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.

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.

---

1This 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 $\text{LeftChild}(x)$ and $\text{RightChild}(x)$, 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, $\text{shortest}(x)$, of a shortest path from $x$ to an external node satisfies the recurrence

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

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

$$
\text{shortest}(\text{LeftChild}(x)) \geq \text{shortest}(\text{RightChild}(x))
$$

for every internal node $x$.

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

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

$$
\text{weight}(\text{LeftChild}(x)) \geq \text{weight}(\text{RightChild}(x))
$$

2
for every internal node $x$.

It is known [CRAN72] that the length, $\text{rightmost}(x)$, of the rightmost root to external node path of any subtree, $x$, of a leftist tree satisfies

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

The same is true for weight biased leftist trees.

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

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

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

Each node of a min-WBLT has the fields: $\text{lsize}$ (number of internal nodes in left subtree), $\text{rsize}$, $\text{left}$ (pointer to left subtree), $\text{right}$, and $\text{data}$. 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 $\text{head}$ with $\text{lsize} = \infty$ and $\text{lchild} = \text{head}$. In addition, a bottom node $\text{bottom}$ with $\text{data.key} = \infty$. All pointers that would normally be $\text{nil}$ are replaced by a pointer to $\text{bottom}$. 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)-L.Ts. 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 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 up to $l_{max}$ ordered chains designated as level 1 chain, level 2 chain, etc. If we currently have $l_{current}$ number of chains, then all $n$ elements of the dictionary are in the level 1 chain and for each $l$, $2 \leq l \leq l_{current}$, 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.size = t.size + 1;
  if (t.left.size < t.size) then
    begin swap t's children; t = t.left; end
  else t = t.right;
end;
x.left = t.right; x.right = bottom;
x.left = t.size; x.right = 0;
if (t.left.size = t.size) then {swap children}
begin
  t.right = t.left;
  t.left = x; t.size = x.size + 1;
end
else
begin t.right = x; t.size = t.size + 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 \((r \neq bottom)\) 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
        begin do symmetric operations on a.right; \( r = a.right; \end \)
      end
    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
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 \( l_{current} = 4 \) 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 \( imax \) pointer fields. So, the \( n \) element nodes have a total of \( n \times imax \) pointer fields even though only about \( n/(1-p) \) pointers are necessary. Since \( imax \) 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 4: 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 current 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 \(l.current\) becomes empty then
    delete this and succeeding empty lists until we reach the first non empty list,
    update \(l.current\);
end;

Figure 8: MSL Delete

any level other than \(l.current\), we expect to examine upto \(c = 1/p - 1\) additional nodes on
the next lower level. Hence, upto \(c(l.current - 2)\) 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
\(c(l.current - 2) + l.current\) additional compares during an insert. The number of \(down\)
pointers that need to be changed during an insert or delete is expected to be \(\sum_{i=1}^{\infty} i p^i = \frac{1}{(1-p)^2}\).
Since \(c\) and \(p\) are constants and \(l.max = \log_{1/p} n\), 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. \(l.max\) 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, … 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
### Table 1: The number of key comparisons

<table>
<thead>
<tr>
<th>n</th>
<th>operation</th>
<th>random inputs</th>
<th>ordered inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SKIP</td>
<td>MSL</td>
</tr>
<tr>
<td>10,000</td>
<td>insert</td>
<td>224757</td>
<td>322499</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>255072</td>
<td>36265</td>
</tr>
<tr>
<td></td>
<td>ins/del</td>
<td>519430</td>
<td>734161</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>256124</td>
<td>349591</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>231745</td>
<td>320594</td>
</tr>
<tr>
<td>50,000</td>
<td>insert</td>
<td>1357076</td>
<td>1950583</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>1537547</td>
<td>1965649</td>
</tr>
<tr>
<td></td>
<td>ins/del</td>
<td>2996512</td>
<td>4142186</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>1501731</td>
<td>2038774</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>1373858</td>
<td>1853671</td>
</tr>
<tr>
<td>100,000</td>
<td>insert</td>
<td>2919371</td>
<td>4146428</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>3188621</td>
<td>4315576</td>
</tr>
<tr>
<td></td>
<td>ins/del</td>
<td>6399463</td>
<td>9103135</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>3225343</td>
<td>4427979</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>2981173</td>
<td>4161994</td>
</tr>
<tr>
<td>200,000</td>
<td>insert</td>
<td>6178596</td>
<td>8927523</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>6697223</td>
<td>9273707</td>
</tr>
<tr>
<td></td>
<td>ins/del</td>
<td>13377747</td>
<td>19370831</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>6680642</td>
<td>9662006</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>6149268</td>
<td>9101721</td>
</tr>
</tbody>
</table>

