Technical Report 54

Faster than Pairing and Fibonacci Heaps?
Rank-Relaxed Weak Queues

Dr. Stefan Edelkamp
TZI, Universität Bremen

TZI-Bericht Nr. 54
2009

TZI-Berichte

Herausgeber:
Technologie-Zentrum Informatik und Informationstechnik
Universität Bremen
Am Fallturm 1
28359 Bremen
Telefon: +49-421-218-7272
Fax: +49-421-218-7820
E-Mail: info@tzi.de
http://www.tzi.de

ISSN 1613-3773

Rank-Relaxed Weak Queues: Faster than Pairing and Fibonacci Heaps?

Stefan Edelkamp
TZI
Universität Bremen
Am Fallturm 1
28357 Bremen
Germany
edelkamp@tzi.de

September 24, 2009

Abstract

A run-relaxed weak queue by Elmasry et al. (2005) is a priority queue data structure with insert and decrease-key in $O(1)$ as well as delete and delete-min in $O(\log n)$ worst-case time. One further advantage is the small space consumption of $3n + O(\log n)$ pointers.

In this paper we propose *rank-relaxed weak queues*, reducing the number of rank violations nodes for each level to a constant, while providing amortized constant time for decrease-key. Compared to run-relaxed weak queues, the new structure additionally gains one pointer per node.

An empirical evaluation shows that the implementation can outperform Fibonacci and pairing heaps in practice even on rather simple data types.

1 Introduction

Priority queues are among the most important non-trivial data structures and essential for many fundamental algorithms, like Dijkstra's approach to

compute shortest paths [3], or minimum spanning tree generation according to Kruskal's algorithm [15]. For a comparison function operating on totally ordered keys, besides providing the dictionary operations insert and delete, priority queues feature extracting the minimum and decreasing the value of a key.

The most prominent implementation of priority queues featured in many text books are Fibonacci heaps [12], which can be roughly characterized as lazy-join versions of binomial queues. They provide insert and decrease-key in $O(1)$ amortized, as well as delete and delete-min in $O(\log n)$ amortized.

Run-relaxed weak queues as proposed in Elmasry et al. [9] are worst-case efficient priority queues, by means that all running times of Fibonacci heaps are worst-case instead of amortized. They have been derived from run-relaxed heaps [4], which have matching performance, but a rather involved and less space-efficient implementation. The core difference between the two is that the latter relies on binomial queues, while the former uses perfect weak-heaps, where weak-heaps [5] have been designed for efficient sorting. Compared to ordinary binary heaps, weak-heaps are less restrictive. A key only needs to be smaller than all keys in its right subtree. As the root node has no left subtree, it contains the minimal key. The efficiencies for sorting, worst and best case inputs, and the construction of a (double-ended) priority queue has been studied by [7].

In this paper we improve run-relaxed weak queues to rank-relaxed weak queues for better practical time and space performance by refining the data structure for storing and reducing potential heap-order violating nodes. The core result is that by sacrificing worst-case for amortized complexity at most 4 potential heap-order violating nodes are needed at each heigth.

As the operation is not to be so important in applications this paper does not discuss an efficient meld of two rank-relaxed weak queues. As the structure for heap-order violation becomes simper for rank-relayed weak-queues compared to run-relaxed weak-queues we expect that a worst case running time of $O(\min\{\log m, \log n\})$ for two structures of n and m elements should be possible to achieve.

Our experiments in a space-optimized implementation show that the efficiency of our implementation can be superior to the performance of Fibonacci and pairing heap priority queue implementations. Moreover, wrt. new developments of processor architectures to support leading zero bit counts, the efficiency might further rise. The price we pay wrt. the original implementation of run-relaxed weak-heaps is that decrease-key is no longer worst-case

but amortized constant time Our approach further shows that the space consumption of relaxed weak queues can be reduced.

2 Run-Relaxed Weak Queues

Run-relaxed weak queues are binary tree variants of run-relaxed heaps [4], and reflect worst-case efficient priority queues (with constant-time efficiencies for insert and decrease-key and logarithmic time for delete and delete-min). Other structures achieving this performances are *Brodal heaps* [2] and *fat heaps* [14]. The fact that distinguishes run-relaxed weak queues from the others is that they are considerably easy to implement [19].

