Equivalence between Priority Queues and Sorting

MIKKEL THORUP

AT&T Labs–Research, Florham Park, New Jersey

Abstract. We present a general deterministic linear space reduction from priority queues to sorting implying that if we can sort up to n keys in $S(n)$ time per key, then there is a priority queue supporting delete and insert in $O(S(n))$ time and find-min in constant time. Conversely, a priority queue can trivially be used for sorting: first insert all keys to be sorted, then extract them in sorted order by repeatedly deleting the minimum. Asymptotically, this settles the complexity of priority queues in terms of that of sorting.

Previously, at SODA'96, such a result was presented by the author for the special case of monotone priority queues where the minimum is not allowed to decrease.

Besides nailing down the complexity of priority queues to that of sorting, and vice versa, our result yields several improved bounds for linear space integer priority queues with find-min in constant time:

Deterministically. We get $O(\log\log n)$ update time using a sorting algorithm of Han from STOC'02. This improves the $O((\log\log n)(\log\log\log n))$ update time of Han from SODA'01.

Randomized. We get $O(\sqrt{\log\log n})$ expected update time using a randomized sorting algorithm of Han and Thorup from FOCS'02. This improves the $O(\log\log n)$ expected update time of Thorup from SODA'96.

Deterministically in AC^0 (without multiplication). For any $\varepsilon > 0$, we get $O((\log\log n)^{1+\varepsilon})$ update time using an AC^0 sorting algorithm of Han and Thorup from FOCS'02. This improves the $O((\log\log n)^2)$ update time of Thorup from SODA'98.

Randomized in AC^0. We get $O(\log\log n)$ expected update time using a randomized AC^0 sorting algorithm of Thorup from SODA'97. This improves the $O((\log\log n)^{1+\varepsilon})$ expected update time of Thorup also from SODA'97.

The above bounds assume that each integer is stored in a single word and that word operations take unit time as in the word RAM.

Categories and Subject Descriptors: E.1 [**Data Structures**]: Lists, stacks, and queues; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and searching*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Priority queues, sorting

A preliminary version of this article appeared in *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, pp. 125–134.

Author's address: AT&T Labs–Research, Florham Park, NJ 07932, e-mail: mthorup@research.att.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2007 ACM 0004-5411/2007/12-ART28 $5.00 DOI 10.1145/1314690.1314692 http://doi.acm.org/10.1145/1314690.1314692

ACM Reference Format:
Thorup, M. 2007. Equivalence between priority queues and sorting. J. ACM 54, 6, Article 28 (December 2007), 27 pages. DOI = 10.1145/1314690.1314692 http://doi.acm.org/10.1145/1314690.1314692

1. *Introduction*

A priority queue is a data structure maintaining the minimum (or maximum) of a dynamic ordered set of keys. Updates may insert and delete keys. The minimum key is returned by a find-min operation. Formally, a key is a unique element with an associated value. Internally, the priority queue data structure may associate additional information such as pointers with a key. Such information is often used when a key is deleted.

A priority queue is one of the fundamental data structures for dynamic ordered sets [Cormen et al. 2001, pp. 138–141]. They are used both directly, for example, in operating systems to schedule jobs on a shared computer, and as subroutines in greedy algorithms such as Dijkstra's single source shortest path algorithm [Dijkstra 1959] (see also Cormen et al. [2001, pp. 595–599]) and Prim's minimum spanning tree algorithm [Prim 1957] (see also Cormen et al. [2001, pp. 570–573]).

In this article, we show that priority queues are computationally equivalent to sorting, that is, asymptotically, the per key cost of sorting is the update time of a priority queue. This equivalence is immediate if keys can only be accessed via comparisons, for then we know that sorting takes $\Theta(n \log n)$ time while a priority queue needs $\Theta(\log n)$ time per update.[1]

On computers, however, integers and floating point numbers are the most common ordered data types. For such data types, represented as lexicographically ordered bitstrings, we can apply classical non-comparison-based techniques such as radix sort and hashing. Historically, radix sort dates back at least to 1929 [Comrie 1929–1930] and hashing dates back at least to 1956 [Dumey 1956], whereas the focus on general comparison-based methods only dates back to 1959 [Ford and Johnson 1959]. For integers, we have known since 1990 [Fredman and Willard 1993] that the comparison based bounds were not tight, but we do not know the inherent complexity of integer sorting and priority queues. This is where our equivalence between the two problems becomes interesting.

Clearly, we can sort with a priority queue as a black box: first we insert all keys to be sorted in the priority queue, and then we extract them in sorted order by repeatedly deleting the minimum. We show here that the converse is also possible. As a reduction from priority queues to sorting, we present a priority queue that uses a sorting routine as a black box, supporting each update in essentially the same time as the sorting routine uses per key it is called on.

Our reduction is nontrivial, making multiple calls to the sorting black box with different subsets of the keys in the priority queue. The quality of the reduction depends on the power of the computational model it is implemented on.

---

[1] We follow here the convention that logarithms are base 2 unless otherwise stated, and that n is the number of keys involved.

1.1. THE WORD RAM. As our most powerful model of computation, we consider the word RAM that reflects what we can program in standard imperative programming languages such as C [Kernighan and Ritchie 1988]. The memory is divided into addressable words of bit length w. Addresses are themselves contained in words, so $w \geq \log n$. Moreover, we have a constant number of registers, each with capacity for one word. The basic instructions are: conditional jumps, direct and indirect addressing for loading and storing words in registers. Moreover we have some computational instructions, such as comparisons, addition, and multiplication, for manipulating words in registers. We are only considering instructions available in C, and refer to these as *standard* instructions. For contrast, on the low level, different processors have quite different computational instructions. Our RAM algorithms can be implemented in C-code which in turn can be compiled to run on all these processors.

The space complexity of our RAM algorithms is the maximal memory address used, and the time complexity is the number of instructions performed. All keys are assumed to be integers represented as binary strings, each fitting in one word. Integers occupying a constant number of words can be handled within the same asymptotic time bounds. An important feature of the RAM is that it can use keys to compute addresses, as opposed to comparison-based models of computation. This feature is commonly used in practice, for example in bucket or radix sort.

The restriction to integers is not as severe as it may seem. Floating point numbers, for example, are sorted correctly simply by perceiving their bit-string representation as representing an integer. Another example of the power of integer ordering is fractions of two one-word integers. Here, we get the right ordering if we carry out the division with floating point numbers with $2w$ bits of precision, and then just perceive the result as a double word integer. These examples illustrate how the integer ordering can capture many seemingly different types of orderings.

THEOREM 1.1. *If for some nondecreasing function S, we can sort up to n integer keys in $S(n)$ time per key, we can implement an integer priority queue supporting* find-min *in constant time, and updates, that is,* insert *and* delete*, in $O(S(n))$ time. Here n is the current number of keys in the queue. The reduction uses linear space so if the sorting uses linear space, so does the resulting priority queue. The reduction assumes a word RAM where each integer is contained in a single word.*

The author has previously presented a reduction of the above format [Thorup 2000], but where the resulting priority queue was monotone, meaning that the minimum had to be non-decreasing. This is an interesting special case, sufficing for applications such as Dijkstra's single source shortest path algorithm [Dijkstra 1959], but it doesn't work for other application such as Prim's minimum spanning tree algorithm [Prim 1957]. More importantly, monotonicity is not satisfied in direct applications of priority queues, say, for task scheduling in operating systems. Furthermore, the monotone priority queues from Thorup [2000] only had amortized time bounds as opposed to our worst-case bounds.

The main implication of Theorem 1.1 is that we can regard the complexity of basic integer priority queues as settled, and just focus on establishing the complexity of integer sorting. However, the reduction also provides some concrete new bounds for linear space integer priority queues with find-min in constant time:

Deterministically, we get $O(\log\log n)$ update time using a sorting algorithm of Han [2004]. This improves the $O((\log\log n)(\log\log\log n))$ update time of Han [2001]

Randomized, we get $O(\sqrt{\log\log n})$ expected update time using a randomized sorting algorithm of Han and Thorup [2002]. This improves $O(\log\log n)$ expected update time of Thorup [2000].

Our reduction is also interesting from a lower-bound perspective, for if we could prove any super-constant lower-bound for priority queues, then Theorem 1.2 would imply a corresponding super-linear lower bound for sorting.

Researchers have been successful in finding tight lower-bounds for many dynamic data structure problems. For example, for union-find, we know that n unions and m finds take $\Theta(m\alpha(m, n))$ time [Tarjan 1975; Fredman and Saks 1989], dynamic predecessor searching takes $\Theta(\sqrt{\log n/\log\log n})$ time per operation [Beame and Fich 2002; Andersson and Thorup 2007], and dynamic rank takes $\Theta(\log n/\log\log n)$ per operation [Fredman and Willard 1993; Fredman and Saks 1989]. From this perspective, one could hope for at least some lower-bound success for priority queues, and thereby for sorting.

Depending on mood, our equivalence can thus be viewed optimistically as opening a new venue for proving lower bounds for sorting via priority queue, or pessimistically, as saying that we can't hope to find super-constant lower-bounds for priority queues because we are clueless how to find super-linear lower-bounds for off-line problems like sorting.

The reduction of Theorem 1.1 uses Fredman and Willard's [1994] atomic heaps that besides being very complicated, use multiplication which is not in AC^0. This issue will be addressed in the next subsection.

1.2. A RECURSIVE REDUCTION FOR WEAK MACHINES. We will now introduce a somewhat weaker recursive reduction using a much less powerful model of computation.

The computational instructions will all be AC^0 operations, meaning that they can be implemented by an $w^{O(1)}$-sized constant depth circuit with $O(w)$ input and output bits. In the circuit we may have negation, and-gates, and or-gates with unbounded fan-in. Addition, shift, and bit-wise Boolean operations are all AC^0 operations. Multiplication, on the other hand, is not an AC^0 operation. As above, we restrict ourselves to standard AC^0 operations available in C.

The recursive reduction will only access keys via constant time comparisons and the sorting black-box. Thus, if the sorting black-box is comparison-based, then so is the resulting priority queue. We can thus get a comparison-based priority queue with $O(\log n)$ update time directly from an $O(n\log n)$ sorting routine.

Finally, the recursive reduction will not do any manipulation of addresses. This means that the reduction itself can be implemented on a simple pointer machine.

We are now ready to state the result provided by our recursive reduction.

