Correspondence Based Data Structures For Double Ended Priority Queues

Kyun-Rak Chong
Department of Computer Engineering
HongIk University
Seoul, Korea
chong@cs.hongik.ac.kr

Sartaj Sahni
Computer & Information Science & Engineering Department
University of Florida
Gainesville, FL 32611
sahni@cise.ufl.edu

November 4, 1998

1 Introduction

A *min priority queue* (*minPQ*) is a data structure which supports the following operations :

- `FindMin(Q)` : return the minimum element in `Q`
- `DeleteMin(Q)` : delete the minimum element in `Q`
- `Insert(Q,x)` : insert x into the minPQ `Q`

A *max priority queue* (*maxPQ*) is an analogous data structure in which the operations `FindMin(Q)` and `DeleteMin(Q)` are replaced by the operations `FindMax(Q)` and `DeleteMax(Q)`. Several implicit and explicit data structures have been developed for minPQs (and hence for maxPQs) [10, 16, 5, 8, 2, 6, 14, 9, 15].

A *min meldable priority queue* (*minMPQ*) is a min priority queue which also supports the operation

- `Meld`(Q_1,Q_2) : return a min priority queue that contains all the elements in minPQs Q_1 and Q_2. Q_1 and Q_2 may be destroyed by the operation.

A *maxMPQ* is defined similarly. Among the known priority structures, the structure, Fast Meldable Priority Queue (FMPQ), has the best asymptotic properties - `DeleteMin`(Q) runs in logarithmic time and the remaining operations take constant time [2].

A double ended priority queue (DEPQ) is a data structure which supports the operations:

- `FindMin(Q)` : return the minimum element in `Q`
- `FindMax(Q)` : return the maximum element in `Q`
- `DeleteMin(Q)` : delete the minimum element in `Q`
- `DeleteMax(Q)` : delete the maximum element in `Q`
- `Insert(Q,x)` : insert `x` into `Q`

Many data structures [1, 3, 4, 7, 11, 12, 17, 2] have been proposed for the representation of a DEPQ. Some of these data structures [12, 7, 2] were developed to also support the `Meld` operation efficiently. For Example, Brodal [2] describes how his FMPQ structure may be used to perform the `DeleteMin` and `DeleteMax` operation in logarithmic time and the remaining operations in constant time.

The purpose of this paper is to demonstrate the generality of two techniques used in [7] to develop an MDEPQ representation from an MPQ representation – height biased leftist trees. These methods – total correspondence and leaf correspondence – may be used to arrive at efficient DEPQ and MDEPQ data structures from PQ and MPQ data structures such as the pairing heap [8, 15], Binomial and Fibonacci heaps [9], and Brodal's FMPQ [2] which also provide efficient support for the operation:

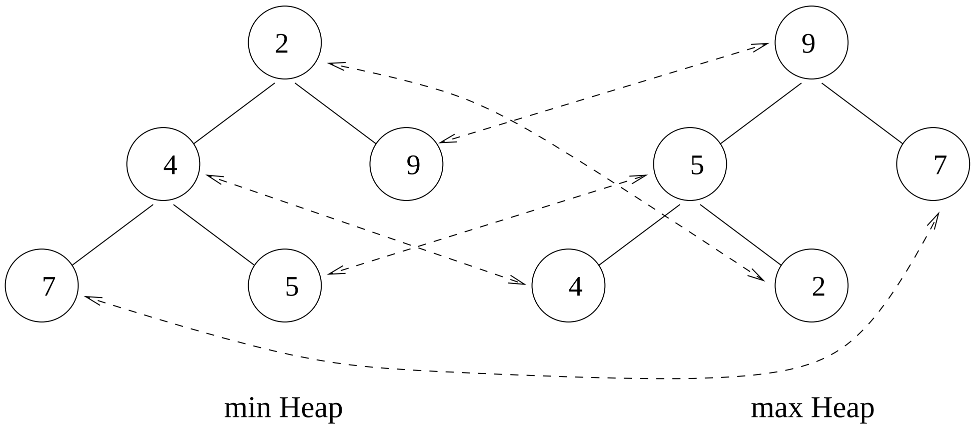


Figure 1: Dual heap structure

- `Delete(Q,p)` : delete and return the element located at `p`

We begin, in Section 2, by reviewing a rather straightforward way, dual priority queues, to obtain a (M)DEPQ structure from a (M)PQ structure. This method [7, 2] simply puts each element into both a minPQ and a maxPQ. In Section 3, we describe the total correspondence method and in Section 4, we describe leaf correspondence. Both sections provide examples of PQs and MPQs and the resulting DEPQs and MDEPQs. Section 5 gives complexity results. In Section 6, we provide the result of experiments that compare the performance of the MDEPQs based on height biased leftist tree [5], pairing heaps [8, 15], and FMPQs [2]. For reference purpose, we also provide run times for the splay tree data structure [13]. Although splay trees were not specifically designed to represent DEPQs, it is easy to use them for this purpose. Note that splay trees do not provide efficient support for the `Meld` operation.

2 Dual Priority Queues

A simple strategy, dual priority queues, to use to arrive at a DEPQ structure from a PQ structure that also supports `Delete(Q, p)` is to maintain both a minPQ `Qmin` and a maxPQ `Qmax`; every element of the PQ is in both `Qmin` and `Qmax`; and there are pointers between the two copies of any element `e` (note that one copy of `e` is in `Qmin` and the other in `Qmax`). For example, if the DEPQ is to contain elements with priorities [5, 9, 2, 4, 7], then we could set up a min heap and a max heap as in Figure 1. Pointers between the two copies of an element are shown by broken lines. When dual priority queues are used, the (M)DEPQ operations are performed as follows.

- `FindMin(Q)` = return `FindMin(Qmin)`
- `FindMax(Q)` = return `FindMax(Qmax)`
- `Insert(Q,x)` = {`Insert(Qmin,x)`; `Insert(Qmax,x)`; `SetPointers()`;}

- `DeleteMin(Q)` = {`Delete(Qmax`, `Pointer(FindMin(Qmin)))`; `DeleteMin(Qmin)`;}
- `DeleteMax(Q)` = {`Delete(Qmin`, `Pointer(FindMax(Qmax)))`; `DeleteMax(Qmax)`;}
- `Meld`(Q_1,Q_2) = {`Meld`(Q_1 `min`,Q_2 `min`); `Meld`(Q_1 `max`,Q_2 `max`);}

`SetPointers`() creates the pointers between the two copies of the newly inserted element. The code to do this task could easily be integrated into the code for `Insert`. `Pointer(y)` gives the pointer to the copy of `y` in the dual priority queue.

If we make the assumption that `Delete(Q,p)` has the same complexity as `DeleteMin` and `DeleteMax`, then the asymptotic complexity of the individual operations for a (M)DEPQ are the same as for the corresponding operations in a (M)PQ. Since this assumption is valid for all PQ structures cited earlier other than the weight biased leftist trees of [6], the concept of dual priority queues may be used to arrive at efficient (M)DEPQ structures from each of the cited (M)PQ structures other than weight biased leftist trees.

Although the notion of dual priority queues is straightforward, it suffers from at least two deficiencies : (1) The number of nodes in the two priority queues is twice the number of elements and (2) Each operation of the (M)DEPQ takes approximately twice the time it takes for the corresponding operation in a PQ because the corresponding operation needs to be done in both the minPQ and the maxPQ. The concepts of total and leaf correspondence overcome both these deficiencies.

A refinement of dual priority queues was proposed by Cho and Sahni [7]. This refinement applies to linked priority queues such as leftist trees and Brodal's FMPQ structure. The two nodes used for each element in ordinary dual priority queues are combined into a single node. So, in refined dual priority queues based on leftist trees, for example, each node will have 1 data field, 2 left child fields (one for the min leftist tree, the other for the max leftist tree), 2 right child fields, and 2 `sh` (length of shortest path to an external node) fields.

3 Total Correspondence

The notion of total correspondence borrows heavily from the ideas used in a twin heap [17]. In the twin heap data structure `n` elements are stored in a min heap using an array `minHeap[1:n]` and `n` other elements are stored in a max heap using the array `maxHeap[1:n]`. The min and max heaps satisfy the inequality `minHeap[i]` $\leq$ `maxHeap[i]`, $1 \leq$ `i` $\leq$ `n`. In this way, we can represent a DEPQ with 2`n` elements. When we must represent a DEPQ with an odd number of elements, one element is stored in a buffer, and the remaining elements are divided equally between the arrays `minHeap` and `maxHeap`.

