Biased Leftist Trees and Modified Skip Lists[1]

Seonghun Cho and Sartaj Sahni

Department of Computer and Information Science and Engineering

University of Florida

Gainesville, FL 32611, U.S.A.

Technical Report 96-002

Abstract

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 as well as experimental results for double ended priority queues are presented.

Keywords and Phrases. leftist trees, skip lists, dictionary, priority queue, double ended priority queue

1 Introduction

Several data structures (e.g., heaps, leftist trees [CRAN72], binomial heaps [FRED87]) have been proposed for the representation of a (single ended) priority queue. Heaps permit one to delete the min element and insert an arbitrary element into an n element priority queue in $O(\log n)$ time. Leftist trees support both these operations and the merging of pairs of priority queues in logarithmic time. Using binomial heaps, inserts and combines take $O(1)$ time and a delete-min takes $O(\log n)$ amortized time. In this paper, we begin in Section 2, by developing the weight biased leftist tree. 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.

[1]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.

Experimental results presented in Section 5 show that weight biased leftist trees provide better performance than provided by leftist trees. The experimental comparisons of Section 5 also include a comparison with heaps and binomial heaps as well as 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. For completeness, we include, in Section 5, a comparison of data structures for double ended priority queues.

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.

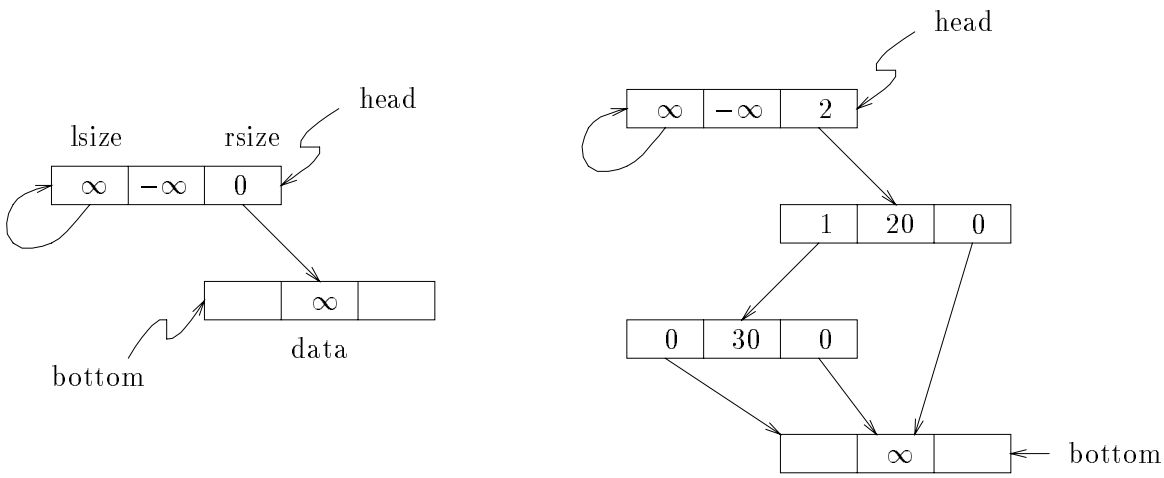
Theorem 1 *Let x be any internal node of a weight biased leftist tree. $rightmost(x) \leq \log_2(w(x) + 1)$.*

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 $rightmost(x) = 1 + rightmost(RightChild(x)) \leq 1 + \log_2((n-1)/2+1) = 1 + \log_2(n+1) - 1 = \log_2(n + 1)$. □

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 combine operation (i.e., join two priority queues together) can also be done in logarithmic time. The algorithms for these operations



(a) Empty min-WBLT

(b) Nonempty min-WBLT

Figure 1: Example min-WBLTs

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 combine 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 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

```
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 ;
```

Figure 2: min-WBLT Insert

```
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 ;
```