THEOREM 1.2. *If, for some nondecreasing function S, we can sort up to n keys in $S(n)$ time per key, we can implement a priority queue supporting* find-min *in constant time and updates in $T(n)$ time where n is the current number of keys in the queue and $T(n)$ satisfies the recurrence:*

$$T(n) = O(S(n)) + T(O(\log n)). \tag{1}$$

The reduction can be implemented in linear space using standard AC^0 instructions only, and it works on a comparison-based pointer machine.

For contrast to (1), we note that the exponential priority queue trees of Andersson and Thorup [2007] imply that

$$T(n) = O(S(n)) + T(n^{4/5}). \tag{2}$$

As an example, with Thorup's $O(n \log\log n)$ randomized AC^0 sorting algorithm [Thorup 2002], we have $S(n) = O(\log\log n)$. Inserting this in (2), we would only get an expected update time of $O((\log\log n)^2)$ whereas we with (1) we get $O(\log\log n)$. In fact, with (1), we get an update time of $O(S(n))$ as long as S is a constantly iterated logarithm. If one day we discover how to sort in linear time with $S(n) = O(1)$, then (2) gives a priority queue update time of $O(\log\log n)$ whereas (1) gives an update time of $O(\log^* n)$. Recall here that using multiplication as in Theorem 1.1, we would get constant update time from linear time sorting.

Using Theorem 1.2, we get the following AC^0 bounds for linear space integer priority queues with find-min in constant time:

Deterministically in AC^0, for any constant $\varepsilon > 0$, we get $O((\log\log n)^{1+\varepsilon})$ update time using an AC^0 sorting algorithm of Han and Thorup [2002]. This improves the $O((\log\log n)^2)$ update time of Thorup [1998].

Randomized in AC^0 we get $O(\log\log n)$ expected update time using a randomized AC^0 sorting algorithm of Thorup [2002]. This improves the AC^0 expected update time of $O((\log\log n)^{1+\varepsilon})$ of Thorup [2002].

1.3. NONSTANDARD AC^0 OPERATIONS. So far, we have restricted ourselves to standard operations available in C. However, Andersson et al. [1999] have argued that if we are allowed to design our own AC^0 operations, then we can implement data structures such as Fredman and Willard's [1994] atomic heaps using AC^0 operations only. This implies that we can implement a priority queue with capacity for $\Theta(\sqrt{\log n})$ keys in constant time per operation. We get down to this base case if we apply the recurrence (1) from Theorem 1.2 twice, and then the overall update time is $O(S(n))$, just as in Theorem 1.1 but with nonstandard AC^0 operations instead of standard non-AC^0 operations such as multiplication. Thorup [2003a, 2003b] have shown that the nonstandard AC^0 operations needed for atomic heaps are often directly available in modern multimedia processors, hence that an AC^0 implementation with update time $O(S(n))$ may fully possible.

1.4. FURTHER REDUCTIONS FOR CONSTANT TIME INSERT AND MELD OPERATIONS. Some further reductions should be mentioned, allowing us to get even more powerful priority queues out of sorting. First, a simple observation of Alstrup et al. [2005] shows that if we have a priority queue doing all updates in $T(n)$ time, then we can actually implement a priority queue with insert in constant time and delete in $O(T(n))$ time. Using this, we can reduce the insert time to constant in all the above mentioned deterministic integer priority queues. In the randomized cases, the insert time becomes expected constant time.

Another reduction of Mendelson et al. [2006] shows that if we have a family of priority queues like those described above, then we can meld any two of them in constant amortized time. The time bounds for the other operations are not increased asymptotically, but the update times do become amortized.

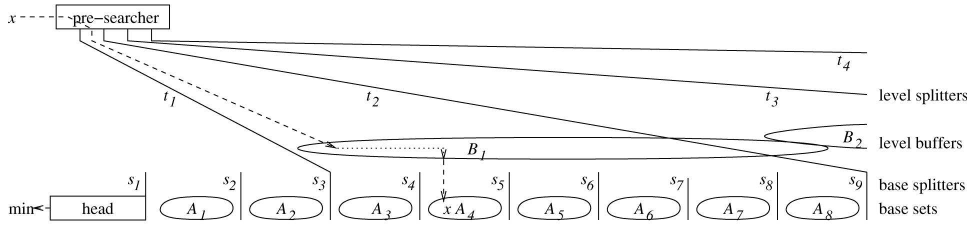

FIG. 1. The components of the priority queue. Note that the level buffers have overlapping ranges.

To illustrate the power of the chain of reductions, starting with the $O(n\sqrt{\log\log n})$ randomized sorting algorithm of Han and Thorup [2002], we get a family of priority queues supporting find-min in constant time, insert and meld in constant expected amortized time, and delete in $O(\sqrt{\log\log n})$ expected amortized time.

2. *Amortized Priority Queues*

In this section, we will prove a simpler amortized version first of Theorem 1.1, and later of Theorem 1.2. We will construct a priority queue with capacity for up to n keys. The actual number of keys is assumed to be $\Omega(n)$. If we can sort up to n keys in $S(n)$ time per key, the priority queue will support each update in $O(S(n))$ amortized time. To get a variable sized priority queue, we can rebuild the queue with twice the capacity whenever it gets full, and with half the capacity whenever it is $3/4$ empty. This does not affect the asymptotic time bounds. We assume that $n \geq 16$.

We will assume that all key values are distinct. This can always be obtained by viewing the unique identifier of a key as a secondary value breaking ties. We use $-\infty$ and ∞ to represent values defined as smaller respectively bigger than any possible key value.

2.1. THE COMPONENTS OF THE PRIORITY QUEUE. The components of our priority queue are illustrated in Figure 1. The keys in the priority queue will be organized around auxiliary values called *splitters* that are not themselves keys, though they are typically copies of current or former keys. Most keys are contained in logarithmic-sized disjoint *base sets* $A_0, A_1, \ldots, A_k$. The keys in the base sets are called *base keys*. The base sets are separated by *base splitters* $s_0, s_1, \ldots, s_k, s_{k+1}$ where $s_0 = -\infty$, $s_{k+1} = \infty$, and for $i = 1, \ldots, k$, $\max A_{i-1} < s_i \leq \min A_i$. The base sets are thus sorted relative to each other and the base splitters, but they are not required to be internally sorted. Concerning the exact size of the base sets, we define $\Phi = \Theta(\log n)$ as the smallest number bigger than $\log n$ that is, divisible by 12. We will have $\Phi/4 \leq |A_i| \leq \Phi$ for each $i < k$ and $|A_k| \leq \Phi$. We refer to the first base set A_0 as the *head*, and it should always contain the minimum key in the priority queue.

A logarithmic number of base splitters are promoted as *level splitters* $t_0, t_1, \ldots t_{\ell+1}$ where $\ell = \Theta(\log n)$, $t_0 = s_0 = -\infty$ and $t_{\ell+1} = s_{k+1} = \infty$. The level splitters are required to be increasing and exponentially spread with around $4^j\Phi$ base keys below t_j. More precisely, we have $\ell = \lfloor \log_4(n/\Phi) \rfloor$ and

(i) For $j = 1, \ldots, \ell$, there are more than $\frac{1}{2}4^j\Phi$ base keys below t_j if $t_j < \infty$.

(ii) For $j = 1, \ldots, \ell$, there are less than $\frac{7}{4}4^j\Phi$ keys below t_j.

Note that the lower bound in (i) only counts base keys while the upper bound in (ii) counts all keys. The two conditions together imply that the finite level splitters are strictly increasing. If the number of keys is smaller than n, we may have several infinite level splitters in the end.

In the text, we will sometimes refer to level splitter t_j as *the level j splitter*. This kind of terminology will become convenient when we later talk about other types of things associated with levels such as background processes in the worst-case construction.

For $j = 1, \ldots, \ell$, we have a *level j buffer B_j* with at most 4^j keys. We require:

(iii) All keys in B_j are in $[t_j, t_{j+2})$. Here $t_{\ell+2} = t_{\ell+1} = \infty$.

The keys in the buffers are called *buffer keys*. All keys in our priority queue are either base keys or buffer keys. The splitters are not themselves stored keys, though they may be copies of such keys.

LEMMA 2.1. *If (i) and (iii) are satisfied, the minimum key is in the head.*

PROOF. From (i) it follows that there are base keys below t_1 and from (iii) it follows that there are no buffer keys before t_1. Hence, the minimum must be found in the first base set, that is, the head. □

2.2. REPRESENTING THE COMPONENTS. To represent some of the above components, we will employ the *atomic heaps* of Fredman and Willard [1994]. These can support updates and searches in sets of $O(\log^2 n)$ size in constant time (in Fredman and Willard [1994], they are only described as having amortized time bounds, but as pointed out by Willard [2000], the time bounds can be made worst-case by background rebuilding). Here, by searching an element y in a set X, we mean finding the largest $x \in X$ with $x \leq y$. The atomic heap can also report its minimum key in constant time. The atomic heaps will later be bypassed in Sections 2.9 and 2.10.

The different components are stored as follows:

—Identifiers i of the base sets are stored in sorted order in a doubly linked list. These identifiers are not integers, but using the pointers from i, we can always find the previous and next identifier, denoted $i - 1$ and $i + 1$, respectively. With the identifier i we associate the base splitter s_i so in effect this gives us the sorted list of base splitters. We also store a pointer to the base set A_i described below.

—Each base set A_i is represented as a doubly linked list where each key knows the identifier i of the base set it is in. The list is not in sorted order. We also store a counter m_i for the number $|A_i|$ of keys in A_i. Recall from the beginning of the introduction that a key is a unique element with which we can store pointers such as list pointers. We can therefore delete and insert keys in the doubly-linked base sets in constant time.

—Besides being in a doubly linked list, the keys in the head A_0 are stored in an atomic heap that provides us the minimum key in constant time.

—Each buffer B_j is stored as a doubly linked list. It has a *level j counter q_j* which is at least as big as the number of elements in B_j.

—The level splitters are stored in an atomic heap, called the *pre-searcher*.

2.3. INITIALIZATION. Before describing any updates, we argue that it takes linear time to initialize our priority queue from a sorted list. First, we divide the sorted list into base sets of size $\Phi/2$, except the last base set which may be smaller. The keys in the first base set, the head, are placed in an atomic heap.

Next, we scan through the list of base splitters, accumulating counters m_i for each base set A_i passed, and identifying level splitters t_j with between $4^j\Phi$ and $(4^j + 1/2)\Phi$ base keys below.

We leave all buffers empty, and each level counter q_j is set to 0. Finally, we place the level splitters in the atomic heap of the pre-searcher. This completes the linear time initialization from a sorted list.