In total correspondence, we remove the positional requirement in the relationship between pairs of elements in the min heap and max heap. The requirement becomes: for each element `a` in minPQ there is a distinct element `b` in maxPQ such that `a` $\leq$ `b` and vice versa. (`a`,`b`) is a corresponding pair of elements. Figure 2(a) shows a twin heap with 11 elements and Figure 2(b) shows a total correspondence heap. The broken arrows connect corresponding pairs of elements.

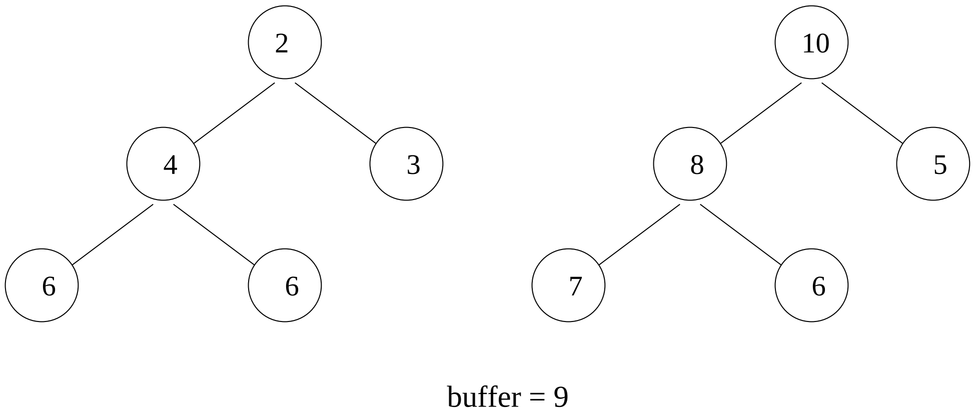
In a twin heap the corresponding pairs (`minHeap[i], maxHeap[i]`) are implicit, whereas in a total correspondence heap these pairs are represented using explicit pointers. The (M)DEPQ operations can be performed on a total correspondence priority queue as below.

```
FindMax(Q) =
if (the buffer is empty)
  return FindMax(Qmax)
else
  return max{buffer, FindMax(Qmax)}
```

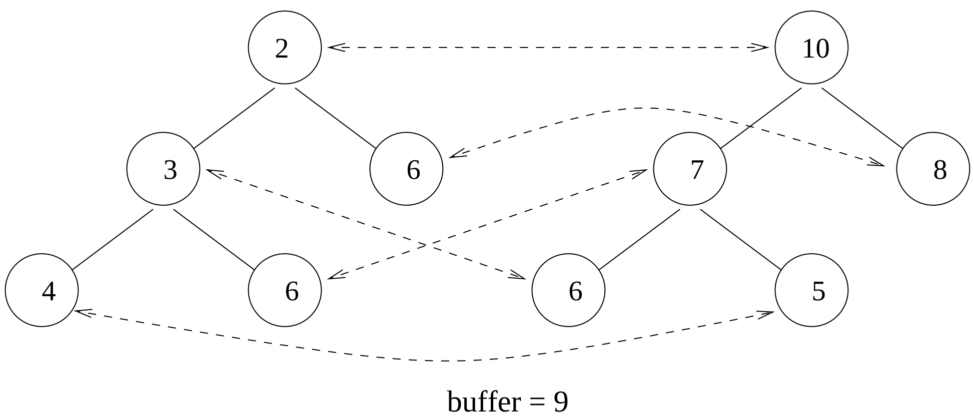
```
FindMin(Q) = similar to FindMax(Q)
```

```
Insert(Q,e) =
if (the buffer is empty)
  put e into the buffer;
else {
  Insert(Qmax,max{buffer,e});
  Insert(Qmin,{buffer,e} - max{buffer,e});
  SetPointers();
  buffer = empty;
}
```

```
DeleteMax(Q) =
if (the buffer is empty) {
  y = FindMax(Qmax);
  DeleteMax(Qmax);
  buffer = Delete(Qmin, Pointer(y));
}
else {
  if (buffer < FindMax(Qmax)) {
    // delete FindMax(Qmax)
    y = FindMax(Qmax);
    DeleteMax(Qmax);
    if (buffer ≥ element at Pointer(y)) {
```



(a) Twin heap



(b) Total correspondence heap

Figure 2: Twin heap and total correspondence heap

```
      Insert(Qmax,buffer);
      SetPointers(); // between Pointer(y) and buffer
    }
    else {
      Insert(Qmax, element at Pointer(y));
      Delete(Qmin, Pointer(y));
      Insert(Qmin, buffer);
      SetPointers();
    }
  }
  buffer = empty;
}
```

`DeleteMin(Q)` = similar to `DeleteMax(Q)`;

`Meld`(Q_1,Q_2) = {`Meld`(Q_1 `min`,Q_2 `min`); `Meld`(Q_1 `max`,Q_2 `max`);}

In a total correspondence (M)DEPQ, the number of nodes is either `n` or `n-1`. The space requirement is half that needed by the dual priority queue representation. The time required is also reduced. For example, if we do a sequence of inserts, every other one simply puts the element in the buffer. The remaining inserts put one element in `Qmax` and one in `Qmin`. So, on average, an insert takes time comparable to an insert in either `Qmax` or `Qmin`. Recall that when dual priority queues are used the insert time is the sum of the times to insert into `Qmax` and `Qmin`. Note also that the size of `Qmax` and `Qmin` together is half that of a dual priority queue.

If we assume that the complexity of the insert operation for priority queues as well as 2 Delete() operations is no more than that of the delete max or min operation (this is true for all known priority queue structures other than weight biased leftist trees), then the complexity of `DeleteMax` and `DeleteMin` for total correspondence (M)DEPQ is the same as for the `DeleteMax` and `DeleteMin` operation of the underlying priority queue data structure. The complexity of the `Meld` operation is the same as that for the underlying priority queue.

Using the notion of total correspondence, we trivially obtain efficient (M)DEPQ structures starting with any of the known priority queue structures (other than weight biased leftist trees). In particular, if we use the FMPQ structure of [2] as the base priority structure, we obtain a total correspondence MDEPQ structure in which `DeleteMax` and `DeleteMin` take logarithmic time, and the remaining operations take constant time. This adaptation is superior to the dual priority queue adaptation proposed in [2] because the space requirements are almost half. Additionally, the total correspondence adaptation is faster (see Section 6).

The `DeleteMax` and `DeleteMin` operations can generally be programmed to run faster than

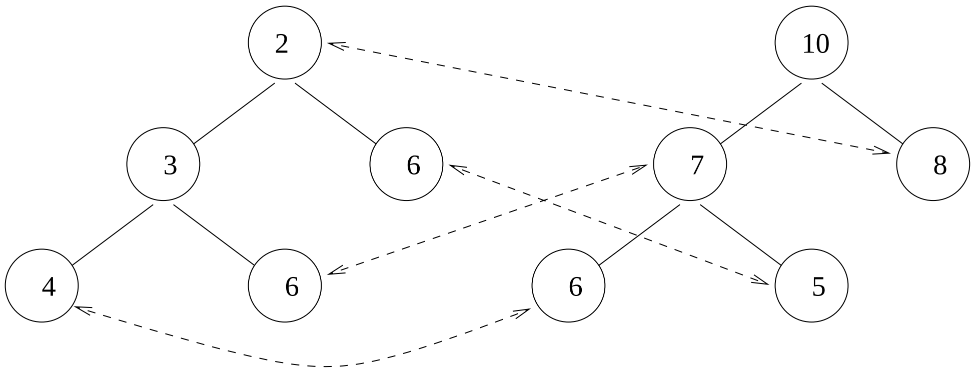


Figure 3: Leaf correspondence heap

suggested by our generic algorithms. This is because, for example, a `DeleteMax` and `Insert` into a maxPQ can often be done faster as a single operation `ChangeMax`. Similarly a Delete and `Insert` can be programmed as a `Change` operation.

