Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or `permissions@acm.org`.

Weight Biased Leftist Trees and Modified Skip Lists

Seonghun Cho

and

Sartaj Sahni

Department of Computer and Information Science and Engineering, University of Florida,

Gainesville, FL 32611, U.S.A.

This research was supported, in part, by the Army Research Office under grant DAA H04-95-1-0111, and by the National Science Foundation under grant MIP91-03379.

We propose the weight biased leftist tree as an alternative to traditional leftist trees [CRAN72] for the representation of mergeable priority queues. A modified version of skip lists [PUGH90] that uses fixed size nodes is also proposed. Experimental results show our modified skip list structure is faster than the original skip list structure for the representation of dictionaries. Experimental results comparing weight biased leftist trees and competing priority queue structures are presented.

Categories and Subject Descriptors: E.1 [**Data Structures**]: trees

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Leftist trees, skip lists, dictionary, priority queue

---

1. INTRODUCTION

Several data structures (e.g., heaps, leftist trees [CRAN72], Fibonacci heaps [FRED87], binomial heaps [BROW78], skew heaps [SLEA86], and pairing heaps [FRED86]) have been proposed for the representation of a (single ended) priority queue. Although find min, insert, and delete min are the primary operations that a priority queue supports, many authors consider additional operations such as delete an arbitrary element (assuming we have a pointer to the element), decrease the key of

an arbitrary element (again assuming we have a pointer to this element), meld two priority queues, and initialize a priority queue with a nonzero number of elements. In this paper, we are concerned primarily with the insert and delete min operations.

The different data structures that have been proposed for the representation of a priority queue differ in terms of the performance guarantees they provide. Some guarantee good performance on a per operation basis while others do this only in the amortized sense. Heaps permit one to delete the min element and insert an arbitrary element into an n element priority queue in $O(\log n)$ time per operation; a find min takes $O(1)$ time. Additionally, a heap is an implicit data structure that has no storage overhead associated with it. All other priority queue structures are pointer-based and so require additional storage for the pointers. Leftist trees also support the insert and delete min operations in $O(\log n)$ time per operation and the find min operation in $O(1)$ time. Additionally, they permit us to meld pairs of priority queues in logarithmic time.

The remaining structures do not guarantee good complexity on a per operation basis. They do, however, have good amortized complexity. Using Fibonacci heaps, binomial queues, or skew heaps, find min, inserts and melds take $O(1)$ time (actual and amortized) and a delete-min takes $O(\log n)$ amortized time. When a pairing heap is used, the amortized complexity is $O(1)$ for find min and insert (provided no decrease key operations are performed) and O(log n) for delete min operations [STAS87].

In this paper, we begin in Section 2, by developing the weight biased leftist tree (WBLT). This is similar to a leftist tree. However biasing of left and right subtrees is done by number of nodes rather than by length of paths. Like the leftist tree, a

WBLT permits one to do a find min in O(1) time and each insert, delete min, and meld operation takes $O(\log n)$ time.

Experimental results presented in Section 5 show that WBLTs provide better performance than provided by leftist trees. In fact, *of the priority queue data structures that provide good per operation performance guarantee, weighted leftist trees have best measured performance.* When the WBLT is compared against the structures that provide good amortized complexity (but do not provide good complexity on a per operation basis), our experiments indicate that the WBLT provides superior performance than binomial queues. It is better than skew heaps except when the keys are inserted in ascending order. However, the pairing heap is the best of the priority queue structures tested. Surprisingly, the splay tree [SLEA85] which supports the more general dictionary operations with good amortized complexity outperforms all priority queue structures when the operations are limited to insert and delete min! This conclusion is consistent with that obtained by Jones in his expermental evaluation of priority queue representations [JONE86]. Note that neither the experimental work of Jones nor our work includes measurements for operation mixes that include operations such as decrease key, arbitrary delete, and meld.

The experimental comparisons of Section 5 also include a comparison with unbalanced binary search trees and the probabilistic structures treap [ARAG89] and skip lists [PUGH90].

In Section 3, we propose a fixed node size representation for skip lists. The new structure is called modified skip lists and is experimentally compared with the variable node size structure skip lists. Our experiments indicate that modified skip

lists are faster than skip lists when used to represent dictionaries.

Modified skip lists are augmented by a thread in Section 4 to obtain a structure suitable for use as a priority queue.

2. WEIGHT BIASED LEFTIST TREES

Let T be an extended binary tree. For any internal node x of T, let $LeftChild(x)$ and $RightChild(x)$, respectively, denote the left and right children of x. The weight, $w(x)$, of any node x is the number of internal nodes in the subtree with root x. The length, $shortest(x)$, of a shortest path from x to an external node satisfies the recurrence

$$shortest(x) = \begin{cases} 0 & \text{if } x \text{ is an external node} \\ 1 + min\{shortest(LeftChild(x)), shortest(RightChild(x))\} & \text{otherwise.} \end{cases}$$

Definition [CRAN72] A *leftist tree* (LT) is a binary tree such that if it is not empty, then

$$shortest(LeftChild(x)) \geq shortest(RightChild(x))$$

for every internal node x.

A weight biased leftist tree (WBLT) is defined by using the weight measure in place of the measure $shortest$.

Definition A *weight biased leftist tree* (WBLT) is a binary tree such that if it is not empty, then

$$weight(LeftChild(x)) \geq weight(RightChild(x))$$

for every internal node x.

It is known [CRAN72] that the length, $rightmost(x)$, of the rightmost root to

external node path of any subtree, x, of a leftist tree satisfies

$$rightmost(x) \leq \log_2(w(x) + 1).$$

The same is true for weight biased leftist trees.

THEOREM 1. *Let x be any internal node of a weight biased leftist tree.* $rightmost(x) \leq \log_2(w(x) + 1)$.

PROOF. The proof is by induction on $w(x)$. When $w(x) = 1$, $rightmost(x) = 1$ and $\log_2(w(x) + 1) = \log_2 2 = 1$. For the induction hypothesis, assume that $rightmost(x) \leq \log_2(w(x)+1)$ whenever $w(x) < n$. When $w(x) = n$, $w(RightChild(x)) \leq (n-1)/2$ and $rightmost(x) = 1+rightmost(RightChild(x)) \leq 1+\log_2((n-1)/2+1) = 1 + \log_2(n + 1) - 1 = \log_2(n + 1)$. $\square$

Definition A min (max)-WBLT is a WBLT that is also a min (max) tree.

Each node of a min-WBLT has the fields: *lsize* (number of internal nodes in left subtree), *rsize*, *left* (pointer to left subtree), *right*, and *data*. While the number of size fields in a node may be reduced to one, two fields result in a faster implementation. We assume a head node *head* with $lsize = \infty$ and $lchild = head$. In addition, a bottom node *bottom* with $data.key = \infty$. All pointers that would normally be *nil* are replaced by a pointer to *bottom*. Figure 1(a) shows the representation of an empty min-WBLT and Figure 1(b) shows an example non empty min-WBLT. Notice that all elements are in the right subtree of the head node.

Min (max)-WBLTs can be used as priority queues in the same way as min (max)-LTs. For instance, a min-WBLT supports the standard priority queue operations of insert and delete-min in logarithmic time. In addition, the melding operation

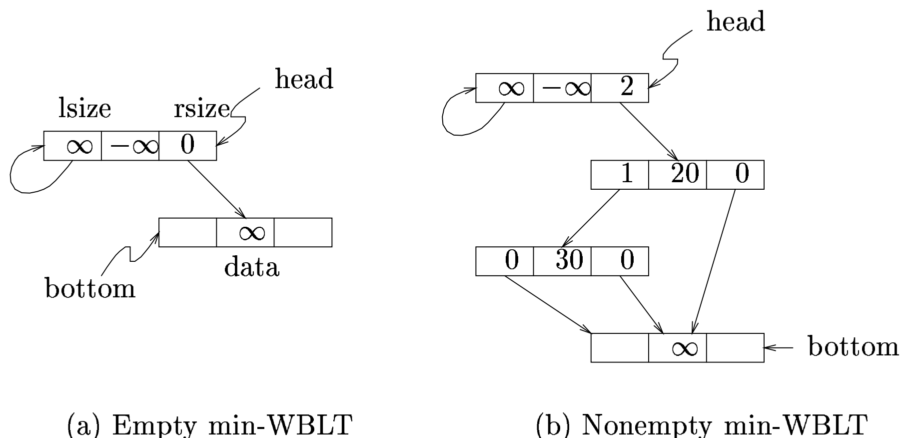

Fig. 1. Example min-WBLTs

(i.e., join two priority queues together) can also be done in logarithmic time. The algorithms for these operations have the same flavor as the corresponding ones for min-LTs. A high level description of the insert and delete-min algorithm for min-WBLT is given in Figures 2 and 3, respectively. The algorithm to meld two min-WBLTs is similar to the delete-min algorithm. The time required to perform each of the operations on a min-WBLT T is $O(rightmost(T))$.

