International Journal of Foundations of Computer Science

MERGEABLE DOUBLE-ENDED PRIORITY QUEUES *

SEONGHUN CHO
Synopsis Inc
700 Middlefield Road
Mountain View, CA 94043
seonghun@synopsis.com

and

SARTAJ SAHNI
CISE Department, University of Florida,
Gainesville, FL 32611, USA
sahni@cise.ufl.edu

Received 2 April 1998
Revised 29 July 1998
Communicated by O. H. Ibarra

ABSTRACT

We show that the leftist tree data structure may be adapted to obtain data structures that permit the double-ended priority queue operations $Insert$, $DeleteMin$, $DeleteMax$, and $Merge$ to be done in $O(\log n)$ time where n is the size of the resulting queue. The operations $FindMin$ and $FindMax$ can be done in $O(1)$ time. Experimental results are also presented.

Keywords: Double-ended priority queues, complexity, performance.

1. Introduction

A double-ended priority queue (DEPQ) is a collection of elements each of which has a priority. The operations that may be performed on a DEPQ are $FindMax$ (find the element with maximum priority), $FindMin$ (find the element with minimum priority), $Insert$ (insert a new element into the DEPQ), $DeleteMax$ (delete an element with maximum priority), and $DeleteMin$ (delete an element with minimum priority).

Several simple and efficient implicit data structures have been proposed for the representation of a DEPQ [13, 2, 4, 10, 5, 9]. In all of these, $FindMax$ and $FindMin$ take $O(1)$ time and the remaining operations take $O(\log n)$ time each (n is the number of elements in the DEPQ).

*This work was supported in part by the Army Research Office under grant DAA H04-95-1-0111.

The twin heaps of Ref. 13, the min-max pair heaps of Ref. 10, the interval heaps of Ref. 9, and the diamond deques of Ref. 5 are virtually identical data structures. In each of these structures, an n element DEPQ is represented by a min heap with $\lceil n/2 \rceil$ elements and a max heap with the remaining $\lfloor n/2 \rfloor$ elements. The two heaps satisfy the property that each element in the min heap is $\leq$ the corresponding element in the max heap. When the number of elements in the DEPQ is odd, the min heap has one element (i.e., element $\lceil n/2 \rceil$) that has no corresponding element (recall that a heap is a complete binary tree [8] and two elements correspond if they occupy the same position in their respective binary trees) in the max heap. In the twin heaps of Ref. 13, this is handled as a special case and one element is kept outside of the two heaps. In min-max pair heaps, interval heaps, and diamond deques, the case when n is odd is handled by requiring element $\lceil n/2 \rceil$ of the min heap to be $\leq$ element $\lfloor n/4 \rfloor$ of the max heap.

In the twin heaps of Ref. 13, the min and max heaps are stored in two arrays MIN and MAX using the standard array representation of a complete binary tree [8]. The correspondence property becomes $MIN[i] \leq MAX[i]$, $1 \leq i \leq \lfloor n/2 \rfloor$. In the min-max pair heaps of Ref. 10 and the interval heaps of Ref. 9, the two heaps are stored in a single array $MINMAX$ and we have $MINMAX[i].min$ being the i'th element of the min heap, $1 \leq i \leq \lceil n/2 \rceil$ and $MINMAX[i].max$ being the i'th element of the max heap, $1 \leq i \leq \lfloor n/2 \rfloor$. In the diamond deque [5], the two heaps are mapped into a single array with the min heap occupying even positions (beginning with position 0) and the max heap occupying odd positions (beginning with position 1). Since this mapping is slightly more complex than the ones used in twin heaps, min-max pair heaps, and interval heaps, actual implementations of the diamond deque are expected to be slightly slower than implementations of the remaining three structures.

In the min-max heap structure [2], all n DEPQ elements are stored in a complete binary tree with alternating levels being min levels and max levels [8]. This structure is more complex than the ones mentioned above and so is expected to run slower than those by a constant factor. The deap structure of Ref. 4 is similar to the two heap structures of Refs. 13,10,9,5. At the conceptual level, we have a min heap and a max heap. However, the distribution of elements between the two is not $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Rather, we begin with an $n+1$ node complete binary tree. Its left subtree is the min heap and its right the max heap. The correspondence property is more involved as several leaf elements of the min heap may not have a corresponding element in the max heap. Let i be a leaf of the min heap. Let j be the corresponding position in the max heap. Element i of the min heap is required to be $\leq$ element j of the max heap. If the max heap has no element at position j, then element i of the min heap is to be $\leq$ element $\lfloor j/2 \rfloor$ of the max heap. While deaps are faster than min-max heaps (by a constant factor), they are slower than twin heaps, min-max pair heaps, and interval heaps.