Weak-heaps [5] are obtained by relaxing the heap requirements. More precisely, a weak-heap satisfies the following three conditions: The root value of any subtree is smaller than or equal to all elements to its right (weak heap dominance property), the root of the entire structure has no left child (optimal root property), and leaf nodes are found on the last two levels only (heap balance property). In *perfect weak-heaps*, the right subtree of the root is a complete binary tree. Weak-heaps have a natural array embedding that utilizes so-called *reverse bits* r_i, $i \in \{0, \ldots, n-1\}$. The index of the left child is located at $2i+r_i$ and the right child is found at $2i+1-r_i$. For this purpose r_i is interpreted as an integer in $\{0, 1\}$, being initialized with value 0. By flipping, the bit the status of being a left and a right child is exchanged, an essential property to realize the join of two weak-heaps in constant time.

As an example take $a = [1, 4, 5, 2, 7, 5, 3, 8, 15, 11, 10, 13, 14, 9, 12]$ and $r = [0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1]$ as an array representation of a weak-heap. Its binary tree equivalent is shown in Fig. 1.

Weak-heaps are state-of-the-art for sequential sorting. For $l = \lceil \log n \rceil$, the worst-case number of comparisons of weak-heap sort [5] is $nl - 2^l + n - 1 \leq n \log n + 0.09n$ [7]. An improvement sorts indexes in $n \log n - 0.91n$ comparisons [6].

An array-based solution is not an option for our studes. One main reason is that it is difficult to efficiently meld two structures.

Weak queues [9] contribute to the observation that perfect weak-heaps obey a one-to-one correspondence to heap-ordered binomial trees as featured in run-relaxed heaps (as well as in binomial queues, Fibonacci heaps, and others), and perfect weak-heaps (as featured in run-relaxed weak-queues). We obsereve that binomial tree *ranks* correspond to weak-heap *heights*. Rea-

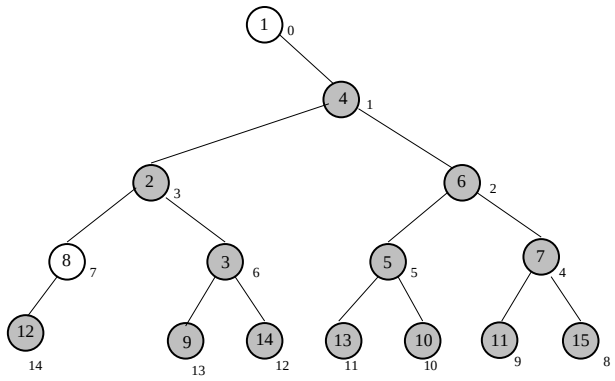

Figure 1: Example of a perfect weak-heap. Reflected nodes are shown in gray.

call that a binomial tree B_n is a tree of height n with 2^n nodes in total and $\binom{n}{i}$ nodes in depth i. The structure of B_n is found by unifying two structures B_{n-1}, where one is added as an additional successor to the second. As an unfortunate side effect, this increases the node branching factor considerably. Operations on perfect weak-heaps are slightly more flexible than on binomial trees. Moreover, binary trees provide a better space consumption, as only two links are necessary to cover the parent and successor relationship. A weak queue storing n elements is a collection of disjoint perfect weak-heaps based on the binary representation of $n = \sum_{i=0}^{\lfloor \log n-1 \rfloor} b_i 2^i$. In its basic form, a weak queue contains a perfect weak-heap H_i of size 2^i if and only if $b_i = 1$.

In *run-relaxed weak-queues* [9], the requirement of having exactly one perfect weak-heap of a given size is relaxed. An additional structure, called the *heap store*, maintains perfect weak-heaps of same height. At most two heaps per height suffice to efficiently realize injection and ejection of perfect weak-heaps. To meet the worst-cased complexity bounds, the join of two perfect weak-heaps of the same height is delayed, while maintaining the following structural property on the sequence of numbers of perfect weak heaps of the same height. The height sequence $(r_0, \ldots, r_k) \in \{0, 1, 2\}^{k+1}$ is regular, if any digit 2 is preceded by a digit 0, possibly having some digits 1 in between. A subsequence of the form (01^l2) is called a block. That is, every digit 2 must be part of a block, but there can be digits, 0s and 1s, that are not part of a block. For example, the height sequence (1011202012) contains three blocks. After the injection of a perfect weak heap, we join the first two of the same size, if there are any. They are found by scanning the height sequence. To

grant $O(1)$ access, a stack of pending joins, the *join schedule* implements the height sequence of pending joins. Then we insert the new weak-heap, while preserving the regularity of the height sequence. For ejection, the smallest weak heap is eliminated from the sequence and, if it forms a pair, the top of the join schedule is also removed.

The heap store can be implemented as a singly-linked list where each node, if it is (the first of) a 2, has a jump pointer to the next 2. This implementation is proposed in [1].