Notice that while the insert and delete-min operations for min-LTs require a top-down pass followed by a bottom-up pass, these operations can be performed by a single top-down pass in min-WBLTs. Hence, we expect min-WBLTs to outperform min-LTs.

3. MODIFIED SKIP LISTS

Skip lists were proposed in [PUGH90] as a probabilistic solution for the dictionary problem (i.e., represent a set of keys and support the operations of search, insert, and delete). The essential idea in skip lists is to maintain upto $lmax$ ordered chains designated as level 1 chain, level 2 chain, etc. If we currently have $lcurrent$ number

```
procedure Insert(d) ;
{insert d into a min-WBLT}
begin
create a node x with x.data = d ;
t = head ; {head node}
while (t.right.data.key < d.key) do
      begin
      t.rsize = t.rsize + 1 ;
      if (t.lsize < t.rsize) then
            begin swap t's children ; t = t.left ; end
      else t = t.right ;
      end ;
x.left = t.right ; x.right = bottom ;
x.lsize = t.rsize ; x.rsize = 0 ;
if (t.lsize = t.rsize) then {swap children}
      begin
      t.right = t.left ;
      t.left = x ; t.lsize = x.lsize + 1 ;
      end
else
      begin t.right = x ; t.rsize = t.rsize + 1 ; end ;
end ;
```

Fig. 2. min-WBLT Insert

of chains, then all n elements of the dictionary are in the level 1 chain and for each l, $2 \leq l \leq lcurrent$, approximately a fraction p of the elements on the level $l-1$ chain are also on the level l chain. Ideally, if the level $l-1$ chain has m elements then the approximately $m \times p$ elements on the level l chain are about $1/p$ apart in the level $l-1$ chain. Figure 4 shows an ideal situation for the case $lcurrent = 4$ and $p = 1/2$.

While the search, insert, and delete algorithms for skip lists are simple and have probabilistic complexity $O(\log n)$ when the level 1 chain has n elements, skip lists suffer from the following implementational drawbacks:

(1) In programming languages such as Pascal, it isn't possible to have variable size nodes. As a result, each node has one *data* field, and *lmax* pointer fields. So, the n element nodes have a total of $n \times lmax$ pointer fields even though only about $n/(1-p)$ pointers are necessary. Since *lmax* is generally much larger

```
procedure Delete-min ;
begin
x = head.right ;
if (x = bottom) then return ; {empty tree}
head.right = x.left ; head.rsize = x.lsize ;
a = head;
b = x.right ; bsize = x.rsize ;
delete x ;
if (b = bottom) then return ;
r = a.right ;
while (r ≠ bottom) do
    begin
    s = bsize + a.rsize ; t = a.rsize ;
    if (a.lsize < s) then  {work on a.left}
        begin
        a.right = a.left ; a.rsize = a.lsize ; a.lsize = s ;
        if (r.data.key > b.data.key) then
            begin a.left = b ; a = b ; b = r ; bsize = t ; end
        else
            begin a.left = r ; a = r ; end
        end
    else
        do symmetric operations on a.right ;
    r = a.right ;
    end ;
if (a.lsize < bsize) then
    begin
    a.right = a.left ; a.left = b ;
    a.rsize = a.lsize ; a.lsize = bsize ;
    end
else
    begin a.right = b ; a.rsize = bsize ; end ;
end ;
```

Fig. 3. min-WBLT Delete-min

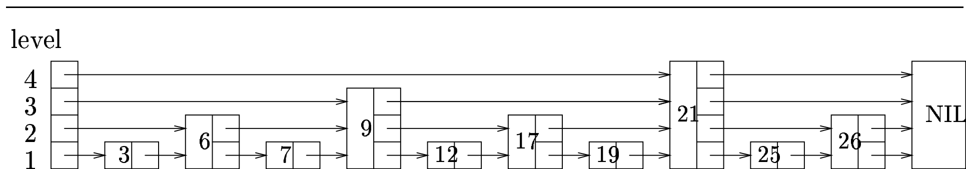

Fig. 4. Skip Lists

than 3 (the recommended value is $\log_{1/p} nMax$ where $nMax$ is the largest number of elements expected in the dictionary), skip lists require more space than WBLTs.

(2) While languages such as C and C++ support variable size nodes and we can construct variable size nodes using simulated pointers [SAHN93] in languages such as Pascal that do not support variable size nodes, the use of variable size nodes requires more complex storage management techniques than required by the use of fixed size nodes. So, greater efficiency can be achieved using simulated pointers and fixed size nodes.

With these two observations in mind, we propose a modified skip list (MSL) structure in which each node has one *data* field and three pointer fields: $left$, *right*, and *down*. Notice that this means MSLs use four fields per node while WBLTs use five (as indicated earlier this can be reduced to four at the expense of increased run time). The $left$ and *right* fields are used to maintain each level l chain as a doubly linked list and the *down* field of a level l node x points to the leftmost node in the level $l-1$ chain that has key value larger than the key in x. Figure 5 shows the modified skip list that corresponds to the skip list of Figure 4. Notice that each element is in exactly one doubly linked list. We can reduce the number of pointers in each node to two by eliminating the field $left$ and having *down* point one node the left of where it currently points (except for head nodes whose down fields still point to the head node of the next chain). However, this results in a less time efficient implementation. H and T, respectively, point to the head and tail of the level *lcurrent* chain.

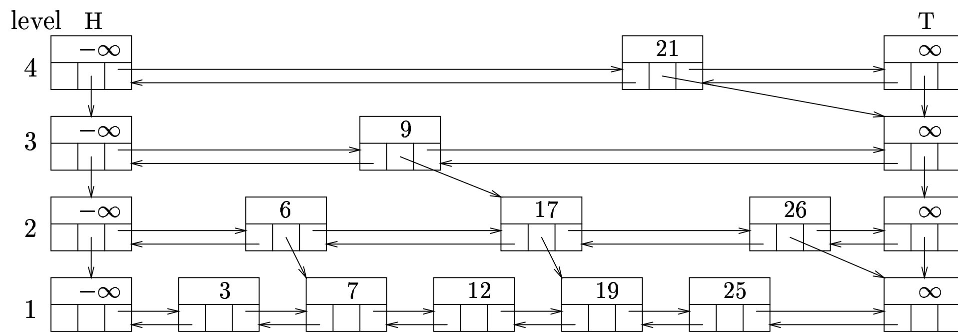

Fig. 5. Modified Skip Lists

```
procedure Search(key) ;
begin
p = H ;
while (p ≠ nil) do
      begin
      while (p.data.key < key) do
            p = p.right ;
      if (p.data.key = key) then report and stop
      else p = p.left.down ; {1 level down}
      end ;
end ;
```

Fig. 6. MSL Search

A high level description of the algorithms to search, insert, and delete are given in Figures 6, 7, and 8. The next theorem shows that their probabilistic complexity is $O(\log n)$ where n is the total number of elements in the dictionary.

THEOREM 2. *The probabilistic complexity of the MSL operations is* $O(\log n)$.