Figure 3: min-WBLT Delete-min

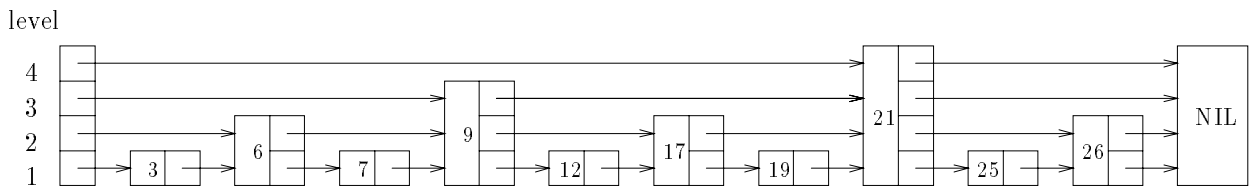


Figure 4: Skip Lists

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 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

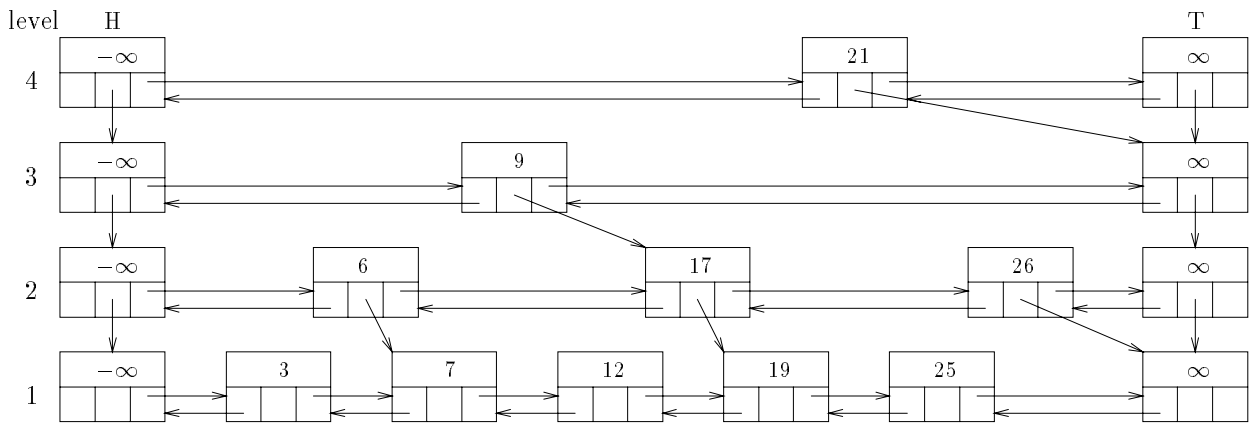


Figure 5: Modified Skip Lists

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.

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

```
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 ;
```

Figure 6: MSL Search

```
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 ≠ 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 ;
```