### Table 2: Number of levels

<table>
<thead>
<tr>
<th>n</th>
<th>random inputs</th>
<th>ordered inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SKIP</td>
<td>MSL</td>
</tr>
<tr>
<td>10,000</td>
<td>8,8</td>
<td>7,7</td>
</tr>
<tr>
<td>50,000</td>
<td>9,9</td>
<td>7,7</td>
</tr>
<tr>
<td>100,000</td>
<td>9,9</td>
<td>7,8</td>
</tr>
<tr>
<td>200,000</td>
<td>9,9</td>
<td>8,9</td>
</tr>
<tr>
<td>n</td>
<td>operation</td>
<td>random inputs</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>---------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SKIP</td>
</tr>
<tr>
<td>10,000</td>
<td>insert</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>ins/del</td>
<td>0.45</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>0.16</td>
</tr>
<tr>
<td>50,000</td>
<td>insert</td>
<td>1.36</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>1.25</td>
</tr>
<tr>
<td></td>
<td>ins/del</td>
<td>2.73</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>1.16</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>1.10</td>
</tr>
<tr>
<td>100,000</td>
<td>insert</td>
<td>2.84</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>2.63</td>
</tr>
<tr>
<td></td>
<td>ins/del</td>
<td>6.13</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>2.61</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>2.41</td>
</tr>
<tr>
<td>200,000</td>
<td>insert</td>
<td>6.25</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>5.85</td>
</tr>
<tr>
<td></td>
<td>ins/del</td>
<td>13.29</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>5.81</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>5.35</td>
</tr>
</tbody>
</table>

Table 3: Run time
time. In the case of MSLs, the min element is the first one in one of the \textit{current} 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-decreasing 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 $\text{current} - 1$) and then follow the threaded chain.

\textbf{Theorem 3} The expected complexity of an insert and delete-min operation in a TMSL is $O(\log n)$ and $O(1)$, respectively.

\textbf{Proof} Follows from the notion of a thread, Theorem 2, and [PUGH90]. \qed
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 \( ((k > lcurent) \text{ and } (lcurent \neq lmax)) \) then
  begin
    lcurent = lcurent + 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 \( lcurent - 1 \) 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 \( lcurent \) becomes empty then
  delete this and succeeding empty lists until we reach the first non empty list,
  update \( lcurent \) ;
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 \( p\text{.down} \) for nodes on level \( k + 1 \) as necessary;
    if the list at level \( l\text{current} \) becomes empty then
        delete this and succeeding empty lists until we reach the first non empty list,
        update \( l\text{current} \);
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

17
Table 4: The number of key comparisons

<table>
<thead>
<tr>
<th>BST</th>
<th>Heap</th>
<th>B/-Heap</th>
<th>LT</th>
<th>WBLT</th>
<th>TRP</th>
<th>SKIP</th>
<th>AVL</th>
<th>TMSL</th>
<th>AVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>835041</td>
<td>10169239</td>
<td>6938659</td>
<td>4005082</td>
<td>54235734</td>
<td>300206</td>
<td>1001000</td>
<td>501000</td>
<td>2000312</td>
<td>4138341</td>
</tr>
<tr>
<td>65469321</td>
<td>81263489</td>
<td>3734982</td>
<td>12859802</td>
<td>3218795</td>
<td>300206</td>
<td>1001000</td>
<td>501000</td>
<td>2000312</td>
<td>4138341</td>
</tr>
<tr>
<td>34567959</td>
<td>83569875</td>
<td>12365972</td>
<td>3128795</td>
<td>3218795</td>
<td>12859802</td>
<td>3734982</td>
<td>65469321</td>
<td>81263489</td>
<td>3734982</td>
</tr>
<tr>
<td>83569875</td>
<td>12365972</td>
<td>3128795</td>
<td>3218795</td>
<td>12859802</td>
<td>65469321</td>
<td>3734982</td>
<td>83569875</td>
<td>12365972</td>
<td>3128795</td>
</tr>
</tbody>
</table>