A mergeable DEPQ (MDEPQ) supports all of the operations supported by a DEPQ as well as the operation $Merge$ which merges (combines) two DEPQs into one. To merge two DEPQs in less than linear time, it is essential that the DEPQs

be represented using explicit pointers (rather than implicit ones as in the array representation of a heap) as otherwise a linear number of elements need to be moved from their initial to their final locations. Olariu et al. [10] have shown that when the min-max pair heap is represented in such a way, an n element DEPQ may be merged with a k element one ($k \leq n$) in $O(\log(n/k) * \log k)$ time. When $k = \sqrt{n}$, this is $O(\log^2 n)$. For the remaining DEPQ structures, no sublinear $Merge$ method is known. In fact, Hasham and Sack [7] have shown that the complexity of merging two min-max heaps of size n and k, respectively, is $\Omega(n + k)$. Brodal [3] has developed an MDEPQ implementation that allows one to find the min and max elements, insert an element, and merge two priority queues in O(1) time. The time needed to delete the minimum or maximum element is O($\log n$). Although the asymptotic complexity provided by this data structure are the best one can hope for [3], the data structure has practical limitations. First, each element is represented twice using a total of 16 fields per element. Second, even though the delete operations have O($\log n$) complexity, the constant factors are very high and the data structure will not perform well unless find, insert, and merge are the primary operations.

In this paper, we show that leftist trees [6, 8, 12] may be adapted to obtain a simple representation for MDEPQs in which $Merge$ takes $O(\log nk) = O(logn)$ time and the remaining operations have the same asymptotic complexity as when any of the aforementioned DEPQ representations is used. In Section 2, we present a simple adaption that does this. This is refined, in Section 3, to obtain a more space efficient adaptation. The adaptation of Section 3, the twin leftist tree, represents each element just once and requires 6 fields per element (in contrast to the 16 fields required by Brodal's structure). Experimental results are presented in Section 4. These results show that our data structure is very practical. In fact, it outperforms the data structures deveoped earlier for DEPQs even on operation mixes that do not contain any merge operations.

2. First Adaptation Of Leftist Trees

Crane [6] has proposed a simple data structure, the *leftist tree*, for the representation of single ended mergeable priority queues. Let T be a binary tree. The extended binary tree T' corresonding to T is obtained by replacing each null child of T by an external node. For any node x in T, let $sh(x)$ be the length of a shortest path from x to an external node in the subtree of T' rooted at x. Let $sh(x) = 0$ for x an external node. T is a leftist tree iff sh(leftchild x) $\geq sh$(rightchild x) for all nodes x of T. Let $sh(T) = sh$(root of T).

Lemma 1 (Refs. 6,8,12) *For any n node leftist tree T, $sh(T) = O(\log n)$ and the rightmost root to external node path has length $sh(T)$.*

In a min (max) leftist tree, the priority of the element in each node x is $\leq$ ($\geq$) that of all elements in the subtree rooted at x. In a min (max) leftist tree, $FindMin$ ($FindMax$) can be done in $O(1)$ time, $Insert$ and $DeleteMin$ ($DeleteMax$) take $O(\log n)$ time, and a merge takes $O(\log nk)$ time.

A simple way to use leftist trees to represent an MDEPQ is to simultaneously

maintain both a min and a max leftist tree of all n MDEPQ elements. Figure 1 shows an example representation for the case when $n = 7$ and the element priorities are (1, 2, 4, 5, 5, 6, 8). Let us call this representation of an MDEPQ the *dual leftist tree* representation.

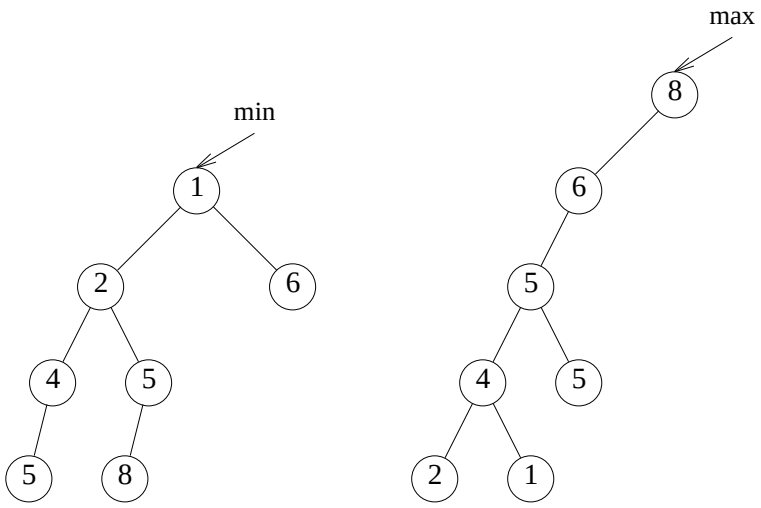


Fig. 1. Dual leftist trees

We obtain a space efficient representation of a dual leftist tree when we use nodes with the fields: $L1$ (left child for min leftist tree), $L2$, (left child for max leftist tree) $R1$ (right child for min leftist tree), $R2$ (right child for max leftist tree), $P1$ (parent for min leftist tree), $P2$ (parent for max leftist tree), $sh1$ (sh() value for min leftist tree), $sh2$ (sh() value for max leftist tree), and *data*. $L1$, $R1$, $P1$, and $sh1$ ($L2$, $R2$, $P2$, and $sh2$) are used to form the min (max) leftist tree. $L1$ ($R1$, $P1$) points to the left child (right child, parent) of the node in the min leftist tree. $sh1$ ($sh2$) is the sh value of the node when viewed as a member of the min (max) leftist tree. With this node structure, the dual leftist tree representation of an n element MDEPQ requires space for n data fields, $6n$ pointer fields, and $2n$ sh fields.

