Performance Evaluation of Approximate Priority Queues

Yossi Matias *   Süleyman Cenk Şahinalp †   Neal E. Young ‡

June 9, 1997

Abstract

We report on implementation and a modest experimental evaluation of a recently introduced priority-queue data structure. The new data structure is designed to take advantage of fast operations on machine words and, as appropriate, reduced key-universe size and/or tolerance of *approximate* answers to queries. In addition to standard priority-queue operations, the data structure also supports successor and predecessor queries. Our results suggest that the data structure is practical and can be faster than traditional priority queues when holding a large number of keys, and that tolerance for approximate answers can lead to significant increases in speed.

*Bell Laboratories, 600 Mountain Ave. Murray Hill, NJ 07974; matias@research.bell-labs.com.

†Bell Laboratories, 600 Mountain Ave. Murray Hill, NJ 07974; jenk@research.bell-labs.com. The author was also affiliated with Department of Computer Science, University of Maryland, College Park, MD 20742 when the experiments were conducted.

‡Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510; ney@cs.dartmouth.edu.

speedup (n) : An employer's demand for accelerated output without increased pay.

- Webster's dictionary.

1 Introduction

A priority queue is an abstract data type consisting of a dynamic set of data items with the following operations:

- inserting an item,
- deleting the item whose key is the maximum or minimum among the keys of all items.

The variants described in this paper also support the following operations: deleting an item with any given key; finding the item with the minimum or the maximum key; checking for an item with a given key; checking for the item with the largest key smaller than a given key; and checking for the item with the smallest key larger than a given key.

Priority queues are widely used in many applications, including shortest path algorithms, computation of minimum spanning trees, and heuristics for NP-hard problems (e.g. TSP). For many applications, priority queue operations (especially deleting the minimum or maximum key) are the dominant factor in the performance of the application. Priority queues have been extensively analyzed and numerous implementations and empirical studies exist; see, e.g., [AHU74, Meh84, Jon86, SV87, LL96].

This note presents an experimental evaluation of one of the priority queues recently introduced in [MVY94]: the *(word-based) radix tree*. This data structure is qualitatively different than traditional priority queues in that its time per operation does not increase as the number of keys grows (in fact it can decrease). Instead, as in the van Emde Boas data structure [vKZ77], the running time depends on the size of the universe – the set of possible keys – which is assumed to be $\{0, 1, \ldots, U\}$, for some integer $U > 0$. Most notably, the data structure is designed to take advantage of any tolerance the application has for working with *approximate*, rather than exact, key values. The greater the tolerance for approximation, the smaller the effective universe size and so the greater the speed-up.

More specifically, suppose that the data structure is implemented on a machine whose basic word size is b bits. When no approximation error is allowed, the running time of each operation on the data structure is expected to be close to $c_0(\lg U)/(\lg b)$, for some $c_0 > 0$. It is expected to be significantly faster when a relative error tolerance, $\epsilon > 0$, is allowed. In this case, each key k is mapped to a smaller (approximate) key $k_\epsilon \approx \lceil \lg_{1+\epsilon} k \rceil$, resulting in an effective universe U_ϵ that is considerably smaller. In particular, when the universe size U is smaller than 2^b, and the relative error is $\epsilon = 2^{-j}$ (for non-negative integer j), then the running time of each operation on the data structure is expected to be close to $c(1 + j/\lg b)$, for some constant $c > 0$.

The data structure is also unusual in that it is designed to take advantage of fast bit-based operations on machine words. Previously, such operations were used in the context of theoretical priority queues, such as those of Fredman and Willard [FW90a, FW90b, Wil92], which are often considered non-practical. In contrast, the data structure considered here appears to be simple to implement and to have small constants; it is therefore expected (at least in theory) to be a candidate for a competitive, practical implementation, as long as the bit-based operations can be supported efficiently.

In this paper we report on an implementation of the word-based radix tree, on a demonstration of the effectiveness of approximation for improved performance, and on a modest experimental testing which was designed to study to what extent the above theoretical expectations hold in practice. Specifically, our experimental study considers the following questions:

1. What may be a typical constant, c_0, for the radix tree data structure with no approximation tolerance, in the estimated running time of $c_0(\lg U)/(\lg b)$;

2. What may be a typical constant, c, for the radix tree data structure with approximation error $\epsilon = 2^{-\alpha} > 0$, for which the estimated running time is approximately $c(1 + \alpha/\lg b)$; and

3. How does the performance of the radix tree implementations compare with traditional priority queue implementations, and what may be a typical range of parameters for which it may become competitive in practice.

We study two implementations of the radix tree: one *top-down*, the other *bottom-up*. The latter variant is more expensive in terms of memory usage, but it has better time performance. The main body of the tests timed these two implementations on numerous randomly generated sequences, varying the universe size U over the set $\{2^{10}, 2^{15}, 2^{20}, 2^{25}\}$, the number of operations N over the set $\{2^{15}, 2^{18}, 2^{21}, 2^{24}\}$, and the *approximation tolerance*, ϵ, over the set $\{2^{-5}, 2^{-10}, \ldots, U/2^5, 0\}$, where the case $\epsilon = 0$ corresponds to no tolerance for approximation. The tests were performed on a single node of an SGI Power Challenge with 2Gbytes of main memory.

The sequences are of three given types: INSERTs only, INSERTs followed by DELETE-MINIMUMs, and INSERTS followed by mixed operations. Based on the measured running times, we identify reasonable estimates to the constants c_0 and c, for which the actual running time deviates from the estimated running time by at most about 50%.

