, \ldots, n$). Since we can get different trees even from a fixed permutation, the outcomes were averaged over 100 tests for each value of n.

The results are summarized in the following table (each displayed value is the factor c in expression $c \log(n+1)$):

	Almost sorted permutation		Random permutation		Almost inverted permutation	
n	h	meld	h	meld	h	meld
50	0.85	1.34	0.79	1.28	0.63	0.79
500	0.80	1.39	0.76	1.31	0.50	0.65
5000	0.78	1.41	0.74	1.32	0.40	0.49
15000	0.77	1.42	0.73	1.32	0.36	0.41

It turns out that in case of a tree obtained from a random permutation the value of h is just $\frac{3}{4}$ of the value for the full tree. Moreover, the total length of paths traversed while melding two such trees is about 15% smaller than doubled value of h, as used for an estimation in Lemma 1. (This is not surprising because only one of two random paths is traversed to the very end while melding.)

5 Conclusions

Before the concluding remarks let us note that the flexibility of the randomized heap can be increased by *scaling* it in the manner similar to well known d-ary heaps. Let us fix an integer $d \geq 2$ and make the underlying structure of the heap be a tree with at most d children in each node (kept in an array of size d). The only change to operation MELD is that instead of tossing a symmetric coin we choose value t from $\{1, \ldots, d\}$ at random and recursively meld the tree with the bigger key at the root with t-th subtree of the other tree. An easy adaptation of the proofs from Section 3 gives the following estimates for the complexity of operations on a randomized d-heap with at most n nodes:

– MAKEQUEUE, FINDMIN — $O(1)$
– MELD, DECREASEKEY — $O(\log_d n)$
– INSERT — $O(d + \log_d n)$ (we have to initialize d pointers in the new node to NULL)
– DELETEMIN, DELETE — $O(d \log_d n)$ (we have to meld $O(d)$ heaps)

We have presented a very simple randomized data structure capable to support all meldable priority queue operations in logarithmic time with high probability. The experiments show that the constant factors in the complexity of

the operations are in fact even smaller than those derived from the theoretical analysis. Simplicity, flexibility and small memory overhead make the randomized heap seem to be a practical choice for a meldable priority queue with worst-case performance guarantees.

The following question looks as a good starting point for further research: does the randomized approach allow us to lower the asymptotic complexity of some meldable priority queue operations while keeping the data structure simple?

References

1. T. Altman, B. Chlebus, A. Malinowski, M. Ślusarek, manuscript.
2. C. R. Aragon, R. G. Seidel, Randomized search trees. *Algorithmica* 16(1996), 464-497.
3. G. S. Brodal, Fast meldable priority queues, *Proc. 4th Workshop on Algorithms and Data Structures,* vol. 955 of *Lecture Notes in Computer Science,* 282-290, Springer-Verlag, Berlin, 1995.
4. G. S. Brodal, Worst-case efficient Priority Queues, *Proc. 17th ACM-SIAM Symposium on Discrete Algorithms,* 1996, 52-58.
5. J. R. Driscoll, H. N. Gabow, R. Shrairman, R. E. Tarjan, Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Comm. ACM* 31(1988), 1343-1354.
6. M. L. Fredman, R. Sedgewick, D. D. Sleator, R. E. Tarjan, The pairing heap: A new form of self-adjusting heap. *Algorithmica* 1(1986), 111-129.
7. D. W. Jones, An empirical comparison of priority queue and event set implementations, *Comm. ACM* 29(1986), 300-311.
8. D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching,* Addison-Wesley, Reading, MA, 1973.
9. A. M. Liao, Three priority queue applications revisited. *Algorithmica* 7(1992), 415-427.
10. K. Mehlhorn, A. K. Tsakalidis, Data Structures, In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity,* MIT Press/Elsevier, 1990.
11. W. Pugh, Skip Lists: A probabilistic alternative to balanced trees, *Comm. ACM*, 33(1990), 668-676.
12. D. D. Sleator, R. E. Tarjan, Self-adjusting binary trees, *Proc. 15th ACM Symp. on Theory of Computing,* 1983, 235-246.
13. D. D. Sleator, R. E. Tarjan, Self-adjusting heaps, *SIAM J. Comput.* 15(1986), 52-69.