Let Q be an MDEPQ represented as a dual leftist tree. Let $Q.min$ and $Q.max$, respectively, point to the min and max leftist trees of Q. The MDEPQ $FindMax$ ($FindMin$) operation is done in $O(1)$ time by simply invoking the max (min) leftist tree $FindMax$ ($FindMin$) operation on the leftist tree $Q.max$ ($Q.min$). To insert a new item into Q, we insert it into $Q.max$ and $Q.min$ using the $O(\log n)$ time codes to insert into a max and min leftist tree. Hence, the $MDEPQ$ insert operation takes $O(\log n)$ time. To merge an MDEPQ $Q1$ with n elements with an MDEPQ $Q2$ with k elements, we merge $Q1.min$ with $Q2.min$ and $Q1.max$ with $Q2.max$. Each of these merges takes $O(\log nk)$ time. So, the two $MDEPQ$s may be merged in $O(\log nk)$ time. For the MDEPQ $DeleteMin$ ($DeleteMax$) operation, let x be the node at the root of the min (max) leftist tree $Q.min$ ($Q.max$). We need to perform

a $DeleteMin$ ($DeleteMax$) in the min (max) leftist tree and also delete node x from the max (min) leftist tree. Since $DeleteMax$ ($DeleteMin$) is a standard max (min) leftist tree operation that takes $O(\log n)$ time, we need only be concerned with the deletion of an arbitrary node x from a max and a min tree. Figure 2 shows how to do this from the min leftist tree $Q.min$. The notation $x \rightarrow P1$ means the $P1$ filed of the node pointed at by x. The algorithm for deletion of x from a max leftist tree is similar.

Step 1: **if** ($x == Q.min$) do a min leftist tree $DeleteMin$ on $Q.min$; **return**;

Step 2: $p = x \rightarrow P1$; //parent of x in min leftist tree

Step 3: Change appropriate child field ($L1$ or $R1$) of p from x to $x \rightarrow L1$.
Follow path from p towards root of min leftist tree using $P1$ pointers and swapping subtrees ($L1$ and $R1$) as needed to ensure leftist tree property and adjusting $sh1$ values.
Stop when this has been done at the root or when we reach a node whose $sh1$ value doesn't change.

Step 4: Merge the min leftist trees $Q.min$ and $x \rightarrow R1$ ($x \rightarrow R1$ may be null).

Step 5: Delete the node x;

Fig. 2. Deletion of node x from the min leftist tree $Q.min$

For the complexity of this algorithm, we note that steps 1, 2, 4, and 5 together take $O(\log n)$ time. For Step 3, note that even though the path from p to the root may be $O(n)$ long, only $O(\log n)$ nodes on this path can be reached as the reached nodes (except possibly the last one) have increasing $sh1$ value following Step 3 and the maximum $sh1$ value is $O(\log n)$.

3. Second Adaptation Of Leftist Trees

3.1. Strategy

This adaptation is called the *twin leftist tree* adaptation. We borrow ideas used in the twin heaps representation of Ref. 13. At most one of the n MDEPQ elements will be kept in a buffer and the remaining elements distributed between a min ($MinLT$) and a max ($MaxLT$) leftist tree. Elements in the min and max leftist trees satisfy the following correspondence property:

> For each element e in $MinLT$ ($MaxLT$), there is at least one element in $MaxLT$ ($MinLT$) whose priority is $\geq$ ($\leq$) that of e.

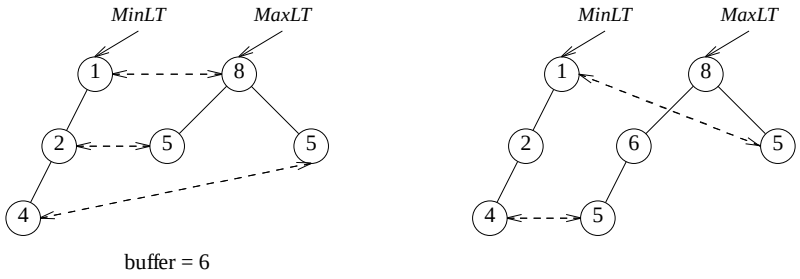
As a result of this correspondence property, the element with least (max) priority is either the one (if any) in the buffer or the one in the root of $MinLT$ ($MaxLT$). By comparing these two elements, $FindMin$ ($FindMax$) can be accomplished in $O(1)$ time.

Each leftist tree node has the following fields: L, R, P, C, sh, and $data$. L (R, P) points to the left child (right child, parent); C points to a node (called the corresponding node) in the other leftist tree; and sh is as defined for a leftist tree. For any node x in the min (max) leftist tree, $x \rightarrow C$ is either $NULL$ or points to a node in the max (min) leftist tree that has priority $\geq$ ($\leq$) that of x. Furthermore, if $x \rightarrow C$ is not null, then x equals $x \rightarrow C \rightarrow C$.