The bottom-up radix tree was generally faster than the top-down one, sometimes significantly so. For sequences with enough insertions, the radix trees were faster than the traditional priority queues, sometimes significantly so.

2 The Data Structure

We describe the the *top-down* and the *bottom-up* variants of the *(word-based) radix tree* priority queue.

Both implementations are designed to take advantage of any tolerance of approximation in the application. Specifically, if an approximation error of $\epsilon > 0$ is tolerated, then each given key k is mapped before use to the *approximate key* $k_\epsilon \approx \lceil \lg_{1+\epsilon} k \rceil$ (details are given below). This ensures that, for instance, the FIND-MINIMUM operation will return an element at most $1+\epsilon$ times the true minimum.

Operating on approximate keys reduces the universe size to the *effective universe size* U_ϵ. For completeness, we give the following general expression for U_ϵ (according to our implementation):

$$U_\epsilon \doteq \begin{cases} 2^{\lceil \lg \lceil \lg (U+1) \rceil \rceil + \lceil \lg (1/\epsilon) \rceil} - 1 & \text{if } \epsilon > 0 \\ U & \text{if } \epsilon = 0. \end{cases} \tag{1}$$

Both the top-down and bottom-up implementations of the word-based radix-tree maintain a complete b-ary tree with U_ϵ leaves, where b is the number of bits per machine word on the host computer (here we test only $b = 32$). Each leaf holds the items having a particular approximate key value k_ϵ. The time per operation for both implementations is upper bounded by a constant times the number of levels in the tree, which is

$$\text{LEVELS}(U, \epsilon) \doteq \lceil \lg_b (U_\epsilon + 1) \rceil \approx \begin{cases} \lg_b \lg U + \lg_b (1/\epsilon) & \text{if } \epsilon > 0 \\ \lg_b U & \text{if } \epsilon = 0. \end{cases} \tag{2}$$

For $U \leq 2^b$, $\epsilon = 2^{-\alpha}$, and for b a power of two, we get

$$\text{LEVELS}(U,\epsilon) \leq \begin{cases} \lceil 1 + \alpha/\lg b \rceil & \text{if } \epsilon > 0 \\ \lceil 1 + \lg(U)/\lg b \rceil & \text{if } \epsilon = 0. \end{cases}$$

For all parameter values we test, the above inequality is tight, with the number of levels ranging from 2 to 5 and the number of leaves ranging from 2^{10} to 2^{25}.

As mentioned above, each leaf of the tree holds the items having a particular (approximate) key value. Each interior node holds a single machine word whose ith bit indicates whether the ith subtree is empty. The top-down variant allocates memory only for nodes with non-empty subtrees. Each such node, in addition to the machine word, also holds an array of b pointers to its children. An item with a given key is found by starting at the root and following pointers to the appropriate leaf. It examines a node's word to navigate quickly through the node. For instance, to find the index of the leftmost non-empty subtree, it determines the most significant bit of the word. If a leaf is inserted or deleted, the bits of the ancestors' words are updated as necessary.

The bottom-up variant pre-allocates a single machine word for every interior node, empty or not. The word is stored in a fixed position in a global array, so no pointers are needed. The items with identical keys are stored in a doubly linked list which is accessible from the corresponding leaf. Each operation is executed by updating the appropriate item of the leaf, and then updating the ancestors' words by moving up the tree. This typically will terminate before reaching the root, for instance an INSERT needs only to update the ancestors whose sub-trees were previously empty.

2.1 Details on implementations of the data structure and word operations

Below we describe how the basic operations are implemented in each of our data structures. We also describe implementations of the word-based operations we require.

2.1.1 Computing approximate keys

Given a key k, this is how we compute the approximate key k_ϵ, assuming $\epsilon = 2^{-j}$ for some non-negative integer j. Let $a_{B-1}a_{B-2}\ldots a_0$ be the binary representation of a key $k \leq U$, where $B = \lceil \lg(U+1) \rceil$. Let $m = \max\{i : a_i = 1\}$ be the index of the most significant bit of k. Let n be the integer whose j-bit binary representation is $a_{m-1}a_{m-2}\ldots a_{m-j}$ (i.e., the j bits following the most significant bit). Then

$$k_\epsilon \doteq m2^j + n.$$

Implicitly, this mapping partitions the universe into sets whose elements differ by at most a multiplicative factor of two, then partitions each set into subsets whose elements differ by at most an appropriate additive amount. Under this mapping, the universe of approximate keys is a subset of $\{0, 1, \ldots, U_\epsilon\}$, where U_ϵ is the largest integer whose binary representation has a number of bits equal to $\lceil \lg \lceil \lg(U+1) \rceil \rceil + j$.[1]

2.1.2 Bottom-up implementation

The bottom-up implementation of the radix tree is a complete 32-ary tree with $L = \lceil \lg_{32} U_\epsilon \rceil$ levels (level ℓ consists of those nodes at distance ℓ from the bottom of the tree, so that the leaves form level zero).

[1]This mapping can be improved a bit by using $k'_\epsilon \doteq (m'-j)2^j + n'$, where $m' \doteq \max\{j, m\}$ and $n = a_{m'-1}a_{m'-2}\ldots a_{m'-j}$.

The leaves in the tree correspond to the approximate key values. For each leaf there is a pointer to a linked list of the items (if any) whose keys equal the corresponding value. These pointers are maintained in an array[2] of size 32^L.