Resolving weak-heap order violations is delayed. The primary purpose of a *node store* is to keep track and reduce the number of potential violation nodes at which the key may be smaller than the key of the (binomial tree) parent. A node that is a potential violation node is said to be marked. A marked node is tough if it is the left child of its parent and also the parent is marked. A chain of consecutive tough nodes followed by a single non-tough marked node is called a run. All tough nodes of a run are called its members; the single non-tough marked node of that run is called its leader. A marked node that is neither a member nor a leader of a run is called a singleton. To summarize, we can divide the set of all nodes into four disjoint type categories: unmarked nodes, run members, run leaders, and singletons. A pair $(type, height)$ with *type* being either unmarked, member, leader, or singleton and *height* being a value in $\{0, 1, \ldots, \lfloor \log n \rfloor - 1\}$ denotes the *state* of a node, where the *height* of a node r is the height of the subtree rooted at r. Transformations induce a constant number of state transitions. A simple example of such a transformation is a join, where the height of the new root must be increased by one. Other operations (see Fig. 2) are cleaning, parent, sibling and pair transformations. A cleaning transformation rotates a marked left child to a marked right one, provided its neighbor and parent are unmarked. A parent transformation reduces the number of marked nodes or pushes the marking one level up. A sibling transformation reduces the markings by eliminating two markings in one level, while generating a new marking one level up. A pair transformation has a similar effect, but also operates on disconnected trees. These four primitive transformations are combined to a singleton or run transformation.

We briefly recall the two transformations from [9] as their application is crucial for our approach. In a *singleton transformation*, we assume that two marked nodes q and s do not have the same parent and that they are of the same height. Furthermore, we assume that q and s are the right children of their respective parents p and r, which both are unmarked. This