4 Leaf Correspondence

In leaf correspondence (M)DEPQs, for every leaf element `a` in minPQ, there is a distinct element `b` in maxPQ such that $\mathtt{a} \leq \mathtt{b}$ and for every leaf element `c` in maxPQ there is a distinct element `d` in minPQ such that $\mathtt{d} \leq \mathtt{c}$. Figure 3 shows a leaf correspondence heap.

Efficient leaf correspondence (M)DEPQs may be constructed easily from (M)PQs which satisfy the following requirements:

(a) The (M)PQ supports the operation `Delete(Q,p)` efficiently.

(b) When an element is inserted into the (M)PQ, no nonleaf node becomes a leaf node (except possibly the node for the newly inserted item).

(c) When an element is deleted (using `Delete`, `DeleteMax` or `DeleteMin`) from the (M)PQ, no nonleaf node (except possibly the parent of the deleted node) becomes a leaf node.

(d) The `Meld` operation (if supported) should not create new leaf nodes.

Some of the (M)PQ structures that satisfy these requirements are height biased leftist tree, pairing heaps, and Fibonacci heaps. Requirements (b) and (c) are not satisfied, for example, by ordinary heaps and the FMPQ structure of [2].

The `FindMax`, `FindMin`, and `Meld` algorithms for a leaf correspondence (M)DEPQ are the same as those for a total correspondence (M)DEPQ. The `Insert` and `DeleteMax` algorithms are given below. `DeleteMin` is similar to `DeleteMax`.

```
Insert(Q,x) =
if (the buffer is empty)
  buffer = x;
else {
  small = min {buffer, x};
  large = {buffer, x} - {small};
  Insert(Qmin, small);
  if (small is a leaf) {
    Insert(Qmax, large);
    SetPointers(); // between small and large
    buffer = empty;
  }
  else buffer = large;
}


DeleteMax(Q) =
if (the buffer is empty) {
  y = FindMax(Qmax);
  DeleteMax(Qmax);
  if (Pointer(y) ≠ null)
    if (Pointer(y) is not a leaf)
      Pointer(Pointer(y)) = null;
    else { // must establish leaf correspondence
      p = Parent(Pointer(y));
      y = Delete(Qmin,Pointer(y));
      if (p is now a leaf and Pointer(p) = null) {
        Insert(Qmax,y);
        SetPointers(); // between p and y
      }
      else buffer = y;
    }
}
else { // buffer is not empty
  y = FindMax(Qmax);
  if (buffer ≥ y)
    buffer = empty;
  else { // delete from Qmax
    DeleteMax(Qmax);
```

```
if (Pointer(y) ≠ null) {
  if (Pointer(y) is a leaf) {
    // must establish leaf correspondence
    if (buffer ≥ element at Pointer(y)) {
      Insert(Qmax,buffer);
      SetPointers(); // between Pointer(y) and buffer
      buffer = empty;
    }
    else {
      p = Parent(Pointer(y));
      z = Delete(Qmin, Pointer(y));
      Insert(Qmax,z);
      if (either p or z has become a leaf and Pointer(p) is null)
        SetPointers(); // between p and z
      else
        if (z has become a leaf) { // Pointer(p) is not null
          Insert(Qmin, buffer);
          SetPointers(); // between z and buffer
          buffer = empty;
        }
        else Pointer(z) = null;
    }
  else Pointer(Pointer(y)) = null;
 }
}
```

The operations on leaf correspondence height biased leftist trees and pairing heaps have the same asymptotic complexity as when total correspondence is used.

Although heaps and Brodal's FMPQ structure do not satisfy the requirements of our generic approach to build a leaf correspondence (M)DEPQ structure from a priority queue, we can nonetheless arrive at leaf correspondence heaps and leaf correspondence FMPQs using a customized approach.

4.1 Leaf Correspondence Heaps

We assume familiarity with the top-down delete and bottom-up insert algorithms for min and max heaps [10]. We first describe a way to establish correspondence between two nodes `P` and `Q`, `P` is in the max heap, `Q` is in the min heap, one or both are leaves, and both presently have null correspondence pointers. If the element, `data(P)`, in node `P` is such that `data(P)` $\geq$ `data(Q)`,

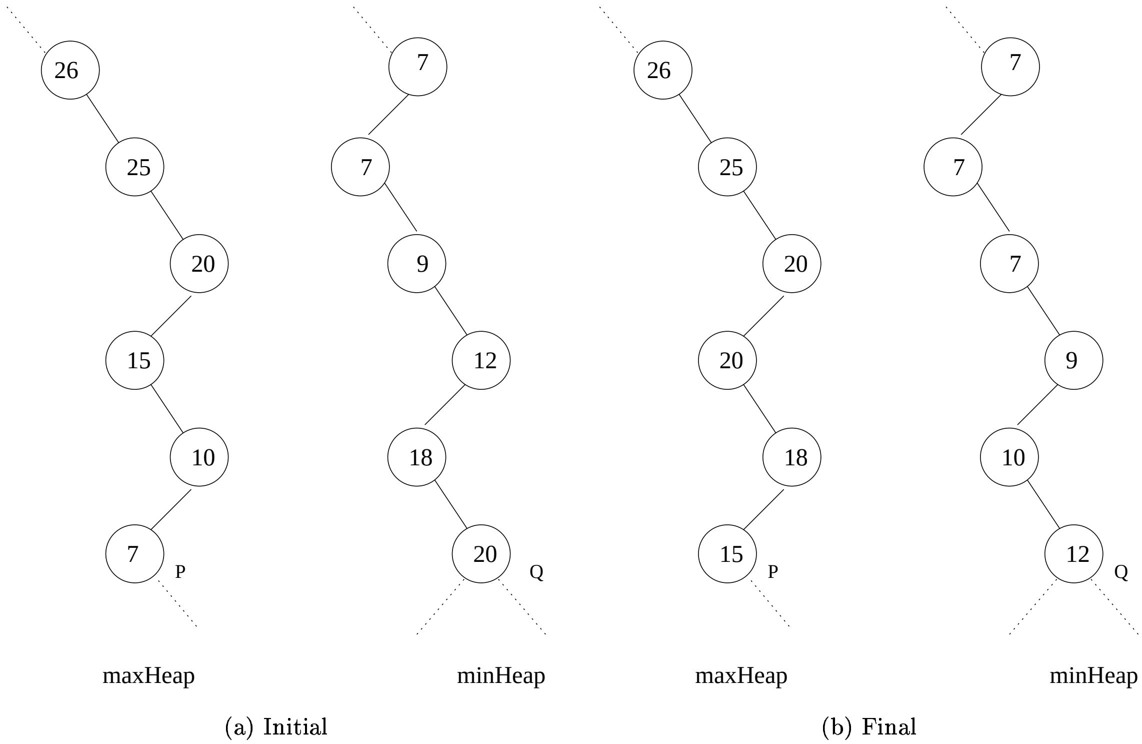
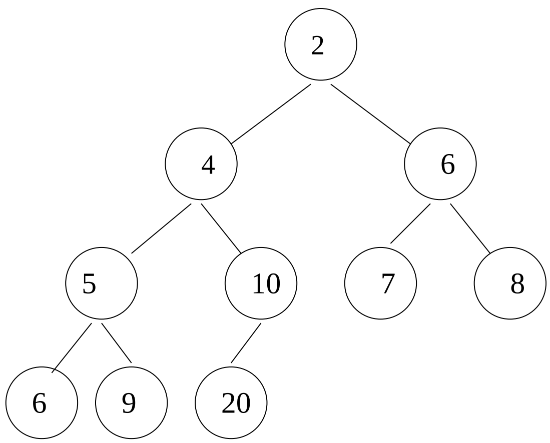


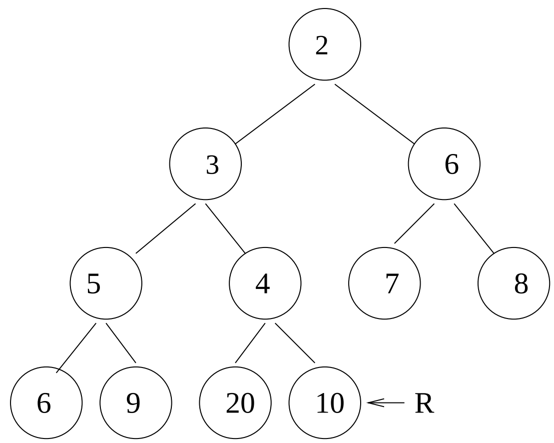
Figure 4: Estabilishing correspondence between P and Q

then we can simply set correspondence pointers between `P` and `Q`. So, suppose that `data(P)` $<$ `data(Q)`. To establish a correspondence between node `P` and `Q`, we must change the elements in `P` and/or `Q` so that `data(P)` $\geq$ `data(Q)`. To this end, we traverse the path from `P` to the root of the max heap `maxHeap` collecting elements that are $<$ `data(Q)`. In the example of Figure 4(a), the elements 7, 10, and 15 are collected.