The linear time initialization is used for rebuilding and is needed for a later recursive variant avoiding the use of atomic heaps (cf., Section 2.10).

2.4. INITIAL UPDATES. We are now ready to start describing how updates are implemented. Recall that delete(x) provides a pointer to the place where x is stored. Ignoring sizes initially, we can implement delete(x) simply by deleting x from whatever base set or buffer it resides in. If x was in a base set A_i, we also decrement m_i.

As our initial implementation of insert(x), we first check whether x is smaller than the first level splitter t_1. In that case, we check against all base splitters below t_1 to find the right base set. By (ii), there are at most $\frac{7}{4}4\Phi$ base keys below t_1, hence at most $\frac{7}{4}4\Phi/(\Phi/4) = 28$ base splitters below t_1, so finding the right base set takes constant time. If $x \geq t_1$, we use the pre-searcher to find the level j such that $t_j \leq x < t_{j+1}$, place x in the level j buffer B_j, and increment the level j counter q_j by one.

After each insert or delete, we may have to move keys around so as to satisfy all the size constraints from Section 2.1. This is the topic of the next subsections.

2.5. FLUSHING BUFFERS. Recall that the level j counter q_j is at least the size of the level j buffer B_j. When q_j reaches 4^j, we *flush* buffer B_j as follows. First, we sort B_j in $O(|B_j|S(|B_j|)) = O(|B_j|S(n))$ time. Next, we merge the sorted B_j with the sorted list of base splitters. This will tell us which buffer keys belong in which base sets. Obviously, the merging stops when we reach a base splitter bigger than the biggest key from B_j. When done with the flushing, we set $q_j = 0$.

LEMMA 2.2. *If (ii) and (iii) are satisfied before we flush B_j, then merging takes $O(4^j)$ time.*

PROOF. We know that $|B_j| \leq 4^j$, so it suffices to prove that we only need to scan $O(4^j)$ base splitters before reaching one bigger than the biggest key in B_j. By (iii), we know that t_{j+2} is bigger than the biggest key in B_j, and by (ii), we have at most $\frac{7}{4}4^{j+2}\Phi$ base keys below t_{j+2}, hence at most $\frac{7}{4}4^{j+2}\Phi/(\Phi/4) = O(4^j)$ base splitters below t_{j+2}. $\square$

2.6. REBALANCING BASE SETS. As a result of a deletion or the flushing of a buffers, some base sets may get too big or too small. By joining and splitting neighboring base sets, it is standard to maintain base sets of size $\Theta(\Phi)$. Recall that Φ is divisible by 12. If a base set gets up to Φ keys, we split it around a median s. The median becomes a new base splitter. Conversely, if a base set gets down to

$\Phi/4$ keys, we consider any one of its neighbors. If they have at most $\frac{10}{12}\Phi$ keys in total, we join them removing the base splitter between them from the base splitter list. If they have more than $\frac{10}{12}\Phi$ keys, we join them and split them around their median. Their median replaces the previous base splitter between them.

We note that any base set coming out of a join or split has between $\frac{5}{12}\Phi$ and $\frac{10}{12}\Phi$ keys, so it takes at least $\frac{1}{6}\Phi$ new updates before the set will initiate a new join or split. The join or split is implemented in $O(\Phi)$ time, which is amortized as constant time per updates to the base sets. This time bound also suffices to update the atomic heap of the head base set if it participates in a join and split.

In our application, we have to be a little careful about a join for we should not join over a base splitter that is also a level splitter. However, from (i) and (ii), it follows that before an update we have more than Φ base keys below the first level splitter and between any two level splitters. If a base set gets down to $\Phi/4$ keys, it is because of a deletion that removed only one base key. Hence, there must exist a neighboring base set that is not on the other side of a level splitter. An exception is a base set after the last finite level splitter, but since we allowed the last base set to be arbitrarily small, it does not matter if it is alone.

Another issue is that the merging provides a whole list of keys to be inserted in a base set, and the above argument assumes that keys are inserted one at the time, splitting whenever a base set gets too big. We solve this by integration with the merging process as described below.

In the beginning of a merge step, we have a suffix of the sorted list of buffer keys and a suffix of the sorted list of base splitters. If the next buffer key is smaller than the next base splitter, we insert it in the preceding base set. If this base set gets too big, and gets split, we push the new base splitter in front of our remaining base splitters, and consider this base splitter for the next merge step.

For the analysis, each merge step will either insert the next buffer key in its base set, or it will skip the next base splitter. Obviously, we have at most $|B_j| \leq 4^j$ insertions of buffer keys. Moreover, when we skip a base splitter, we will never return to it, and we know that there is at least $\Phi/4$ base keys between consecutive base splitters. We can then apply the analysis from Lemma 2.2 including buffer keys from B_j.

2.7. UPDATING THE LEVEL SPLITTERS. The most interesting problem is how to update the level splitters so as to maintain $\Theta(4^j\Phi)$ keys below t_j as detailed in (i) and (ii). This requires that t_j is reset once for every $O(4^j\Phi)$ updates to the priority queue. In addition, we have to maintain (iii) stating that $B_j \subseteq [t_j, t_{j+2})$. Most of these conditions will be maintained implicitly via careful scheduling. However, to maintain $\min B_j \geq t_j$, we make sure to always flush B_j before changing t_j. The condition is then trivially satisfied because $B_j = \emptyset$.

We will thus link changes to t_j with the flushing of B_j, resetting t_j whenever we have flushed B_j and rebalanced the base sets. To reset t_j, we then simply scan through the base splitter list, adding up the counters for the base sets, until we reach a base splitter s_i with $4^j\Phi$ base keys below or $s_i = s_k = \infty$. We then make s_i the new level splitter t_j. Since our scanning adds one base set at the time, we may end up with up to $(4^j + 1)\Phi$ base keys below t_j.

LEMMA 2.3. *The scanning for the new level j splitter t_j takes $O(4^j)$ time.*

PROOF. As in the proof of Lemma 2.2, we note that we only spend constant time for each base set added below t_j, and we add up at most $4^j\Phi/(\Phi/4) = O(4^j)$ base sets. □

Note that above time bound for scanning on level j is dominated by the time it took to sort and flush B_j in Section 2.5.

We will now describe the system used to schedule the updates of level splitters so that t_j is reset once in every $O(4^j\Phi)$ updates. The scheduling is based on the level counters q_j. Currently, q_j is incremented every time a key is inserted in B_j. In addition, in round robin fashion, we are going to increment some level counter q_r in connection with each update to the priority queue. Afterward, the index r is incremented, or set it to 1 if $r = \ell$.

Whenever a level counter q_j reaches 4^j, we flush buffer B_j, set $q_j = 0$, rebalance the base sets, and scan to reset level splitter t_j as described above.

LEMMA 2.4. *On level j, we flush buffer B_j and reset level splitter t_j at least once in every $4^j(\Phi/2-1)$ updates.*

PROOF. We increment q_j at least once for every ℓ updates and $\ell = \lfloor \log_4(n/\Phi) \rfloor < (\log n)/2 - 1 \leq \Phi/2 - 1$. The inequalities use that $\Phi > \max\{4, \log n\}$. Moreover, we flush B_j when q_j reaches 4^j and reset q_j to 0. □

2.8. ANALYSIS. For the correctness of the above priority queue implementation, we have to prove the following lemma:

LEMMA 2.5. *The invariants (i), (ii), and (iii) restated below remain satisfied.*

(*i*) *For $j = 1, \ldots, \ell$, there are more than $\frac{1}{2}4^j\Phi$ base keys below t_j if $t_j < \infty$.*
(*ii*) *For $j = 1, \ldots, \ell$, there are less than $\frac{7}{4}4^j\Phi$ keys below t_j.*
(*iii*) *All keys in B_j are in $[t_j, t_{j+2})$. Here $t_{\ell+2} = t_{\ell+1} = \infty$.*

PROOF. First, we prove (i). When we sat t_j with a scan, we had scanned at least $4^j\Phi$ base keys below t_j. Base keys can only be lost because of deletes, and by Lemma 2.4, we have less than $4^j\Phi/2$ deletes before t_j is reset, so the number of base keys below t_j remains bigger than $4^j\Phi - 4^j\Phi/2 \geq \frac{1}{2}4^j\Phi$. This settles (i).

Next we show for all j that $t_j < t_{j+1}$ if $t_j < \infty$, that is, the finite level splitters are strictly increasing. This follows because the scan fixes t_j with at most $(4^j+1)\Phi$ base keys below, and above we proved that t_{j+1} always has at least $\frac{1}{2}4^{j+1}\Phi > (4^j+1)\Phi$ keys below.

For the lower bound in (iii), we note that when a key x is inserted in B_j, we have $x \in [t_j, t_{j+1})$. Moreover, B_j is flushed just before t_j is changed, so buffer keys in B_j remain at least t_j. This establishes the lower bound of (iii). Since the level splitters are strictly increasing, it follows that the buffers on level j or higher cannot contain keys below t_j.

We will now prove (ii). As mentioned above, we set t_j with at most $(4^j + 1)\Phi$ base keys below. Additional base keys below t_j can come either from the current buffers or from new inserts. We just proved that buffers on level j or higher cannot contain keys below t_j. Hence, we can bound the number of buffer keys

below t_j as

$$\sum_{h=1}^{j-1} |B_h| \leq \sum_{h=1}^{j-1} 4^h < \frac{1}{3} 4^j.$$

By Lemma 2.4, the maximal number of new inserts before t_j is reset is at most $4^j(\Phi/2 - 1)$. Thus, the maximal number of keys below t_j is bounded by

$$(4^j + 1)\Phi + \frac{1}{3} 4^j + 4^j(\Phi/2 - 1) < \left(\frac{3}{2} 4^j + 1\right) \Phi \leq \frac{7}{4} 4^j \Phi.$$

The last inequality uses that $j \geq 1$. This completes the proof of (i).

Finally, we prove the upper bound in (iii), that is, we show that no key x in B_j can be bigger than t_{j+2}. By definition $x < \infty$, and we know that there is always at least $\frac{1}{2} 4^{j+2}\Phi$ base keys below t_{j+2} if $t_{j+2} < \infty$. We want to show that there are less keys below x. We know that when x was inserted, it was smaller than t_{j+1}, and above we saw that t_{j+1} had at most $(\frac{3}{2} 4^{j+1} + 1)\Phi$ keys below. By Lemma 2.4, there are less than $4^j \Phi/2$ additional keys inserted before x is flushed from B_j, so the number of keys below x remains bounded by

$$\left(\frac{3}{2} 4^{j+1} + 1\right) \Phi + 4^j \Phi/2 \leq \left(\frac{13}{8} 4^{j+1} + 1\right) \Phi \leq \left(\frac{27}{16} 4^{j+1} \Phi\right).$$

This is strictly less than $\frac{1}{2} 4^{j+2}\Phi$, as desired, completing the proof of (iii). □

Finally, we conclude the following theorem:

THEOREM 2.6. *Using atomic heaps, the above priority queue implements* find-min *in constant time and* insert *and* delete *in $O(S(n))$ amortized time.*

PROOF. First, we note that the initial cost of the updates is constant. Also, since each key is inserted and deleted at most once from the base sets, the cost of rebalancing base sets is constant per key.