PROOF. We establish this by showing that our algorithms do at most a logarithmic amount of additional work than do those of [PUGH90]. Since the algorithms of [PUGH90] has probabilistic $O(\log n)$ complexity, so also do ours. During a search, the extra work results from moving back one node on each level and then moving down one level. When this is done from any level other than $lcurrent$, we expect to examine upto $c = 1/p - 1$ additional nodes on the next lower level. Hence, upto

procedure Insert(d) ;
begin
randomly generate the level k at which d is to be inserted ;
search the MSL H for $d.key$ saving information useful for insertion ;
if $d.key$ is found **then** fail ; {duplicate}
get a new node x and set $x.data = d$;
if $((k > lcurrent)$ **and** $(lcurrent \neq lmax))$ **then**
    begin
    $lcurrent = lcurrent + 1$;
    create a new chain with a *head* node, node x, and a *tail* and
    connect this chain to H ;
    update H ;
    set $x.down$ to the appropriate node in the level $lcurrent - 1$ chain (to *nil* if $k = 1$) ;
    end
else
    begin
    insert x into the level k chain ;
    set $x.down$ to the appropriate node in the level $k - 1$ chain (to *nil* if $k = 1$) ;
    update the *down* field of nodes on the level $k + 1$ chain (if any) as needed ;
    end ;
end ;

Fig. 7. MSL Insert

procedure Delete(z) ;
begin
search the MSL H for a node x with $data.key = z$ saving information useful for deletion;
if not found **then** fail ;
let k be the level at which z is found ;
for each node p on level $k + 1$ that has $p.down = x$, set $p.down = x.right$;
delete x from the level k list ;
if the list at level $lcurrent$ becomes empty **then**
    delete this and succeeding empty lists until we reach the first non empty list,
    update $lcurrent$;
end ;

Fig. 8. MSL Delete

$c(lcurrent - 2)$ additional nodes get examined. During an insert, we also need to verify that the element being inserted isn't one of the elements already in the MSL. This requires an additional comparison at each level. So, MSLs may make upto $c(lcurrent - 2) + lcurrent$ additional compares during an insert. The number of *down* pointers that need to be changed during an insert or delete is expected to be $\sum_{i=1}^{\infty} ip^i = \frac{1}{(1-p)^2}$. Since c and p are constants and $lmax = \log_{1/p} n$, the expected additional work is $O(\log n)$. $\square$

The relative performance of skip lists and modified skip lists as a data structure for dictionaries was determined by programming the two in C. Both were implemented using simulated pointers. The simulated pointer implementation of skip lists used fixed size nodes. This avoided the use of complex storage management methods and biased the run time measurements in favor of skip lists. For the case of skip lists, we used $p = 1/4$ and for MSLs, $p = 1/5$. These values of p were found, experimentally, to work best for each structure. $lmax$ was set to 16 for both structures. In determining the level assigned to a new element upon insertion, we used the "fix-the-dice" approach suggested in [PUGH90].

We experimented with $n =$ 10,000, 50,000, 100,000, and 200,000. For each n, the following five part experiment was conducted:

(a) start with an empty structure and perform n inserts;

(b) search for each item in the resulting structure once; items are searched for in the order they were inserted

(c) perform an alternating sequence of n inserts and n deletes; in this, the n elements inserted in (a) are deleted in the order they were inserted and n new elements are

n	operation	random inputs		ordered inputs	
		SKIP	MSL	SKIP	MSL
10,000	insert	225	322	247	319
	search	255	363	257	339
	ins/del	519	734	355	560
	search	256	350	251	339
	delete	232	321	84	185
50,000	insert	1357	1951	1422	1912
	search	1537	1966	1467	1837
	ins/del	2997	4142	1973	3204
	search	1502	2039	1450	1990
	delete	1374	1854	486	9320
100,000	insert	2919	4146	2926	4276
	search	3189	4316	2971	4082
	ins/del	6399	9103	4406	6896
	search	3225	4428	3277	4346
	delete	2981	4162	961	2053
200,000	insert	6179	8928	6403	9023
	search	6697	9274	6448	8946
	ins/del	13378	19371	9054	9062
	search	6681	9662	6458	9198
	delete	6149	9102	1995	4838

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

inserted

(d) search for each of the remaining n elements in the order they were inserted

(e) delete the n elements in the order they were inserted.

For each n, the above five part experiment was repeated ten times using different random permutations of distinct elements. For each sequence, we measured the total number of element comparisons performed and then averaged these over the ten sequences. The average number of comparisons (in thousands) for each of the five parts of the experiment are given in Table 1.

Also given in this table is the number of comparisons using ordered data. For this data set, elements were inserted and deleted in the order $1, 2, 3, \ldots$. For the case of random data, MSLs make 40% to 50% more comparisons on each of the five parts of the experiment. On ordered inputs, the disparity is even greater with

n	random inputs		ordered inputs	
	SKIP	MSL	SKIP	MSL
10,000	8,8	7,7	8,8	7,7
50,000	9,9	7,7	9,9	7,7
100,000	9,9	7,8	9,9	7,8
200,000	9,9	8,9	9,9	8,9

Table 2. Average number of levels

MSLs making 30% to 140% more comparison. Table 2 gives the number of levels in SKIP and MSL. The first number of each entry is the number of levels following part (a) of the experiment and the second the number of levels following part (b). As can be seen, the number of levels is very comparable for both structures. MSLs generally had one or two levels fewer than SKIPs had.

Despite the large disparity in number of comparisons, MSLs generally required less time than required by SKIPs (see Table 3). Integer keys were used for our run time measurements. In many practical situations the observed time difference will be noticeably greater as one would need to code skip lists using more complex storage management techniques to allow for variable size nodes. Note that while MSLs require less storage than ordinary skip lists when used in languages that do not support the dynamic construction of variable size arrays, they require more storage when used in programming languages such as C, C++, and Java that do permit dynamic allocation of variable size arrays.

4. MSLS AS PRIORITY QUEUES

At first glance, it might appear that skip lists are clearly a better choice than modified skip lists for use as a priority queue. The min element in a skip list is the first element in the level one chain. So, it can be identified in O(1) time and then deleted in $O(\log n)$ probabilistic time. In the case of MSLs, the min element is the

n	operation	random inputs		ordered inputs	
		SKIP	MSL	SKIP	MSL
10,000	insert	0.24	0.18	0.20	0.17
	search	0.18	0.12	0.12	0.07
	ins/del	0.45	0.35	0.20	0.20
	search	0.18	0.12	0.13	0.07
	delete	0.16	0.12	0.07	0.05
50,000	insert	1.36	1.22	0.92	0.80
	search	1.25	0.98	0.62	0.38
	ins/del	2.73	2.53	1.07	1.08
	search	1.16	1.00	0.62	0.42
	delete	1.10	0.83	0.27	0.23
100,000	insert	2.84	2.86	1.72	1.60
	search	2.63	2.39	1.23	0.85
	ins/del	6.13	5.80	2.43	2.28
	search	2.61	2.33	1.35	0.92
	delete	2.41	2.02	0.55	0.52
200,000	insert	6.25	6.49	3.52	3.47
	search	5.85	5.34	2.70	1.87
	ins/del	13.29	13.02	5.13	4.75
	search	5.81	5.51	2.72	1.92
	delete	5.35	4.85	1.12	1.18

Table 3. Average run time

first one in one of the *lcurrent* chains. This can be identified in logarithmic time using a loser tree whose elements are the first element from each MSL chain. By using an additional pointer field in each node, we can thread the elements in an MSL into a chain. The elements appear in non-decending order on this chain. The resulting threaded structure is referred to as TMSL (threaded modified skip lists). A delete min operation can be done in O(1) expected time when a TMSL is used. The expected time for an insert remains $O(\log n)$. The algorithms for the insert and delete min operations for TMSLs are given in Figures 9 and 10, respectively. The last step of Figure 9 is implemented by first finding the largest element on level 1 with key $< d.key$ (for this, start at level $lcurrent - 1$) and then follow the threaded chain.

THEOREM 3. *The expected complexity of an insert and delete-min operation in*

```
procedure Insert(d) ;
begin
randomly generate the level k at which d is to be inserted ;
get a new node x and set x.data = d ;
if ((k > lcurrent) and (lcurrent ≠ lmax)) then
      begin
      lcurrent = lcurrent + 1 ;
      create a new chain with a head node, node x, and a tail and
      connect this chain to H ;
      update H ;
      set x.down to the appropriate node in the level lcurrent − 1 chain (to nil if k = 1) ;
      end
else
      begin
      insert x into the level k chain ;
      set x.down to the appropriate node in the level k − 1 chain (to nil if k = 1) ;
      update the down field of nodes on the level k + 1 chain (if any) as needed ;
      end ;
find node with largest key < d.key and insert x into threaded list ;
end ;
```

Fig. 9. TMSL Insert

```
procedure Delete-min ;
begin
delete the first node x from the thread list ;
let k be the level x is on ;
delete x from the level k list (note there are no down fields on level k + 1
that need to be updated) ;
if the list at level lcurrent becomes empty then
      delete this and succeeding empty lists until we reach the first non empty list,
      update lcurrent ;
end ;
```

Fig. 10. TMSL Delete-min

```
procedure Delete-max ;
begin
delete the last node x from the thread list ;
let k be the level x is on ;
delete x from the level k list updating p.down for nodes on level k + 1 as necessary ;
if the list at level lcurrent becomes empty then
      delete this and succeeding empty lists until we reach the first non empty list,
      update lcurrent ;
end ;
```

Fig. 11. TMSL Delete-max

a TMSL is O($\log n$*) and O(1), respectively.*

PROOF. Follows from the notion of a thread, Theorem 2, and [PUGH90]. □

TMSLs may be further extended by making the threaded chain a doubly linked list. This permits both delete-min and delete-max to be done in $\Theta(1)$ expected time and insert in $O(\log n)$ expected time. With this extension, TMSLs may be used to represent double ended priority queues.