For each level $\ell > 0$, we maintain an array A_ℓ of $32^{L-\ell}$ 32-bit words, one for each node in level ℓ. For convenience and locality of reference, the nodes are stored in the arrays so that the binary representation of the index of each node is the common prefix of the binary representation of the indices of its descendants. That is, the index of a leaf with key value k_ϵ is just k_ϵ. The index of the ancestor of the leaf at level ℓ has binary representation $a_{D-1}a_{D-2}\ldots a_{5\ell}$, where $a_{D-1}a_{D-2}\ldots a_0$ is the binary representation of k_ϵ ($D = \lceil \lg(U_\epsilon + 1) \rceil$). This allows the index of an ancestor or child to be computed quickly from the index of a given node.

As described previously, we maintain the invariant that the rth bit of a node's word is set iff the subtree of the rth child is non-empty. For brevity, we call r the *rank* of the child.

INSERT First insert the item (with key k_ϵ) into the linked list of the corresponding leaf. Then set the appropriate bits of the ancestors' words to 1. Specifically, starting at the leaf, check the rth bit of the parent's word, where r is the rank of the leaf (note that r is the five least significant bits of k_ϵ). If the bit is already set, stop, otherwise set it, move up to the parent, and continue. Stop at the root or upon finding an already-set bit.

SEARCH To test whether an item with a given key k_ϵ is present, simply check whether the leaf's linked list is empty. To find an item with the given key, search the linked list of the leaf.

DELETE First remove the item from the linked list of its leaf. If the list becomes empty, update the appropriate bits of the ancestors' words. Specifically, starting at the leaf, zero the rth bit of the parent's word, where r is the rank of the node. If the entire word is still non-zero, stop. Otherwise, move up to the parent and continue. Stop at the root or upon finding a word that is non-zero after its bit is zeroed.

MINIMUM Starting from the root, compute the index r of the least significant bit (**lsb**) of the root's word. Descend to the child of rank r and continue. Stop at a leaf.[3]

MAXIMUM This is similar to MINIMUM, except it uses the most, rather than least, significant bits.

SUCCESSOR (Given a key k_ϵ, return an item with smallest key as large as k_ϵ.)

If the leaf corresponding to k_ϵ has a non-empty list, return an item from the list. Otherwise, search up the tree from the leaf to the first ancestor A with a *non-empty* sibling R_A of rank at least the rank of A. To find R_A, examine the word of the parent of A to find the most significant bit (if any) set among those bits encoding the emptiness of the subtrees of those siblings that have rank at least the rank of A. (For this use the *nearest-one-to-right* operation, **nor**, as described later in this section.)

Next find the leftmost non-empty leaf in R's subtree, proceeding as in the MINIMUM operation above. Return an item from that leaf's list.

[2]If this array is large and only sparsely filled, it can instead be maintained as a hash table.

[3]In retrospect, this query should probably have been done in a bottom-up fashion, similar to the SUCCESSOR query below.

PREDECESSOR PREDECESSOR is similar to SUCCESSOR, but is based on the *nearest-one-to-left* (**nol**) operation.

2.1.3 Top-down implementation

This implementation also maintains a 32-ary tree with L levels. The implementation differs from the bottom-up one in that memory is allocated on a per-node basis and operations start at the root, rather than at the leaves.

Specifically, each interior node rooting a non-empty subtree consists of a 32-bit word (used as in the bottom-up implementation) and an array of 32 pointers to its children nodes (where a pointer is null if the child's subtree is empty).

INSERT Starting from the root, follow pointers to the appropriate leaf, instantiating new nodes and setting node's word's bits as needed. Specifically, before descending into a rank-r child, set the rth bit of the node's word.

SEARCH Follow pointers from the root as in INSERT, but stop upon reaching any empty subtree and return NULL. If a leaf is reach, return a node from the leaf's list.

DELETE Follow pointers as in SEARCH. Upon finding the leaf, remove an item from the leaf's list. Retrace the path followed down the tree in reverse, zeroing bits as necessary to represent subtrees that have become empty.

MINIMUM Starting at the root, use the node's word to find the leftmost child with a non-empty subtree (as in the bottom-up implementation of INSERT). Move to that child by following the corresponding pointer. Continue in this fashion down the tree. Stop upon reaching a leaf and return an item from the leaf's list.

MAXIMUM Similar to MINIMUM.

SUCCESSOR Starting at the root, let r be the rank of the child whose subtree has the leaf corresponding to the given key k_ϵ. Among the children of rank at least r with non-empty subtrees, find the one of smallest rank. Do this by applying the *nearest-one-to-right* word-operation (**nor**, defined below) to the node's word. Follow the pointer to that child. Continue until reaching a leaf.

PREDECESSOR Similar to SUCCESSOR, except use the **nol** operation instead of **nor**.

2.1.4 Implementation of binary operations

Below we describe how we implemented the binary operations that are not supported by the machine hardware.

msb(key). The *most significant bit* operation, or **msb** is defined as follows: Given a key k, **msb**(k) is the index of the most significant bit that is equal to 1 in k. We have three implementations of the operation:

- The first method is starting from the $\lg U_\epsilon^{th}$ **msb** of the key and linearly searching the **msb** by checking next **msb** by shift left operation. We expect this method to be fast (just a few shifts will be needed) for most values of the words.

- A method for obtaining a worst-case performance of $\lg\lg U_\epsilon$ is based on binary search. We apply at most $\lg\lg U_\epsilon$ masks and comparisons with 0 to be to determine the index of **msb** This method is not used in any our experiments as it was slower than the above method in preliminary experiments.