The remaining cost relates to the flushing of buffers and scanning for level splitters. When the level counter q_j reaches 4^j, we first sort the buffer in $O(|B_j| S(|B_j|)) = O(4^j S(n))$ time. Next we merge in $O(4^j)$ time by Lemma 2.2. Finally, we scan for the new t_j in $O(4^j)$ time by Lemma 2.3. This adds up to $O(4^j S(n))$ time. However, we also decreases the level j counter q_j from 4^j to 0, so we can charge the time spent as $O(S(n))$ per increment to a level counter. Since each update increments at most two level counters by one, this gives us an $O(S(n))$ amortized cost per update. □

With Theorem 2.6, we have now established Theorem 1.1 with amortized time bounds.

2.9. BYPASSING THE ATOMIC HEAP IN THE PRE-SEARCHER.

As a first step towards proving an amortized version of Theorem 1.2, we will now show how to bypass the atomic heap in the pre-searcher. The idea is very simple. Instead of the atomic heap, we use a pre-search buffer in the pre-searcher to collect a batch of up to Φ keys to be inserted. Once in every Φ updates, we sort the keys of this buffer together with the $\ell < \Phi$ level splitters. The sorting presorts the keys in the sense

of telling which level buffer each key belongs in. The sorting time is $O(\Phi \cdot S(\Phi))$, which amortized over Φ updates is $O(S(\Phi)) = O(S(n))$ per update. Thus we do not increase our overall update time.

To fix the details above, we need to prove the following robustness to the ordering of the level splitters:

LEMMA 2.7. *If we consider any period of at most Φ updates, then the initial value of t_j is less than the final value of t_{j+1}, and the final value of t_j is less than the initial value of t_{j+1}.*

PROOF. First, if $j = 0$, the result is true by definition since $t_0 = -\infty < t_1$. If $j > 0$, Lemma 2.5 (ii), there are always less than $\frac{7}{4} 4^j \Phi$ keys below t_j, and by Lemma 2.5 (i), there are always more than $\frac{1}{2} 4^{j+1} \Phi$ keys below t_{j+1}. Thus we have a difference of more than

$$\frac{1}{2} 4^{j+1} \Phi - \frac{7}{4} 4^j \Phi = \frac{1}{4} 4^j \Phi \geq \Phi$$

keys, which would have to be made up for by new updates. □

Now, suppose the sorting places x between t_j and t_{j+1}. We note that x could be anywhere in the batch of Φ keys to be inserted, one by one, and as we insert these other keys, the level splitters may move. However, by Lemma 2.7, x will remain between t_{j-1} and t_{j+2}. Hence, it follows that x belongs in B_{j-1}, B_j, or B_j, and we can determine which by comparing x to t_j and t_{j+1}. As a special case, if the sorting places x below t_2, we have to check if $x < t_1$, and if so, insert x directly in the relevant base set as described in the initial insert in Section 2.4.

A last concern is to prove that the minimum remains in the head as stated in Lemma 2.1. If a key x is placed in the pre-search buffer, it is bigger than t_1, and by (i), there are at least 2Φ keys below t_1. However, there can be at most Φ deletes before x is moved from the pre-search buffer. Hence, x cannot become the minimum before it is transferred from the pre-search buffer.

It is important to notice that the rest of the priority queue does not need to know that we have replaced the atomic heap with a pre-search buffer collecting inserts in batches. This may change the order in which updates arrive, but when they arrive, they are treated exactly as described in the preceding sections.

2.10. A RECURSIVE HEAD. The remaining atomic heap is the one representing the head base set. All we need from the head is access to its minimum key, so we can replace the atomic heap with a recursively defined priority queue with capacity Φ. Intuitively, this should give an amortized update time of

$$T(n) = O(S(n)) + T(\Phi),$$

implying the recurrence (1) from Theorem 1.1. However, we need to be a little careful with the rebalancing of base sets, for when we join or split the head, we cannot implement the keys moved as general recursive updates. The next lemma deals with this issue.

LEMMA 2.8. *Given a set of at most n keys, we can initialize our recursive priority queue in $O(n \cdot S(n))$ time and linear space, with no potential work to be done by future updates.*

PROOF. First, we sort all the keys in $O(n \cdot S(n))$. With sorted keys, we can initialize the priority queue in linear time as described in Section 2.3. The head, however, has to be initialized recursively from its sorted list of keys. Thus, with sorted keys, we get an initialization time of

$$Init(n) = O(n) + Init(\Phi) = O(n)$$

Finally, we note that no potential work is left for future updates. More precisely, the initialization from Section 2.3 gives each base set size $\Phi/2$, and when amortizing the rebalancing of base sets in Section 2.6, we only assume that new base sets start with sizes between $\frac{5}{12}\Phi$ and $\frac{10}{12}\Phi$. Also, the initialization starts with empty level buffers and all level counters $q_j = 0$, thus leaving us with no potential level work (cf. proof of Theorem 2.6). □

We now note that the head only needs to participate in a split or join if this is initiated by the head itself getting too many or too few keys. More precisely, by (i), there are always at least 2Φ keys below t_1. Moreover, the head has at most Φ keys, so if the second base set gets down to $\Phi/4$ keys, there must be a third base set before t_1 that the second base set can join with. Thus, from the analysis in Section 2.6, it follows that we have at least $\frac{1}{6}\Phi$ updates to the head between each split or join involving the head. To finish such a split or join, using Lemma 2.8, we initialize the priority queue in the new head in $O(\Phi \cdot S(\Phi))$ time. Since this cost is amortizing over $\frac{1}{6}\Phi$ updates to the head, it does not affect the overall $O(S(n))$ cost per update. We conclude that (1) is satisfied by our recursive amortized update time, matching the recurrence in Theorem 1.2.

Finally, concerning find-min, to find the minimum in constant time, it is important that this is not done recursively. We will therefore store a reference to the minimum key that can be returned in constant time. The recursive priority queue in the head will also have such a reference, and we can finish each update by copying this recursive reference. Now find-min is supported in constant time. This completes the proof of an amortized version of Theorem 1.2.

3. *Worst-Case Priority Queues*

In this section, we will de-amortize the priority queue from the previous section so as to prove Theorems 1.1 and 1.2. The challenge is that we cannot take a break, say, to flush a full buffer. Everything has to happen in parallel: the flushing of buffers from all levels, the scanning for new level splitters, and the rebalancing of base sets. All this requires careful coordination.

As in the amortized case, we will construct a priority queue with capacity for up to n keys. If we can sort up to n keys in $S(n)$ time per key, the new priority queue will support each update in $O(S(n))$ worst-case time. A variable-sized priority can then be obtained by standard background rebuilding. The exact details can be found in Thorup [2000, Lemma 2.1]. The update time of the variable-sized priority queue remains $O(S(n))$.

3.1. THE COMPONENTS OF THE WORST-CASE PRIORITY QUEUE.

The components of our worst-case priority queue are similar to those of the amortized one from Section 2.1, and we shall mostly focus on the differences.

We still have most keys in logarithmic-sized disjoint *base sets* $A_0, A_1, \ldots, A_k$ where $\Phi/4 \le |A_i| \le \Phi$ for $i < k$, and $|A_k| \le \Phi$, that is, the last base set may be smaller. This time we define Φ as the smallest number bigger than $2 \log n$ that is divisible by 84. This divisibility will be needed for our worst-case rebalancing of base sets in Section 3.5.

The base and level splitters are defined as before. That is, we have base splitters $s_0, s_1, \ldots, s_k, s_{k+1}$ where $s_0 = -\infty$, $s_{k+1} = \infty$, and for $i = 1, \ldots, k-1$, $\max A_{i-1} < s_i \le \min A_i$. Also, a logarithmic number of base splitters are promoted as level splitters $t_0, t_1, \ldots, t_{\ell+1}$ where $\ell = \Theta(\log n)$, $t_0 = s_0 = -\infty$ and $t_{\ell+1} = s_{k+1} = \infty$. This time, however, there will be around $4^{j+1}\Phi$ base keys below t_j. More precisely, we have $\ell = \lfloor \log_4(n/\Phi) \rfloor - 1$ and

(i) For $j = 1, \ldots, \ell$, there are more than $3.5 \cdot 4^j \Phi$ base keys below t_j if $t_j < \infty$.
(ii) For $j = 1, \ldots, \ell$, there are less than $8 \cdot 4^j \Phi$ keys below t_j.

By (i) and (ii), the finite level splitters are strictly increasing. We note that these invariants for the worst-case construction are slightly different from the corresponding invariants for the amortized construction.

For $j = 2, \ldots, \ell$, we have a level j buffer B_j which this time may have up to $3 \cdot 4^j$ keys. We require:

(iii) All keys in B_j are in $[t_{j-1}, t_{j+2})$. Here $t_{\ell+2} = t_{\ell+1} = \infty$.

Contrasting the amortized version, we note that the first buffer is on level 2, and that the buffer keys on level j may be smaller than t_j, though not smaller than t_{j-1}.

LEMMA 3.1. *If (i) and (iii) are satisfied, the minimum key is in the head.*

PROOF. Consider any buffer key $x \in B_j$. By (iii), we have that $x \ge t_{j-1}$. However, $j \ge 2$, so by (i), there are base keys below x. The minimum key in the priority queue must therefore be a base key in the first base set, the head. □

The above components are represented as in the amortized version (cf., Section 2.2) with the exception of the level buffers described below.

3.2. WORST-CASE LEVEL BUFFERS. The level j buffer is divided into three sections, each with at most 4^j keys. We have an *entrance*, a *sorter*, and a *merger*. The level j buffer works in periods of 4^j *level steps*. These correspond to the increments of the level j counter q_j in the amortized construction.

In the beginning of a period, the entrance is empty, the sorter has an unsorted set of at most 4^j keys, and the merger has a sorted list of at most 4^j keys. During the period, up to 4^j keys are inserted in the entrance. Meanwhile, the sorter sorts its list of keys, and the merger inserts all its keys in the base sets. At the end of the period, the entrance hands over its keys to the sorter, who in turns hands its sorted list of keys to the merger.