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

- Random
- Increasing
- Decreasing

\( n = \text{the number of elements in initial data structures} \)
Table 2: Height/level of the structures

Total number of operations performed = 100.000

\( n \) = the number of elements in initial data structures

<table>
<thead>
<tr>
<th></th>
<th>AVL</th>
<th>BST</th>
<th>s-heap</th>
<th>i-heap</th>
<th>TMSL</th>
<th>TBP</th>
<th>LT</th>
<th>WBLT</th>
<th>TRP</th>
<th>B/-Heap</th>
<th></th>
<th>inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increasing</td>
<td>1.71</td>
<td>1.71</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
</tr>
<tr>
<td>Decreasing</td>
<td>1.71</td>
<td>1.71</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
</tr>
<tr>
<td>Random</td>
<td>1.71</td>
<td>1.71</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
</tr>
<tr>
<td></td>
<td>1.71</td>
<td>1.71</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
</tr>
</tbody>
</table>
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

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

Another way to interpret the time results is in terms of the ratio \( m/n \) \((m = \text{number of operations})\). 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
Tables 7, 8, and 9 provide an experimental comparison of BSTs, AVL trees, MMHs (min-max heaps) [ATK186], Deaps [CARL187], 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.
<table>
<thead>
<tr>
<th>inputs</th>
<th>$n$</th>
<th>BST</th>
<th>MMH</th>
<th>Deap</th>
<th>TRP</th>
<th>SKIP</th>
<th>TMSL</th>
<th>AVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>random1</td>
<td>100</td>
<td>13,12</td>
<td>7,6</td>
<td>7,6</td>
<td>13,11</td>
<td>4,4</td>
<td>4,4</td>
<td>8,7</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>22,22</td>
<td>10,10</td>
<td>10,10</td>
<td>23,22</td>
<td>6,6</td>
<td>6,6</td>
<td>12,12</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>32,31</td>
<td>14,14</td>
<td>14,14</td>
<td>33,32</td>
<td>8,8</td>
<td>8,8</td>
<td>16,16</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>41,41</td>
<td>17,17</td>
<td>17,17</td>
<td>41,42</td>
<td>9,9</td>
<td>9,9</td>
<td>20,20</td>
</tr>
<tr>
<td>random2</td>
<td>100</td>
<td>13,13</td>
<td>7,7</td>
<td>7,7</td>
<td>14,12</td>
<td>4,4</td>
<td>4,4</td>
<td>8,7</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>23,69</td>
<td>10,10</td>
<td>10,10</td>
<td>22,60</td>
<td>6,6</td>
<td>6,6</td>
<td>12,11</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>38,93</td>
<td>14,14</td>
<td>14,14</td>
<td>35,82</td>
<td>8,7</td>
<td>8,7</td>
<td>16,15</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>147,199</td>
<td>17,17</td>
<td>17,17</td>
<td>135,186</td>
<td>9,9</td>
<td>9,9</td>
<td>19,19</td>
</tr>
<tr>
<td>increasing</td>
<td>100</td>
<td>–</td>
<td>7,8</td>
<td>7,8</td>
<td>11,16</td>
<td>4,5</td>
<td>4,5</td>
<td>7,8</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>–</td>
<td>10,11</td>
<td>10,11</td>
<td>24,27</td>
<td>6,7</td>
<td>6,7</td>
<td>10,11</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>–</td>
<td>14,14</td>
<td>14,14</td>
<td>33,33</td>
<td>8,8</td>
<td>8,8</td>
<td>14,14</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>–</td>
<td>17,17</td>
<td>17,17</td>
<td>46,43</td>
<td>9,9</td>
<td>9,9</td>
<td>17,17</td>
</tr>
<tr>
<td>decreasing</td>
<td>100</td>
<td>–</td>
<td>7,7</td>
<td>7,7</td>
<td>11,15</td>
<td>4,5</td>
<td>4,5</td>
<td>7,8</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>–</td>
<td>10,10</td>
<td>10,10</td>
<td>24,21</td>
<td>6,7</td>
<td>6,7</td>
<td>10,11</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>–</td>
<td>14,14</td>
<td>14,14</td>
<td>33,36</td>
<td>8,8</td>
<td>8,8</td>
<td>14,14</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>–</td>
<td>17,17</td>
<td>17,17</td>
<td>46,43</td>
<td>9,9</td>
<td>9,9</td>
<td>17,17</td>
</tr>
</tbody>
</table>