- The last method is dividing the key to 4 blocks of 8 bit each, V_4, V_3, V_2, V_1, by shifts and masks, and then search the **msb** of the most significant non-zero block. This approach could be seen as a compromise between the two methods described above, and is used for the computation of the maximum and minimum keys.

lsb(key). The *least significant bit* operation, or **lsb** is defined as follows: Given a key k, $\mathbf{lsb}(k)$ is the index of the least significant bit that is equal to 1 in k. The implementations for **lsb** are similar to those of **msb** (essentially, replacing above "most" with "least", "**msb**" with "**lsb**", and "left" with "right").

We note that by using the above implementations for **msb** and **lsb** in our experiments, we exploited the fact that the input is random. For general applicability, we would need an implementation that is efficient for arbitrary input. This report is concerned mainly with the performance of the radix tree data structure when efficient bit-based operations are available, and therefore exploiting the nature of input for efficient **msb** and **lsb** executions is appropriate. Nevertheless, we mention an alternative technique (which we have not tested), that enables an **msb** and **lsb** implementation using a few operations: one subtraction, one bit-wise XOR, one MOD, and one lookup to a table of size b.

We describe the **lsb** implementation for a given value x. First, note that $x \mapsto (x\, \mathbf{bitwise\text{-}xor}\, (x-1))$ maps x to $2^{\mathbf{lsb}(x)+1} - 1$. Next, note that there will be a small prime p slightly larger than b such that $2^i \bmod p$ is unique for $i \in \{0, ..., p-1\}$, because 2 generates the multiplicative mod-p group. Thus, the function $x \mapsto ((x\, \mathbf{bitwise\text{-}xor}(x-1)) \bmod p)$ maps x to a number in $\{0, ..., p-1\}$ that is uniquely determined by $\mathbf{lsb}(x)$. By precomputing an array T of $p \approx b$ b-bit words such that $T[(x\, \mathbf{bitwise\text{-}xor}(x-1)) \bmod p] = \mathbf{lsb}(x)$, the **lsb** function can subsequently be computed in a few operations. Examples of good p include 523 ($b = 512$), 269 ($b = 256$), 131 ($b = 128$), 67 ($b = 64$), 37 ($b = 32$), 19 ($b = 16$), and 11 ($b = 8$).

nol(index,key). The operation *nearest one to left*, or **nol**, is defined as follows: Given a key k and an index i, $\mathbf{nol}(i, k)$ is the index of the first bit that is equal to 1 to the left of (more significant than) the given index. The **nol** operation is implemented by using the operations shift right by index and **msb**.

nor(index,key). The operation *nearest one the right*, or **nor** , is defined as follows: Given a key k and an index i, the $\mathbf{nor}(i, k)$ is the index of the first bit that is equal to 1 to the right of (least significant than) the given index. The **nor** operation is implemented by using the operations shift left by index and **lsb**.

3 The Tests

In this section we describe in detail what kind of tests we performed, the data sets we used and the testing results. We verify from the testing results how well our theoretical expectations were supported and interpret the variations.

3.1 The test sets

The main body of the test timed two implementations on numerous sequences, varying the universe size U over the set $\{2^{10}, 2^{15}, 2^{20}, 2^{25}\}$, the number of operations N over the set $\{2^{15}, 2^{18}, 2^{21}, 2^{24}\}$, and the approximation tolerance, ϵ, over the set $\{2^{-5}, 2^{-10}, \ldots, U/2^5, 0\}$. We note that the case $\epsilon = 0$ corresponds to no tolerance for approximation.

The sequences were of three kinds: N insertions; $N/2$ insertions followed by $N/2$ DELETE-MINIMUMs; or $N/2$ insertions, followed by $N/2$ mixed operations (half INSERTs, one quarter DELETEs, and one quarter DELETE-MINIMUMs). Each insertion was of an item with a random key from the universe. Each deletion was of an item with a random key from the keys in the priority queue at the time of the deletion.

For comparison, we also timed more "traditional" priority queues: the binary heap, the Fibonacci heap, the pairing heap and the 32-ary heap from the LEDA library [LED], as well as a plain binary search tree with no balance condition. None of these data structures are designed to take advantage of U or ϵ, so for these tests we fixed $U = 2^{20}$ and $\epsilon = 0$ and let only N vary.[4]

3.2 The test results

In summary, we observed that the times per operation were roughly linear in the number of levels as expected. Across the sequences of a given type (INSERTs, INSERTs then DELETE-MINIMUMs, or INSERTS then mixed operations), the "constant" of proportionality varied up or down typically by as much as 50%. The bottom-up radix tree was generally faster than the top-down one, sometimes significantly so. For sequences with enough insertions, the radix trees were faster than the traditional priority queues, sometimes significantly so.

3.3 Estimating the constants c_0 and c

The following table summarizes our best estimates of the running time (in microseconds) per operation as a function of the number of levels in the tree $L = \text{LEVEL}(U, \epsilon)$ (see Equation (2)) for the two radix-tree implementations on the three types of sequences, as N, U, and ϵ vary. These estimates are based on the sequences described above, and are chosen to minimize the ratio between the largest and smallest deviations from the estimate. The accuracy of these estimates in comparison to the actual times per operation is presented Figures 1 and 2.