Figure 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 ;
```

Figure 8: MSL Delete

any level other than $lcurrent$, we expect to examine upto $c = 1/p - 1$ additional nodes on the next lower level. Hence, upto $c(lcurrent - 2)$ 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 $O(\log n)$. □

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 to work best for each structure. $lmax$ was set to 16 for both structures.

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 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 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 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 and Figure 9). 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.

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

n	operation	random inputs		ordered inputs	
		SKIP	MSL	SKIP	MSL
10,000	insert	224757	322499	247129	318854
	search	255072	362865	256706	339019
	ins/del	519430	734161	354566	560219
	search	256124	349591	250538	339121
	delete	231745	320594	84392	185489
50,000	insert	1357076	1950583	1422120	1911818
	search	1537547	1965649	1467217	1836713
	ins/del	2996512	4142186	1973416	3204400
	search	1501731	2038774	1449810	1989550
	delete	1373858	1853671	486498	931975
100,000	insert	2919371	4146428	2925618	4275880
	search	3188621	4315576	2970715	4082193
	ins/del	6399463	9103135	4406427	6895510
	search	3225343	4427979	3277089	4345874
	delete	2981173	4161994	961283	2052638
200,000	insert	6178596	8927523	6403207	9022631
	search	6697223	9273707	6448304	8946474
	ins/del	13377747	19370831	9054078	9062233
	search	6680642	9662006	6458321	9197714
	delete	6149268	9101721	1995215	4837867

Table 1: The number of key comparisons

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: Number of levels

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: Run time

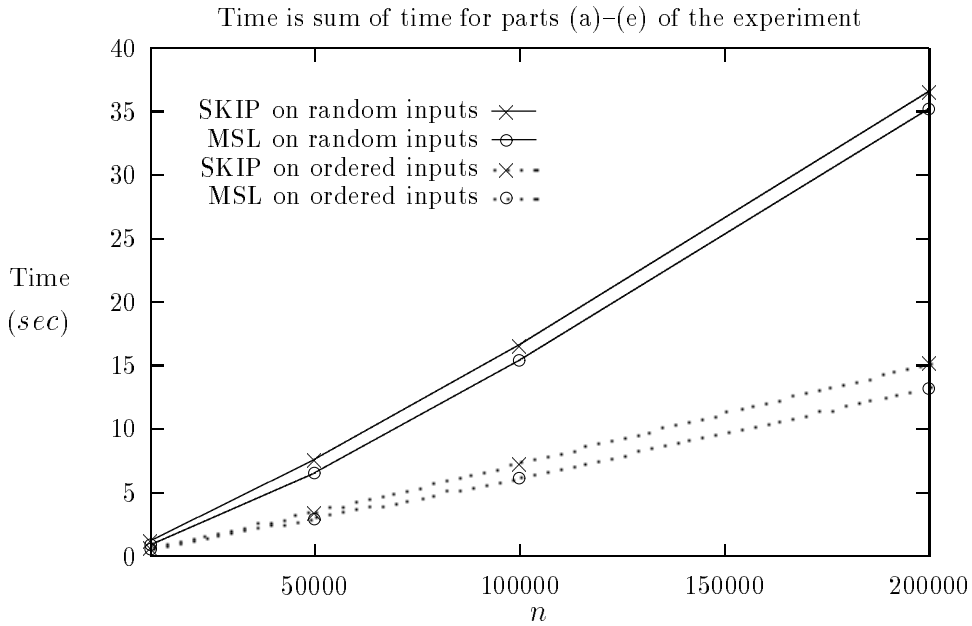


Figure 9: Run time

time. In the case of MSLs, the min element is the 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 10 and 11, respectively. The last step of Figure 10 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 a TMSL is O($\log n$) and O(1), respectively.*

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

```
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 ;
```

Figure 10: 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 ;
```