Note that it is by definition that a period has to be completed in 4^j level steps. In principle this could be done spending linear time in the last level step of a period. However, we are later going to show that we can implement each level step in $O(S(n))$ time and yet be sure to complete a period in time. Trivially, this is the case for sorter that has to sort at most 4^j keys, and the entrance is just receiving keys. However, the merger is more tricky as it will have to synchronize with other processes.

From (iii), we know that all buffer keys in B_j are in (t_{j-1}, t_{j+2}). Hence, the merger can start from t_{j-1} in the base splitter list, and it will never reach t_{j+2}.

The merging is divided into individual merge steps, and we may have multiple of these in a level step. A merge step is atomic in the sense that no other action takes place in the priority queue while a merge step is being performed. At the beginning of a merge step, the level j merger will be at some base splitter s_i and have some next buffer key x to insert in the base sets. It is guaranteed that $x \geq s_{i-1}$. If $x < s_i$, the merger inserts x in the base set A_{i-1} preceding s_i. It will consider the buffer key after x in the next merge step. If $s_i \leq x$, the merger moves to the next base splitter s_{i+1}.

If the merge step inserts a key in a base set, there may be a subsequent rebalancing between base sets, and if a join removes a base splitter, then any merger at that base splitter will have to be moved. In that context, we shall need the following invariant:

(iv) The level j merger remains strictly between t_{j-2} and t_{j+2}.

The invariant implies that there can be at most 4 mergers at any given base splitter $s_i \in [t_j, t_{j+1})$; namely those from levels $j-1$, j, $j+1$, and $j+2$.

3.3. WORST-CASE LEVEL SCANNERS. In the amortized construction, to reset level j splitter t_j, we scanned the base splitter list, adding up the counters of the base sets until we found an appropriate base splitter s_i with $\Phi 4^j$ base keys below (cf., Section 2.7). In our worst-case construction, this scanning is a background process referred to as the *level j scanner*.

The level j scanner works in the same period of 4^j level steps as the level j buffer. By the end of each period, it will have assigned a new value to the level splitter t_j.

Like the level j merger, the level j scanner will start at level splitter t_{j-1} in the base splitter list. The scanner uses a counter n_j to count passed base keys. It ignores all base keys preceding t_{j-1}, thus start with $n_j = 0$.

In each individual scan step, the level j scanner will start at some base splitter s_i. It will move to s_{i+1}, passing A_i, and adding $m_i = |A_i|$ to its counter n_j. If n_j reaches $4^j\Phi$, we would like to promote s_{i+1} to be the new level j splitter t_j. However, as we shall see in Section 3.5, we may have to promote a subsequent base splitter instead. We shall need the following invariant:

(v) The level j scanner will always be strictly between t_{j-2} and t_{j+1}.

This invariant implies that there can never be more than 3 level scanners at any base splitter.

3.4. INITIAL UPDATES. As in the amortized construction, initially, we want to implement delete(x) simply by deleting x from whatever base set or buffer it resides in. However, if x is in the middle of being sorted, we may have to wait taking it out until the sorting is completed. Removing all such elements from a sorted list takes linear time, so this does not affect the overall sorting time. We note that between merge steps, it only helps the merger if the next element to be inserted is deleted.

As our initial implementation of insert(x), we first check if x is smaller than the second level splitter t_2. In that case, we check against all base splitters below t_2 to find the right base set. By (i), there are at most $3.5 \cdot 4^2\Phi$ base keys below t_2, hence at most $3.5 \cdot 4^2\Phi/(\Phi/4) = 224$ base splitters below t_2, so finding the correct base

set takes constant time. If $x \geq t_2$, we use the pre-searcher to find the level j such that $t_j \leq x < t_{j+1}$, and place x in the entrance of the level j buffer B_j. Finally, we perform a level step in the level j period.

In addition, each update performs a level step on one level r in a round robin fashion. That is, after the level step, we increment r, or set it to 1 if $r = \ell$. Corresponding to Lemma 2.4, we get the following lemma:

LEMMA 3.2. *Each level j period is completed in fewer than $4^j \Phi/4$ updates.*

PROOF. A level j period takes 4^j steps on level j, and does a level j step at least once for every ℓ updates. Moreover $\ell = \lfloor \log_4(n/\Phi) \rfloor < (\log n)/2 \leq \Phi/4$ since $\Phi \geq 2 \log n$. □

Note that the Φ in Lemma 3.2 is twice as big as it was in Lemma 2.4.

3.5. REBALANCING BASE SETS. We now need to balance the base sets in constant worst-case time per update to the base sets. In principle, this is standard using background rebuilding as described in Willard and Lueker [1985]. However, the details are rather messy. Instead we will use a general join-split protocol from Andersson and Thorup [2007] to keep the sizes of the base sets between $\Phi/4$ and Φ. The last base set is allowed to be smaller, and we will play a special role as described in Section 3.5.2.

3.5.1. *A Protocol for a Weight Balancing Game.* The protocol from Andersson and Thorup [2007, Section 3.3.1] is defined as a combinatorial game on a family of lists of non-negative integer weights. First, we describe the game on a single such list. The players are *us* against an adversary. Our goal is to maintain balance in the sense that for some parameter b, called the *latency*, we want all weights to be of size $\Theta(b)$. More precisely, for some concrete constants $c_1 < c_2 < c_3 < c_4$, the list is *balanced* if all weights are in the interval $[c_1 b, c_4 b]$. Initially, the list is *neutral* in the sense that all weights are in the smaller interval $[c_2 b, c_3 b]$.

The goal of the adversary is to upset balance, pushing some weight outside $[c_1 b, c_4 b]$. The adversary has the first turn, and when it is his turn, he can change the value of an arbitrary weight by 1. He has the restriction that he may not take the sum of all weights in the list down below $c_2 b$.

After the adversary has changed a weight, we get to work locally on balance. We may join neighboring weights w_1 and w_2 into a single weight $w = w_1 + w_2$, or split a weight w into w_1 and w_2. The adversary decides on the values of w_1 and w_2, but he must fulfill that $w_1 + w_2 = w$ and that $|w_1 - w_2| \leq \Delta b$. Here Δ is called the *split error*.

Each split or join is followed by a postprocessing requiring b steps during which the involved weights may not participate in any other split or join. When it is our turn, we get to perform one such step. Moreover, the step may only be performed *locally* in the sense of being on a postprocessing involving the weight last changed by the adversary, or a neighboring weight. The idea of the postprocessing steps is to give the adversary a break to perform some computations while knowing that the weights are not involved in other splits or joins.

As described in Andersson and Thorup [2007, Section 3.3.3], we will allow the adversary to be a bit lazy about the splitting. Instead of performing the split as soon as it is requested by the protocol, the adversary can perform the actual split during any step of the split postprocessing. The point is to allow the adversary to use some

of the postprocessing steps to identify a good split. The requirements of the actual split are unchanged: if the weight is w at the step where the actual split is about to be performed, the resulting weights w_1 and w_2 should satisfy that $w_1 + w_2 = w$ and that $|w_1 - w_2| \leq \Delta b$.

We will now extend the game to a family of lists. The adversary wins if he manages to unbalance *some* list in the family. In addition to the weight updates, he can now perform list operations: He may add or remove a neutral list, that is, a list with all weights in $[c_2 b, c_3 b]$. He may also cut or concatenate lists. When it is his turn, he can first perform as many list operations as he desires, and then a single weight update in one of the lists. When it is our turn, we get to work locally on balance around his weight update as described in the single list game.

The adversary may not choose a cut arbitrarily. He may not cut between two weights coming out of a split until the split postprocessing is completed. Moreover, we have the power to *tie* some neighboring weights, and the adversary cannot cut ties. We ourselves can only tie and untie the last weight updated by the adversary or one of its neighbors.

An *uncuttable* segment is a segment of weights that the adversary may not cut. Our tying of weights is restricted in that for some constant $c_5 \geq c_4$, we have to make sure that the total weight of any uncuttable segment is at most $c_5 b$. In particular, this implies that no uncuttable segment contains more than $c_5/c_1 = O(1)$ weights.

A *protocol* for the balancing game is a winning strategy against any adversary. Also, to be a protocol it should be efficient identifying its moves in constant time.

LEMMA 3. 3 [ANDERSSON AND THORUP 2007, PROPOSITION 3.9 AND REMARK 3.11]. *For any latency b, split error Δ, and number μ, we have a protocol for the balancing game with $c_1 = \mu b$, $c_2 = (\mu + 3)b + 1$, $c_3 = (2\mu + \Delta + 9)b - 1$, $c_4 = (3\mu + \Delta + 14)b$, and $c_5 = (5\mu + \Delta + 20)b$. In particular, for $\Delta = 7$ and $\mu = 21$, we assume that neutral lists have weights strictly between $24b$ and $58b$, and that the total weight of any list is more than $24b$. Then, we guarantee that all weights stay between $21b$ and $84b$ and that the maximum uncuttable segment is of size at most $132b$.*

3.5.2. *Applying the Protocol.* In our priority queue construction, the weights of in the above game will be the sizes of the base sets, possibly excluding the last weight which may be smaller. Each list will be the sequence of base sets between two level splitters. We will use the protocol from Lemma 3.3 with $b = \Phi/84$, $\Delta = 7$ and $\mu = 21$. Recall here that Φ was chosen to be divisible by 84. We will manage the priority queue to conform with the rules of the adversary. The protocol then ascertains that the base set sizes remain between $\Phi/4$ and Φ.

The role of the uncuttable segments is that if two neighboring base sets are in the same uncuttable segment, then the base splitter between them cannot be promoted to a level splitter. However, the protocol guarantees that an uncuttable segment has weight at most $132b$. In addition, to avoid creating lists of close to the minimal legal size $24b$, we will not cut off an end of weight less than $25b$. In effect this gives us uncuttable ends of weight at most $157b < 2\Phi$. Thus, we can never have more than 2Φ base keys between consecutive promotable splitters. This has the following effect on the completion of a scan from Section 3.3: when the counter n_j of the level j scanner reach the target of $4^{j+1}\Phi$ passed base keys, it moves to the first promotable base splitter, and promotes it to be the new level splitter t_j. Consequently, we end up with $4^j\Phi \leq n_j \leq 4^j\Phi + 2\Phi$. Moreover, since base

sets have at least $\Phi/4$ keys, we have to pass less than $2\Phi/(\Phi/4) = 8$ extra base splitters to reach a promotable one, so this completion of a scan takes constant time. As a special case, we note that the last base set may have less than $\Phi/4$ keys, but then it cannot be in the game, and then the base splitter in front of it is promotable.