There are several ways [13, 4, 8] in which we can enforce the correspondence property. Two of these are given below:

1. *Total Correspondence.* In this, no node in either $MaxLT$ or $MinLT$ has a null correspondence pointer C. As a result of this and the requirement $x \rightarrow C \rightarrow C$ equal x for every node with a non-null C pointer, the number of nodes in $MaxLT$ equals that in $MinLT$. An example total correspondence twin leftist tree with $n = 7$ and priorities (1, 2, 4, 5, 5, 6, 8) is shown in Figure 3(a).

2. *Leaf Correspondence.* In this, no leaf of $MinLT$ or $MaxLT$ may have a null pointer. Figure 3(b) gives an example of a leaf correspondence twin leftist tree.



(a) Total correspondence (b) Leaf correspondence

Broken arrows denote correspondence pointers
Parent fields not shown

Fig. 3. Twin leftist trees

The storage requirements for the twin leftist tree adaptation are $4n$ pointer fields, n sh fields, and n data fields (including that in the buffer).

3.2. Algorithms For Total Correspondence

To insert an element e when the buffer is empty, we simply put e into the buffer. When the buffer is not empty, the smaller of e and the element in the buffer is inserted into $MinLT$ and the larger into $MaxLT$. Correspondence pointers between these two elements are also set up. Hence an insertion takes either $O(1)$ time (when the buffer is empty) and $O(\log n)$ time.

To merge two MDEPQs $Q1$ and $Q2$ to get the MDEPQ $Q3$, the two min leftist trees $Q1.MinLT$ and $Q2.MinLT$ are merged to get $Q3.MinLT$; the two max leftist

trees $Q1.MaxLT$ and $Q2.MaxLT$ are merged to get $Q3.MaxLT$; $Q3.buffer$ is set empty; the elements (if any) in $Q1.buffer$ and $Q2.buffer$ are inserted into $Q3$. Since two min (max) leftist trees can be merged in $O(\log nk)$ time and since an insertion into a total correspondence MDEPQ takes $O(\log n)$ time, the two MDEPQs are merged in $O(\log nk)$ time.

The $DeleteMax$ operation may be implemented as in Figure 4. This requires that we be able to delete an arbitrary node x from a min leftist tree. In Section 2, we saw how this could be done in logarithmic time. So, the complexity of the $DeleteMax$ algorithm of Figure 4 is $O(\log n)$. $DeleteMin$ is similar.

Step 1: If the buffer is empty, do a max leftist tree $DeleteMax$ on $Q.MaxLT$. Delete the corresponding node from $Q.MinLT$ and put this element in the buffer. Return.

Step 2: [Buffer is not empty] If the element in the buffer has priority $\geq$ that of the one in the root of $Q.MaxLT$ empty the buffer and return.

Step 3: Do a max leftist tree $DeleteMax$ on $Q.MaxLT$. If the element in the buffer has priority $\geq$ that of the corresponding node, insert the buffer element into $Q.MaxLT$ and set a correspondence between these two. If not, delete the corresponding node from $Q.MinLT$ and insert it into $Q.MaxLT$; insert the buffer element into $Q.MinLT$; set a correspondence between the two.

Fig. 4. $DeleteMax$ from a total correspondence twin leftist tree

3.3. Algorithms For Leaf Correspondence

With leaf correspondence twin leftist trees, we can avoid some of the work done when total correspondence leftist trees are used. For example, when inserting an element into an MDEPQ with a non empty buffer, it may be possible to get away with a single insert into the min leftist tree rather than an insert into both the min and max leftist trees (see Figure 5). The asymptotic complexity remains $O(\log n)$.

Step 1: If buffer is empty, put the new element e into it and return.

Step 2: Let b be the element in the buffer. Insert the smaller of b and e (call this element *small*) into $Q.MinLT$

Step 3: If *small* is not a leaf of $Q.MinLT$, put $\{b, e\} - small$ into the buffer and return.

Step 4: [*small* is a leaf] Insert $\{b, e\} - small$ into $Q.MaxLT$ and set correspondence pointers between b and e. Set the buffer to empty.

Fig. 5. Insertion into a leaf correspondence twin leftist tree

Two leaf correspondence twin leftist trees may be merged in $O(\log nk)$ time

using the same procedure as used to merge two total correspondence leftist trees. To see that this works, we note that when two leftist trees are merged using the standard merge algorithm [8], no nonleaf node becomes a leaf. As a result, no new correspondences need to be established.

The $DeleteMax$ algorithm is given in Figure 6. Like the $DelteMax$ algorithm for total correspondence twin leftist trees, this makes a non-standard deletion from a leftist tree. This time, however, the non-standard deletions are restricted to leaf node deletions which can be done by a simple path traversal towards the root stopping when we reach a node whose sh value doesn't change. The complexity of the algorithm of Figure 6 is $O(\log n)$. The $DeleteMin$ algorithm is similar.

Step 1: [Empty buffer] If the buffer is empty, do a max leftist tree $DeleteMax$ on $Q.MaxLT$.
If the correspondence pointer of the deleted node is null return.
If the corresponding node, x, is not a leaf of $Q.MinLT$, set its correspondence pointer to $NULL$ and return.
Let p be the parent (if any) of x in $MinLT$.
Delete the leaf x from $MinLT$.
If p has now become a leaf of $MinLT$ and p's correspondence pointer is null, then insert x into $MaxLT$ and establish correspondence pointers between p and x. Otherwise, put the element in x into the buffer.
Return.

