Weight Biased Leftist Trees and Modified Skip Lists*

Seonghun Cho and Sartaj Sahni

University of Florida, Gainesville, FL 32611, USA

Abstract. We propose the weight biased leftist tree as an alternative to traditional leftist trees [2] for the representation of mergeable priority queues. A modified version of skip lists [5] 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.

1 Introduction

Several data structures (e.g., heaps, leftist trees [2], binomial heaps [3]) 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. Experimental results show that weight biased leftist trees provide better performance than provided by leftist trees. The experimental comparisons also include a comparison with heaps and binomial heaps as well as with unbalanced binary search trees and the probabilistic structures treap [1] and skip lists [5].

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.

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

* 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.

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 1 [2]. 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 2. 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 [2] 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 3. *Let x be any internal node of a weight biased leftist tree. Then, $rightmost(x) \leq \log_2(w(x) + 1)$.*

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 4. 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*. 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 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 1 and 2, 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 $O(rightmost(T))$.

```
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. 1. min-WBLT Insert

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.

The single-ended priority queue structures min heap (Heap), binomial heap (B-Heap), leftist trees (LT) and weight biased leftist trees (WBLT) 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, 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

```
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. 2. min-WBLT Delete-min

(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 Table 1. 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.

The measured run times on a Sun Sparc 5 are given in Table 1. For this, the codes were compiled using the cc compiler in optimized mode. 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 $n = 10,000$, 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 1). 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.

Table 1. Run time using integer keys

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

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

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 1, $m = 100,000$. 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.

3 Modified Skip Lists

Skip lists were proposed in Pugh [5] 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 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 3 shows an ideal situation for the case $lcurrent = 4$ and $p = 1/2$.

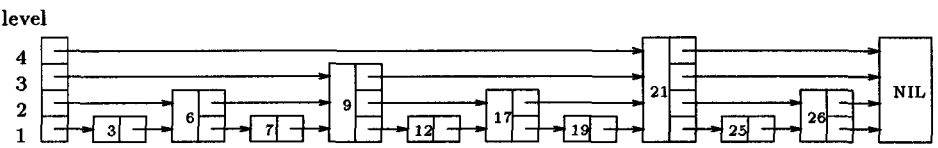

Fig. 3. Skip Lists

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 [6] 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 4 shows the modified skip list that corresponds to the skip list of Figure 3. 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.

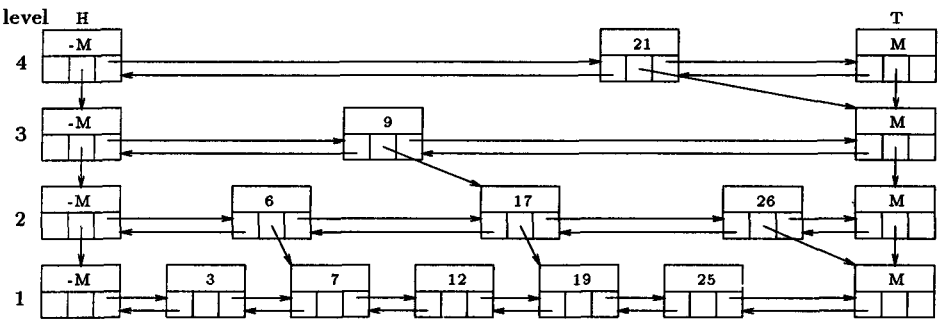

Fig. 4. Modified Skip Lists

A high level description of the algorithms to search, insert, and delete are given in Figure 5. The next theorem shows that their probabilistic complexity is $O(\log n)$ where n is the total number of elements in the dictionary.

Theorem 5. *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 Pugh [5]. Since the algorithms of Pugh [5] 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 $c(lcurrent - 2)$ 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 $O(\log n)$. □

```
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 ;

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 ;

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. 5. MSL Operations

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.

Despite the large disparity in number of comparisons, MSLs generally required less time than required by SKIPs (see Table 2). 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 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.

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 and SKIP 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 Gonnet and Baeza-Yates [4, p228] where leftist trees are reported to take approximately four times as much time as heaps. We suspect

Table 2. Run time

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

this difference in results is because of different programming techniques (recursion vs. iteration, dynamic vs. static memory allocation, etc.) used in Gonnet and Baeza-Yates [4] for the different structures. In our experiments, all structures were coded using similar programming techniques.

References

1. C. R. Aragon and R. G. Seidel: Randomized Search Trees, Proc. 30th Ann. IEEE Symposium on Foundations of Computer Science, pp. 540-545, October 1989.
2. C. Crane: Linear Lists and Priority Queues as Balanced Binary Trees, Tech. Rep. CS-72-259, Dept. of Comp. Sci., Stanford University, 1972.
3. M. Fredman and R. Tarjan: Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms, JACM, vol. 34, no. 3, pp. 596-615, 1987.
4. G. H. Gonnet and R. Baeza-Yates: Handbook of Algorithms and Data Structures, 2nd Edition, Md.: Addison-Wesley Publishing Company, 1991.
5. W. Pugh: Skip Lists: a Probabilistic Alternative to Balanced Trees, Communications of the ACM, vol. 33, no. 6, pp.668-676, 1990.
6. S. Sahni: Software Development in Pascal, Florida: NSPAN Printing and Publishing Co., 1993.