Below, we will first describe our implementations of the joins and splits requested by the protocol. At the end, we will consider the issues of neutral lists of minimal size.

3.5.2.1. IMPLEMENTING JOIN. Suppose the protocol request us to join two neighboring base sets A and A', removing the base splitter s between them. Let s' be the base splitter after A'. Then, s' will be the base splitter after the joined base set $A \cup A'$. However, before removing s, we move all level mergers and scanners at s to s'. By (iv) and (v) there can be at most 4 mergers and 3 scanners at s, and we will deal with each of them in constant time: if a merger is at s, we simply move it to s', and if a level j scanner is at s, we move it to s' adding $|A'|$ to its counter n_j for the number of base keys passed. If $n_j \geq 4^{j+1}\Phi$, we complete the scan as described above, finding the first unprotected base splitter and promoting it to become the new level splitter t_j. Having moved all mergers and scanners from s, we remove s from the base splitter list.

So far, we have only spent constant time on the join, and this time bound also suffices if we want to concatenate neighboring base sets. However, if A is the head with its atomic heap, we need to insert the keys from A' in its atomic heap. The protocol ascertains that it takes $\Phi/84$ updates to the head or its neighbors before the head gets involved in another join or split. Hence, it suffices to insert 84 keys in the atomic heap during each of these base updates. This is still only constant time per base update.

3.5.2.2. IMPLEMENTING SPLIT. The split is essentially symmetric to a join. We want to split a base set A, creating a new base splitter s. We have $\Phi/84$ base updates available for the split. The join-split protocol that we use from Lemma 3.3 requires that the difference in size between the two resulting sets is at most $\Phi/12$. To implement the split, we first scan A to create a copy. The copy is not expected to be current, but it will contain every key that stays in A for the whole split postprocessing. We now find a median s of the copy. We will use s as our new base splitter, so we scan the current set A again, moving all keys as big as s to a new base set A'. All keys arriving to A after s is chosen are compared with s and if bigger, placed in A'. If A is the head, we remove all the moved keys from its atomic heap. All this takes $O(\Phi)$ time, so it can be implemented spending constant time in each of the $\Phi/84$ base updates available for the split postprocessing. We know that s is the median of a copy of A, and since all keys are assumed distinct, we know that it divides the copy with at most one in difference between the sizes of the two sides. Each of the $\Phi/84$ updates done during the split can increase the difference by one, so the maximal difference is $1 + \Phi/84 < \Phi/12$, as required.

Let s' be the base splitter that was after A before the split. After the split s' will be after A', and we want to insert s in front of s' in the base splitter list. This does not affect any scanner, but if there is a merger at s', we have to check if the next update key x precedes s, and if so, move the merger back to s. By (iv), there were at most 4 mergers at s' so this last part takes constant time.

3.5.2.3. NEUTRAL LISTS OF MINIMAL SIZE. Recall from Lemma 3.3 that each list in the game needs to has to have weight at least $24b = \frac{24}{84}\Phi$ and that we can only involve a new list in the balancing game if it is neutral in the sense that each weight is between $24b = \frac{24}{84}\Phi$ and $58b = \frac{58}{84}\Phi$. It is to satisfy these conditions that we have made it optional to include the last weight w_k in the game. Recall that this last weight is the only one that is allowed to be less than $\Phi/4$.

As a first simple case, if the total number of keys is below $\frac{25}{84}\Phi$, they will all be in the head base set. We will then not play the balancing game at all. The balancing game starts with a single list with one weight; namely that of the head when its size reaches $\frac{24}{84}\Phi$. The balancing game is abandoned if we at some later stage get down below $\frac{24}{84}\Phi$ keys.

In general, we will maintain the invariant that the last weight w_k is in the game if $w_k > \frac{25}{84}\Phi$. Note that if w_k is in the game, then like any other weight in the game, it is at least $\Phi/4$.

The invariants (i) and (ii) ensure that we have more than 24Φ base keys between any two finite level splitters, so it is only the list after the last finite level splitter that may end up having weight as low as $\frac{24}{84}\Phi$, and hence have to be taken out of the game. Since we never cut off an end of size below $\frac{25}{84}\Phi$, the small last list must be a result of deletions, each decreasing a weight by one. Since each weight in the game has weight at least $\Phi/4$, the small list must consist of a single weight. If this single weight gets down to $\frac{24}{84}\Phi$, we take the small list out of the game. If the single weight was the very last weight w_k, we are done. However, it may be that we already had a last isolated weight w_k, and now we have also isolated w_{k-1}. We then join these two weights into a single weight w as described in Section 3.5.2. The join takes constant time because it does not involve the head which we treated as a special case above. Now w comprises all the weight after the last finite level splitter. Our invariant for the last isolated weight states that $w_k \leq \frac{24}{84}\Phi$ and we know that $w_{k-1} = \frac{24}{84}\Phi$, so $w = w_{k-1} + w_k \leq \frac{48}{84}\Phi$. We now have two cases. If $w < \frac{25}{84}\Phi$, we leave it as a single isolated last weight. Then, we have no list in the game after the last level splitter. On the other hand, if $w \geq \frac{25}{84}\Phi$, we use it to create a new neutral singleton list that joins the game as the list after the last level splitter.

The only case left to consider is when we have an isolated last weight w_k which is incremented to $\frac{25}{84}\Phi$. We then use it to create a new neutral singleton list that joins the game, and concatenate with the current last list in the game.

3.6. ANALYSIS. For the correctness of the above priority queue implementation, we have to prove the following lemma verifying our worst-case invariants:

LEMMA 3.3. *The invariants* (*i*), (*ii*), (*iii*), (*iv*) *and* (*v*) *restated below remain satisfied.*

(*i*) *For* $j = 1, \ldots, \ell$, *there are more than* $3.5 \cdot 4^j \Phi$ *base keys below* t_j *if* $t_j < \infty$.
(*ii*) *For* $j = 1, \ldots, \ell$, *there are less* $8 \cdot 4^j \Phi$ *keys below* t_j.
(*iii*) *All keys in* B_j *are in* (t_{j-1}, t_{j+2}). *Here* $t_{\ell+2} = t_{\ell+1} = \infty$.
(*iv*) *The level* j *merger remains strictly between* t_{j-1} *and* t_{j+2}.
(*v*) *The level* j *scanner remains strictly between* t_{j-1} *and* t_{j+1}.

PROOF. Internal to the proof, we will introduce an extra, more technical invariant:

(vi) If x is in a buffer B_{j^*} when we start the scan for t_j, $j < j^*$, then the scan for t_j cannot reach x.

We will prove that there cannot be a first violation to any of the invariants. Hence, we will assume that no invariant has been violated in the past.

First, we prove (i). Consider some instance of t_j. We know that when level j scanned for t_j, it passed at least $4^{j+1}\Phi$ base keys. Some of these may be deleted in the period of the scan for t_j or in the period in which t_j is active before it is replaced by a new t_j. By Lemma 3.2, we have fewer than $2 \cdot 4^j \Phi/4 = 4^j \Phi/2$ deletes in these two periods. Thus, while t_j is active, there are more than $4^{j+1}\Phi - 4^j\Phi/2 \geq 3.5 \cdot 4^j \Phi$ base keys below t_j. This completes the proof of (i).

Next, we prove (vi). When x was inserted, we had $x \geq t_{j^*}$, so by (i), there were at least $3.5 \cdot 4^{j^*}\Phi$ base keys below x. We know that x can reside in B_{j^*} for at most 3 level j^* periods. Also, the scan for t_j lasts one level j period. Hence, by Lemma 3.2, we lose at most $3 \cdot 4^{j^*}\Phi/4 + 4^j\Phi/4$ keys between the insertion of x in B_{j^*} and the end of the scan for t_j. Thus, we have at least

$$3.5 \cdot 4^{j^*}\Phi - (3 \cdot 4^{j^*}\Phi/4 + 4^j\Phi/4) = 2.75 \cdot 4^{j^*}\Phi - 4^j\Phi/4 \geq 10.75 \cdot 4^j\Phi$$

base keys below x that have been alive throughout the scan for t_j.

However, by (ii), there are at most $8 \cdot 4^{j-1}\Phi$ base keys below t_{j-1} when we start the scan, and the scan itself passes at most $4^{j+1}\Phi + 2\Phi$ base keys. Since

$$8 \cdot 4^{j-1}\Phi + 4^{j+1}\Phi + 2\Phi \leq 6.5 \cdot 4^j\Phi < 8.25 \cdot 4^j\Phi,$$

we conclude that t_j cannot reach x. This completes the proof of (vi).

To prove (ii), we count the keys below an active level splitter t_j. When we say that keys "started" somewhere, we refer to the state when we started scanning for t_j, which is the period before t_j became active. Any key below t_j while active must be in at least one of the following groups:

—Keys starting below t_{j-1} of which there are at most $8 \cdot 4^{j-1}\Phi$ by (ii).
—Live base keys passed in the scan for t_j. By definition of a scan, there can be at most $(4^{j+1} + 2)\Phi$ such keys.
—Keys starting in one of the buffers. We know from (vi) that we only need to consider buffers at level j or lower, and the total number of keys starting in those buffers is bounded by

$$\sum_{h=2}^{j} |B_h| \leq \sum_{h=2}^{j}(3 \cdot 4^h) < \frac{4}{3} \cdot 3 \cdot 4^j = 4^{j+1}.$$

—Keys inserted since the scan started, either in the period of the scan, or the period where t_j is active. By Lemma 3.2, there are at most $4^j\Phi/2$ such keys.

Adding up, we conclude that there are at most

$$\begin{aligned} &8 \cdot 4^{j-1}\Phi + (4^{j+1} + 2)\Phi + 4^{j+1} + 4^j\Phi/2 \\ &\leq\ 7 \cdot 4^j\Phi + 4^{j+1} \\ &<\ 8 \cdot 4^j\Phi \end{aligned}$$

keys below t_j. The last inequality uses that $\Phi > 4$. This completes the proof of (ii).