	INSERT	INS/DEL	INS/mixed
top-down $\epsilon > 0$	$-3.1 + 3.3L$	$-2.9 + 3.4L$	$-7.9 + 7.6L$
$\epsilon = 0$	$-14.9 + 5.2L$	$-11.4 + 4.4L$	$-12.4 + 6.6L$
bottom-up $\epsilon > 0$	$1.3 + 1.3L$	$0.8 + 1.3L$	$1.1 + 1.6L$
$\epsilon = 0$	$-4.8 + 1.8L$	$-5.6 + 2.4L$	$-4.8 + 2.2L$

The variation of the actual running times on the sequences in comparison to the estimate is typically on the order of 50% (lower or higher) of the appropriate estimate. The bottom-up implementation is generally faster than the top-down implementation. The $\epsilon = 0$ case is typically slower *per level* than the $\epsilon > 0$ case (especially for the bottom-up implementation).

The table below presents for comparison an estimate of the time for the pairing heap (consistently the fastest of the traditional heaps we tested) as a function of $\lg N$.

[4]The use of approximate keys reduces the number of *distinct* keys. By using approximate keys, together with an implementation of a traditional priority queue that "buckets" equal elements, the time per operation for DELETE and DELETE-MIN could be reduced from proportional to $\lg N$ down to proportional to the logarithm of the number of distinct keys. We don't explore this here.

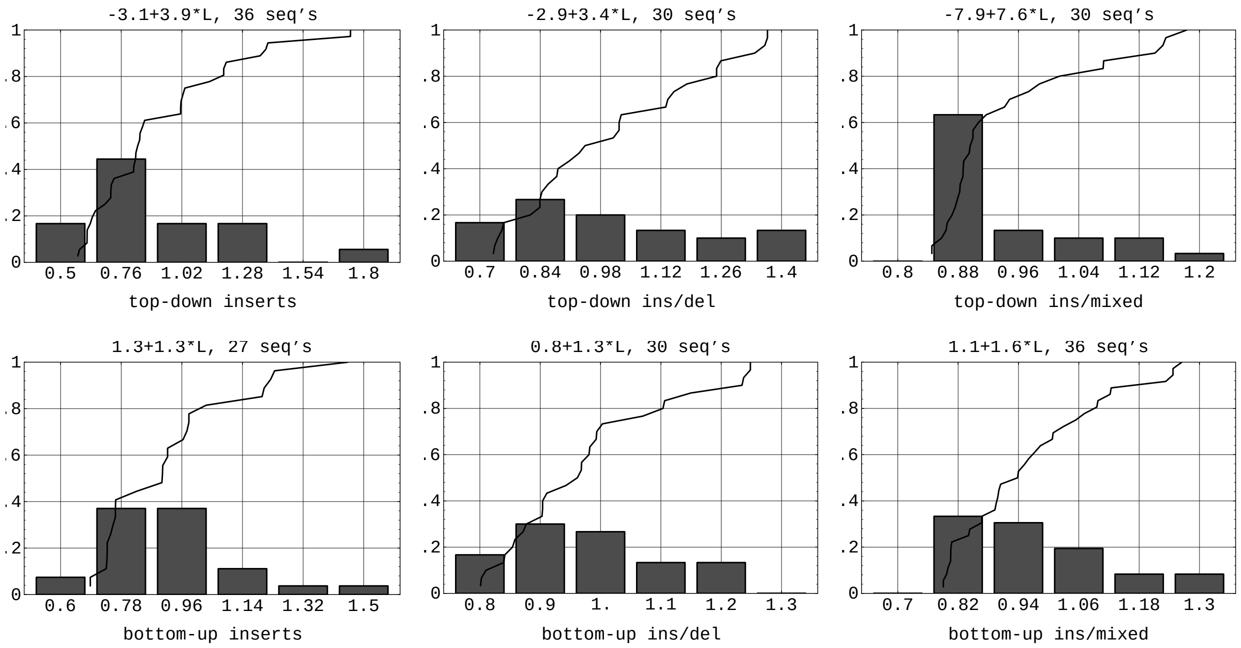

Figure 1: For the plot in the upper left, the top-down radix tree was run on sequences of N insertions of a universe of size U with approximation tolerance ϵ for the various values of N, U, and $\epsilon > 0$. Above the plot is an estimate (chosen to fit the data) of the running time per operation (in microseconds as a function of the number of levels L) for the top-down implementation on sequences of insertions. To the right of the function is the number of sequences. Each sequence yields an "error ratio" — the actual time for the sequence divided by the time predicted by the estimate. The plot shows the distribution of the set of error ratios. Each histogram bar represents the fraction of error ratios near the corresponding x-axes label x. The curve above represents the cumulative fraction of sequences with error ratios x or less. There is one plot for each implementation/sequence-type pair. Typically, the error ratios are between 0.5 and 1.5.

	INSERT	INS/DEL	INS/mixed
best traditional	$1.69 + .009 \lg N$	$-29.6 + 2.45 \lg N$	$-16.8 + 1.28 \lg N$

3.4 Demonstrating the effect of approximation

Our experiments suggest that the radix tree might be very suitable for exploiting approximation tolerance to obtain better performance. In the following table we provide timing results for three sequences of operations applied to the bottom up implementation with approximation tolerance $\epsilon = 2^{-5}, 2^{-10}, 2^{-15}, 2^{-20}, 0$, and the universe size $U = 2^{25}$. The first sequence consist of $N = 2^{21}$ insertions, the second one consists of $N/2 = 2^{20}$ insertions followed by $N/2 = 2^{20}$ delete-minimums, and finally the third one consists of $N/2 = 2^{20}$ insertions followed by $N/2 = 2^{20}$ mixed operations.