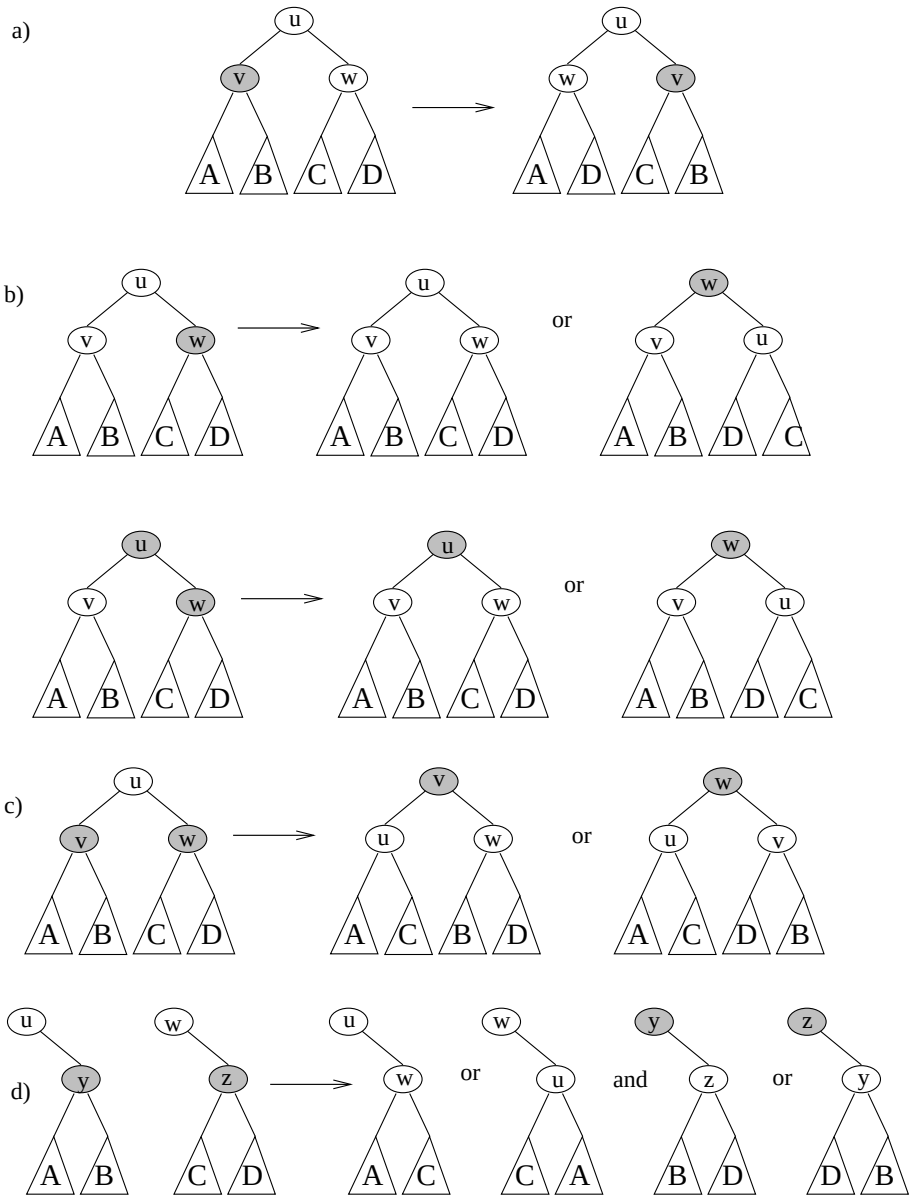

Figure 2: Primitives used in a λ-reduction: a) cleaning transformation, b) parent transformation, c) sibling transformation, and d) pair transformation.

transformation involves three steps. First, the subheaps rooted at p and r are split. Second, the produced subheaps rooted at p and r are joined and the resulting subheap is put in the place of the subheap originally rooted at p or r, depending on which becomes the root of the resulting subheap. Third, the two remaining subheaps rooted at q and s are joined and the resulting subheap is put in the place of the subheap originally rooted at p or r, depending on which is still unoccupied after the second step. If after the third step q or s becomes a root, the node is unmarked. By this transformation at least one marked node is eliminated.

The purpose of a *run transformation* is to move the two top-most marked nodes of a run upwards and at the same time remove at least one marking. Assume now that q is the leader of a run taken from the leader-object list and that r is the first member of that run. There are two cases depending on the position of q. In Case 1 q is a right child. Apply the parent transformation to q. If the number of marked nodes decreased, stop. Now the parent of r is unmarked. If the sibling of r is marked, apply the sibling transformation to r and its sibling, and stop. Thereafter, apply the parent transformation once or twice to r to reduce the number of marked nodes. In Case 2 q is a left child. If the sibling of q is marked, apply the sibling transformation to q and its sibling, and stop. Otherwise, apply the cleaning transformation to q, thereby making it a right child. Now the parent of r is unmarked. If the sibling of r is marked, apply the sibling transformation to r and its sibling, and stop. Otherwise, apply the cleaning transformation followed by the parent transformation to r. Now q and r are marked siblings with an unmarked parent; apply the sibling transformation to them to reduce the number of marked nodes.

The singleton transformation reduces the number of marking in a given level by 1, not generating a marking in the level above; or by 2, generating a marking in the level above. A similar statement is valid for run transformations, so that for both functions, the number of markings is reduced by at least 1 in constant amount of work. A *λ-reduction* is invoked once for each decrease-key and twice for each delete and delete-min operation. It calls either a singleton or a run transformation and bounds the number of marked nodes to at most $\lfloor \log n \rfloor - 1$.

In an implementation one would need a list of run leaders, a list of singleton leaders, for each singleton team a list of its members, and an array of pointers to the beginning of each singleton team list.

An implementation of run-relaxed weak queues is due to Rasmussen [19].

The code uses primitives of the standard template library STL . In the implementation the node store consists of different list items containing the type of the node marking, which can either be a *fellow*, a *chairman*, a *leader*, or a *member* of a run, where fellows and chairmen refine the concept of singletons. A fellow is a marked node, with an unmarked parent, if it is a left child. If more than one fellow has a certain height, one of them is elected as a chairman. The list of chairmen is required for a pair transformation. Nodes that are left children of a marked parent are members, while the parent of such runs is entitled the leader. The list of leaders is needed for a run transformation.

An implementation of the λ-reduction routine that realizes the above case study with these two lists is shown in Fig. 3. As the pseudo code transparently refers to the transformation routines and not to the actual marking and unmarking procedures underneath (that are called on-the-fly), given the four primitives displaying in Fig.2, the complex case study should be easy to walk through. For additional information on the implementation we kindly refer the reader to the original description in [19]

3 Rank-Relaxed Weak Queues

Rank-relaxed weak queues improve the run-relaxed weak queues by *eager λ-reductions*; yielding a more efficient node store. Instead of executing at most one reduction at a time, we eliminate all leaders and chairmen in one operation, thus performing transformations until both lists are empty. In such an iterated reduction, all runs are destroyed and no more than two singleton remain.

The modified implementation of procedure is shown in Fig. 4. The changes wrt. the implementation of Rasmussen in Fig. 3 are modereate. The most important change is the embedding of the original λ reduction in an additional loop (**while** ($leaders \cup chairmen \neq \emptyset$)). Moreover, we have exchanged the order of singleton and run transformations, so that run transformations are preferred. Last, but not leasta line that terminates a run transformation in case a singleton one becomes applicable.