Next, we collect elements on the path from `Q` to the root of `minHeap` that are $>$ `data(P)`, the elements 20, 18, 12, and 9 are collected. The two lists of collected elements are merged to get the list 7, 9, 10, 12, 15, 18, 20 and these elements are reassigned to the nodes of `minHeap` and `maxHeap`. The first four elements are put in `minHeap` because four elements of the list came from `minHeap`, the remaining elements are put into `maxHeap`. The resulting configuration is shown in Figure 4(b). This element reassignment process replaces elements on the path from `P` to the root of `maxHeap` by possibly larger ones and those on the path from `Q` to the root of `minHeap` by possibly smaller ones. Consequently, the heap property is not violated. Further `data(P)` $\geq$ `data(Q)` and we can set correspondence pointers between `P` and `Q`. Note that correspondence pointers in nodes on the paths from `P` and `Q` to their respective roots are still valid. We shall refer to this method of establishing correspondence as "establish PQ correspondence". Note that we can establish PQ correspondence in $O(\log n)$ time, where n is the total number of elements in the leaf correspondence heap.



(a) Initial



(b) After Insertion

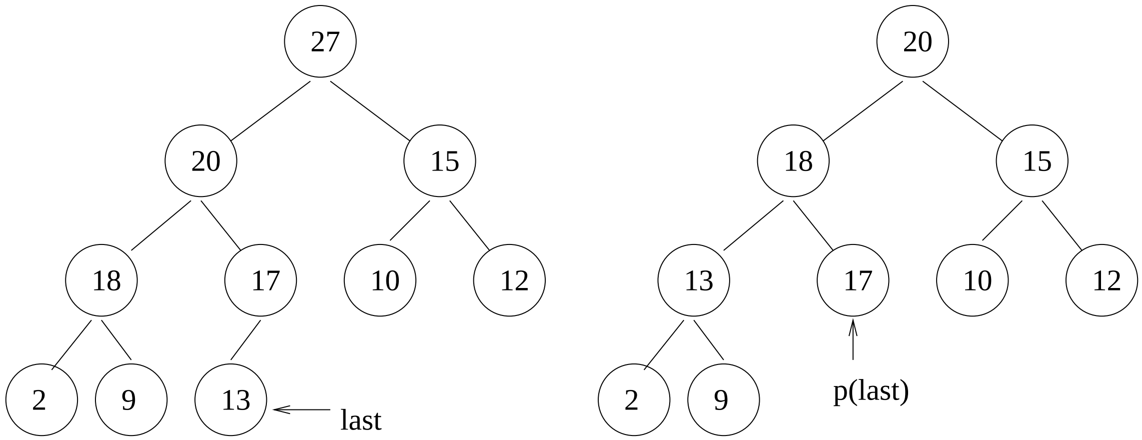
Figure 5: Heap insertion

4.1.1 Inserting into a Leaf Correspondence Heap

When a new element is inserted into a nonempty min heap or max heap, it is possible for a nonleaf existing element to become a leaf. This can happen only when we insert an element into a heap that has an even number of elements. Figure 5(a) shows a min heap with 10 elements. If we insert 3 into this min heap, the result is the min heap of Figure 5(b).

Element 10 which is a nonleaf of Figure 5(a) becomes a leaf because of the insertion. The new node `R` is the right child of its parent `p(R)`. During the insertion of element `x` into a min heap a nonleaf becomes a leaf iff (a) the new node `R` is the right child of its parent and (b) the original element in `p(R)` $>$ `x`. A similar observation may be made about insertion into a max heap. With this knowledge, we arrive at the following algorithm to insert an element `x` into a leaf correspondence heap.

```
InsertLCH(Q,e) =
if (the buffer is empty)
  buffer = x;
else {
  small = min {buffer, x};
  large = {buffer, x} - {small};
  insert large into maxHeap using the max heap insertion algorithm;
  if (a nonleaf of the original maxHeap is now a leaf that has a null correspondence pointer)
    remove the new leaf and put it in the buffer;
  insert small into minHeap using the min heap insertion algorithm;
  if (small is a leaf)
    set correspondence pointers between small and large;
```



(a) Initial (b) After deletion

Figure 6: Deletion from a max heap

```
  else
    if (a nonleaf R (see Figure 5) with null corresponding pointer becomes a leaf)
      establish PQ correspondence with P = R and Q = large;
    else
      if (large is a leaf)
        set correspondence pointers between small and large;
}
```

The time required to insert an element into an LCH is $O(\log n)$.

4.1.2 Deleting the Maximum Element from a Leaf Correspondence Heap

Next, consider deleting the maximum element from a LCH (deleting the minimum element is similar). The maximum element is either in the buffer or in the root of `maxHeap`. The case when the maximum element is in the buffer is handled by simply emptying the buffer. When the maximum element is in the root of the `maxHeap`, we first use the delete max algorithm for max heaps. This algorithm takes the last element, `data(last)`, out of the max heap and reinserts this element into the max heap in a top down manner (see Figure 6).

As a result of this deletion process, the former parent, `p(last)`, of the last element may become a leaf. When `p(last)` has a null correspondence pointer, we need to establish correspondence for this new leaf node. Notice that the deletion process moves an element from a leaf node to a nonleaf node. This element is `data(last)` in Figure 6(a), and is `data(new Parent(data(last)))` when `data(last)` is a leaf node following reinsertion. Let `C` be the corresponding node in the `minHeap` for the former leaf. Establish PQ correspondence with `P =`

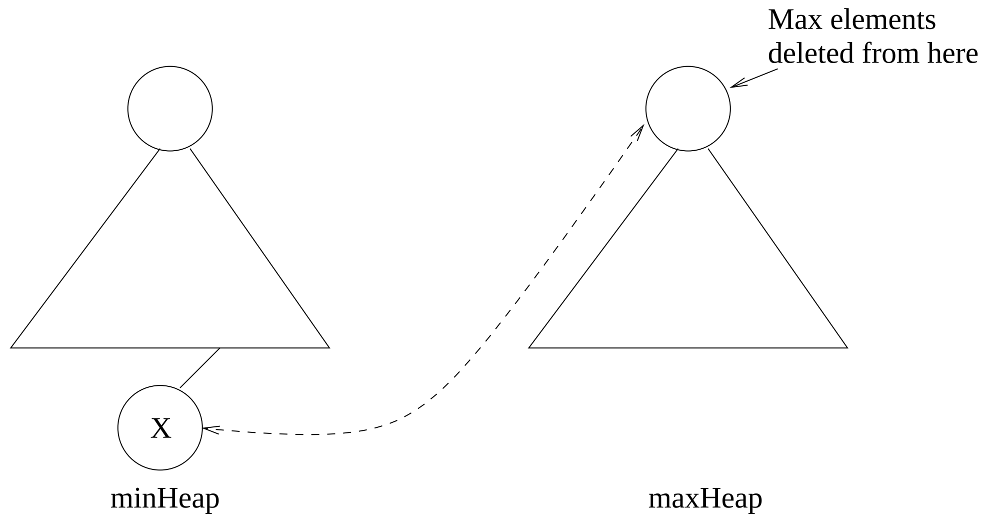


Figure 7: Establishing correspondence for the correponding node

`p(last)` and `Q = C`.

Having taken care of possible correspondence problems in `maxHeap`, we proceed to take care of such problems in `minHeap`. Problems of this type arise only when the deleted max element is the corresponding element for a leaf element `data(x)` in `minHeap`. We must establish correspondence for leaf node `x` of `minHeap`.