5. EXPERIMENTAL RESULTS FOR PRIORITY QUEUES

The single-ended priority queue structures min heap (Heap), binomial queue (BQueue), leftist trees (LT), weight biased leftist trees (WBLT), pairing heap (Pair), skew heaps (Skew), and TMSLs (MSL) were programmed in C. In addition, priority queue versions of unbalanced binary search trees (BST), AVL trees (AVL), splay trees (Splay), treaps (TRP), and skip lists (SKIP) were also programmed. The priority queue version of these structures differed from their normal dictionary versions in that the delete operation was customized to support only a delete min. (Note: for pairing heaps, skew heaps, and splay trees, we used the codes developed by Jones [JONE86]). For skip lists and TMSLs, the level allocation probability p was set to 1/4. While BSTs are normally defined only for the case when the keys are distinct, they are easily extended to handle multiple elements with the same

key. In our extension, if a node has key x, then its left subtree has values $< x$ and its right values $\geq x$. To minimize the effects of system call overheads, all structures (other than Heap) were programmed using simulated pointers. The min heap was programmed using a one-dimensional array.

For our experiments, we began with structures initialized with n = 100, 1,000, and 10,000 elements and then performed random sequences of 100,000, 500,000, and 1,000,000 operations. This random sequence consists of approximately 50% insert and 50% delete min operations. The results are given in Tables 4-19. In the data sets 'random1' and 'random2', the elements to be inserted were randomly generated while in the data set 'increasing' an ascending sequence of elements was inserted and in the data set 'decreasing', a descending sequence of elements was used. Since BSTs have very poor performance on the last two data sets, we excluded it from this part of the experiment. In the case of both random1 and random2, ten random sequences were used and the average of these ten is reported. The random1 and random2 sequences differed in that for random1, the keys were integers in the range $0..(10^6 - 1)$ while for random2, they were in the range 0..999. So, random2 is expected to have many more duplicates. Also, random2 is expected to have a more uniform distribution as the random numbers used by us were obtained by extracting bits 5 through 15 of the number generated by the C random number generating function *rand*.

Tables 4 and 5 give the total number of comparisons (in thousands) made by each of the methods. On the two random data tests as well as on the decreasing order data set, leftist trees, weight biased leftist trees, pairing heaps, and splay trees required the fewest number of comparisons. With ascending data, splay trees

did best.

The structure height initially and following the randon sequences of operations is given in Tables 6 and 7. For BQueues, the height of the tallest tree is given. For SKIPs and TMSLs, table 7 gives the number of levels. In the case of LT and WBLT, table 6 gives the length of the rightmost path following initialization and the average of its length following each of the operations in the sequence. The two leftist structures are able to maintain their rightmost paths so as to have a length much less than $\log_2(n+1)$.

The measured run times on a Sun Sparc 5 are given in Tables 8 and 9 for integer keys and in Tables 10 and 11 for double precision keys. The codes were compiled using the cc compiler in optimized mode. Of the tested structures that provide a good performance guarantee on a per operation basis, the WBLT generally performed best on the random1, random2, and decreasing data sets while the min heap did best on the increasing data set. When considered along with data structures that provide a good amortized performance guarantee, the splay tree did best when integer keys were used. However, with double precision keys, the WBLT remained best for the random1 and random2 data sets when n was 100 and 1000, and the splay tree was better when n was 10,000. For the increasing data set, splay trees performed better and for the decreasing data set, splay trees and WBLTs have almost the same performance.

The standard deviations in the data reported in Tables 4-11 is given in Tables 12-19. These standard deviations are relatively small, rarely exceeding 10%. Therefore, we have confidence in the measured results and our conclusions.

6. CONCLUSION

We have developed two new data structures: weight biased leftist trees and modified skip lists. Experiments indicate that WBLTs have better performance (i.e., run time characteristic and number of comparisons) than LTs as a data structure for single ended priority queues and MSLs have a better performance than skip lists as a data structure for dictionaries.

Of the tested data structures that provide good performance guarantee on a per operation basis, WBLTs have best performance except when keys are inserted in ascending order. In this latter case, heaps have best performance. When amortized performance guarantees are sufficient, the splay tree is the best data structure to use. Note, however, that the splay tree does not support amortized log time melds and so in priority queue applications that perform frequent meld operations, the WBLT would outperform the splay tree.

Our experimental results for single ended priority queues are in marked contrast to those reported in [GONN91, p183] where leftist trees are reported to take approximately four times as much time as heaps. We suspect this difference in results is because of different programming techniques (recursion vs. iteration, dynamic vs. static memory allocation, etc.) used in [GONN91] for the different structures. In our experiments, all structures were coded using similar programming techniques.

On the other hand, our results are in agreement with those of Jones [JONE86]. The relative performance we observed for the three codes developed by Jones–splay trees, skew heaps, and pairing heaps–was the same as reported in [JONE86]; splay trees are better than pairing heaps, which in turn are better than skew heaps.

inputs	m	n	Heap	BQueue	LT	WBLT	Pair	Skew
rand1	100K	100	1092	256	63	63	62	391
		1,000	1504	284	119	118	114	497
		10,000	1751	642	454	446	417	730
	500K	100	5491	1258	267	267	266	1942
		1,000	7687	1305	348	346	340	2392
		10,000	9547	2633	987	973	906	3179
	1M	100	10990	2509	519	518	517	3871
		1,000	15427	2518	610	608	601	4751
		10,000	19456	4949	1384	1367	1284	6139
rand2	100K	100	744	254	62	62	62	372
		1,000	1318	288	115	114	114	476
		10,000	1388	677	424	414	417	681
	500K	100	3010	1260	264	264	265	1652
		1,000	4727	1288	335	333	338	2109
		10,000	5356	2626	891	866	897	2830
	1M	100	5492	2509	514	514	514	3229
		1,000	8883	2568	591	588	596	3956
		10,000	9976	4994	1243	1211	1271	5268
inc	100K	100	822	574	704	701	306	651
		1,000	1063	735	937	930	314	806
		10,000	1386	1073	1238	1229	316	983
	500K	100	4111	2870	3521	3502	1528	3258
		1,000	5319	3673	4682	4649	1567	4041
		10,000	6936	5327	6196	6145	1583	5034
	1M	100	8223	5741	7041	7002	3055	6517
		1,000	10638	7346	9365	9299	3134	8085
		10,000	13877	10645	12396	12290	3166	10098
dec	100K	100	1100	150	50	50	50	150
		1,000	1550	201	50	50	50	150
		10,000	1999	410	50	50	50	150
	500K	100	5500	750	250	250	250	750
		1,000	7750	1001	250	250	250	750
		10,000	9999	2010	250	250	250	7500
	1M	100	11000	1500	500	500	500	1500
		1,000	15500	2001	500	500	500	1500
		10,000	19999	4010	500	500	500	1500

m = the number of operations performed

n = the number of elements in initial data structures

Table 4: The number of key comparisons (in thousands)

inputs	m	n	BST	TRP	Skip	MSL	AVL	Splay	WBLT
rand1	100K	100	350	422	243	513	388	66	63
		1,000	490	527	350	681	499	124	118
		10,000	643	752	712	1163	664	423	446
	500K	100	1711	2213	1149	2469	1928	271	267
		1,000	2181	2487	1590	3283	2464	358	346
		10,000	2933	3316	2439	4463	3204	973	973
	1M	100	3363	4331	2360	5072	3840	523	518
		1,000	4339	5054	3086	6451	4921	623	608
		10,000	5583	6335	4472	8636	6352	1382	1367
rand2	100K	100	254	349	242	511	374	66	62
		1,000	363	447	360	711	491	120	114
		10,000	961	762	674	1053	666	376	414
	500K	100	876	1305	1170	2512	1832	270	264
		1,000	1516	2016	1543	3192	2354	346	333
		10,000	4531	3128	2392	4389	3123	845	866
	1M	100	1425	2209	2345	5045	3646	523	514
		1,000	2607	3610	2930	6147	4604	606	588
		10,000	8128	5639	4452	8500	6142	1206	1211
inc	100K	100	–	415	786	1026	456	50	701
		1,000	–	502	1031	1353	583	50	929
		10,000	–	646	1315	1688	749	50	1229
	500K	100	–	2102	3910	5141	2278	250	3502
		1,000	–	2539	5104	6678	2912	250	4649
		10,000	–	3097	6636	8484	3742	250	6145
	1M	100	–	4197	7816	10302	4555	500	7002
		1,000	–	5069	10215	13363	5824	500	9299
		10,000	–	6202	13246	17252	7484	500	12290
dec	100K	100	–	570	250	537	400	50	50
		1,000	–	326	300	637	551	50	50
		10,000	–	452	400	838	701	50	50
	500K	100	–	2850	1250	2684	2000	250	250
		1,000	–	1625	1500	3184	2751	250	250
		10,000	–	2256	2000	4184	3501	250	250
	1M	100	–	5699	2500	5367	4000	500	500
		1,000	–	3248	3000	6367	5501	500	500
		10,000	–	4512	4000	8368	7000	500	500