Proposition 1 *The loop increases the worst-case time for reduce to $O(\log n)$.*

Proof. Eliminating all leaders and all singleton pairs may yield a ripple effect. As an example, consider that for each height we have already stored

Procedure λ-Reduce

```
if (leaders ≠ ∅)                                        ;; Leader exists on some level
  leader ← leaders.first ; leaderparent ← parent(leader) ;; Select leader and parent
  if (leader = leaderparent.right)                      ;; Leader is right child
    parenttrans(leaderparent)                           ;; Transform into left child
    if (¬marked(leaderparent) ∧ marked(leader))         ;; Parent also marked
      if (marked(leaderparent.left) siblingtrans(leaderparent); return
      parenttrans(leaderparent)
    if (marked(leaderparent,right)) parenttrans(leader)
  else                                                  ;; Leader is left child
    sibling ← leaderparent.right                        ;; Temporary variable
    if (marked(sibling)) siblingtrans(leaderparent); return
    cleaningtrans(leaderparent)                         ;; Toggle marking of leader's children
    if (marked(sibling.right)) siblingtrans(sibling); return
    cleaningtrans(sibling)                              ;; Toggle marking of sibling's children
    parenttrans(sibling)
    if (marked(leaderparent.left)) siblingtrans(leaderparent)
else if (chairmen ≠ ∅)                                  ;; Fellow pair on some level
  first ← chairmen.first; firstparent ← parent(first)
  if (firstparent.left = first and marked(firstparent.right) or   ;; 2 children
    firstparent.left ≠ first and marked(firstparent.left))        ;; ...
      siblingtrans(firstparent); return
  second ← chairmen.second; secondparent ← parent(second)
  if (secondparent.left = second and marked(secondparent.right) or ;; 2 children
    secondparent.left ≠ second and marked(secondparent.left))     ;; marked
      siblingtrans(secondparent); return
  if (firstparent.left = first) cleaningtrans(firstparent)  ;; Toggle children marking
  if (secondparent.left = second) cleaningtrans(secondparent)
  if (marked(firstparent) or root(firstparent))         ;; Parent also marked
    parenttrans(firstparent); return
  if (marked(secondparent) or root(secondparent))       ;; Parent also marked
    parenttrans(secondparent); return
    pairtrans(firstparent, secondparent)
```

Figure 3: Reducing number of marked nodes in a run-relaxed weak-queue.

```
Procedure Eager λ-Reduce
while (leaders ∪ chairmen ≠ ∅)                                   ;; New loop
  if (chairmen ≠ ∅)          ;; New ordering: first singletons, then run members
    first ← chairmen.first; firstparent ← parent(first)
    if (firstparent.left = first and marked(firstparent.right) or
      firstparent.left ≠ first and marked(firstparent.left))
        siblingtrans(firstparent); continue
    second ← chairmen.second; secondparent ← parent(second)
    if (secondparent.left = second and marked(secondparent.right) or
      secondparent.left ≠ second and marked(secondparent.left))
        siblingtrans(secondparent); continue
    if (firstparent.left = first) cleaningtrans(firstparent)
    if (secondparent.left = second) cleaningtrans(secondparent)
    if (marked(firstparent) or root(firstparent))
      parenttrans(firstparent); continue
    if (marked(secondparent) or root(secondparent))
      parenttrans(secondparent); continue
      pairtrans(firstparent, secondparent)
  else if (leaders ≠ ∅)
    leader ← leaders.first ; leaderparent ← parent(leader)
    if (leader = leaderparent.right)
      parenttrans(leaderparent)
      if (¬marked(leaderparent) ∧ marked(leader))
        if (marked(leaderparent.left) siblingtrans(leaderparent); continue
        parenttrans(leaderparent)
      if (marked(leaderparent,right)) parenttrans(leader)
    else
      sibling ← leaderparent.right
      if (marked(sibling)) siblingtrans(leaderparent); continue
      cleaningtrans(leaderparent)
      if (chairmen) continue                                     ;; New case
      if (marked(sibling.right)) siblingtrans(sibling); continue
      cleaningtrans(sibling)
      parenttrans(sibling)
      if (marked(leaderparent.left)) siblingtrans(leaderparent)
```

Figure 4: Reducing number of marked nodes in the rank-relaxed weak-queue.

one singleton. Adding another singleton at height 0 we have to perform a transformation, such that its elimination introduces the generation of another one at height 1, and so on, until we reach the root node. As there are at most $O(\log n)$ marked nodes in the store, and each applicable reduction eliminates one marked node, the worst-case of at most $O(\log n)$ steps is immediate. q.e.d.