We will now prove (iii). We consider $x \in B_j$. First we prove $x > t_{j-1}$. By (ii), the number of keys below t_{j-1} is less than $8 \cdot 4^{j-1}\Phi = 2 \cdot 4^j\Phi$. However, when x was inserted in B_j, by (i), there where at least $3.5 \cdot 4^j\Phi$ base keys below $t_j \leq x$. That was less than three periods ago, so by Lemma 3.2, at most $3 \cdot 4^j\Phi/4$ of these keys have been deleted. Since

$$3.5 \cdot 4^j\Phi - 3 \cdot 4^j\Phi/4 = 2.75 \cdot 4^j\Phi > 2 \cdot 4^j\Phi,$$

we conclude that $x > t_{j-1}$. Next we prove $x < t_{j+1}$ with a very similar calculation. By (i), the number of base keys below t_{j+2} is more than $3.5 \cdot 4^{j+2}\Phi$. However, when x was inserted in B_j, by (ii), there where less than $8 \cdot 4^{j+1}\Phi$ keys below $t_{j+1} \geq x$. However, that is less than three periods ago, so by Lemma 3.2, at most $3 \cdot 4^j\Phi/4$ more keys can have been inserted below x. Since

$$8 \cdot 4^{j+1}\Phi + 3 \cdot 4^j\Phi/4 < 3.5 \cdot 4^{j+2}\Phi,$$

we conclude that $x < t_{j+2}$.

Finally, we prove (iv) and (v). To see that the level j scanner and merger cannot by "overtaken" by t_{j-2}, we note that they started at a value $x = t_{j-1}$. The proof of (iii) above really showed that $t_{j-2} < x$ for at least three periods. However, the scan or merge only lasts a single period, and the value can only increase. Hence, it follows that both scanner and merger remain above t_{j-2}.

The merger cannot reach t_{j+2} as it stops as soon as it has placed its last buffer key x, and by (iii), we have $x < t_{j+2}$. This completes the proof of (iv).

Finally, to see that the scanner cannot reach t_{j+1}, we note that the proof of (vi) allows for $x \in B_{j+1}$ to have value t_{j+1}. This completes the proof of (v), hence of Lemma 3.4. □

Next, we have to show that the merging and scanning can be done efficiently.

LEMMA 3.5. *Each level step can perform the necessary merging and scanning in constant time.*

PROOF. Recall that for level j, we have 4^j steps to complete a period. We have to complete both merging and scanning in such a period, spending only constant time on each level step.

Currently, we have described the merging and scanning as divided into atomic steps, each taking constant time. However, a merge step may insert a key in a base set and follow this by some constant time rebalancing of base sets, which in turns may move a constant number of scanners and splitters (but do not insert any more base keys). All this is included in the constant time we spend on a merge step.

Our goal is to show that it suffices to perform a constant number of merge and scan steps in each level step.

First, we consider scanning, which is almost trivial. Each scan step moves us to the next splitter, adding at least $\Phi/4$ to our counter n_j. Joins and splits of the base set may move the scanning but they can never decrease the counter. The scan terminates before the counter reaches $(4^{j+1}+2)\Phi$, and this takes at most $(4^{j+1}+2)\Phi/(\Phi/4) \leq 18 \cdot 4^j$ atomic scan steps. Thus, it suffices to perform 18 atomic scan steps in each merge step.

The analysis for the merger is a bit more subtle. By definition, we have at most 4^j buffer keys to insert in the base sets during the merge. The tricky part is to measure the progress done when a merge step moves us from one base splitter to the next, for there are cases where a split of a base set can pull a merger back to a new preceding base splitter.

We define the minimum of the merger as the minimum of the next buffer key and the next base splitter. Now, returning to split, it takes a base set A with subsequent base splitter s, and creates a new base splitter s' from A in front of s. If a merger at s has its next buffer key $x < s'$, the split pulls the merger back to s'. However, this does not affect our minimum which is still x. We note that the join cannot decrease the minimum of the merger either, though it can increase the minimum when it removes a base splitter, moving any merger to the next base splitter. Whenever the minimum of the merger increases from x to x', we say that it passes the base keys that are currently live in $[x, x')$.

We now consider an atomic merge step. It can happen at most 4^j times that a buffer key is the minimum to be inserted in the step. Otherwise, the merger will skip the next base splitter. If the new minimum is the next buffer key, then we are back to the previous case, and that can happen at most 4^j times. If the new minimum is a base splitter, we have passed a base set with at least $\Phi/4$ base keys.

The next question is how many base keys the merger can pass. Since the merger stops when the buffer is empty, we will only pass base keys smaller than the largest buffer key x, and by Lemma 3.4(iii), we had $x < t_{j+2}$ when the period started. By Lemma 3.4(ii), there were less than $8 \cdot 4^j \Phi$ keys below t_{j+2} when the period started. By Lemma 3.2, at most $4^j \Phi/2$ keys can arrive during the period. Hence it follows that the merger passes less $8.5 \cdot 4^j \Phi$ keys. Thus, we can have at most $8.5 \cdot 4^j \Phi/(\Phi/4) = 34 \cdot 4^j$ merge step moving the minimum from one base splitter to the next.

All in all we end up with at most $36 \cdot 4^j$ atomic merge steps to be completed in a period, so it suffices to do 36 of these in each level j step.

We remark that via Lemma 3.2 and 3.4, the above analysis used that each period completed its merge and scan, and we used this to prove that each period completed its merge and scan. The reason that this is not a circular argument is that we are really only assuming that each preceding period has completed its work correctly. About the current period, we just used that the length of the period limited how many updates there could be, and from that we concluded that the merging and scanning would be completed by the time the period ended. □

We are now ready to complete the analysis of the construction.

LEMMA 3.6. *The above priority queue uses $O(n)$ space, supplies the minimum in constant time, and supports updates in $O(S(n))$ time.*

PROOF. The general correctness follows from Lemma 3.4. The space bound is trivial in that we use constant space for each key plus sublinear space for all the

splitters. By Lemma 3.1, the minimum key is in the head, from which it can be accessed in constant time.

For the update time we note that we initially spend constant time on each update. For an insert, this includes placing the key in a buffer, and for a delete, this includes possible rebalancing of the base sets. Moreover, each update results in at most two level steps. A level step spends $O(S(n))$ time in the sorter. Moreover, by Lemma 3.5, it spends constant time on merging, including rebalancing of base sets, and scanning. This adds up to $O(S(n))$ time per update. □

Lemma 3.6 immediately implies the statement of Theorem 1.1 for n a fixed capacity, and stated earlier, we get variable size from Thorup [2000, Lemma 2.1]. This completes the proof of Theorem 1.1.

3.7. A WORST-CASE PRE-SEARCH BUFFER. We will now show how to bypass the atomic heap in the pre-searcher. The construction is very similar to the amortized construction from Section 2.9. We shall refer to the priority queue described above, but without the atomic heap in the pre-searcher, as the basic priority queue. We will construct the pre-searcher as a front-end to the basic priority queue.

As in the amortized construction, the basic idea is to make a pre-search buffer that we can sort to find the right levels. For a worst-case construction, we divide the pre-search buffer in an entrance, a sorter, and an exit, each with capacity of Φ keys. The pre-searcher works in periods of Φ updates. When a period starts, the pre-search entrance is empty. If an update inserts a key $x \geq t_2$, the key is placed in the entrance in a set X, represented as a doubly-linked list.

When the next period starts, a pointer to the set X is moved to the pre-search sorter, leaving the entrance empty. This move takes constant time. The sorter also copies the level splitters. This is done in a scan through the level splitter list. We note the level splitters copied may not be alive at the same time as the scan is done over multiple updates. Let t'_j be the copy that was made of the level j splitter t_j. Now, the sorter sorts the keys it got from the entrance together with its level splitter copies. Finally, the sorter runs through the sorted list, removing keys $x \in X$ that were deleted while the sorting took place. The final result of the sorting is that we for each remaining key $x \in X$ know the level j such that $x \in [t'_j, t'_{j+1})$. The level j is stored with x when X is moved to the exit at end of the period.

The work done by the pre-search buffer in a period is dominated by the sorting which takes $O(\Phi \cdot S(\Phi))$ time, and this work is distributed as $O(S(\Phi))$ time at each of the Φ updates.

The way the overall priority queue works is that when an update is made, it is applied directly to the basic priority queue unless it is an insert of a key $x \geq t_2$, in which case we place x in the pre-search entrance. Moreover, if there are keys in the pre-search exit, we take one and insert it in the basic priority queue. One important thing to note here is that an update to the overall priority queue can generate two updates for the basic priority queue.

We now need some lemmas to show how the keys from the pre-search exit can be added to the basic priority queue in constant time.

LEMMA 3.7. *If we consider any period of at most* 24Φ *updates to the basic priority queue, then the initial value of* t_j *is less than the final value of* t_{j+1}*, and the final value of* t_j *is less than the initial value of* t_{j+1}*.*

PROOF. First, if $j = 0$, the result is true by definition since $t_0 = -\infty < t_1$. If $j > 0$, by Lemma 3.4(ii), there are always less than $8 \cdot 4^j \Phi$ keys below t_j, and by Lemma 3.4(i), there are more than $3.5 \cdot 4^{j+1}\Phi$ keys below t_{j+1}. Thus, we have a difference of more than

$$3.5 \cdot 4^{j+1}\Phi - 8 \cdot 4^j \Phi = 6 \cdot 4^j \Phi \geq 24\Phi$$

keys, which would have to be made up for by new updates. □

LEMMA 3.8. *With a key from the pre-search exit, we can perform an initial insert in constant time.*

PROOF. The initial update from Section 3.4 took constant time. The only thing we are missing now is the atomic heap in the pre-searcher which for a key x found the level j such that $x \in [t_j, t_{j+1})$. However, when x is from the pre-search exit, we have a level j' such that $x \in [t'_{j'}, t'_{j'+1})$.

We know that there are at most 2 periods of Φ updates to the overall priority queue from the time that t'_j and t'_{j+1} were copied from t_j and t_{j+1}. These updates to the overall priority queue lead to at most 4Φ updates to the basic priority queue. It now follows from Lemma 3.7 that $x \geq t'_{j'} > t_{j'-1}$ and that $x < t'_{j'+1} < t_{j'+2}$. Hence $x \in [t_j, t_{j+1})$ for j equal to $j' - 1$, j', or $j' + 1$, and we can check which by comparing x with $t_{j'}$ and $t_{j'+1}$. These comparisons take constant time as desired. □

We also need to make sure that the minimum key stays in the head of the basic priority queue.