m = the number of operations performed

n = the number of elements in initial data structures

Table 5: The number of key comparisons (in thousands)

inputs	m	n	Heap	BQueue	LT	WBLT	Pair	Skew
rand1	100K	100	7,8	1,8	4,1	4,1	33,1	4,1
		1,000	10,11	1,11	5,1	5,2	174,2	5,2
		10,000	14,14	1,14	5,5	9,5	7581,11	6,5
	500K	100	7,8	1,8	4,1	5,1	85,1	6,1
		1,000	10,11	1,11	6,1	5,1	834,1	8,1
		10,000	14,14	1,14	8,1	6,1	2380,1	9,1
	1M	100	7,8	1,8	4,1	4,1	88,1	6,1
		1,000	10,11	1,11	8,1	8,1	535,1	13,1
		10,000	14,14	1,14	9,1	8,1	7957,1	11,1
rand2	100K	100	7,8	1,8	5,1	4,1	49,1	7,1
		1,000	10,11	1,11	7,1	7,1	159,1	9,1
		10,000	14,14	1,14	6,2	6,2	530,3	9,2
	500K	100	7,8	1,8	5,1	4,1	35,1	6,1
		1,000	10,11	1,11	6,1	6,1	259,1	7,1
		10,000	14,14	1,14	4,1	4,1	719,1	5,1
	1M	100	7,8	1,8	5,1	5,1	43,1	7,1
		1,000	10,11	1,11	5,1	4,1	93,1	7,1
		10,000	14,14	1,14	8,1	7,1	592,1	9,1
inc	100K	100	7,8	1,8	6,5	6,7	99,7	100,6
		1,000	10,11	1,11	9,6	9,7	999,5	1000,11
		10,000	14,14	1,14	13,9	13,9	9999,6	10000,11
	500K	100	7,8	1,8	6,5	6,6	99,5	100,6
		1,000	10,11	1,11	9,7	9,6	999,4	1000,6
		10,000	14,14	1,14	13,7	13,8	9999,5	10000,9
	1M	100	7,8	1,8	6,5	6,6	99,199	100,6
		1,000	10,11	1,11	9,8	9,6	999,4	1000,8
		10,000	14,14	1,14	13,9	13,8	9999,5	10000,11
dec	100K	100	7,8	1,8	1,1	1,1	1,1	1,1
		1,000	10,11	1,11	1,1	1,1	1,1	1,1
		10,000	14,14	1,14	1,1	1,1	1,1	1,1
	500K	100	7,8	1,8	1,1	1,1	1,1	1,1
		1,000	10,11	1,11	1,1	1,1	1,1	1,1
		10,000	14,14	1,14	1,1	1,1	1,1	1,1
	1M	100	7,8	1,8	1,1	1,1	1,1	1,1
		1,000	10,11	1,11	1,1	1,1	1,1	1,1
		10,000	14,14	1,14	1,1	1,1	1,1	1,1

m = the number of operations performed

n = the number of elements in initial data structures

Table 6: Height/level of the structures

inputs	m	n	BST	TRP	Skip	MSL	AVL	Splay	WBLT
rand1	100K	100	13,16	13,17	3,4	4,5	8,9	14,19	4,1
		1,000	23,23	24,22	5,5	6,6	12,12	22,26	5,2
		10,000	31,29	31,31	7,7	8,8	16,16	33,36	9,5
	500K	100	11,15	12,15	3,4	4,5	8,9	12,19	5,1
		1,000	26,24	21,23	5,5	6,6	12,12	27,27	5,1
		10,000	30,30	31,31	7,7	8,8	16,16	31,37	6,1
	1M	100	13,16	16,16	3,4	4,5	8,9	14,21	4,1
		1,000	23,22	19,22	5,5	6,6	12,12	22,27	8,1
		10,000	32,31	28,31	7,7	8,8	16,16	31,36	8,1
rand2	100K	100	13,60	13,54	3,4	4,5	8,9	15,33	4,1
		1,000	20,71	19,64	5,5	6,6	12,12	24,43	7,1
		10,000	35,92	36,85	7,6	8,7	16,15	39,56	6,2
	500K	100	14,199	12,199	3,4	4,5	8,8	13,27	4,1
		1,000	20,271	23,250	5,5	6,6	12,11	26,122	6,1
		10,000	35,303	36,273	7,7	8,8	16,15	38,137	4,1
	1M	100	11,199	21,199	3,4	4,5	8,8	20,44	5,1
		1,000	26,513	22,508	5,5	6,6	12,11	22,221	4,1
		10,000	40,560	35,520	7,7	8,8	16,15	35,242	7,1
inc	100K	100	–	11,16	3,4	4,5	7,8	100,57	6,7
		1,000	–	24,24	5,5	6,6	10,11	1000,543	9,7
		10,000	–	33,34	7,7	8,8	14,14	10000,9561	13,9
	500K	100	–	11,15	3,4	4,5	7,8	100,77	6,6
		1,000	–	24,18	5,4	6,5	10,11	1000,707	9,6
		10,000	–	33,31	7,7	8,8	14,14	10000,7601	13,8
	1M	100	–	11,14	3,3	4,4	7,8	100,200	6,6
		1,000	–	24,19	5,5	6,6	10,11	1000,363	9,6
		10,000	–	33,32	7,6	8,7	14,14	10000,5151	13,8
dec	100K	100	–	11,15	3,4	4,5	7,8	100,200	1,1
		1,000	–	24,25	5,5	6,6	10,11	1000,1100	1,1
		10,000	–	33,33	7,7	8,8	14,14	10000,10100	1,1
	500K	100	–	11,14	3,4	4,5	7,8	100,200	1,1
		1,000	–	24,25	5,5	6,6	10,11	1000,1100	1,1
		10,000	–	33,33	7,7	8,8	14,14	10000,10100	1,1
	1M	100	–	11,14	3,4	4,5	7,8	100,200	1,1
		1,000	–	24,25	5,5	6,6	10,11	1000,1100	1,1
		10,000	–	33,33	7,7	8,8	14,14	10000,10100	1,1

m = the number of operations performed

n = the number of elements in initial data structures

Table 7: Height/level of the structures

inputs	m	n	Heap	BQueue	LT	WBLT	Pair	Skew
rand1	100K	100	0.16	0.21	0.09	0.09	0.09	0.14
		1,000	0.21	0.23	0.11	0.11	0.10	0.19
		10,000	0.27	0.47	0.33	0.27	0.23	0.34
	500K	100	0.81	1.01	0.43	0.41	0.42	0.70
		1,000	1.05	1.11	0.47	0.45	0.45	0.95
		10,000	1.40	1.78	0.88	0.76	0.69	1.47
	1M	100	1.63	2.03	0.85	0.83	0.84	1.38
		1,000	2.11	2.10	0.90	0.87	0.88	1.87
		10,000	2.80	3.27	1.40	1.25	1.17	2.83
rand2	100K	100	0.13	0.20	0.09	0.09	0.09	0.13
		1,000	0.19	0.23	0.12	0.10	0.10	0.19
		10,000	0.24	0.47	0.32	0.26	0.23	0.32
	500K	100	0.59	1.01	0.42	0.41	0.42	0.62
		1,000	0.78	1.08	0.46	0.44	0.45	0.83
		10,000	0.95	1.73	0.82	0.71	0.69	1.30
	1M	100	1.13	2.01	0.84	0.83	0.85	1.24
		1,000	1.49	2.11	0.89	0.86	0.87	1.56
		10,000	1.78	3.18	1.30	1.17	1.16	2.37
inc	100K	100	0.13	0.37	0.25	0.23	0.13	0.18
		1,000	0.18	0.55	0.45	0.38	0.15	0.27
		10,000	0.22	0.80	0.72	0.58	0.17	0.40
	500K	100	0.62	1.77	1.32	1.13	0.65	0.88
		1,000	0.88	2.60	2.27	1.85	0.75	1.40
		10,000	1.12	4.02	3.60	2.87	0.83	2.10
	1M	100	1.25	3.40	2.48	2.12	1.30	1.83
		1,000	1.75	4.98	4.38	3.72	1.58	2.78
		10,000	2.25	8.03	7.20	5.73	1.70	4.15
dec	100K	100	0.18	0.15	0.08	0.07	0.10	0.08
		1,000	0.22	0.17	0.08	0.08	0.07	0.08
		10,000	0.28	0.35	0.08	0.08	0.08	0.10
	500K	100	0.82	0.82	0.42	0.42	0.42	0.45
		1,000	1.05	0.88	0.40	0.42	0.38	0.40
		10,000	1.40	1.72	0.38	0.40	0.43	0.43
	1M	100	1.62	1.63	0.82	0.83	0.83	0.87
		1,000	2.12	1.78	0.82	0.82	0.77	0.78
		10,000	2.78	3.43	0.80	0.80	0.83	0.87