First, consider the case when `x` is the last node of `minHeap`. If the removal of `x` does not cause the parent node `p(x)` to become a leaf or if `p(x)` has a non null correspondence pointer, we remove node `x` from `minHeap` and insert `data(x)` back into the LCH using `InsertLCH`. If the removal of `x` cause `p(x)` to become a leaf with a null correspondence pointer, we insert `data(x)` into `maxHeap` using the bottom up insertion algorithm for a max heap. This insertion creates a new leaf. If this new leaf has a null correspondence pointer, we establish PQ correspondence with `P` = new leaf and `Q = p(x)`; otherwise, we establish PQ correspondence with `P` = node that contains `data(x)` and `Q = p(x)`.

The second and final case to consider is when `x` is not the last node of `minHeap`. Let `last` $\neq$ `x` be the last node. If `p(last)` has a non null correspondence pointer or `p(last)` does not become a leaf when node `last` is removed, then remove node `last`; establish PQ correspondence with `P` = node of `maxHeap` that corresponds to the former leaf `last` and `Q = x`; insert `data(last)` into the LCH using `InsertLCH`. If `p(last)` has a null correspondence pointer and `p(last)` becomes a leaf following removal of the node `last`, move `data(last)` and `correspondence(last)` to `p(last)`; remove node `last`; and insert the original `data(p(last))` into `maxHeap`. This creates a new leaf in `maxHeap`. If this new leaf has a null correspondence pointer, we establish PQ correspondence with `P` = new leaf and `Q = x`; otherwise, we establish PQ correspondence with `P` = node that contains the original `data(p(last))` and `Q = x`.

The complexity of the `DeleteMaxLCH` process described above is $O(\log n)$.

Table 1: Complexity of the (M)DEPQ operations

	FindMax/Min	DeleteMax/Min	Insert	Meld
Dual Correspondence	$O(t_{FindMax})$	$O(t_{DeleteMax})$	$O(t_{Insert})$	$O(t_{Meld})$
Total Correspondence	$O(t_{FindMax})$	$O(t_{DeleteMax} + t_{Insert} + t_{Delete})$	$O(t_{Insert})$	$O(t_{Meld})$
DLT/TLT/LLT DPH/TPH/LPH DFMPQ/TFMPQ/LFMPQ	$O(t_{FindMax})$	$O(t_{DeleteMax} + t_{Insert} + t_{Delete})$	$O(t_{Insert})$	$O(t_{Meld})$

4.2 Leaf Correspondence FMPQs

The generic leaf correspondence algorithms may be applied to leaf correspondence FMPQs. However, the application of these algorithms may leave behind leaves that have a null correspondence pointer. To overcome this problem, newly created leaves with null correspondence pointer are detached from their trees and reinserted into the leaf correspondence FMPQ. It may be shown that $O(1)$ such reinsertions are needed. Therefore, the asymptotic complexity of each MDEPQ operation is the same as for the corresponding operation in an FMPQ.

5 Complexity of Correspondence (M)DEPQs

Let $t_{FindMax}(= t_{FindMin})$, $t_{DeleteMax}(= t_{DeleteMin})$, t_{Delete}, t_{Insert} and t_{Meld} be the complexity of FindMax, DeleteMax, Delete, Insert and Meld operations for the (M)PQ upon which a correspondence DEPQ is based. Table 1 summarizes the complexity of the (M)DEPQ operations when the generic and customized correspondence algorithms are used. In this table, DLT refers to dual correspondence leftist trees, TLT to total correspondence leftist trees, and LLT to leaf correspondence leftist trees; PH is an abbreviation for the pairing heap data structure.

As far as space complexity is concerned, dual and refined dual correspondence require approximately twice as much space as taken by total and leaf correspondence; the space requirements of total and leaf correspondence are the same.

6 Experimental Results

We compared the runtime performance of dual, total and leaf correspondence double ended priority queue structures. Our experiements were limited to correspondence structures based on height biased leftist trees, pairing heaps, and fast meldable priority queues; the first of these permits $O(\log n)$ time melds while in the other two, a meld takes $O(1)$ time. For comparison purposes, our experimental study also includes the splay tree [13]. This tree was adapted to perform the DEPQ operations. We chose the splay tree because it is the fastest binary search tree structure [6].

inputs	m	n	DLT	TLT	LLT	DPH	TPH	LPH	DFD	TFD	LFD	Splay
		1K	432	401	313	306	319	274	1968	1611	1384	262
	100K	10K	782	744	603	585	582	528	2734	2308	2037	491
		100K	1681	1576	1339	1527	1343	1276	3331	2913	2683	1013
		1M	2416	2226	1913	5493	3516	3442	4175	3703	3433	1391
		1K	825	764	594	580	609	518	3933	3212	2755	496
	200K	10K	1290	1225	979	934	949	847	5427	4560	3974	803
		100K	2955	2798	2355	2397	2228	2101	6576	5704	5209	1809
RD1		1M	4699	4348	3733	6958	5001	4856	8325	7371	6831	2733
		1K	1994	1842	1429	1396	1468	1245	9863	8057	6882	1194
	500K	10K	2614	2466	1942	1847	1911	1672	13503	11316	9733	1604
		100K	5794	5550	4589	4327	4213	3921	16201	13918	12412	3601
		1M	11005	10274	8788	11069	9164	8814	20576	18154	16850	6503
		1K	3934	3633	2816	2752	2898	2451	19704	16090	13734	2354
	1M	10K	4666	4373	3420	3269	3410	2946	26972	22564	19259	2836
		100K	9298	8940	7277	6694	6675	6130	32135	27425	23967	5791
		1M	20165	18996	16168	17106	15277	14604	40581	35618	33262	12091
		1K	429	397	310	305	318	272	1953	1597	1373	257
	100K	10K	744	705	572	569	565	511	2646	2229	1955	438
		100K	1457	1359	1152	1421	1236	1169	3102	2707	2460	755
		1M	1819	1662	1424	5209	3231	3158	3823	3371	3090	862
		1K	817	755	587	577	605	514	3920	3202	2738	489
	200K	10K	1224	1156	924	904	918	816	5292	4442	3848	718
		100K	2546	2396	2012	2201	2030	1903	6172	5347	4790	1348
RD2		1M	3537	3247	2778	6403	4442	4298	7617	6708	6147	1696
		1K	1977	1822	1414	1390	1460	1235	9813	8009	6839	1179
	500K	10K	2492	2336	1839	1789	1850	1611	13287	11121	9524	1458
		100K	4964	4713	3887	3921	3802	3512	15376	13192	11519	2694
		1M	8268	7653	6526	9754	7835	7489	18927	16605	15285	4047
		1K	3899	3589	2782	2737	2878	2428	19631	16019	13663	2325
	1M	10K	4484	4177	3265	3181	3317	2853	26674	22296	18974	2629
		100K	7975	7586	6161	6045	6012	5474	30868	26325	22661	4394
		1M	15107	14105	11965	14654	12800	12137	37552	32792	30253	7549
		1K	705	407	199	228	244	190	1941	1501	1218	58
	100K	10K	913	495	200	244	231	199	2694	2110	1673	58
		100K	1149	603	200	419	307	287	3244	2571	2017	58
		1M	1482	757	200	2219	1207	1187	3839	3062	2408	58
		1K	1402	815	399	456	497	382	3824	2967	2427	118
	200K	10K	1824	993	400	472	472	391	5381	4230	3394	118
		100K	2249	1183	400	640	515	474	6461	5124	4020	117
INC		1M	2894	1481	400	2440	1416	1374	7689	6153	4814	117
		1K	3512	2036	1000	1138	1261	953	9589	7442	6091	295
	500K	10K	4559	2494	1000	1156	1222	964	13443	10564	8563	294
		100K	5623	2949	1001	1309	1160	1041	16089	12761	10024	294
		1M	7007	3592	1001	3101	2039	1937	19183	15324	12052	294
		1K	7023	4086	1999	2273	2530	1907	19204	14908	12189	589
	1M	10K	9117	5000	2001	2294	2511	1920	26911	21162	17188	589
		100K	11221	5900	2001	2450	2316	1997	31936	25453	20066	588
		1M	13729	7056	2001	4201	3080	2872	38353	30641	24095	587
		1K	701	413	210	232	258	202	1937	1506	1276	63
	100K	10K	906	551	329	284	312	232	2825	2256	1926	104
		100K	986	656	478	372	386	278	3661	2998	2686	324
		1M	986	656	478	372	386	278	4409	3609	3250	2124
		1K	1400	821	402	459	513	394	3852	2990	2536	120
	200K	10K	1811	1062	429	530	587	429	5515	4365	3668	147
		100K	2093	1421	549	939	872	551	7263	5952	5337	285
DEC		1M	2093	1421	549	2683	1659	551	8806	7201	6516	618
		1K	3491	2039	1007	1136	1265	938	9541	7404	6290	299
	500K	10K	4537	2553	1127	1191	1374	1032	13573	10693	8923	340
		100K	5554	3457	2348	1713	1814	1335	17485	14313	12574	755
		1M	5652	3632	2584	1863	1939	1394	22057	18048	16345	2622
		1K	6999	4081	2007	2272	2533	1998	19112	14847	12586	593
	1M	10K	9094	5068	2129	2329	2714	2032	27025	21307	17707	635
		100K	11151	6468	3637	2852	3188	2329	33291	26984	22737	1050
		1M	11947	7591	5337	3729	3922	2786	44148	36123	33130	3244