$n = \text{the number of elements in initial data structures}$

Total number of operations performed $= 100,000$

Table 8: Height/level of the structures

![Figure 15: Run time on random1](image_url)
<table>
<thead>
<tr>
<th>inputs</th>
<th>$n$</th>
<th>BST</th>
<th>MMH</th>
<th>Deap</th>
<th>TRP</th>
<th>SKIP</th>
<th>TMSL</th>
<th>AVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>random1</td>
<td>100</td>
<td>0.29</td>
<td>0.42</td>
<td>0.39</td>
<td>0.44</td>
<td>0.45</td>
<td>0.42</td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>0.32</td>
<td>0.62</td>
<td>0.57</td>
<td>0.49</td>
<td>0.56</td>
<td>0.51</td>
<td>0.57</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>0.34</td>
<td>0.83</td>
<td>0.81</td>
<td>0.65</td>
<td>0.87</td>
<td>0.74</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>0.64</td>
<td>1.18</td>
<td>1.05</td>
<td>1.17</td>
<td>1.51</td>
<td>1.45</td>
<td>0.99</td>
</tr>
<tr>
<td>random2</td>
<td>100</td>
<td>0.27</td>
<td>0.42</td>
<td>0.39</td>
<td>0.47</td>
<td>0.51</td>
<td>0.46</td>
<td>0.56</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>0.47</td>
<td>0.64</td>
<td>0.59</td>
<td>0.54</td>
<td>0.53</td>
<td>0.48</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>0.59</td>
<td>0.85</td>
<td>0.78</td>
<td>0.72</td>
<td>0.89</td>
<td>0.71</td>
<td>0.69</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>4.22</td>
<td>1.07</td>
<td>1.01</td>
<td>1.91</td>
<td>1.50</td>
<td>1.47</td>
<td>1.01</td>
</tr>
<tr>
<td>increasing</td>
<td>100</td>
<td>-</td>
<td>0.38</td>
<td>0.38</td>
<td>0.35</td>
<td>0.48</td>
<td>0.38</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>-</td>
<td>0.60</td>
<td>0.63</td>
<td>0.42</td>
<td>0.65</td>
<td>0.53</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>-</td>
<td>0.88</td>
<td>0.82</td>
<td>0.47</td>
<td>0.80</td>
<td>0.63</td>
<td>0.62</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>-</td>
<td>1.12</td>
<td>1.15</td>
<td>0.57</td>
<td>0.92</td>
<td>0.85</td>
<td>0.77</td>
</tr>
<tr>
<td>decreasing</td>
<td>100</td>
<td>-</td>
<td>0.37</td>
<td>0.40</td>
<td>0.35</td>
<td>0.35</td>
<td>0.42</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>-</td>
<td>0.63</td>
<td>0.62</td>
<td>0.43</td>
<td>0.33</td>
<td>0.38</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>-</td>
<td>0.83</td>
<td>0.83</td>
<td>0.50</td>
<td>0.35</td>
<td>0.40</td>
<td>0.62</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>-</td>
<td>1.05</td>
<td>1.10</td>
<td>0.63</td>
<td>0.40</td>
<td>0.45</td>
<td>0.83</td>
</tr>
</tbody>
</table>

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
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