Time Unit : *sec*

m = the number of operations performed

n = the number of elements in initial data structures

Table 8: Run time using integer keys

inputs	m	n	BST	TRP	Skip	MSL	AVL	Splay	WBLT
rand1	100K	100	0.13	0.20	0.13	0.16	0.26	0.09	0.09
		1,000	0.15	0.24	0.15	0.18	0.28	0.10	0.11
		10,000	0.21	0.32	0.32	0.36	0.39	0.21	0.27
	500K	100	0.59	1.00	0.64	0.74	1.34	0.40	0.41
		1,000	0.70	1.11	0.71	0.87	1.42	0.41	0.45
		10,000	0.91	1.36	1.05	1.23	1.70	0.63	0.76
	1M	100	1.19	1.99	1.28	1.51	2.68	0.78	0.83
		1,000	1.40	2.21	1.39	1.68	2.86	0.80	0.87
		10,000	1.77	2.54	1.88	2.25	3.29	1.06	1.25
rand2	100K	100	0.10	0.18	0.13	0.15	0.25	0.08	0.09
		1,000	0.13	0.21	0.15	0.18	0.28	0.09	0.10
		10,000	0.33	0.36	0.31	0.35	0.40	0.18	0.26
	500K	100	0.43	0.84	0.63	0.75	1.28	0.38	0.41
		1,000	0.59	1.03	0.70	0.85	1.40	0.41	0.44
		10,000	1.55	1.43	1.04	1.22	1.70	0.56	0.71
	1M	100	0.84	1.61	1.28	1.51	2.54	0.79	0.83
		1,000	1.11	1.95	1.37	1.67	2.76	0.79	0.86
		10,000	2.88	2.64	1.87	2.22	3.23	0.97	1.17
inc	100K	100	–	0.20	0.20	0.20	0.32	0.10	0.23
		1,000	–	0.25	0.27	0.27	0.42	0.13	0.38
		10,000	–	0.27	0.32	0.28	0.50	0.12	0.58
	500K	100	–	1.02	1.05	0.97	1.62	0.52	1.13
		1,000	–	1.15	1.30	1.23	2.03	0.58	1.85
		10,000	–	1.33	1.60	1.47	2.47	0.63	2.87
	1M	100	–	2.00	2.12	1.92	3.35	1.07	2.12
		1,000	–	2.32	2.60	2.45	4.07	1.22	3.72
		10,000	–	2.63	3.17	2.95	4.93	1.30	5.73
dec	100K	100	–	0.23	0.12	0.15	0.40	0.08	0.07
		1,000	–	0.18	0.18	0.17	0.40	0.08	0.08
		10,000	–	0.22	0.15	0.17	0.48	0.08	0.08
	500K	100	–	1.23	0.63	1.00	1.85	0.42	0.42
		1,000	–	0.85	0.67	0.92	2.05	0.35	0.42
		10,000	–	0.95	0.73	0.88	2.43	0.42	0.40
	1M	100	–	2.15	1.47	1.50	4.08	0.82	0.83
		1,000	–	1.77	1.35	1.90	4.08	0.70	0.82
		10,000	–	1.65	1.48	2.00	4.83	0.80	0.80

Time Unit : *sec*

m = the number of operations performed

n = the number of elements in initial data structures

Table 9: Run time using integer keys

inputs	m	n	Heap	BQueue	LT	WBLT	Pair	Skew
rand1	100K	100	0.29	0.24	0.11	0.10	0.10	0.18
		1,000	0.39	0.28	0.15	0.12	0.13	0.24
		10,000	0.54	0.57	0.41	0.32	0.29	0.43
	500K	100	1.46	1.20	0.52	0.47	0.50	0.88
		1,000	1.96	1.27	0.58	0.52	0.55	1.17
		10,000	2.75	2.11	1.09	0.90	0.86	1.87
	1M	100	2.93	2.42	1.04	0.93	1.03	1.75
		1,000	3.93	2.49	1.10	0.99	1.06	2.33
		10,000	5.53	3.89	1.73	1.45	1.45	3.59
rand2	100K	100	0.23	0.24	0.11	0.10	0.11	0.17
		1,000	0.35	0.27	0.14	0.13	0.13	0.24
		10,000	0.46	0.57	0.39	0.31	0.29	0.42
	500K	100	1.01	1.19	0.53	0.48	0.51	0.80
		1,000	1.39	1.26	0.57	0.51	0.54	1.02
		10,000	1.79	2.05	1.01	0.85	0.86	1.64
	1M	100	1.92	2.39	1.02	0.93	1.01	1.56
		1,000	2.66	2.50	1.08	0.98	1.06	1.93
		10,000	3.37	3.77	1.61	1.37	1.44	2.98
inc	100K	100	0.23	0.42	0.33	0.37	0.17	0.25
		1,000	0.33	0.52	0.57	0.50	0.20	0.38
		10,000	0.43	0.95	0.88	0.70	0.23	0.53
	500K	100	1.13	2.02	1.70	1.83	0.85	1.28
		1,000	1.65	2.57	2.75	2.48	1.02	1.88
		10,000	2.07	4.18	4.43	3.50	1.10	2.73
	1M	100	2.28	4.12	3.40	3.48	1.62	2.30
		1,000	3.32	5.12	5.45	5.02	2.05	3.73
		10,000	4.15	8.47	8.83	7.00	2.32	5.57
dec	100K	100	0.30	0.27	0.13	0.10	0.12	0.10
		1,000	0.38	0.22	0.10	0.10	0.10	0.10
		10,000	0.55	0.43	0.10	0.08	0.10	0.12
	500K	100	1.45	0.97	0.48	0.45	0.50	0.55
		1,000	1.97	1.07	0.52	0.47	0.53	0.55
		10,000	2.80	1.63	0.50	0.47	0.47	0.52
	1M	100	2.93	1.98	1.02	0.93	1.03	1.08
		1,000	3.93	2.25	1.02	0.90	0.93	1.02
		10,000	5.58	3.28	1.00	0.95	0.92	1.02

Time Unit : *sec*
m = the number of operations performed
n = the number of elements in initial data structures
Table 10: Run time using real (double) keys