Figure 11: 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 ;
```

Figure 12: TMSL Delete-max

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 heap (B-Heap), leftist trees (LT), weight biased leftist trees (WBLT), and TMSLs were programmed in C. In addition, priority queue versions of unbalanced binary search trees (BST), AVL trees, 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. 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, 100,000, and 100,000 elements and then performed a random sequence of 100,000 operations. This

random sequence consists of approximately 50% insert and 50% delete min operations. The results are given in Tables 4, 5, and 6. 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 BST 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.

Table 4 gives the total number of comparisons made by each of the methods. On the two random data tests, weight biased leftist trees required the fewest number of comparisons except when $n = 100,000$. In this case, AVL trees required the fewest. With ascending data, treaps did best and with descending data, LTs and WBLTs did best. For both, each insert could be done with one comparison as both structures build a left skewed tree.

The structure height initially and following the 100,000 operations is given in Table 5 for BSTs, Heaps, TRPs and AVL trees. For B-Heaps, the height of the tallest tree is given. For SKIPs and TMSLs, this table gives the number of levels. In the case of LT and WBLT, this table gives the length of the rightmost path following initialization and the average of its length following each of the 100,000 operations. 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 Table 6. For this, the codes were compiled using the cc compiler in optimized mode. The run time for the data set random1 is graphed in Figure 13. The run time for the data set random2 and Heap, LT, WBLT, SKIP, TMSL, and AVL is graphed in Figure 14. For the data sets random1 and random2 with $n = 100$ and 1,000, WBLTs required least time. For random1 with $n = 10,000$, BSTs took least time while when $n = 100,000$, both BSTs and Heaps took least time. For random2 with

inputs	n	BST	Heap	B-Heap	LT	WBLT	TRP	SKIP	TMSL	AVL
random1	100	621307	823685	268224	165771	165525	407326	270584	503567	373683
	1,000	677570	1317728	383285	203550	202274	537418	397181	729274	542789
	10,000	726875	1693955	645664	476534	468713	757946	711869	1104805	685422
	100,000	1067670	1746376	1153516	1199207	1171181	1119554	1327083	1778579	877131
random2	100	522728	781808	260209	164288	164031	384721	261169	485995	358885
	1,000	612630	1273828	389333	199886	198720	518022	386837	707997	534406