Step 2: [Buffer is not empty] If the element in the buffer has priority $\geq$ that of the one in the root of $Q.MaxLT$ empty the buffer and return.

Step 3: Do a max leftist tree $DeleteMax$ on $Q.MaxLT$.
If the correspondence pointer of the deleted node is null return.
If the corresponding node, x, is not a leaf of $Q.MinLT$, set its correspondence pointer to $NULL$ and return.
If the buffer element, b, has priority $\geq$ that of x, insert it into $MaxLT$, establish correspondences pointers between x and b, return.
Let p be the parent (if any) of x in $MinLT$.
Delete the leaf x from $MinLT$ and insert it into $MaxLT$.
If p has now become a leaf of $MinLT$ and p's correspondence pointer is null, then establish correspondence pointers between p and x and return.
If x is a leaf, insert the buffer element, b, into $MinLT$ and establish correspondence pointers between x and b and return.
Set x's correspondence pointer to null.

Fig. 6. $DeleteMax$ from a leaf correspondence twin leftist tree

4. Experimental Results

We compared the performance of our second adaptation, the twin leftist tree (TLT), of leftist trees to that of other structures proposed for the representation of double-ended priority queues. Specifically, for the experimental comparison, we

considered the array based structures min-max heaps (MMH) [2] and deaps (Deap) [4] as well as the pointer based structure min-max pair heap (MMP) [10]. Additionally, the pointer based dictionary structures unbalanced binary search tree (BST), AVL tree (AVL), and treap (TRP) [1] were adapted to directly support the deletion of the minimum and maximum elements. All of the structures were coded in C and run on a SUN SPARC-5. The code was compiled using the UNIX cc compiler in optimization mode. The AVL code was based on that of Ref. 11. The pointer codes saved deleted nodes on an internal chain so that calls to **malloc** were made only when an insert caused the number of elements to exceed the largest number of elements previously in the structure.

Since the TLT is the only structure in our test set that supports logarithmic merging of two double-ended priority queues, we know apriori that any experiment involving a large number of merge operations will favor the TLT structure. Instead, we focussed our experimental study on determining how well the TLT performs relative to the other structures when there are no merge operations. So, in the experiments, we began with a structure with n elements for $n = 100$, 1000, and 10000 and performed a sequence of m DEPQ operations (insert, delete max, and delete min). The choices for m were 50,000, 100,000, and 200,000. For each m a random sequence of insert, delete max, and delete min operations was also generated. The inserts were generated with probablity 0.5 and each type of delete operation had probability 0.25. This tended to keep the structure size close to the initial value of n. The keys of the elements being inserted were generated in four different ways: (a) random1 $\cdots$ random keys in the range 1 through 10^6; (b) random2 $\cdots$ random keys in the range 1 through 1000; (c) increasing keys; and (d) decreasing keys. Data set *random*2 is expected to have many duplicate keys.

In our first experiment, we obtained the number of key comparisons performed by each of the methods. These are reported in Table 1. The reported numbers for the data sets random1 and random2 are the average of ten runs. The standard deviation in the measured number of comparisons for these ten data sets is given in Table 2. For the increasing and decreasing data sets, the BST structure was excluded from the experiments as on these data BST's exhibit their worst performance and the run times are too large. As can be seen from the table, the TLT structure consistently performs fewer comparisons than any of the others. So, even when no merges are to be performed, the TLT is expected to outperform the other structures provided the cost of a key comparison is "sufficiently high".

While the key comparison results are independent of machine architecture, run time results are not. These are quite sensitive to the caching strategies, compiler optimization capabilities, relative cost of array and pointer accesses, etc. Our run time experiments were done once using integer keys and again using floating point keys. The data for the floating point case was the same as for the integer case except that the type of the key field was changed from **int** to **float**. The measured run times using integer keys are shown in Table 4. As was the case for the number of comparisons, the times for the data sets random1 and random2 are the average of ten runs. The standard deviation in the measured times for these ten data sets

inputs	m	n	BST	MMH	Deap	MMP	TLT	AVL	TRP
		100	263	492	307	397	138	184	202
	50,000	1000	311	837	465	556	171	272	282
		10000	384	1148	547	634	372	349	405
		100	492	961	601	783	267	363	403
random1	100,000	1000	649	1662	932	1114	314	545	546
		10000	709	2329	1165	1342	603	693	746
		100	1125	2119	1298	1662	538	776	853
	200,000	1000	1412	3405	1914	2278	593	1103	1068
		10000	1408	4670	2433	2792	981	1379	1402
		100	233	439	281	372	130	172	194
	50,000	1000	368	805	454	543	167	268	281
		10000	607	1117	553	639	349	351	444
		100	539	988	615	796	265	368	411
random2	100,000	1000	855	1660	939	1119	309	536	574
		10000	1326	2235	1182	1354	571	692	871
		100	1247	2082	1281	1643	533	765	837
	200,000	1000	2489	3372	1910	2271	584	1064	1148
		10000	3318	4502	2475	2822	922	1362	1785
		100	–	446	322	409	97	178	179
	50,000	1000	–	821	547	634	100	282	251
		10000	–	1196	753	839	100	363	306
		100	–	976	688	862	196	375	371
decreasing	100,000	1000	–	1661	1107	1280	200	563	502
		10000	–	2395	1509	1683	200	725	656
		100	–	1980	1392	1741	392	756	747
	200,000	1000	–	3341	2229	2575	400	1123	998
		10000	–	4789	3031	3378	400	1451	1255
		100	–	525	373	462	99	195	190
	50,000	1000	–	860	567	656	100	283	252
		10000	–	1199	766	854	100	363	302
		100	–	1046	743	921	197	387	375
increasing	100,000	1000	–	1736	1144	1321	200	563	495
		10000	–	2399	1535	1712	200	726	631
		100	–	1936	1397	1753	391	736	735
	200,000	1000	–	3429	2265	2620	399	1126	1003
		10000	–	4800	3081	3436	399	1451	1248