inputs	m	n	BST	TRP	Skip	MSL	AVL	Splay	WBLT
rand1	100K	100	0.16	0.22	0.16	0.21	0.30	0.11	0.10
		1,000	0.21	0.27	0.19	0.25	0.34	0.12	0.12
		10,000	0.29	0.41	0.40	0.47	0.46	0.28	0.32
	500K	100	0.81	1.14	0.78	1.00	1.53	0.51	0.47
		1,000	0.96	1.25	0.90	1.18	1.68	0.54	0.52
		10,000	1.26	1.64	1.32	1.67	2.04	0.84	0.90
	1M	100	1.66	2.24	1.58	2.08	3.08	1.04	0.93
		1,000	1.95	2.51	1.76	2.36	3.38	1.05	0.99
		10,000	2.39	3.08	2.35	3.06	3.94	1.42	1.45
rand2	100K	100	0.14	0.21	0.16	0.21	0.29	0.11	0.10
		1,000	0.18	0.24	0.19	0.26	0.34	0.12	0.13
		10,000	0.46	0.43	0.39	0.46	0.48	0.24	0.31
	500K	100	0.61	0.90	0.79	1.04	1.47	0.52	0.48
		1,000	0.81	1.13	0.88	1.18	1.63	0.53	0.51
		10,000	2.16	1.74	1.30	1.66	2.02	0.75	0.85
	1M	100	1.14	1.69	1.58	2.05	2.93	1.03	0.93
		1,000	1.52	2.17	1.71	2.33	3.27	1.05	0.98
		10,000	4.02	3.10	2.34	3.04	3.87	1.29	1.37
inc	100K	100	–	0.23	0.30	0.28	0.38	0.13	0.37
		1,000	–	0.28	0.35	0.38	0.42	0.15	0.50
		10,000	–	0.32	0.45	0.43	0.53	0.15	0.70
	500K	100	–	1.15	1.48	1.42	1.90	0.68	1.83
		1,000	–	1.32	1.77	1.78	2.13	0.75	2.48
		10,000	–	1.52	2.18	2.17	2.65	0.78	3.50
	1M	100	–	2.25	2.98	2.82	3.85	1.28	3.48
		1,000	–	2.63	3.50	3.57	4.22	1.53	5.02
		10,000	–	3.08	4.28	4.40	5.37	1.65	7.00
dec	100K	100	–	0.30	0.17	0.20	0.40	0.10	0.10
		1,000	–	0.22	0.17	0.22	0.40	0.10	0.10
		10,000	–	0.22	0.22	0.25	0.57	0.10	0.08
	500K	100	–	1.42	0.80	1.00	2.03	0.53	0.45
		1,000	–	1.05	0.83	1.50	2.05	0.55	0.47
		10,000	–	1.10	1.03	1.27	2.62	0.47	0.47
	1M	100	–	2.48	1.78	2.07	4.08	1.07	0.93
		1,000	–	2.13	1.65	2.20	4.30	1.10	0.90
		10,000	–	2.20	2.03	2.98	5.27	0.93	0.95

Time Unit : *sec*
m = the number of operations performed
n = the number of elements in initial data structures
Table 11: Run time using real (double) keys

inputs	m	n	Heap	BQueue	LT	WBLT	Pair	Skew
rand1	100K	100	231	1507	410	407	400	3245
		1,000	337	5160	467	527	543	1591
		10,000	1200	7601	1648	1591	1574	910
	500K	100	243	560	481	477	409	19286
		1,000	1574	804	830	713	748	6698
		10,000	1789	13858	2967	2572	2410	2164
	1M	100	219	1616	453	457	390	30894
		1,000	691	46054	889	859	961	14056
		10,000	1970	33915	3012	2745	2386	4143
rand2	100K	100	9060	6517	199	201	170	3795
		1,000	33050	415	670	573	648	1258
		10,000	7282	675	1062	1213	1159	786
	500K	100	39099	231	206	214	212	50846
		1,000	48193	736	972	824	1254	9104
		10,000	13193	1456	2697	2652	2271	5513
	1M	100	27007	352	194	170	185	232797
		1,000	62067	662	983	1072	894	31907
		10,000	22050	1889	2902	2459	2917	12523

m = the number of operations performed
n = the number of elements in initial data structures
Table 12: Standard deviation of the number of key comparisons

inputs	m	n	BST	TRP	Skip	MSL	AVL	Splay	WBLT
rand1	100K	100	36762	36288	18586	37161	3489	469	407
		1,000	5373	32736	36943	76955	7180	599	527
		10,000	1675	29307	14277	28758	1277	1720	1591
	500K	100	169243	201371	122009	243296	27912	547	477
		1,000	50562	84867	198150	402161	15179	958	713
		10,000	8714	87236	75259	172594	33256	2580	2572
	1M	100	241816	266066	232022	463443	49374	454	457
		1,000	207562	270670	327040	657720	46798	1111	859
		10,000	24862	258054	151337	317804	18710	2595	2745
rand2	100K	100	20492	18310	13423	27287	3146	271	201
		1,000	1844	19903	15212	31057	2132	892	573
		10,000	2577	27004	14445	33077	1914	1030	1213
	500K	100	44415	28946	94090	187855	7195	291	214
		1,000	15861	130893	94076	190376	31954	1307	824
		10,000	15713	114530	102129	217412	7398	2312	2652
	1M	100	42520	28828	313077	625333	11095	330	170
		1,000	15137	189764	151862	303610	34972	1146	1072
		10,000	35855	232912	271014	549434	29618	2767	2459

m = the number of operations performed
n = the number of elements in initial data structures
Table 13: Standard deviation of the number of key comparisons

inputs	m	n	Heap	BQueue	LT	WBLT	Pair	Skew
		100	0.0,0.0	0.0,0.0	0.92,0.0	1.11,0.0	23.30,0.0	1.34,0.0
	100K	1,000	0.0,0.0	0.0,0.0	0.0,1.20	0.0,1.50	0.0,3.00	0.0,3.00
		10,000	0.0,0.0	0.0,0.0	0.0,4.02	0.0,4.07	0.0,10.71	0.0,3.92
		100	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0
rand1	500K	1,000	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0
		10,000	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0
		100	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0
	1M	1,000	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0
		10,000	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0
		100	0.0,0.0	0.0,0.0	1.11,0.0	0.78,0.0	30.56,0.0	2.79,0.0
	100K	1,000	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0
		10,000	0.0,0.0	0.0,0.0	0.0,1.50	0.0,2.40	0.0,4.50	0.0,2.40
		100	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0
rand2	500K	1,000	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0
		10,000	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0
		100	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0
	1M	1,000	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0
		10,000	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0	0.0,0.0

m = the number of operations performed
n = the number of elements in initial data structures
Table 14: Standard deviation of height/level of the structures

inputs	m	n	BST	TRP	Skip	MSL	AVL	Splay	WBLT
		100	1.40,1.02	1.84,2.97	0.0,0.64	0.0,0.64	0.0,0.0	1.42,1.67	1.11,0.0
	100K	1,000	0.0,1.11	0.0,1.45	0.0,1.17	0.0,1.17	0.0,0.30	0.0,2.00	0.0,1.50
		10,000	0.0,1.02	0.0,1.68	0.0,0.30	0.0,0.30	0.0,0.0	0.0,1.62	0.0,4.07
		100	0.0,1.67	0.0,1.11	0.0,0.67	0.0,0.67	0.0,0.0	0.0,1.67	0.0,0.0
rand1	500K	1,000	0.0,1.62	0.0,2.33	0.0,0.89	0.0,0.89	0.0,0.0	0.0,2.50	0.0,0.0
		10,000	0.0,0.49	0.0,1.49	0.0,0.40	0.0,0.40	0.0,0.0	0.0,3.75	0.0,0.0
		100	0.0,1.60	0.0,1.37	0.0,0.83	0.0,0.83	0.0,0.0	0.0,2.76	0.0,0.0
	1M	1,000	0.0,1.30	0.0,1.14	0.0,0.94	0.0,0.94	0.0,0.30	0.0,3.69	0.0,0.0
		10,000	0.0,1.27	0.0,1.60	0.0,0.66	0.0,0.66	0.0,0.0	0.0,1.90	0.0,0.0
		100	1.10,5.06	1.11,5.78	0.0,0.67	0.0,0.67	0.0,0.46	2.04,2.96	0.78,0.0
	100K	1,000	0.0,4.59	0.0,3.29	0.0,0.40	0.0,0.40	0.0,0.50	0.0,2.14	0.0,0.0
		10,000	0.0,2.57	0.0,4.72	0.0,0.63	0.0,0.63	0.0,0.0	0.0,3.88	0.0,2.40
		100	0.0,0.0	0.0,0.0	0.0,0.90	0.0,0.90	0.0,0.46	0.0,6.59	0.0,0.0
rand2	500K	1,000	0.0,12.76	0.0,15.34	0.0,0.54	0.0,0.54	0.0,0.0	0.0,11.76	0.0,0.0
		10,000	0.0,12.37	0.0,7.57	0.0,0.81	0.0,0.81	0.0,0.0	0.0,3.74	0.0,0.0
		100	0.0,0.0	0.0,0.0	0.0,0.90	0.0,0.90	0.0,0.30	0.0,10.73	0.0,0.0
	1M	1,000	0.0,16.14	0.0,19.31	0.0,0.66	0.0,0.66	0.0,0.0	0.0,8.46	0.0,0.0
		10,000	0.0,14.73	0.0,17.74	0.0,0.49	0.0,0.49	0.0,0.0	0.0,5.26	0.0,0.0

m = the number of operations performed
n = the number of elements in initial data structures
Table 15: Standard deviation of height/level of the structures

inputs	m	n	Heap	BQueue	LT	WBLT	Pair	Skew
rand1	100K	100	0.008	0.011	0.008	0.010	0.007	0.010
		1,000	0.008	0.005	0.005	0.008	0.007	0.008
		10,000	0.011	0.008	0.007	0.008	0.008	0.008
	200K	100	0.011	0.017	0.008	0.011	0.017	0.036
		1,000	0.007	0.049	0.007	0.007	0.005	0.010
		10,000	0.000	0.014	0.009	0.008	0.008	0.008
	1M	100	0.008	0.025	0.012	0.019	0.031	0.046
		1,000	0.008	0.031	0.012	0.013	0.005	0.017
		10,000	0.009	0.045	0.014	0.007	0.000	0.008
rand2	100K	100	0.007	0.007	0.011	0.007	0.011	0.007
		1,000	0.008	0.009	0.011	0.008	0.007	0.008
		10,000	0.007	0.008	0.012	0.008	0.007	0.008
	200K	100	0.008	0.064	0.012	0.008	0.020	0.039
		1,000	0.011	0.034	0.008	0.013	0.009	0.011
		10,000	0.000	0.022	0.009	0.008	0.008	0.012
	1M	100	0.012	0.130	0.020	0.042	0.026	0.076
		1,000	0.008	0.057	0.011	0.008	0.013	0.017
		10,000	0.011	0.049	0.012	0.011	0.011	0.013