	10,000	1027346	1576921	642968	448379	439014	768783	710612	1131421	685926
	100,000	5641638	1713676	1146214	1032043	978785	1746732	1339410	1800404	890190
increasing	100	–	564332	410119	552081	536796	363045	629085	836223	382513
	1,000	–	946659	655107	917664	882490	496223	1018473	1331317	584422
	10,000	–	1284712	814412	1234622	1192325	645765	1313298	1685844	747581
	100,000	–	1617645	923048	1550741	1560866	723825	1568819	1939835	902437
decreasing	100	–	836361	202402	50010	50010	362723	194245	425238	334965
	1,000	–	1394286	313587	50010	50010	515579	300298	637341	512942
	10,000	–	1950062	413840	50010	50010	558222	400082	836879	672910
	100,000	–	2400032	534821	50010	50010	648730	450090	936986	835044

n = the number of elements in initial data structures
Total number of operations performed = 100,000

Table 4: The number of key comparisons

inputs	n	BST	Heap	B-Heap	LT	WBLT	TRP	SKIP	TMSL	AVL
random1	100	13,13	7,6	1,6	4,2	4,2	12,11	4,4	4,4	8,7
	1,000	22,22	10,10	1,10	7,2	7,2	22,24	6,6	6,6	12,12
	10,000	31,32	14,14	1,14	8,4	8,4	33,30	8,7	8,7	16,16
	100,000	40,41	17,17	1,17	10,9	10,9	42,42	9,9	9,9	20,20
random2	100	13,16	7,7	1,7	4,2	4,2	14,17	4,4	4,4	8,7
	1,000	23,69	10,10	1,10	5,2	5,2	23,63	6,5	6,5	12,11
	10,000	39,93	14,14	1,14	6,4	6,4	36,83	8,7	8,7	16,15
	100,000	147,201	17,17	1,17	6,8	6,8	133,183	9,9	9,9	19,19
increasing	100	–	7,8	1,8	6,4	6,4	11,15	4,5	4,5	7,8
	1,000	–	10,11	1,11	9,7	9,7	24,24	6,6	6,6	10,11
	10,000	–	14,14	1,14	13,9	13,9	33,34	8,8	8,8	14,14
	100,000	–	17,17	1,17	16,11	16,11	46,41	9,9	9,9	17,17
decreasing	100	–	7,7	1,7	1,1	1,1	11,15	4,4	4,4	7,7
	1,000	–	10,10	1,10	1,1	1,1	24,24	6,6	6,6	10,10
	10,000	–	14,14	1,14	1,1	1,1	33,33	8,8	8,8	14,14
	100,000	–	17,17	1,17	1,1	1,1	46,46	9,9	9,9	17,17

n = the number of elements in initial data structures
Total number of operations performed = 100,000

Table 5: Height/level of the structures

inputs	n	BST	Heap	B-Heap	LT	WBLT	TRP	SKIP	TMSL	AVL
random1	100	0.32	0.32	0.53	0.30	0.23	0.56	0.36	0.37	0.50
	1,000	0.34	0.44	0.59	0.29	0.25	0.56	0.35	0.39	0.56
	10,000	0.38	0.57	0.98	0.62	0.49	0.71	0.62	0.59	0.70
	100,000	0.66	0.66	1.90	1.77	1.26	1.26	1.33	1.32	0.96
random2	100	0.29	0.30	0.55	0.27	0.25	0.55	0.32	0.35	0.50
	1,000	0.32	0.44	0.58	0.27	0.23	0.53	0.32	0.36	0.53
	10,000	0.49	0.51	0.93	0.57	0.44	0.70	0.59	0.57	0.67
	100,000	3.83	0.68	1.92	1.44	0.99	1.70	1.32	1.34	1.02
increasing	100	–	0.22	0.63	0.50	0.40	0.42	0.43	0.42	0.62
	1,000	–	0.35	0.92	0.95	0.70	0.48	0.78	0.58	0.58
	10,000	–	0.47	1.05	1.25	0.95	0.55	0.72	0.60	0.63
	100,000	–	0.60	1.30	1.83	1.40	0.60	0.83	0.65	0.78
decreasing	100	–	0.30	0.38	0.13	0.12	0.45	0.23	0.28	0.40
	1,000	–	0.45	0.47	0.13	0.10	0.50	0.28	0.33	0.48