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

Table 2: The number of key comparisons

inputs	m	n	DLT	TLT	LLT	DPH	TPH	LPH	DFD	TFD	LFD	Splay
RD1	100K	1K	2.47	2.92	2.28	1.89	2.26	2.08	9.67	9.44	6.92	1.88
		10K	2.39	1.97	1.81	1.49	1.68	1.53	2.57	4.78	4.75	1.24
		100K	2.52	3.16	1.93	2.15	2.10	2.12	8.78	11.24	6.39	1.49
		1M	1.55	1.79	1.83	1.33	1.58	1.51	7.14	7.32	10.01	1.22
	200K	1K	2.66	2.88	2.28	1.94	2.28	2.14	16.75	15.28	10.46	1.63
		10K	4.69	4.70	4.08	3.66	3.69	3.51	5.14	8.39	8.38	3.17
		100K	5.11	4.14	4.20	3.26	3.29	2.79	11.80	12.05	21.82	2.57
		1M	3.74	3.65	2.54	1.79	1.64	1.72	8.85	11.10	9.05	1.95
	500K	1K	6.45	7.62	6.14	5.12	6.07	5.36	30.63	28.69	18.18	5.04
		10K	5.10	5.29	4.07	4.22	4.64	3.94	7.77	13.14	12.56	3.91
		100K	6.90	7.26	5.81	4.06	4.62	4.04	28.19	33.94	52.05	4.52
		1M	6.28	6.63	5.45	4.50	4.19	3.84	10.35	10.94	42.49	3.21
	1M	1K	6.89	8.42	7.29	5.75	7.04	6.27	30.97	29.73	18.86	5.95
		10K	6.95	7.17	6.44	5.99	6.40	5.67	12.12	15.77	20.90	6.01
		100K	10.16	8.10	7.52	6.51	7.65	6.32	22.56	28.77	98.10	6.50
		1M	9.17	10.15	8.58	7.59	7.64	6.88	17.44	21.98	122.12	6.35
RD2	100K	1K	2.45	2.11	2.00	1.78	1.75	1.79	8.74	7.42	5.38	1.87
		10K	1.88	1.75	1.56	1.38	1.35	1.36	3.85	5.36	3.76	1.27
		100K	3.25	2.54	2.46	2.14	2.12	2.19	7.47	8.72	6.60	1.51
		1M	2.52	1.91	1.65	1.82	1.09	1.47	7.15	7.87	7.76	0.87
	200K	1K	2.31	2.52	2.25	1.83	2.26	1.75	16.76	14.36	7.71	1.71
		10K	3.34	3.53	2.34	2.75	2.94	2.50	4.20	7.15	10.11	2.40
		100K	3.25	3.70	3.36	2.84	3.27	2.90	17.14	19.67	17.18	1.78
		1M	3.87	4.31	2.85	2.05	2.16	2.41	10.01	9.91	8.93	1.01
	500K	1K	4.12	4.58	3.26	3.23	3.70	3.36	30.02	25.23	16.51	2.96
		10K	3.83	4.52	3.03	3.21	4.14	3.52	8.92	12.77	16.16	3.05
		100K	6.46	6.93	5.21	5.74	5.84	4.63	27.60	33.38	39.81	3.65
		1M	6.42	5.92	3.13	4.28	4.61	3.52	11.21	12.28	76.52	2.40
	1M	1K	8.01	8.74	6.76	5.79	7.20	5.85	39.59	31.80	22.46	5.74
		10K	6.17	5.68	5.25	4.36	4.86	4.31	12.53	16.43	19.51	4.72
		100K	10.74	11.35	9.30	8.77	9.32	9.44	38.54	42.82	81.32	6.14
		1M	12.00	11.08	8.47	8.60	7.76	7.55	16.51	21.36	54.28	4.41