Time Unit : *sec*
m = the number of operations performed
n = the number of elements in initial data structures
Table 16: Standard deviation of run time using integer keys

inputs	m	n	BST	TRP	Skip	MSL	AVL	Splay	WBLT
rand1	100K	100	0.008	0.011	0.008	0.017	0.008	0.008	0.010
		1,000	0.008	0.010	0.009	0.013	0.005	0.009	0.008
		10,000	0.008	0.015	0.007	0.017	0.008	0.008	0.008
	200K	100	0.030	0.052	0.024	0.035	0.033	0.015	0.011
		1,000	0.018	0.015	0.024	0.057	0.027	0.008	0.007
		10,000	0.012	0.057	0.018	0.025	0.024	0.010	0.008
	1M	100	0.038	0.053	0.062	0.034	0.064	0.037	0.019
		1,000	0.039	0.055	0.052	0.089	0.040	0.012	0.013
		10,000	0.042	0.140	0.022	0.102	0.065	0.009	0.007
rand2	100K	100	0.009	0.011	0.007	0.015	0.007	0.008	0.007
		1,000	0.000	0.008	0.007	0.007	0.007	0.008	0.008
		10,000	0.008	0.011	0.011	0.007	0.007	0.007	0.008
	200K	100	0.012	0.013	0.018	0.045	0.026	0.020	0.008
		1,000	0.011	0.042	0.019	0.037	0.045	0.008	0.013
		10,000	0.013	0.059	0.023	0.046	0.022	0.005	0.008
	1M	100	0.023	0.028	0.092	0.093	0.056	0.032	0.042
		1,000	0.015	0.041	0.019	0.041	0.050	0.008	0.008
		10,000	0.024	0.088	0.068	0.089	0.085	0.008	0.011

Time Unit : *sec*
m = the number of operations performed
n = the number of elements in initial data structures
Table 17: Standard deviation of run time using integer keys

inputs	m	n	Heap	BQueue	LT	WBLT	Pair	Skew
rand1	100K	100	0.008	0.008	0.008	0.000	0.012	0.008
		1,000	0.008	0.010	0.007	0.008	0.005	0.008
		10,000	0.011	0.012	0.008	0.008	0.008	0.005
	200K	100	0.012	0.016	0.014	0.013	0.015	0.022
		1,000	0.013	0.034	0.007	0.009	0.005	0.011
		10,000	0.005	0.015	0.011	0.015	0.008	0.012
	1M	100	0.013	0.059	0.021	0.017	0.036	0.038
		1,000	0.010	0.038	0.014	0.015	0.017	0.015
		10,000	0.007	0.074	0.019	0.005	0.008	0.013
rand2	100K	100	0.007	0.011	0.008	0.005	0.008	0.009
		1,000	0.010	0.008	0.008	0.008	0.007	0.008
		10,000	0.008	0.021	0.008	0.008	0.008	0.011
	200K	100	0.008	0.020	0.011	0.011	0.017	0.027
		1,000	0.012	0.050	0.011	0.011	0.008	0.011
		10,000	0.008	0.018	0.011	0.010	0.010	0.011
	1M	100	0.012	0.126	0.021	0.013	0.032	0.083
		1,000	0.017	0.089	0.011	0.008	0.015	0.018
		10,000	0.010	0.027	0.011	0.012	0.013	0.023

Time Unit : *sec*
m = the number of operations performed
n = the number of elements in initial data structures
Table 18: Standard deviation of run time using real keys

inputs	m	n	BST	TRP	Skip	MSL	AVL	Splay	WBLT
rand1	100K	100	0.016	0.017	0.011	0.013	0.008	0.008	0.000
		1,000	0.008	0.012	0.013	0.012	0.008	0.008	0.008
		10,000	0.008	0.011	0.014	0.008	0.011	0.010	0.008
	200K	100	0.047	0.060	0.030	0.043	0.043	0.015	0.013
		1,000	0.017	0.037	0.040	0.077	0.018	0.008	0.009
		10,000	0.008	0.030	0.015	0.045	0.031	0.008	0.015
	1M	100	0.115	0.073	0.046	0.114	0.094	0.053	0.017
		1,000	0.072	0.092	0.067	0.136	0.033	0.032	0.015
		10,000	0.035	0.076	0.045	0.092	0.035	0.018	0.005
rand2	100K	100	0.013	0.008	0.010	0.011	0.008	0.008	0.005
		1,000	0.007	0.008	0.008	0.008	0.008	0.011	0.008
		10,000	0.009	0.012	0.011	0.013	0.008	0.008	0.008
	200K	100	0.014	0.019	0.020	0.060	0.031	0.028	0.011
		1,000	0.011	0.043	0.026	0.046	0.026	0.013	0.011
		10,000	0.020	0.041	0.025	0.049	0.025	0.010	0.010
	1M	100	0.022	0.020	0.119	0.189	0.053	0.053	0.013
		1,000	0.014	0.060	0.033	0.166	0.030	0.029	0.008
		10,000	0.023	0.077	0.064	0.111	0.040	0.024	0.012

Time Unit : *sec*
m = the number of operations performed
n = the number of elements in initial data structures
Table 19: Standard deviation of run time using real keys

REFERENCES

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.

M. Atkinson, J. Sack, N. Santoro, and T. Strothotte, Min-max Heaps and Generalized Priority Queues, Communications of the ACM, vol. 29, no. 10, pp. 996-1000, 1986.

M. Brown, Implementation and analysis of binomial queue algorithms, SIAM Jr. on Computing, 7, 3, 1978, 298-319.

S. Carlsson, The Deap: a Double-Ended Heap to Implement Double-Ended Priority Queues, Information processing letters, vol. 26, pp.33-36, 1987.

C. Crane, Linear Lists and Priority Queues as Balanced Binary Trees, Tech. Rep. CS-72-259, Dept. of Comp. Sci., Stanford University, 1972.

M. Fredman, R. Sedgewick, D. Sleator, and R.Tarjan, The pairing heap: A new form of self-adjusting heap. Algorithmica 1, pp. 111-129, 1986.

M. Fredman and R. Tarjan, Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms, JACM, vol. 34, no. 3, pp. 596-615, 1987.

G. H. Gonnet and R. Baeza-Yates, Handbook of Algorithms and Data Structures, 2nd Edition, Md.: Addison-Wesley Publishing Company, 1991.

E. Horowitz and S. Sahni, Fundatamentals of Data Structures in Pascal, 4th Edition, New York: W. H. Freeman and Company, 1994.

D. Jones, An empirical comparison of priority-queue and event-set implementations, Communications of the ACM, 29, 4, pp. 300-311, 1986.

W. Pugh, Skip Lists: a Probabilistic Alternative to Balanced Trees, Communications of the ACM, vol. 33, no. 6, pp.668-676, 1990.

S. Sahni, Software Development in Pascal, Florida: NSPAN Printing and Publishing Co., 1993.

D. Sleator and R. Tarjan, Self-adjusting binary search trees, JACM, 32, 3, pp. 652-686, 1985.

D. Sleator and R. Tarjan, Self-adjusting heaps, SIAM Jr. on Computing, 15, 1, pp. 52-69, 1986.

J. Stasko and J. Vitter, Pairing heaps: Experiments and analysis, Communications of the ACM, 30, 3, pp. 234-249, 1987.