	10,000	–	0.58	0.55	0.12	0.12	0.52	0.32	0.38	0.67
	100,000	–	0.70	0.73	0.12	0.12	0.55	0.35	0.45	0.80

Time Unit : *sec*
n = the number of elements in initial data structures
Total number of operations performed = 100,000

Table 6: Run time using integer keys

$n = 10,000$, WBLTs were fastest while for $n = 100,000$, Heap was best. On the ordered data sets, BSTs have a very high complexity and are the poorest performers (times not shown in Table 6). For increasing data, Heap was best for n = 100, 1,000 and 10,000 and both Heap and TRP best for $n = 100,000$. For decreasing data, WBLTs were generally best. On all data sets, WBLTs always did at least as well (and often better) as LTs. Between SKIP and TMSL, we see that SKIP generally did better for small n and TMSL for large n.

Another way to interpret the time results is in terms of the ratio m/n (m = number of operations). In the experiments reported in Table 6, $m = 100,000$. As m/n increases, WBLTs and LTs perform better relative to the remaining structures. This is because as m increases, the (weight biased) leftist trees constructed are very highly skewed to the left and the length of the rightmost path is close to one.

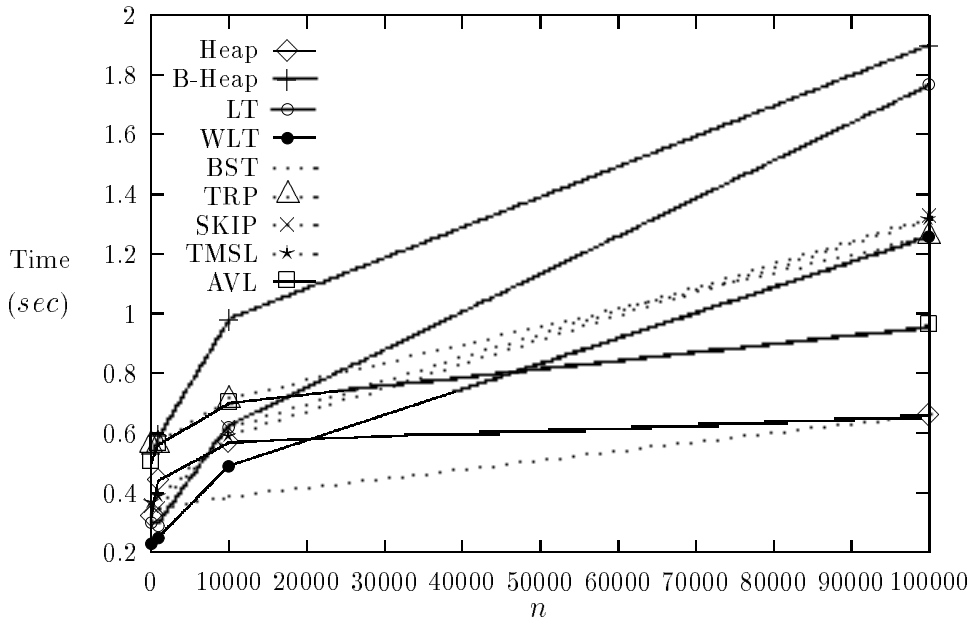


Figure 13: Run time on random1

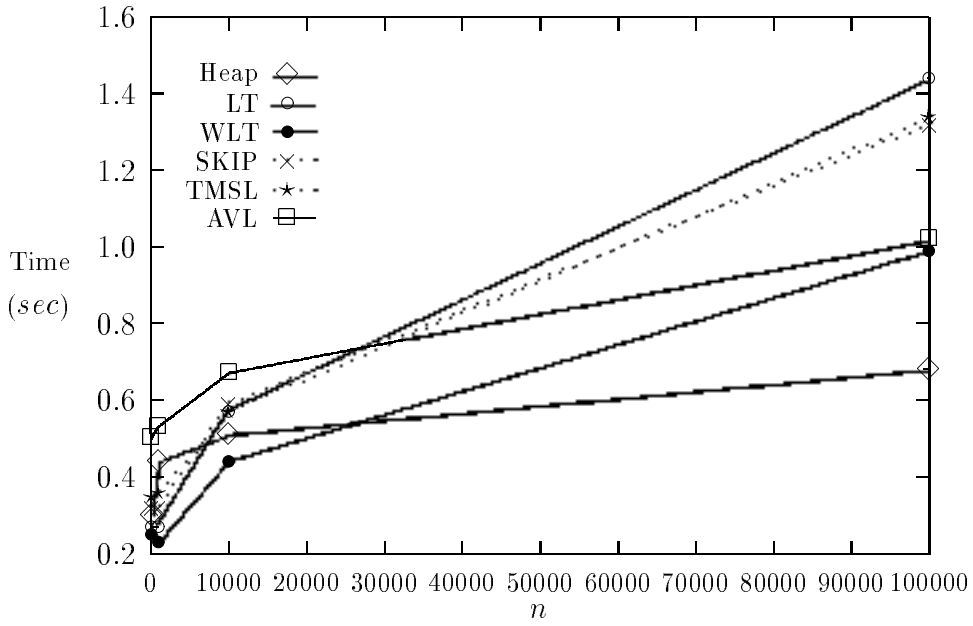


Figure 14: Run time on random2

inputs	n	BST	MMH	Deap	TRP	SKIP	TMSL	AVL
random1	100	534197	994374	581071	402987	462690	666845	371435
	1,000	676634	1677964	912282	550363	698132	996774	545600
	10,000	738100	2328247	1122599	759755	1034669	1437925	693935
	100,000	1068250	2795369	1123423	1127616	1439387	1862864	878709
random2	100	514680	941925	557549	396910	447909	650533	357556
	1,000	908437	1651830	909747	564414	658284	949230	537616
	10,000	1339444	2239519	1145724	881889	1003300	1379690	689415
	100,000	5956407	2754760	1125210	1873922	1454138	1875706	891332
increasing	100	–	926017	592503	364026	624430	803024	363399
	1,000	–	1660945	1052894	507812	999886	1304223	562873
	10,000	–	2392627	1411640	614534	1373332	1766792	726087
	100,000	–	3015425	1849318	679120	1487465	1866498	878978
decreasing	100	–	926041	615944	360030	193021	413284	355999
	1,000	–	1711076	1062010	490022	287887	603089	563085
	10,000	–	2400014	1480876	676190	352780	732673	725079
	100,000	–	3035292	1851845	698740	450090	927278	878244

n = the number of elements in initial data structures
Total number of operations performed = 100,000

Table 7: The number of key comparisons