Proposition 2 *The amortized costs for eager λ-reductions is constant.*

Proof. The critical observation is that with each reduction that generates a new marking at a certain depth, it eliminates more than one with smaller height value. If we assign a account for the constant amount of work needed for applying one reduction with each insertion of an element to the node store, these saved efforts can be exploited to cover the work needed for iterating the λ-reduction. q.e.d.

Proposition 3 *At any given time, there are at most four marked nodes of the same height.*

Proof. By the preference of singleton to run reductions at the time of each run reduction we have at most one marked singleton at each height. The critical case is that a cleaning transformation of the leader at height h to convert it to a left child, will disconnect it from its marked left child and can change it to a singleton, given that the left child of its destination is not marked, so that two singletons could appear in height $h + 1$. With the extra line in the code we participate from the fact that now a singleton transformation applies. As a result, at height $h + 1$ we grant space for a potential second fellow that is needed to finalize the transformation. All other cases ensure that at most one new marking is generated in height $h+1$, or $h + 2$. Continuing with singleton transformations we satisfy the invariant that after executing reduce, we have no run, and at most one singleton for each height. Moreover, in between two such iterated reductions for each height, at most 2 nodes are stored as a singleton. Similarly, at most 2 nodes appear as a member of a run at any given height. q.e.d.

The major gain of our approach of eager λ-reductions is that we can limit the number of markings at a given height. An efficient implementation avoids lists of marked nodes at each height.

Instead, we maintain marked nodes in a vector of quadruples; one for each level. The first 2 links are for runs, where a leader can be either of the 2 links. The second 2 links are for singletons.

As the leader and singleton lists are doubly-linked, we need 4 additinal links per level. At each node we maintain its height and its type Knowing the type, there are at most 2 positions at which a link to a node can be found, so that marking and unmarking remain in $O(1)$.

Maintaining pointers for the leaders and chairmen in doubly-linked list can be avoided by using a bit-vector set implementation. To find any member in the set we compute any (or the most significant) bit that is set to 1.

We additionally observe that a refined implementation can save 1 link per node. First of all, the height of a node (already present in the implementation of Rasmussen [19]) can be packed into a single byte. A closer look shows that its representation requires $\log \log n$ bits. This is much less than a link, since with six bits we can cope with heaps of $2^{64} = 1.844 \cdot 10^{19}$ nodes, which is sufficient for all practical purposes. Maintaining the type of a node requires two additional bits. This allows to pack the heights and the types into a single byle. More precisely, using a bit-array implementation (as available in C/C++), both informations still require only one byte per node in addition to successor and parent links. Hence, we save one link per node. Essentially, with our refinement, we require $2n + O(\log n)$ words and n bytes[1].

4 Experiments

We conducted experiments on 32-bit and 64-bit Linux PCs. We optimized the GCC binary (with flag `-O2`). As competitors to rank-relaxed weak queues, we chose Fibonacci heaps, and k-ary heaps from the LEDA library [16] (we used the publically available free 32-bit version for this purpose). We also adapted an efficient pairing heap implementation of Irit Katriel (based on work of [20]) that was used in [17].

Our space optimized implementation of rank-relaxed weak queues assumes that pointers to the elements for decreasing a key and deleting an element to modify are known. For a more flexible access, one would need a pointer/iterator to the elements to track their actual moves.

[1] As a time-space trade-off, the actual implementation does use left, right and parent pointers yielding a space requiremnt of $3n + O(\log n)$ words and n bytes.

	25,000,000 Integers			50,000,000 Integers		
	Ins	DecKey	DelMin	Ins	DecKey	DelMin
Rank-Rel.	0.048	0.223	4.38	0.049	0.223	5.09
Pairing	0.010	0.020	6.71	0.009	0.020	8.01
Fibonacci	0.062	0.116	6.98	-	-	-
k-ary	0.136	0.091	5.32	0.138	0.088	6.45

Table 1: Performances per operation for 32-bit priority queues.

4.1 32-Bit CPU

Our first set of experiments is conducted on a CPU of 3.2 GHz (AMD Athlon), with 2GB RAM. As this is a 32-bit machine, one can construct a 64K-sized table with 65,536 entries denoting the most significant bit of all 16-bit numbers.[2]

In Table 1 we measured the time for inserting n integers, randomly assigned to values from n to $2n-1$. Next, we decreased their value by 10 and continue deleting all n minima. CPU user times are provided in μ-seconds per operation. The bottom entries of the tables refer to results of LEDA, the top ones are alternative implementations. The lack of results in one row is due to the fact that Fibonacci heaps ran out of space.