m = the number of operations performed
n = the number of elements in initial data structures

Table 1. The number of key comparisons (in thousands)

inputs	m	n	BST	MMH	Deap	MMP	TLT	AVL	TRP
		100	37171	49653	23503	23623	2947	13065	10892
	50,000	1000	13485	16765	8177	8196	1786	2106	9250
		10000	7021	1170	1040	1129	2322	1172	14308
		100	1036	418	481	578	139	491	2424
random1	100,000	1000	12643	623	451	630	328	2432	5451
		10000	4421	69	197	308	375	1799	9020
		100	7770	1174	485	659	144	93	614
	200,000	1000	14384	886	73	343	497	296	5254
		10000	2437	112	120	79	296	4424	2283
		100	723	1188	602	600	285	385	2223
	50,000	1000	11047	321	267	260	205	1369	4903
		10000	9505	1013	183	183	535	1159	7506
		100	11281	91	150	259	313	406	1792
random2	100,000	1000	29586	203	172	119	331	473	15151
		10000	3040	716	291	413	283	568	22437
		100	5399	86	222	217	384	193	2503
	200,000	1000	21103	1125	263	196	194	3458	17357
		10000	21495	68	172	279	287	1307	11545

m = the number of operations performed
n = the number of elements in initial data structures

Table 2. Standard deviation of the number of key comparisons

is given in Table 4. The corresponding times using floating point keys are given in Table 5 and 6. Once again, for the increasing and decreasing data sets, the BST structure was excluded from the experiments as on these data BST's exhibit their worst performance and the run times are too large.

Table 4 shows that the BST has best performance on the random1 data set. For the random2 data set, BST is best when $n = 100$. For the remaining values of n, BST and TLT were tied on one data set, TLT was best on four, and Deap was best on two. On the increasing and decreasing data sets, TLT was consistently best. Since the BST is very slow on ordered data (i.e., increasing or decreasing), it is not recommended for general applications. So, it is worthwhile to see which structure did second best the times when BST was best. The TLT held second palce on 11 of these data sets and the Deap did this on the remaining three.

The change to floating point keys increased the cost of a key comparison. However, this increase was not enough for TLT to outperform BST on the random1 data set. BST remained best on seven of the eight data sets and TLT was best on one. On the random2 data set, there was no clear winner. The BST, TLT, AVL, TRP data structures were competitive. The TLT remained consistently superior of the decreasing data set and was generally best on the increasing data. On this latter data set, the randomized structure TRP tied the TLT three times and was faster twice.

5. Conclusion

We have proposed two adaptations of the leftist tree structure to support mergeable double-ended priority queues. Both permit one to perform each of the operations: insert, delete max, delete min, and merge in logarithmic time. For applications in which good amortized complexity suffices, the Fibonacci heap data structure described in Refs. 8,12 may be adapted to obtain a twin Fibonacci heap. Using this data structure, inserts and merges can be done in $\Theta(1)$ time while delete min and delete max have amortized complexity $O(\log n)$. The second of our adaptations, the twin leftist tree, is more space efficient than the first (dual leftist tree). Experimental results obtained by us indicate that the twin leftist tree is actually more efficient (i.e., faster) than data structures (e.g., Deap and MMH) previously proposed for non-mergeable double-ended priority queues.

References

1. C. R. Aragon and R. G. Seidel, Randomized Search Trees, Proc. 30th Ann. IEEE Symposium on Foundations of Computer Science, pp. 540-545, October 1989.
2. M. Atkinson, J. Sack, N. Santoro, and T. Strothotte, Min-max heaps and generalized priority queues, *Communications of the ACM*, 29, 996-1000, 1986.
3. G. Brodal, Fast meldable priority queues, *Workshop on Algorithms and Data Structures*, 1995.
4. S. Carlsson, The deap – A double ended heap to implement double ended priority queues, *Information Processing Letters*, 26, 33-36, 1987.
5. S. Chang and M. Du, Diamond deque: A simple data structure for priority deques,