Tables 7, 8, and 9 provide an experimental comparison of BSTs, AVL trees, MMHs (min-max heaps) [ATKI86], Deaps [CARL87], TRPs, SKIPs, and TMSLs as a data structure for double ended priority queues. The experimental setup is similar to that used for single ended priority queues. However, this time the operation mix was 50% insert, 25% delete-min, and 25% delete-max. On the comparison measure, treaps did best on increasing data (except when $n = 100$) and skip lists did best when decreasing data was used. On all other data, AVL trees did best. As far as run time is concerned, BSTs did best on the random data tests except when $n = 100,000$ and the set random2 was used. In this case, deaps and AVL trees took least time. For increasing data, treaps were best and for decreasing data, skip lists were best. The run time for the data set random1 is graphed in Figure 15. The run time for the data set random2 and MMH, Deap, SKIP, TMSL, and AVL is graphed in Figure 16.

inputs	n	BST	MMH	Deap	TRP	SKIP	TMSL	AVL
random1	100	13,12	7,6	7,6	13,11	4,4	4,4	8,7
	1,000	22,22	10,10	10,10	23,22	6,6	6,6	12,12
	10,000	32,31	14,14	14,14	33,32	8,8	8,8	16,16
	100,000	41,41	17,17	17,17	41,42	9,9	9,9	20,20
random2	100	13,13	7,7	7,7	14,12	4,4	4,4	8,7
	1,000	23,69	10,10	10,10	22,60	6,6	6,6	12,11
	10,000	38,93	14,14	14,14	35,82	8,7	8,7	16,15
	100,000	147,199	17,17	17,17	135,186	9,9	9,9	19,19
increasing	100	–	7,8	7,8	11,16	4,5	4,5	7,8
	1,000	–	10,11	10,11	24,27	6,7	6,7	10,11
	10,000	–	14,14	14,14	33,33	8,8	8,8	14,14
	100,000	–	17,17	17,17	46,43	9,9	9,9	17,17
decreasing	100	–	7,7	7,7	11,15	4,5	4,5	7,8
	1,000	–	10,10	10,10	24,21	6,7	6,7	10,11
	10,000	–	14,14	14,14	33,36	8,8	8,8	14,14
	100,000	–	17,17	17,17	46,43	9,9	9,9	17,17

n = the number of elements in initial data structures
Total number of operations performed = 100,000

Table 8: Height/level of the structures

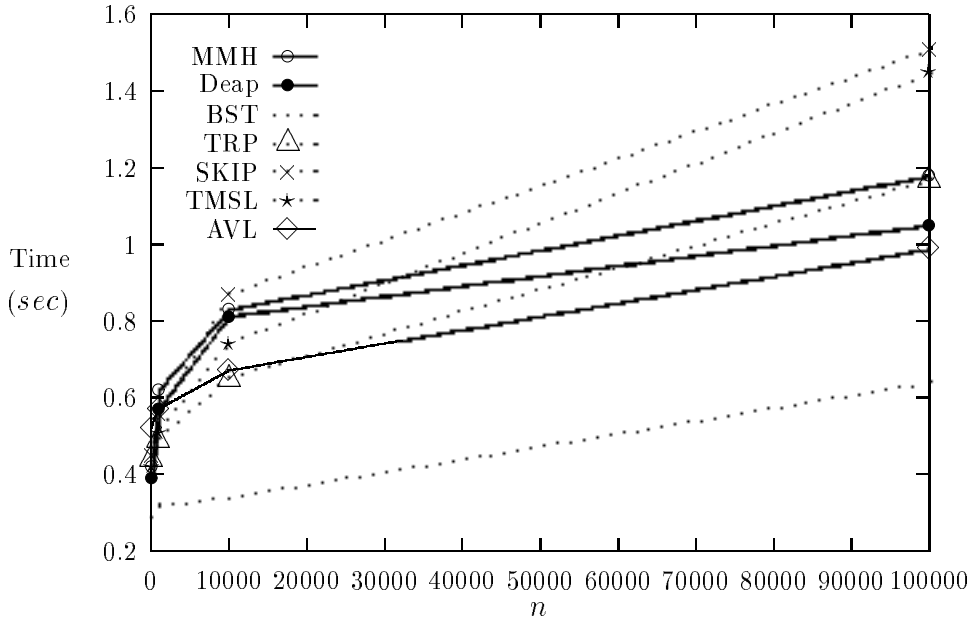


Figure 15: Run time on random1

inputs	n	BST	MMH	Deap	TRP	SKIP	TMSL	AVL
random1	100	0.29	0.42	0.39	0.44	0.45	0.42	0.52
	1,000	0.32	0.62	0.57	0.49	0.56	0.51	0.57
	10,000	0.34	0.83	0.81	0.65	0.87	0.74	0.67
	100,000	0.64	1.18	1.05	1.17	1.51	1.45	0.99
random2	100	0.27	0.42	0.39	0.47	0.51	0.46	0.56
	1,000	0.47	0.64	0.59	0.54	0.53	0.48	0.55
	10,000	0.59	0.85	0.78	0.72	0.89	0.71	0.69
	100,000	4.22	1.07	1.01	1.91	1.50	1.47	1.01
increasing	100	–	0.38	0.38	0.35	0.48	0.38	0.60
	1,000	–	0.60	0.63	0.42	0.65	0.53	0.53
	10,000	–	0.88	0.82	0.47	0.80	0.63	0.62
	100,000	–	1.12	1.15	0.57	0.92	0.85	0.77
decreasing	100	–	0.37	0.40	0.35	0.35	0.42	0.53
	1,000	–	0.63	0.62	0.43	0.33	0.38	0.55
	10,000	–	0.83	0.83	0.50	0.35	0.40	0.62
	100,000	–	1.05	1.10	0.63	0.40	0.45	0.83

Time Unit : *sec*
n = the number of elements in initial data structures
Total number of operations performed = 100,000

Table 9: Run time using integer keys

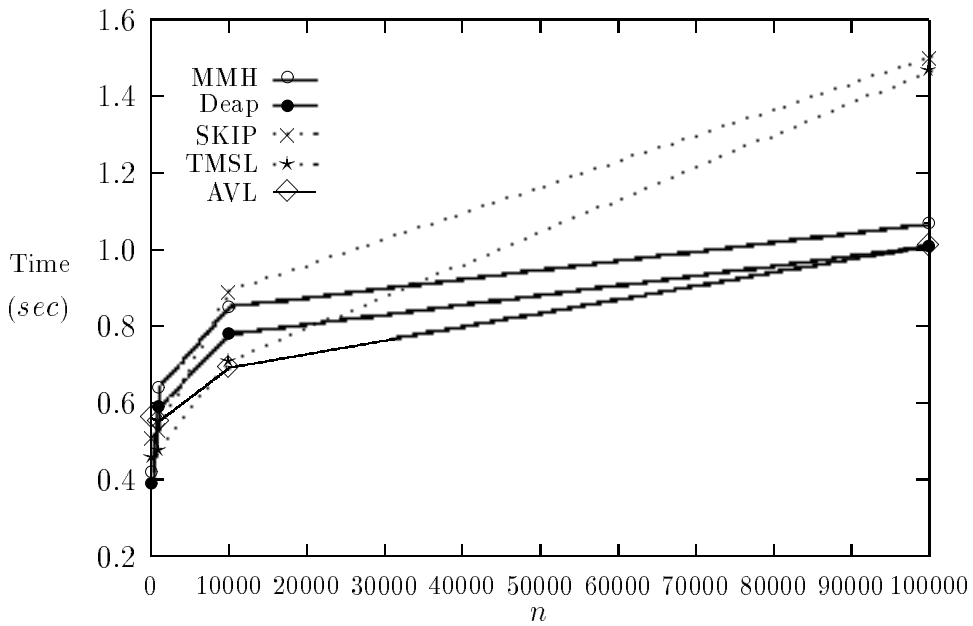


Figure 16: Run time on random2