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

Table 3: Standard deviation of the number of key comparisons

inputs	m	n	DLT	TLT	LLT	DPH	TPH	LPH	DFD	TFD	LFD	Splay
RD1	100K	1K	0.112	0.102	0.080	0.105	0.103	0.089	1.112	0.803	0.651	0.058
		10K	0.250	0.190	0.160	0.264	0.231	0.213	2.219	1.396	1.194	0.138
		100K	0.758	0.650	0.577	1.020	0.755	0.726	3.682	2.838	2.453	0.376
		1M	1.311	1.199	1.029	5.093	2.983	2.914	5.283	4.174	3.648	0.615
	200K	1K	0.222	0.199	0.159	0.197	0.190	0.162	2.213	1.595	1.288	0.106
		10K	0.405	0.320	0.266	0.415	0.377	0.336	4.378	2.731	2.289	0.216
		100K	1.401	1.197	1.033	1.476	1.200	1.123	7.259	5.525	4.803	0.672
		1M	2.552	2.320	2.110	6.115	3.851	3.853	10.830	8.463	7.361	1.225
	500K	1K	0.540	0.482	0.391	0.458	0.440	0.382	5.539	4.016	3.213	0.250
		10K	0.789	0.647	0.526	0.713	0.677	0.613	10.832	6.676	5.495	0.402
		100K	2.787	2.431	2.059	2.460	2.088	2.029	17.753	13.420	11.118	1.238
		1M	6.234	5.758	5.028	8.804	6.410	6.211	27.560	21.522	18.825	2.795
	1M	1K	1.071	0.965	0.779	0.897	0.871	0.762	11.046	7.973	6.392	0.498
		10K	1.399	1.173	0.953	1.209	1.143	1.015	21.601	13.236	10.814	0.673
		100K	4.317	3.628	3.068	3.660	3.247	3.032	34.490	25.901	21.095	1.979
		1M	11.865	11.189	9.838	12.741	10.071	9.658	54.401	42.460	37.698	5.208
RD2	100K	1K	0.114	0.100	0.082	0.102	0.098	0.087	1.098	0.797	0.645	0.056
		10K	0.233	0.185	0.152	0.247	0.217	0.201	2.148	1.349	1.134	0.114
		100K	0.640	0.547	0.473	0.957	0.706	0.664	3.380	2.566	2.204	0.233
		1M	0.951	0.856	0.737	5.037	2.851	2.794	4.694	3.661	3.171	0.285
	200K	1K	0.219	0.196	0.156	0.192	0.183	0.161	2.201	1.584	1.276	0.103
		10K	0.384	0.302	0.249	0.385	0.343	0.308	4.266	2.635	2.220	0.179
		100K	1.189	0.993	0.841	1.336	1.046	0.979	6.713	5.142	4.237	0.399
		1M	1.921	1.671	1.479	5.808	3.516	3.489	9.545	7.516	6.324	0.544
	500K	1K	0.530	0.482	0.390	0.452	0.438	0.382	5.520	3.974	3.187	0.244
		10K	0.761	0.625	0.512	0.677	0.637	0.567	10.639	6.477	5.333	0.340
		100K	2.324	1.933	1.650	2.227	1.912	1.734	16.517	12.514	10.095	0.766
		1M	4.711	4.203	3.677	7.916	5.591	5.373	24.580	18.840	16.231	1.303
	1M	1K	1.048	0.946	0.765	0.878	0.854	0.744	11.014	7.928	6.350	0.480
		10K	1.326	1.120	0.915	1.153	1.091	0.956	21.251	12.964	10.542	0.587
		100K	3.660	3.112	2.693	3.329	2.751	2.534	32.839	24.610	19.589	1.191
		1M	8.899	7.945	6.888	11.037	8.439	7.918	48.973	37.792	32.394	2.335
INC	100K	1K	0.136	0.095	0.051	0.067	0.071	0.060	1.110	0.748	0.632	0.036
		10K	0.227	0.131	0.057	0.084	0.078	0.065	1.980	1.222	1.023	0.039
		100K	0.341	0.174	0.056	0.224	0.140	0.140	2.294	1.505	1.280	0.064
		1M	0.587	0.298	0.057	2.162	1.100	1.083	3.247	2.261	1.849	0.574
	200K	1K	0.271	0.194	0.106	0.168	0.161	0.136	2.224	1.495	1.270	0.069
		10K	0.401	0.240	0.107	0.204	0.172	0.148	4.010	2.395	2.029	0.073
		100K	0.674	0.335	0.112	0.368	0.237	0.221	4.498	2.954	2.494	0.095
		1M	1.156	0.593	0.120	2.390	1.247	1.229	6.437	4.464	3.629	0.606
	500K	1K	0.693	0.490	0.264	0.421	0.420	0.340	5.580	3.768	3.217	0.201
		10K	1.040	0.612	0.275	0.485	0.431	0.366	10.795	6.151	5.341	0.218
		100K	1.742	0.887	0.284	0.689	0.504	0.457	11.920	7.895	6.600	0.280
		1M	2.776	1.405	0.290	2.662	1.489	1.423	16.113	10.976	9.087	0.775
	1M	1K	1.387	0.986	0.535	0.842	0.842	0.688	11.214	7.545	6.444	0.416
		10K	2.214	1.276	0.567	0.986	0.896	0.723	22.258	12.640	10.983	0.451
		100K	3.734	1.849	0.580	1.204	0.951	0.858	26.736	17.595	14.623	0.539
		1M	5.646	2.862	0.592	3.167	1.910	1.806	32.060	21.748	18.056	1.231
DEC	100K	1K	0.139	0.098	0.056	0.084	0.083	0.068	1.115	0.760	0.606	0.035
		10K	0.199	0.135	0.086	0.103	0.099	0.079	2.150	1.339	1.090	0.045
		100K	0.235	0.164	0.117	0.131	0.119	0.089	3.201	2.264	1.958	0.087
		1M	0.250	0.160	0.112	0.138	0.120	0.096	4.559	3.556	2.843	0.724
	200K	1K	0.271	0.195	0.105	0.166	0.166	0.133	2.230	1.517	1.210	0.071
		10K	0.410	0.262	0.112	0.213	0.201	0.145	4.360	2.604	2.076	0.079
		100K	0.595	0.375	0.140	0.482	0.338	0.191	6.638	4.760	4.138	0.173
		1M	0.582	0.364	0.135	2.407	1.204	0.188	9.276	7.275	5.907	0.688
	500K	1K	0.699	0.492	0.265	0.424	0.420	0.331	5.562	3.760	2.975	0.207
		10K	1.058	0.638	0.300	0.468	0.472	0.368	11.172	6.396	5.134	0.220
		100K	1.704	0.955	0.637	0.665	0.604	0.489	16.824	12.012	10.101	0.365
		1M	1.816	1.031	0.715	0.721	0.646	0.502	23.953	18.327	15.226	1.085
	1M	1K	1.396	0.988	0.530	0.841	0.845	0.679	11.079	7.515	5.971	0.419
		10K	2.191	1.262	0.571	0.918	0.921	0.707	22.387	12.717	10.145	0.449
		100K	3.549	1.929	1.036	1.168	1.089	0.854	33.389	23.052	18.158	0.623
		1M	4.100	2.335	1.535	1.433	1.276	0.996	47.838	36.422	31.004	1.710

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

Table 4: Run time using real (double) keys

inputs	m	n	DLT	TLT	LLT	DPH	TPH	LPH	DFD	TFD	LFD	Splay
RD1	100K	1K	0.006	0.006	0.004	0.005	0.005	0.004	0.015	0.008	0.008	0.004
		10K	0.013	0.007	0.007	0.018	0.013	0.012	0.026	0.027	0.015	0.007
		100K	0.064	0.075	0.067	0.085	0.069	0.063	0.094	0.098	0.081	0.028
		1M	0.088	0.067	0.052	0.224	0.163	0.138	0.128	0.112	0.123	0.040
	200K	1K	0.006	0.006	0.005	0.007	0.005	0.004	0.023	0.020	0.017	0.005
		10K	0.014	0.009	0.009	0.032	0.022	0.019	0.049	0.047	0.031	0.011
		100K	0.124	0.117	0.100	0.104	0.102	0.077	0.257	0.152	0.127	0.045
		1M	0.183	0.163	0.159	0.230	0.201	0.453	0.282	0.174	0.143	0.076
	500K	1K	0.015	0.018	0.012	0.011	0.008	0.008	0.048	0.150	0.047	0.004
		10K	0.015	0.017	0.015	0.026	0.027	0.024	0.146	0.138	0.059	0.015
		100K	0.186	0.242	0.184	0.099	0.095	0.139	0.492	0.277	0.250	0.071
		1M	0.329	0.366	0.235	0.403	0.359	0.309	0.748	0.373	0.376	0.173
	1M	1K	0.024	0.029	0.025	0.021	0.014	0.017	0.120	0.090	0.062	0.008
		10K	0.046	0.022	0.022	0.047	0.028	0.024	0.380	0.229	0.217	0.018
		100K	0.212	0.268	0.178	0.205	0.210	0.184	0.514	0.359	0.346	0.120
		1M	0.332	0.405	0.360	0.476	0.355	0.388	0.687	0.525	0.563	0.211
RD2	100K	1K	0.006	0.005	0.004	0.005	0.004	0.005	0.012	0.013	0.008	0.005
		10K	0.009	0.011	0.006	0.017	0.012	0.014	0.031	0.021	0.019	0.005
		100K	0.054	0.074	0.059	0.086	0.073	0.064	0.098	0.079	0.090	0.011
		1M	0.036	0.042	0.035	0.163	0.123	0.127	0.122	0.106	0.081	0.015
	200K	1K	0.008	0.006	0.006	0.005	0.005	0.005	0.023	0.013	0.013	0.005
		10K	0.011	0.012	0.008	0.030	0.023	0.016	0.053	0.043	0.124	0.011
		100K	0.133	0.116	0.096	0.085	0.066	0.061	0.165	0.148	0.126	0.021
		1M	0.108	0.087	0.089	0.208	0.154	0.174	0.182	0.345	0.147	0.023
	500K	1K	0.015	0.017	0.013	0.013	0.008	0.011	0.074	0.054	0.029	0.007
		10K	0.024	0.014	0.015	0.032	0.031	0.032	0.139	0.082	0.060	0.011
		100K	0.165	0.171	0.137	0.145	0.164	0.118	0.240	0.292	0.178	0.036
		1M	0.225	0.252	0.192	0.263	0.216	0.250	0.559	0.300	0.243	0.063
	1M	1K	0.029	0.016	0.020	0.017	0.014	0.015	0.116	0.054	0.041	0.009
		10K	0.021	0.029	0.025	0.054	0.042	0.037	0.235	0.184	0.093	0.016
		100K	0.215	0.204	0.228	0.209	0.143	0.133	0.443	0.354	0.321	0.029
		1M	0.411	0.338	0.322	0.298	0.251	0.206	0.720	0.461	0.424	0.087
INC	100K	1K	0.007	0.005	0.003	0.005	0.002	0.000	0.007	0.005	0.004	0.005
		10K	0.011	0.009	0.013	0.007	0.004	0.005	0.030	0.028	0.016	0.002
		100K	0.022	0.016	0.005	0.026	0.008	0.018	0.029	0.023	0.009	0.011
		1M	0.039	0.018	0.006	0.237	0.103	0.096	0.088	0.054	0.044	0.018
	200K	1K	0.007	0.005	0.005	0.004	0.004	0.006	0.017	0.009	0.010	0.002
		10K	0.015	0.009	0.005	0.017	0.009	0.010	0.070	0.053	0.018	0.005
		100K	0.053	0.025	0.004	0.025	0.009	0.010	0.045	0.031	0.047	0.011
		1M	0.092	0.045	0.008	0.131	0.082	0.076	0.165	0.118	0.091	0.024
	500K	1K	0.024	0.011	0.006	0.022	0.012	0.012	0.037	0.031	0.033	0.008
		10K	0.033	0.019	0.006	0.034	0.020	0.015	0.097	0.060	0.048	0.009
		100K	0.147	0.065	0.010	0.032	0.019	0.018	0.089	0.116	0.089	0.022
		1M	0.199	0.089	0.008	0.109	0.058	0.038	0.189	0.137	0.125	0.027
	1M	1K	0.046	0.022	0.010	0.022	0.024	0.019	0.088	0.054	0.037	0.012
		10K	0.097	0.073	0.025	0.048	0.031	0.021	0.136	0.198	0.109	0.015
		100K	0.326	0.127	0.022	0.036	0.035	0.021	0.192	0.146	0.181	0.022
		1M	0.407	0.212	0.015	0.101	0.062	0.060	0.292	0.154	0.188	0.049
DEC	100K	1K	0.006	0.005	0.005	0.007	0.005	0.005	0.014	0.008	0.007	0.005
		10K	0.008	0.005	0.007	0.006	0.005	0.007	0.034	0.042	0.017	0.005
		100K	0.014	0.007	0.004	0.008	0.006	0.008	0.059	0.032	0.031	0.013
		1M	0.019	0.006	0.004	0.010	0.008	0.019	0.155	0.111	0.110	0.028
	200K	1K	0.009	0.007	0.006	0.009	0.005	0.009	0.027	0.018	0.014	0.004
		10K	0.015	0.007	0.007	0.018	0.011	0.007	0.064	0.065	0.024	0.005
		100K	0.062	0.023	0.007	0.035	0.024	0.011	0.176	0.131	0.104	0.024
		1M	0.052	0.023	0.009	0.115	0.074	0.009	0.273	0.223	0.193	0.028
	500K	1K	0.022	0.007	0.007	0.020	0.012	0.014	0.035	0.029	0.016	0.009
		10K	0.039	0.020	0.015	0.032	0.016	0.018	0.113	0.052	0.054	0.012
		100K	0.139	0.031	0.022	0.022	0.019	0.022	0.207	0.197	0.149	0.040
		1M	0.124	0.064	0.039	0.031	0.023	0.019	0.282	0.202	0.262	0.088
	1M	1K	0.050	0.023	0.010	0.034	0.019	0.016	0.068	0.049	0.042	0.012
		10K	0.068	0.034	0.017	0.040	0.027	0.031	0.167	0.134	0.057	0.016
		100K	0.293	0.138	0.040	0.036	0.036	0.029	0.410	0.195	0.236	0.035
		1M	0.268	0.131	0.072	0.037	0.024	0.026	0.433	0.256	0.309	0.091