inputs	m	n	BST	MMH	Deap	MMP	TLT	AVL	TRP
		100	0.11	0.20	0.14	0.13	0.12	0.18	0.16
	50,000	1000	0.12	0.29	0.20	0.20	0.16	0.22	0.19
		10000	0.16	0.39	0.24	0.29	0.35	0.28	0.28
		100	0.21	0.38	0.28	0.27	0.24	0.36	0.31
random1	100,000	1000	0.26	0.58	0.39	0.38	0.28	0.43	0.37
		10000	0.31	0.79	0.49	0.56	0.58	0.54	0.50
		100	0.45	0.81	0.58	0.56	0.49	0.73	0.64
	200,000	1000	0.55	1.19	0.79	0.77	0.55	0.87	0.73
		10000	0.60	1.57	0.99	1.08	0.96	1.05	0.93
		100	0.10	0.18	0.13	0.13	0.12	0.17	0.15
	50,000	1000	0.15	0.28	0.19	0.19	0.15	0.22	0.18
		10000	0.27	0.38	0.25	0.29	0.33	0.29	0.31
		100	0.22	0.38	0.28	0.27	0.24	0.36	0.32
random2	100,000	1000	0.33	0.59	0.40	0.38	0.28	0.44	0.38
		10000	0.57	0.75	0.49	0.57	0.55	0.55	0.58
		100	0.48	0.80	0.58	0.55	0.49	0.72	0.63
	200,000	1000	0.91	1.17	0.78	0.77	0.55	0.86	0.75
		10000	1.38	1.51	1.01	1.11	0.90	1.06	1.13
		100	–	0.18	0.15	0.13	0.08	0.17	0.15
	50,000	1000	–	0.30	0.23	0.22	0.10	0.20	0.18
		10000	–	0.40	0.28	0.28	0.10	0.23	0.20
		100	–	0.38	0.30	0.27	0.20	0.32	0.32
decreasing	100,000	1000	–	0.60	0.45	0.48	0.20	0.45	0.40
		10000	–	0.82	0.57	0.60	0.20	0.48	0.40
		100	–	0.80	0.62	0.57	0.37	0.65	0.62
	200,000	1000	–	1.22	0.88	0.93	0.40	0.83	0.73
		10000	–	1.65	1.17	1.23	0.40	0.93	0.78
		100	–	0.20	0.17	0.15	0.10	0.17	0.17
	50,000	1000	–	0.30	0.23	0.23	0.10	0.22	0.18
		10000	–	0.42	0.30	0.28	0.12	0.25	0.22
		100	–	0.42	0.32	0.28	0.22	0.33	0.32
increasing	100,000	1000	–	0.63	0.45	0.45	0.23	0.43	0.35
		10000	–	0.83	0.58	0.60	0.22	0.50	0.40
		100	–	0.80	0.60	0.53	0.43	0.67	0.62
	200,000	1000	–	1.25	0.92	0.92	0.48	0.85	0.72
		10000	–	1.68	1.18	1.22	0.47	0.95	0.78

Time Unit : *sec*
m = the number of operations performed
n = the number of elements in initial data structures

Table 3. Run time using integer keys

inputs	m	n	BST	MMH	Deap	MMP	TLT	AVL	TRP
		100	0.015	0.015	0.013	0.009	0.008	0.013	0.008
	50,000	1000	0.008	0.011	0.007	0.007	0.011	0.007	0.011
		10000	0.009	0.008	0.008	0.008	0.005	0.011	0.012
		100	0.008	0.007	0.008	0.009	0.008	0.008	0.007
random1	100,000	1000	0.009	0.005	0.008	0.008	0.009	0.007	0.010
		10000	0.008	0.010	0.011	0.008	0.007	0.011	0.008
		100	0.000	0.008	0.009	0.011	0.011	0.014	0.011
	200,000	1000	0.007	0.008	0.010	0.008	0.009	0.008	0.007
		10000	0.008	0.013	0.011	0.005	0.011	0.009	0.009
		100	0.005	0.008	0.009	0.007	0.000	0.008	0.008
	50,000	1000	0.005	0.005	0.008	0.010	0.005	0.011	0.009
		10000	0.007	0.000	0.007	0.008	0.011	0.008	0.008
		100	0.009	0.007	0.007	0.007	0.008	0.008	0.017
random2	100,000	1000	0.012	0.013	0.007	0.005	0.009	0.015	0.011
		10000	0.008	0.007	0.008	0.009	0.007	0.007	0.011
		100	0.008	0.009	0.013	0.000	0.007	0.011	0.005
	200,000	1000	0.015	0.011	0.005	0.008	0.010	0.010	0.012
		10000	0.008	0.005	0.016	0.015	0.000	0.008	0.011

Time Unit : *sec*
m = the number of operations performed
n = the number of elements in initial data structures