In Table 2 we measured the time for inserting n strings, randomly assigned to ASCII values from $100n$ to $101n-1$ (which avoids underflows). Next, we decrease the key by a random value in $[0, n-1]$ and successively delete n minima. We see that Fibonacci and other heap implementations are inferior and pairing heaps are less effective on a larger set of elements.

4.2 64-Bit CPU

Our second set of experiments is conducted on one core of the Intel i7-920 CPU [3] with 2,66 GHz; and 12GB RAM. We used the same setting as before,

[2]There are some alternative options to quickly compute the most significant bit in an unsigned int `x`, mostly based on considering `x & -x`. Options to identify the position of the bit in the result include converting it to a float, a modulo computation, or a multiplication. We experimented with the latter and got slightly better results than with the 64K table.

[3]As the i7 architecture supports the population count (POPCNT) command in SSE4.2, but not LZCNT[4], we used a iterative approach to determine the most significant bit in the 64-bit vector, operating in $\log 64 = 6$ steps.

	5,000,000 Strings			20,000,000 Strings		
	Ins	DecKey	DelMin	Ins	DecKey	DelMin
Rank-Rel.	0.334	1.910	7.50	0.390	1.986	9.92
Pairing	0.262	1.002	8.99	0.302	1.043	12.51
Fibonacci	0.388	1.042	12.12	0.439	1.097	16.24
k-ary	0.730	1.404	11.07	0.809	1.494	14.35

Table 2: Performance of 32-bit priority queues on strings.

but limited our attention to the pairing heap and rank-relaxed weak queue implementations.

In Table 3 we scaled the experiment from 25 to 225 million integers, after which RAM became exhausted (for both pairing heaps and rank-relaxed weak queues). As before, pairing heaps are faster in performing insert and delete-key, but slower on delete-min. As the latter dominates the running times, for large number of elements, the performance of pairing heaps is inferior.

	Insert		Decrease-Key		Delete-Min	
Elements	Pairing	Rank-Rel.	Pairing	Rank-Rel.	Pairing	Rank-Rel.
25,000,000	0.009	0.031	0.012	0.516	2.351	2.301
50,000,000	0.008	0.031	0.012	0.531	2.854	2.652
75,000,000	0.008	0.031	0.012	0.546	3.204	2.865
100,000,000	0.009	0.031	0.012	0.544	3.453	3.014
125,000,000	0.008	0.030	0.012	0.532	3.681	3.119
150,000,000	0.009	0.030	0.012	0.548	3.854	3.222
175,000,000	0.009	0.030	0.012	0.548	4.009	3.302
200,000,000	0.008	0.030	0.012	0.549	4.148	3.398
225,000,000	0.008	0.030	0.012	0.553	4.249	3.446

Table 3: Performance of 64-bit priority queues.

Table 4 displays the total number of element comparisons for the experiment (including n inserts, n decrease-keys and n delete-mins). As expected, we see that rank-relaxed weak queues are clearly superior to pairing heaps.

Elements	Pairing	Rank-Rel.
25,000,000	1,117,868,044	969,285,934
50,000,000	2,341,540,962	2,014,524,909
75,000,000	3,604,956,553	3,091,500,382
100,000,000	4,894,251,738	4,178,886,163
125,000,000	6,202,768,881	5,279,851,817
150,000,000	7,526,500,750	6,408,502,237
175,000,000	8,863,051,572	7,524,243,367
200,000,000	10,210,578,621	8,656,277,841
225,000,000	11,567,978,225	9,796,509,293

Table 4: Number of comparisons for priority queues.

5 Conclusion, Discussion and Future Work

To push the practical effectiveness of priority queues we have improved the run-relaxed to rank-relaxed weak queues. They outperform Fibonacci heaps on moderate, and pairing heaps on a larger set of elements or on complex comparisons. The refinement we suggest relies on the property of constantly bounded buckets at each height level.

Our vision is a conceptually simple structure with good theoretical and practical performance for substituting Fibonacci and pairing heaps in text books and libraries. At this point we emphasize that although limited to programmers not only data structure performance, the empirical comparison is among these structures is rather fair, as all three implementations maintain memory for node allocation on their own. On the other hand, by using (resizable) arrays for this purpose, the implementations do affect their theoretical worst-case performance guarantees.

Despite the good practical performance, rank-relaxed weak queues are not a clear-cut winner compared to, e.g., pairing heaps. Consider a graph application where the priority queues are used. The running time of the resulting program is proportional to $m + n \log n$, where m is the number of edges and n is the number of nodes. When m is large, the first term dominates the overall costs. And the constant factor for this term is determined by decrease-key. The decrease-key operation is simply too slow for weak queues and its relatives to beat pairing heaps in this setting. The price we pay similar to rank-relaxed heaps [4] and in contrast to run-relaxed queues, is

that decrease-key now operates in amortized (instead of worst-case) constant time.