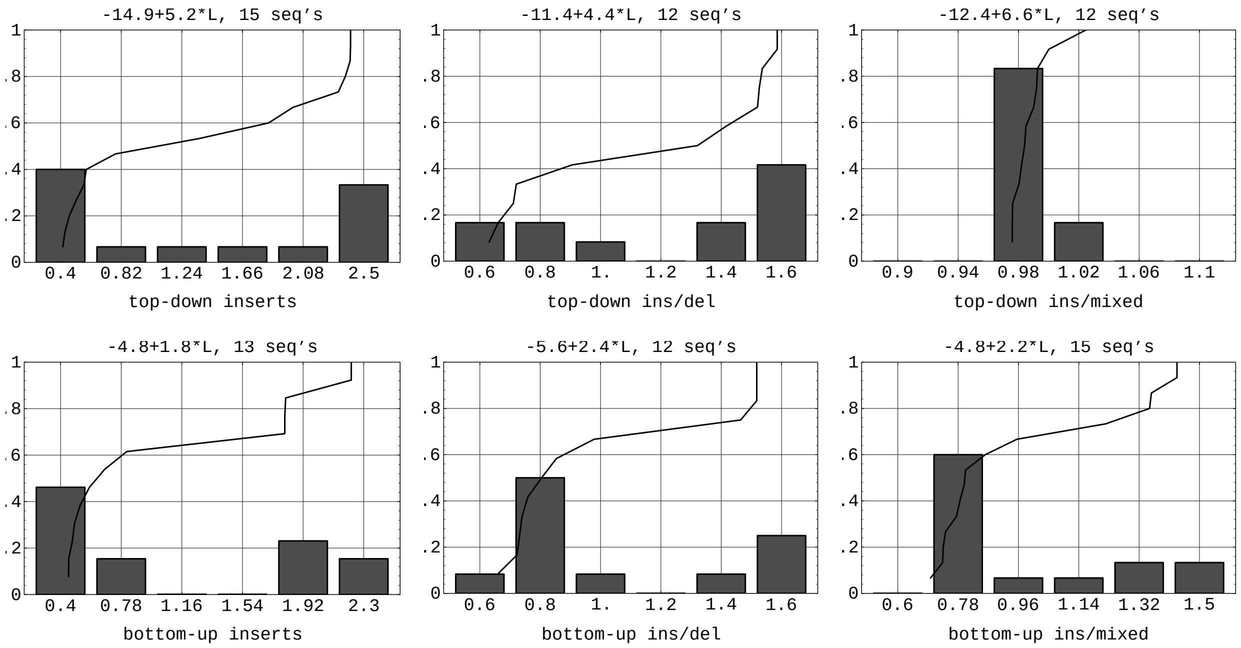

Figure 2: This figure is analogous to Figure 1 but represents the case $\epsilon = 0$. Note the larger variation in INSERT times.

	Sequence 1		Sequence 2		Sequence 3	
	top-down	bottom-up	top-down	bottom-up	top-down	bottom-up
$\epsilon = 0$	47.11	7.01	42.13	14.93	56.02	15.00
$\epsilon = 2^{-20}$	29.7	5.02	29.77	8.99	55.24	18.23
$\epsilon = 2^{-15}$	17.1	2.21	19.65	6.17	41.63	14.28
$\epsilon = 2^{-10}$	14.75	2.08	14.18	4.94	29.23	12.66
$\epsilon = 2^{-5}$	13.77	2.08	11.07	3.27	17.5	11.36

3.5 Some comparisons to traditional heaps

Our experiments provide some indication that the radix tree based priority queue might be of practical interest. Below we provide timing results for the bottom up implementation on three sequences for comparing the performance of the bottom up implementation and the fastest traditional heap implementation from the LEDA library, which consistently occured to be the pairing heap. The first sequence consist of $N = 2^{21}$ insertions, the second one consists of $N/2 = 2^{20}$ insertions followed by $N/2 = 2^{20}$ delete-minimums, and finally the third one consists of $N/2 = 2^{23}$ insertions followed by $N/2 = 2^{23}$ mixed operations; $U = 2^{25}$ and $\epsilon = 0$ for all sequences.

These figures suggest that under appropriate conditions – the size of the universe, the number of elements, and the distribution of the keys over the universe – the radix tree can be competitive with and sometimes better than the traditional heaps.

	Sequence 1	Sequence 2	Sequence 3
Bottom-Up	7.01	14.93	104.38
fastest traditional	3.96	44.43	249.73

The pairing heap is quite fast for the insertion-only sequences. For the other sequences, from the above tables, we can give rough estimates of when, say, the bottom-up radix tree will be faster