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. MSLs have the added advantage of using fixed size nodes.

Our experiments show that binary search trees (modified to handle equal keys) perform best of the tested double ended priority queue structures using random data. Of course, these are unsuitable for general application as they have very poor performance on ordered data. Min-max heaps, deaps and AVL trees guarantee $O(\log n)$ behavior per operation. Of these three, AVL trees generally do best for large n. It is possible that other balanced search structures such as bottom-up red-black trees might do even better. Treaps and skip lists are randomized structures with $O(\log n)$ expected complexity. Treaps were generally faster than skip lists (except for decreasing data) as double ended priority queues.

For single ended priority queues, if we exclude BSTs because of their very poor performance on ordered data, WBLTs did best on the data sets random1 and random2 (except when $n = 100,000$), and decreasing. Heaps did best on the remaining data sets. The probabilistic structures TRP, SKIP and TMSL were generally slower than WBLTs. When the ratio m/n (m = number of operations, n = average queue size) is large, WBLTs (and LTs) outperform heaps (and all other tested structures) as the binary trees constructed tend to be highly skewed to the left and the length of the rightmost path is close to one.

Our experimental results for single ended priority queues are in marked contrast to those reported in [GONN91, p228] 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.

References

[ARAG89] 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.

[ATKI86] 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.

[CARL87] S. Carlsson, The Deap: a Double-Ended Heap to Implement Double-Ended Priority Queues, Information processing letters, vol. 26, pp.33-36, 1987.

[CRAN72] C. Crane, Linear Lists and Priority Queues as Balanced Binary Trees, Tech. Rep. CS-72-259, Dept. of Comp. Sci., Stanford University, 1972.

[FRED87] M. Fredman and R. Tarjan, Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms, JACM, vol. 34, no. 3, pp. 596-615, 1987.

[GONN91] G. H. Gonnet and R. Baeza-Yates, Handbook of Algorithms and Data Structures, 2nd Edition, Md.: Addison-Wesley Publishing Company, 1991.

[HORO94] E. Horowitz and S. Sahni, Fundatamentals of Data Structures in Pascal, 4th Edition, New York: W. H. Freeman and Company, 1994.

[PUGH90] W. Pugh, Skip Lists: a Probabilistic Alternative to Balanced Trees, Communications of the ACM, vol. 33, no. 6, pp.668-676, 1990.

[SAHN93] S. Sahni, Software Development in Pascal, Florida: NSPAN Printing and Publishing Co., 1993.