Time Unit : *sec*

m = the number of operations performed

n = the number of elements in initial data structures

Table 5: Standard deviation of run time using real keys

An experimental comparison of leaf correspondence leftist trees and unbalanced binary search trees, min-max heaps, deaps, AVL trees etc. appears in [6]. The conclusion of [6] for keys of data type double, is that unbalanced binary search trees are the best data structure when keys are selected at random; leaf correspondence leftist trees are the best data structure when keys are in ascending or descending order. Our experimental study is modeled after that used in [7]. Each timing experiment began with a DEPQ with an initial size $n \in \{1000, 10000, 100000, 1000000\}$ and performed a sequence of $m \in \{100000, 200000, 500000, 1000000\}$ DEPQ operations. Insert operations occurred with probability 0.5, and delete max and delete min had probability 0.25 each. The insert keys were selected in four different ways:

- RD1: random double precision keys between 1 and $1,000,000$
- RD2: random double precision keys between 1 and $1,000$
- INC: increasing sequence of double precision keys
- DEC: decreasing sequence of double precision keys

Although the keys are double precision, their actual values are integral, an integer random number generator was used and the numbers typecast to the double data type. All programs were written in C and run on a SUN Ultra Sparc workstation. For each choice of n, m and data set (RD1,RD2) 20 experiments were done, the average results are reported. Table 2 gives the number of key comparisons performed (x1000) and Table 3 gives the standard deviations for RD1 and RD2 (over the 20 experiments). The standard deviations are rather small, boosting our confidence in the reliability of the experiments. For leftist trees, pairing heaps and FMPQs, leaf correspondence made the fewest number of comparisons in all our experiments. For leftist trees and paring heaps, dual correspondence was always inferior to total correspondence, which, in turn, was always inferior to leaf correspondence. In fact, in the INC data set dual correspondence leftist trees made seven times as many comparisons as did leaf correspondence leftist trees for some combinations of n and m. On the comparison count measure, dual correspondence worked better than total correspondence only for pairing haps with $n \in \{1K, 10K\}$ and for data set DEC with $m = 100K$ and all tested n. Of the priority queue structures used by us, leaf correpondence pairing heaps generally outperformed the others. But, even leaf correspondence priority queues were, often, no match for splay trees.

Table 4 gives the runtimes for the various methods, and Table 5 gives the standard deviations in runtime. Once again, the standard deviations are relatively small. The leaf correspondence version of each data structures was, almost always, superior to the total correspondence version; and the total correspondence version was always superior to the dual correspondence version. Of the priority queue structures used by us, leaf correspondence leftist trees took least time almost always. In fact, leaf correspondence leftist trees took one-sixth the time taken by leaf correspondence pairing heaps and one-twentieth the time taken by leaf correspondence FMPQs on some data sets. Even though leaf correspondence leftist trees were faster than the other priority queue structure, they were generally slower than splay trees, at times taking three times as much time. Note, however, that splay trees are not efficiently meldable, whereas leaf correspondence leftist trees may be melded in logarithmic time.

7 Conclusion

We have shown the general applicability of correspondence methods to arrive at double-ended priority queue structures from single-ended priority queue structures. Experimental studies conducted by us indicate that leaf correspondence leftist trees are superior to the other correspondence structures considered. However, even leaf correspondence leftist trees are unable to outperform splay trees on random data.

References

[1] M. Atkinson, J. Sack, N. Santoro, and T. Strothotte, Min-max heaps and generalized priority queues, *Communications of the ACM*, 29, 996-1000, 1986.

[2] G. Brodal, Fast meldable priority queues, *Workshop on Algorithms and Data Structures*, 1995.

[3] S. Carlsson, The deap – A double ended heap to implement double ended priority queues, *Information Processing Letters*, 26, 33-36, 1987.

[4] S. Chang and M. Du, Diamond deque: A simple data structure for priority deques, *Information Processing Letters*, 46, 231-237, 1993.

[5] C. Crane, *Linear lists and priority queues as balanced binary trees*, Technical Report CS-72-259, Computer Science Department, Stanford University,

[6] S. Cho and S. Sahni, Weight biased leftist trees and modified skip lists, to appear in ACM Jr. on Experimental Algorithms.

[7] S. Cho and S. Sahni, Mergeable double ended priority queue, to appear in International Journal on Foundation of Computer Sciences

[8] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, The paring heap : A new form of self-adjusting heap, Algorithmica, 1:111-129, 1986.

[9] M. Fredman and R. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, JACM, 34:3, 596-615, 1987.

[10] E. Horowitz, S. Sahni, D. Mehta, *Fundamentals of Data Structures in C++*, Computer Science Press, NY, 1995.

[11] J. van Leeuwen and D. Wood, Interval heaps, *The Computer Journal*, 36, 3, 209-216, 1993.

[12] S. Olariu, C. Overstreet, and Z. Wen, A mergeable double-ended priority queue, *The Computer Journal*, 34, 5, 423-427, 1991.

[13] D. Sleator and R. Tarjan, Self-adjusting binary search trees, JACM, 32:3, 652-686, 1985.

[14] D. Sleator and R. Tarjan, Self-adjusting heaps, SIAM Journal on Computing, 15,1, 52-69, 1986.

[15] J. T. Stasko and J. S. Vitter, Pairing heaps : Experiments and Analysis, Communication of the ACM, 30:3, 234-249, 1987.

[16] R. Tarjan, *Data structures and network algorithms*, SIAM, Philadelphia, PA, 1983.

[17] J. Williams, Algorithm 232, *Communications of the ACM*, 7, 347-348, 1964.