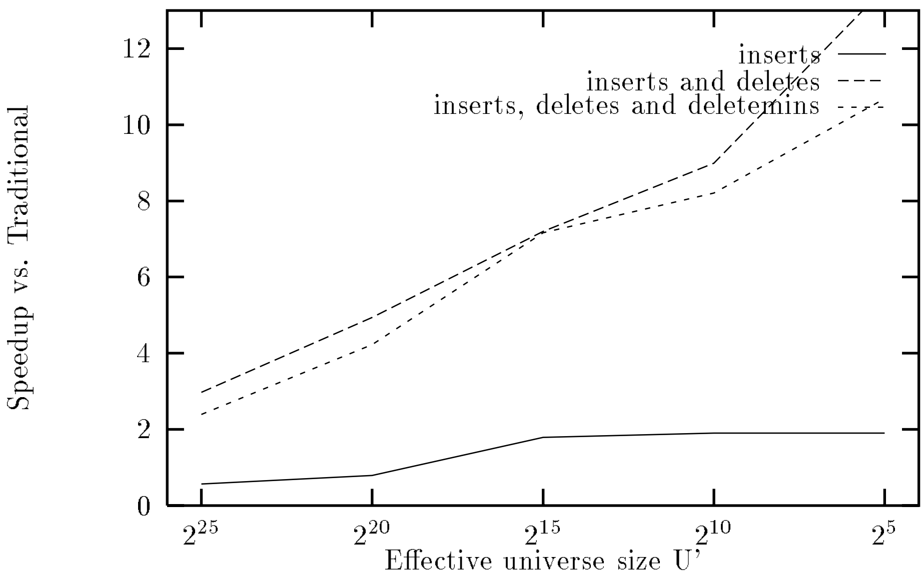

Figure 3: Speedup factor (running time for best traditional priority queue divided by time for bottom-up ϵ-approximate radix-tree) as a function of ϵ for three random test sequences with $N \approx 2^{20}$ and $U \approx 2^{25}$.

---

than the pairing heap. Recall that in the range of parameters we have studied,

$$\text{LEVELS}(U, \epsilon) = \begin{cases} 1 + \lg(1/\epsilon)/5 & \text{if } \epsilon > 0 \\ 1 + \lg(U)/5 & \text{if } \epsilon = 0. \end{cases}$$

For instance, assume a time per operation of $3 + 1.5\text{LEVEL}(U, \epsilon)$ for the radix tree, and $2 \lg N - 25$ for the pairing heap. Even for U as large as 2^{25} and $\epsilon = 0$, the radix tree is faster once N is larger than roughly 43000. If $\epsilon > 1/1000$, then the radix tree is faster once N is larger than roughly 9000. For direct comparisons on actual sequences, see Figure 3, which plots the speed-up factor (ratio of running times) as a function of ϵ on sequences of each of the three types.

We also provide Figures 4, 5, and 6 to demonstrate the conditions under which the radix tree implementations become competitive with the LEDA implementations.

3.6 Discussion

The variation in running times makes us cautious about extrapolating the results.

The effect of big data sets The first source of variation is that in both implementations having more keys in the tree reduces the average time per operation, due to the following factors:

- The top-down variant is slower (by a constant factor) when inserting if the INSERT instantiates many new nodes. The significance of this varies — if a tree is fairly full, typically only a few new nodes, near the leaves, will be allocated. If the tree is relatively empty, nodes will also be allocated further up the tree.

- The bottom-up variant is faster if the tree is relatively full in the region of the operation. As discussed above, bottom-up operations typically step from a leaf only to a (previously or

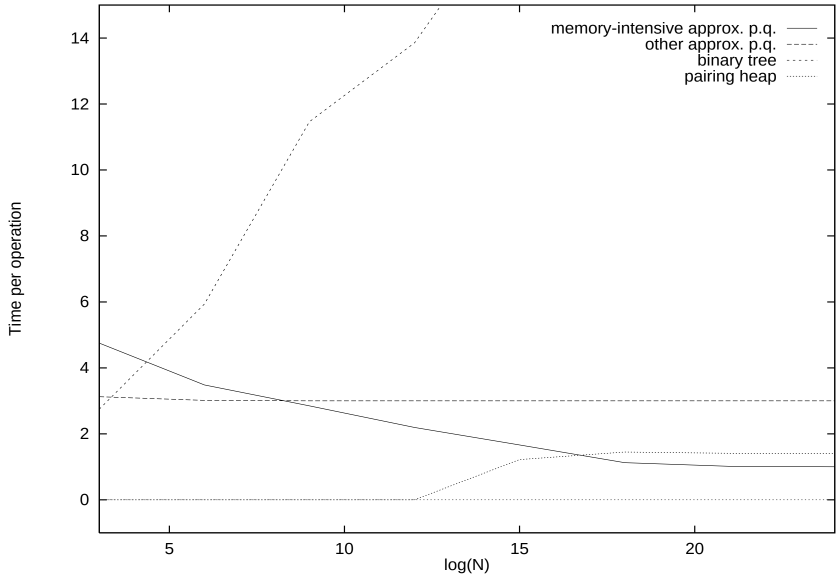

Figure 4: Time per operation (in μ-seconds) varying with $\lg N$, for a sequence of N inserts, for $U = 2^{25}$, with no approximation tolerance. One can observe that the time per operation in the top-down implementation and the pairing heap does not vary with the number of operations, as expected. For the bottom up implementation, the time per operation decreases with the increasing number of items, as explained above.

---

still) non-empty ancestor. This means operations can take time much less than the number of levels in the tree.

- The significance of the above two effects depends on the distribution of keys in the trees. For instance, if the keys are uniformly distributed in the original universe (as in our experiments) then, when $\epsilon > 0$, the *approximate* keys will not be uniformly distributed – there will be more large keys than small keys in the effective universe. Thus, the rightmost subtrees will be fuller (and support faster operations); the leftmost subtrees will be less full (and therefore slower).

The effect of memory issues On a typical computer the memory-access time varies depending on how many memory cells are in use and with the degree of locality of reference in the memory access pattern. A typical computer has a small very fast *cache*, a large fast random access memory (RAM), and a very larger slower virtual memory implemented on top of disk storage.

Our test sequences are designed to allow the data structures to fit in RAM but not in cache. This choice reduces but does not eliminate the memory-access time variation — variation due to caching remains. It is important to note that our results probably do not extrapolate to applications where the data structures do not fit in RAM; when that happens, the relative behaviors could change qualitatively as locality of reference becomes more of an issue.[5] On the other hand, even with fairly large universes and/or a large number of keys, most of these data structures can fit in a typical modern RAM.