LEMMA 3.9. *If x is in the pre-search buffer, then there are at least 56Φ base keys below x. The minimum key of the overall priority queue is hence found in the head of the basic priority queue.*

PROOF. When x was placed in the pre-search buffer, it was bigger than t_2, so by Lemma 3.4(i), there were more than $3.5 \cdot 4^2\Phi = 56\Phi$ base keys below x. Now x stays in the pre-search buffer for at most 3 periods of Φ updates to the overall priority queue. Since no deletes come out of the pre-search buffer, we can lose at most 3Φ keys. Hence we cannot get less than 53Φ base keys below a key x in the pre-search buffer. □

LEMMA 3.10. *The overall priority queue, combining the basic priority queue with the pre-search buffer, uses $O(n)$ space, supplies the minimum in constant time, and supports updates in $O(S(n))$ time.*

PROOF. The search pre-buffer only uses $O(\Phi) = O(\log n)$ space, so the linear space follows from Lemma 3.6. By Lemma 3.9, we get the minimum from the head which as an atomic head supplies the minimum in constant time. In each update, we spend $O(S(\Phi))$ time in the pre-search buffer, and by Lemma 3.8, this replaces the need for an atomic heap in the pre-searcher. Moreover, we do at most two updates to the basic priority queue and each takes $O(S(n))$ time by Lemma 3.6. □

3.8. THE RECURSIVE HEAD. As in the amortized construction, we will replace the atomic priority queue in the head with a recursively defined priority queue. However, first, we add to our priority queue a separate variable with a copy of the minimum key. At the end of each update, the minimum should be found and copied. This way, it is clear that find-min can be computed in constant time. The recursive

version in the head will maintain a copy of its minimum key which we can copy in constant time after any recursive update to the head.

For the base of the recursion, we implement the priority queue as a simple sorted list if $\Phi \geq n/12$. Recall here that Φ was the first number bigger than $2 \log n$ which is divisible by 84. It follows that the priority queue has constant size in this base case, hence that we can implement all updates in constant time.

We will show that update time $T(n)$ of our recursively defined priority queue satisfies the recurrence

$$T(n) = O(S(n)) + T(O(\Phi)).$$

First we show that the head is only affected by updates to smaller keys in the priority queue.

LEMMA 3.11. *The head is only affected by direct updates to the* 3Φ *smallest keys.*

PROOF. The head can be affected either directly when an update is of a key in the head itself, or when the head is being rebalanced with a join or a split. We only work on rebalancing when we update an involved base set or one of its neighbors (cf., Section 2.6). Since the head is the first base set, it is only being rebalanced by updates to the first three base sets, and each base set has at most Φ keys.

From Lemma 3.4(i) and Lemma 3.9, it follows that neither level buffers nor the pre-search buffer can contain any of the 3Φ smallest keys, so these are all found in the base sets. This also implies no updates to the first three base sets can pass through any buffer, so they must be direct updates. □

Next we argue that the rebalancing does not effect our update time.

LEMMA 3.12. *The time we spend on rebalancing the head is* $O(S(\Phi))$ *per update.*

PROOF. We know that we have $\Phi/84$ updates available to implement a join or split of the head. In that time, we have to move less than Φ keys to or from the head, so we may spend $O(S(\Phi))$ time on each key moved. The rebalancing keeps the head of size $\Phi/4$, and it never moves any of the first $\Phi/4$ keys.

Let Ψ be the value of Φ for the recursively defined priority queue. That is, Ψ is the first number bigger than $2 \log \Phi$ which is divisible by 84. If $\Psi < \Phi/12$, the recursive priority queue in the head is in the base case where all updates are implemented in constant time. We can then move each key in constant time.

If $\Psi \geq \Phi/12$, we know that the keys moved are not among the first $\Phi/4 \geq 3\Psi$ keys in the head. Hence, by Lemma 3.11, the updates do not affect the head of the head. Consequently, the update time from Lemma 3.10 applies to the moves to and from the head. Hence, we spend $O(S(\Phi))$ time on each key moved to rebalance the head. □

LEMMA 3.13. *In the above recursive priority queue with capacity of* n *keys uses* $O(n)$ *space. It supplies the minimum in constant time. It spends* $O(S(n))$ *time per update. In addition, we spend* $T(\Phi)$ *time on updates to the head. Here* $T(m)$ *is the update time of a recursive priority queue with capacity* m.

PROOF. From Lemma 3.10, it follows that the total space used is

$$Space(n) = O(n) + Space(\Phi) = O(n).$$

The minimum is stored in a separate variable, and can hence be supplied in constant time. From Lemma 3.10, we know that the update time spent outside the head is $O(S(n))$. By Lemma 3.12, the rebalancing of the head takes $O(S(\Phi)) = O(S(n))$ time per update. For direct updates to the head, we have an additional recursive cost of $T(\Phi)$. Finally, we spend constant time copying the minimum of the head to our variable for the minimum. □

Since $\Phi = O(\log n)$, Lemma 3.13 immediately implies the statement of Theorem 1.2 for n a fixed capacity, and stated earlier, we get variable size from Thorup [2000, Lemma 2.1]. This completes the proof of Theorem 1.2.

4. *Concluding Remarks*

Above, we have shown how a priority queue can be constructed with an update time matching the per key cost of sorting, which is best possible. This allowed us to translate sorting results so as to improve update times for priority queues in several models of computation.

Using the reduction of Alstrup et al. [2005], we can bring the insert time down to constant and only pay the per key cost of sorting for deletes. A further reduction of Mendelson et al. [2006] shows that, if we are satisfied with amortized time bounds, we can also meld priority queues in constant time, preserving the other bounds.

One major open problem is whether there is a general reduction that allows us to decrease the value of a key in constant time. Such a fast decrease-key operation helps in Dijkstra's single source shortest path algorithm [Dijkstra 1959], which may perform many more decrease-key operations than other operations. Recently, Thorup [2004] has presented a priority queue supporting insert and decrease-key in constant time and delete in $O(\log \log n)$ time, and this matches the current best deterministic sorting bound of $O(n \log \log n)$ by Han [2004]. However, randomized we know how to sort in $O(n\sqrt{\log \log n})$ expected time, but we do not know how to get insert and decrease-key in expected constant time and delete in $O(\sqrt{\log \log n})$ expected time.

ACKNOWLEDGMENTS. We thank the referees from *J. ACM* for many good suggestions for improving the presentation of this article.

REFERENCES

ALSTRUP, S., HUSFELDT, T., RAUHE, T., AND THORUP, M. 2005. Black box for constant-time insertion in priority queues (note). *ACM Trans. Algor. 1*, 1, 102–106.

ANDERSSON, A., MILTERSEN, P., AND THORUP, M. 1999. Fusion trees can be implemented with AC^0 instructions only. *Theor. Comput. Sc. 215*, 1-2, 337–344.

ANDERSSON, A., AND THORUP, M. 2007. Dynamic ordered sets with exponential search trees. *J. ACM 54*, 3, Article 13. (Combines results announced at FOCS'96, STOC'00, and SODA'01.)

BEAME, P., AND FICH, F. 2002. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci. 65*, 1, 38–72. (Announced at STOC'99.)

COMRIE, L. J. 1929–30. The hollerith and powers tabulating machines. *Trans. Office Mach. Users' Assoc., Ltd*, 25–37.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms*, Second ed. The MIT Press, Cambridge, MA.

DIJKSTRA, E. W. 1959. A note on two problems in connection with graphs. *Numer. Math. 1*, 269–271.

DUMEY, A. I. 1956. Indexing for rapid random access memory systems. *Comput. Automat. 5*, 12, 6–9.

FORD, L. R., AND JOHNSON, S. M. 1959. A tournament problem. *Amer. Math. Month. 66*, 5, 387–389.

FREDMAN, M. L., AND SAKS, M. E. 1989. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Symposium on Theory of Computing* (*STOC*). ACM, New York, 345–354.

FREDMAN, M. L., AND WILLARD, D. E. 1993. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci. 47*, 424–436. (Announced at STOC'90.)

FREDMAN, M. L., AND WILLARD, D. E. 1994. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci. 48*, 533–551.

HAN, Y. 2001. Improved fast integer sorting in linear space. *Inf. Comput. 170*, 8, 81–94. (Announced at STACS'00 and SODA'01.)

HAN, Y. 2004. Deterministic sorting in $O(n \log \log n)$ time and linear space. *J. Algorithms 50*, 1, 95–105. (Announced at STOC'02.)

HAN, Y., AND THORUP, M. 2002. Integer sorting in $O(n\sqrt{\log \log n})$ expected time and linear space. In *Proceedings of the 43rd Symposium on Foundations of Computer Science* (*FOCS*). IEEE Computer Society Press, Los, Alamitos, CA, 135–144.

KERNIGHAN, B., AND RITCHIE, D. 1988. *The C Programming Language*, Second ed. Prentice-Hall, Englewood Cliffs, NJ.

MENDELSON, R., TARJAN, R., THORUP, M., AND ZWICK, U. 2006. Melding priority queues. *ACM Trans. Algor. 2*, 4, 535–557. (Announced at SODA'03 and SWAT'04.)

PRIM, R. C. 1957. Shortest connection networks and some generalizations. *Bell Syst. Tech. J. 36*, 1389–1401.

TARJAN, R. E. 1975. Efficiency of a good but not linear set union algorithm. *J. ACM 22*, 2, 215–225.

THORUP, M. 1998. Faster deterministic sorting and priority queues in linear space. In *Proceedings of the 9th Symposium on Discrete Algorithms (SODA)*. ACM, New York, 550–555.

THORUP, M. 2000. On RAM priority queues. *SIAM J. Comput. 30*, 1, 86–109. (Announced at SODA'96.)

THORUP, M. 2002. Randomized sorting in $O(n \log \log n)$ time and linear space using addition, shift, and bit-wise boolean operations. *J. Algor. 42*, 2, 205–230. (Announced at SODA'97.)

THORUP, M. 2003a. Combinatorial power in multimedia processors. *ACM SIGARCH Comput. Architect. News 31*, 5, 5–11.

THORUP, M. 2003b. On AC^0 implementations of fusion trees and atomic heaps. In *Proceedings of the 14th Symposium on Discrete Algorithms (SODA)*. ACM, New York, 699–707.

THORUP, M. 2004. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *J. Comput. Syst. Sci. 69*, 3, 330–353. (Announced at STOC'03.)

WILLARD, D. E. 2000. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput. 29*, 3, 1030–1049. (Announced at SODA'92.)

WILLARD, D. E., AND LUEKER, G. S. 1985. Adding range restriction capability to dynamic data structures. *J. ACM 32*, 3, 597–617.

RECEIVED OCTOBER 2005; REVISED APRIL 2007 AND AUGUST 2007; ACCEPTED SEPTEMBER 2007