Table 4. Standard deviation of run time using integer keys

inputs	m	n	BST	MMH	Deap	MMP	TLT	AVL	TRP
		100	0.15	0.28	0.21	0.21	0.16	0.19	0.15
	50,000	1000	0.18	0.44	0.30	0.33	0.21	0.25	0.20
		10000	0.24	0.60	0.37	0.46	0.48	0.33	0.33
		100	0.28	0.56	0.41	0.41	0.33	0.39	0.31
random1	100,000	1000	0.37	0.88	0.59	0.65	0.39	0.49	0.39
		10000	0.44	1.20	0.75	0.90	0.79	0.64	0.59
		100	0.63	1.20	0.86	0.87	0.67	0.81	0.63
	200,000	1000	0.80	1.81	1.21	1.30	0.75	0.99	0.76
		10000	0.85	2.39	1.52	1.71	1.31	1.23	1.06
		100	0.14	0.26	0.19	0.20	0.16	0.18	0.15
	50,000	1000	0.20	0.43	0.29	0.32	0.21	0.25	0.20
		10000	0.40	0.59	0.37	0.47	0.45	0.34	0.36
		100	0.31	0.56	0.41	0.42	0.33	0.40	0.31
random2	100,000	1000	0.47	0.89	0.60	0.66	0.39	0.49	0.40
		10000	0.84	1.17	0.77	0.92	0.76	0.65	0.68
		100	0.68	1.18	0.85	0.87	0.66	0.80	0.63
	200,000	1000	1.34	1.79	1.22	1.29	0.75	0.99	0.80
		10000	2.07	2.32	1.55	1.77	1.24	1.28	1.32
		100	–	0.27	0.20	0.23	0.12	0.20	0.15
	50,000	1000	–	0.48	0.33	0.38	0.13	0.25	0.17
		10000	–	0.63	0.45	0.47	0.13	0.28	0.20
		100	–	0.63	0.47	0.50	0.25	0.40	0.33
decreasing	100,000	1000	–	1.00	0.70	0.85	0.25	0.53	0.40
		10000	–	1.28	0.88	0.97	0.27	0.57	0.43
		100	–	1.18	0.90	0.92	0.50	0.78	0.60
	200,000	1000	–	1.88	1.37	1.62	0.52	1.00	0.75
		10000	–	2.58	1.80	2.02	0.53	1.13	0.87
		100	–	0.32	0.23	0.25	0.15	0.22	0.15
	50,000	1000	–	0.48	0.35	0.40	0.18	0.25	0.18
		10000	–	0.63	0.45	0.47	0.20	0.30	0.20
		100	–	0.60	0.47	0.48	0.33	0.40	0.30
increasing	100,000	1000	–	0.97	0.70	0.82	0.35	0.52	0.38
		10000	–	1.28	0.92	0.97	0.35	0.57	0.43
		100	–	1.18	0.90	0.92	0.65	0.78	0.60
	200,000	1000	–	1.93	1.57	1.58	0.70	1.02	0.73
		10000	–	2.58	1.83	1.98	0.68	1.13	0.83

Time Unit : *sec*
m = the number of operations performed
n = the number of elements in initial data structures

Table 5. Run time using real (double) keys

inputs	m	n	BST	MMH	Deap	MMP	TLT	AVL	TRP
		100	0.014	0.021	0.011	0.015	0.009	0.008	0.008
	50,000	1000	0.011	0.011	0.012	0.011	0.008	0.007	0.011
		10000	0.008	0.005	0.007	0.010	0.007	0.000	0.012
		100	0.007	0.011	0.008	0.008	0.005	0.011	0.008
random1	100,000	1000	0.010	0.000	0.008	0.007	0.011	0.008	0.011
		10000	0.010	0.000	0.010	0.005	0.008	0.011	0.023
		100	0.009	0.010	0.008	0.008	0.015	0.011	0.009
	200,000	1000	0.018	0.011	0.009	0.010	0.007	0.008	0.008
		10000	0.009	0.008	0.008	0.007	0.015	0.007	0.011
		100	0.008	0.008	0.008	0.010	0.008	0.009	0.005
	50,000	1000	0.010	0.008	0.008	0.011	0.008	0.007	0.007
		10000	0.010	0.008	0.008	0.008	0.008	0.008	0.008
		100	0.008	0.010	0.007	0.008	0.010	0.007	0.009
random2	100,000	1000	0.019	0.008	0.007	0.012	0.008	0.008	0.010
		10000	0.008	0.011	0.008	0.011	0.008	0.011	0.018
		100	0.005	0.012	0.007	0.007	0.005	0.005	0.009
	200,000	1000	0.012	0.008	0.009	0.008	0.005	0.008	0.016
		10000	0.017	0.008	0.005	0.011	0.011	0.008	0.013

Time Unit : *sec*
m = the number of operations performed
n = the number of elements in initial data structures

Table 6. Standard deviation of run time using real keys

Information Processing Letters, 46, 231-237, 1993.

6. C. Crane, *Linear lists and priority queues as balanced binary trees*, Technical Report CS-72-259, Computer Science Department, Stanford University, CA, 1972.
7. A. Hasham and J. Sack, Bounds for min-max heaps, *BIT*, 27, 315-323, 1987.
8. E. Horowitz, S. Sahni, D. Mehta, *Fundamentals of Data Structures in C++*, Computer Science Press, NY, 1995.
9. J. van Leeuwen and D. Wood, Interval heaps, *The Computer Journal*, 36, 3, 209-216, 1993.
10. S. Olariu, C. Overstreet, and Z. Wen, A mergeable double-ended priority queue, *The Computer Journal*, 34, 5, 423-427, 1991.
11. T. Papadakis, Skip Lists and Probabilistic Analysis of Algorithms, PhD Dissertation, Univ. of Waterloo, 1993.
12. R. Tarjan, *Data structures and network algorithms*, SIAM, Philadelphia, PA, 1983.
13. J. Williams, Algorithm 232, *Communications of the ACM*, 7, 347-348, 1964.