The apparent question is, if we can get back to worst-case constant time, while providing the effectiveness of constantly bounded lists. Moreover, applying λ-reduction eagerly may result in restructuring transformations that would not be necessary if delayed reductions were applied (e.g., singletons might be eliminated due to an unmarking before the corresponding singleton transformation applies). The increased speed, however, indicates that accelerated restructuring is more important than savings obtained by maintaining a slightly larger node store.

Due to the less complex structure, extensions to *two-tier* [11] (resp. *multipartite* [10]) priority queues with $\log n + O(\log \log n)$ (resp. $\log n + O(1)$) element comparisons for a delete might be easier to realize. However, we expect practical impact only for very complex keys, given that only $\log n$ element comparisons are currently needed to retrieve the minimum element. Other interesting structures to compare with in future are *quickheaps* [18] and *violation heaps* [8]. Moreover, a new variant of pairing heaps, assumed to be simpler, also builds on collections of binary trees [13].

Relaxed heaps structures have been shown to be efficient in the EREW PRAM model for shortest path, minimum spanning trees, minimum cost flow and other graph-related algorithms [4]. This suggests to study if they can effectively operate on graphics processing units in general proposed programming languages environments like NVIDIA's CUDA.

Acknowledgement Thanks to Peter Sanders for insightful comments on advanced bit hacks and new trends in processor architectures and together with his PhD. students Ospinov/Singler for the access to the advanced pairing heap implementation of Irit Katriel that has been used in [17]; Jyrki Katajainen for naming the data structure *rank-relaxed weak queues*, and to initiate a continuation of this research; Jens Rasmussen for providing access to the code; Martin Dietzfelbinger for proof reading. Last but not least, the author wants to thank Jan Vahrenhold, Susanne Albers and Petra Mutzel for the support that this research is worth continuing.

References

[1] G. S. Brodal. Fast meldable priority queues. In *WADS*, pages 282–290, 1995.

[2] G. S. Brodal. Worst-case efficient priority queues. In *Symposium on Discrete Algorithms*, pages 52–58, 1996.

[3] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[4] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31(11), 1988.

[5] R. D. Dutton. Weak-heap sort. *BIT*, 33:372–381, 1993.

[6] S. Edelkamp and P. Stiegeler. Implementing HEAPSORT with $n \log n - 0.9n$ and QUICKSORT with $n \log n + 0.2n$ comparisons. *ACM Journal of Experimental Algorithmics*, 10(5), 2002.

[7] S. Edelkamp and I. Wegener. On the performance of WEAK-HEAPSORT. In *STACS*, pages 254–266, 2000.

[8] A. Elmasry. Violation heaps: A better substitute for Fibonacci heaps. Research report, CoRR, 2008.

[9] A. Elmasry, C. Jensen, and J. Katajainen. Relaxed weak queues: An alternative to run-relaxed heaps. Technical Report CPH STL 2005-2, Department of Computing, University of Copenhagen, 2005.

[10] A. Elmasry, C. Jensen, and J. Katajainen. Multipartite priority queues. *ACM Trans. Algorithms*, 5(1):1–19, 2008.

[11] A. Elmasry, C. Jensen, and J. Katajainen. Two-tier relaxed heaps. *Acta Informatica*, 45(3):193–210, 2008.

[12] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithm. *Journal of the ACM*, 34(3):596–615, 1987.

[13] B. Haeupler, S. Sen, and R. E. Tarjan. Heaps simplified. Technical Report 0903.0116, arXiv.org, 2009. To appear with a different title in ESA-2009.

[14] H. Kaplan, N. Shafrir, and R.E. Tarjan. Meldable heaps and Boolean union-find. In *Symposium on Theory of Computing*, pages 573–582, 2002.

[15] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[16] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[17] V. Osipov, P. Sanders, and J. Singler. The Filter-Kruskal minimum spanning tree algorithm. In *ALENEX*, pages 52–61, 2009.

[18] R. Paredes. *Graphs for Metric Space Searching*. PhD thesis, University of Chile, 2008.

[19] J. Rasmussen. Implementing run-relaxed weak queues. Technical Report CPH STL 2008-1, Department of Computing, University of Copenhagen, 2005.

[20] J. T. Stasko and Jeffrey S. Vitter. Pairing heaps: Experiments and analysis. *Communications of the ACM*, 30(3):234–249, 1987.