Another memory-related issue is the number of bits per machine word b. In this paper, we consider only $b = 32$. On computers with larger words, the times per operation should decrease

[5]Of all the priority queues considered here, the bottom-up radix tree is the most memory-intensive. Nonetheless it seems possible to implement with reasonable locality of reference.

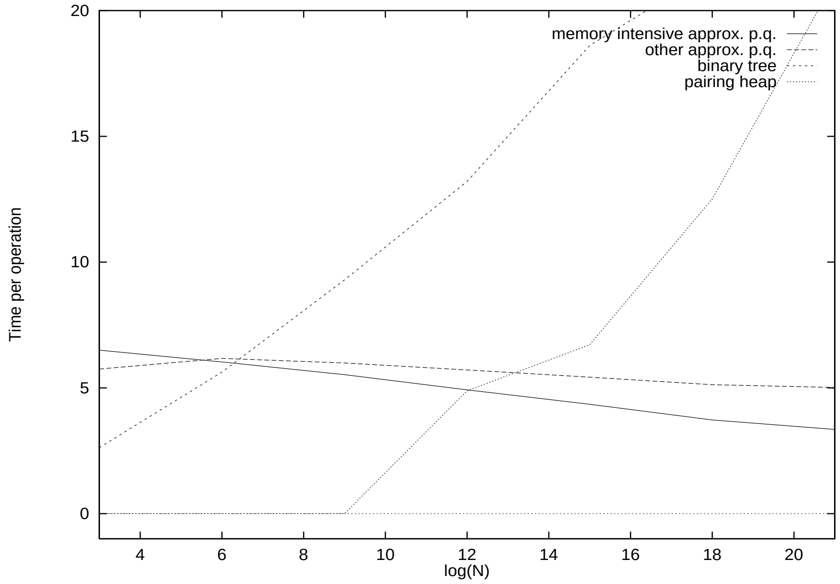

Figure 5: Time per operation (in μ-seconds) varying with $\lg N$, for a sequence of $N/2$ inserts, followed by $N/2$ delete-mins, for $U = 2^{25}$ with no approximation tolerance. One can observe that the time per operation in the top-down and bottom-up implementations is decreasing with time. The non-linear increase in the running time of the pairing heap is possibly due to decreasing cache utilization with increasing number of operations.

---

slightly (all other things being equal). Also, the relative speed of the various bit-wise operations on machine words (as opposed to, for instance, comparisons and pointer indirection) could affect the relative speeds of radix trees in comparison to traditional priority queues.

Acknowledgment

We thank Jon Bright for participating in this research at an early stage, and for contributing to the implementations.

References

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1974.

[FW90a] M.L. Fredman and D.E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proc. 22nd ACM Symp. on Theory of Computing*, pages 1–7, 1990.

[FW90b] M.L. Fredman and D.E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 719–725, 1990.

[Jon86] D. W. Jones. An empirical comparison of priority queue and event set implementations. *Communications of the ACM*, April 1986.

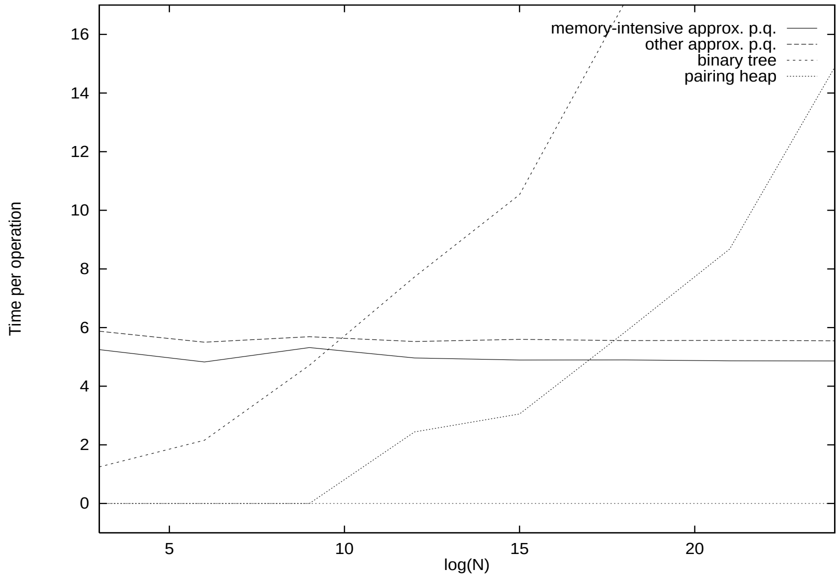

Figure 6: Time per operation (in μ-seconds) varying with $\lg N$, for a sequence of $N/2$ inserts, followed by $N/2$ mixed operations, for $U = 2^{25}$ with no approximation tolerance. The same trend as in Figure 5 can be observed.

[LED] LEDA system manual, version 3.2.3. Technical report, Max Planck Institute fur Informatik, Saarbrucken, Germany.

[LL96] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of heaps. Manuscript, 1996.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms I: Sorting and Searching*. Springer-Verlag, Berlin Heidelberg, 1984. EATCS Monographs on Theoretical Computer Science.

[MVY94] Y. Matias, J.S. Vitter, and N.E. Young. Approximate data structures with applications. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 187–194, January 1994.

[SV87] J. T. Stasko and J. S. Vitter. Pairing heaps: Experiments and analysis. *Communications of the ACM*, March 1987.

[vKZ77] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.

[Wil92] D.E. Willard. Applications of the fusion tree method to computational geometry and searching. In *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 286–295, 1992.