International Journal of Foundations of Computer Science
© World Scientific Publishing Company

STRONGER QUICKHEAPS

GONZALO NAVARRO *
Department of Computer Science, University of Chile, Chile. gnavarro@dcc.uchile.cl

RODRIGO PAREDES
Departamento de Ciencias de la Computación, Universidad de Talca, Chile. raparede@utalca.cl

PATRICIO V. POBLETE
Department of Computer Science, University of Chile, Chile. ppoblete@dcc.uchile.cl

PETER SANDERS
Karlsruhe Institute of Technology, Germany. sanders@kit.edu

Received (Day Month Year)
Accepted (Day Month Year)
Communicated by (xxxxxxxxxx)

The Quickheap (QH) is a recent data structure for implementing priority queues which has proved to be simple and efficient in practice. It has also been shown to offer logarithmic expected amortized complexity for all of its operations. Yet, this complexity holds only when keys inserted and deleted are uniformly distributed over the current set of keys. This assumption is in many cases difficult to verify, and does not hold in some important applications such as implementing some minimum spanning tree algorithms using priority queues. In this paper we introduce an elegant model called a *Leftmost Skeleton Tree (LST)* that reveals the connection between QHs and randomized binary search trees, and allows us to define *Randomized QHs*. We prove that these offer logarithmic expected amortized complexity for all operations regardless of the input distribution. We also use LSTs in connection to α-balanced trees to achieve a practical *α-Balanced QH* that offers worst-case amortized logarithmic time bounds for all the operations. Both variants are much more robust than the original QHs. We show experimentally that randomized QHs behave almost as efficiently as QHs on random inputs, and that they retain their good performance on inputs where that of QHs degrades.

Keywords: Priority Queues; Randomized Data Structures; Amortized Analysis

1991 Mathematics Subject Classification: 22E46, 53C35, 57S20

1. Introduction

A *priority queue* is a data structure which allows maintaining a set of elements in a partially ordered way. Typically, the elements are of the form *(key, item)*, where *key*

*Supported in part by Fondecyt grant 1-080019, Chile.

is the priority and *item* is satellite data (which is not used to sort the set, thus from now on we neglect it). A priority queue enables efficient element insertion (*Insert*), minimum finding (*FindMin*), and minimum extraction (*ExtractMin*) —or, alternatively, maximum finding and extraction. The set of operations can be extended to construct a priority queue from a given array (*Heapify*), increase or decrease the priority of an arbitrary element (*IncreaseKey* and *DecreaseKey*, respectively), delete an arbitrary element from the priority queue (*Delete*), and many others. In the following we focus on the operations we have mentioned explicitly.

The classic priority queue implementation uses a binary heap [$Wil64$]. Wegener [$Weg93$] proposes a *bottom-up deletion* algorithm, which addresses operation *ExtractMin* performing only $\log_2 m + O(1)$ key comparisons per extraction on average, in heaps of m elements. Other well-known priority queues are *sequence heaps* [$San00$], *binomial queues* [$Vui78$], *Fibonacci heaps* [$FT87$], *pairing heaps* [$FSST86$], *skew heaps* [$ST86$], and *van Emde Boas queues* [$vEBKZ77$].

In previous work [$NP10$], a so-called IncrementalQuicksort algorithm (IQS) was developed, and a data structure for implementing priority queues based on IQS, coined *Quickheaps* (QHs), was proposed and empirically tested, showing that QHs are very competitive in practice. Moreover, they offer logarithmic expected amortized complexity for all the operations (except *DecreaseKey*), when keys inserted and deleted are uniformly distributed over the current set of keys. This assumption on the key distribution is not only somewhat awkward and difficult to verify, but also it does not hold in some emblematic applications, such as implementing Prim's Minimum Spanning Tree algorithm [$Pri57$]. Thus, the quest for more robust QHs, which retain practicality, was raised.

In this paper we introduce the *Leftmost Skeleton Tree (LST)*, a simple and elegant model that reveals the connection between QHs and randomized binary search trees. The LST allows us to define stronger QHs based on randomization, coined *Randomized QH*, which override the QH key distribution assumption. We prove that Randomized LST-based QHs enable efficient implementation of all the operations, offering logarithmic expected amortized complexity for all of them regardless of the input distribution. This makes them much more robust than the original QHs. We also use LSTs in connection to α-balanced trees to achieve a practical *α-Balanced QH* that offers worst-case amortized logarithmic time bounds for all of the operations. Like the original QH, both Randomized and α-Balanced QH versions require to store only $O(\log m)$ (expected or worst-case, respectively) extra integers for a queue of m elements. Furthermore, they exhibit a local access pattern, which makes them excellent alternatives for a secondary memory implementation.

We experimentally compare randomized LSTs-based QHs with the original ones, showing that the randomized variants are at most 12% slower when the uniformity assumption holds (in which case the original QHs were already shown to be competitive with the state of the art [$NP10$]). Furthermore, the randomized QHs retain their good performance on inputs where that of original QHs degrades.

The rest of this paper is organized as follows. In the next section we describe the original Quickheap data structure. In Section 3 we present the Leftmost Skeleton Tree and its basic operations. Then, in Section 4 we show how to use LSTs in order to implement a randomized priority queue, and in Section 5 we empirically show that it is competitive in practice. In Section 6 we combine LSTs and General Balanced Trees [*And*99] in order to provide worst-case amortized guarantees, and in Section 7 we analyze our variants under the cache-oblivious model, showing that their locality of reference makes them apt for a secondary-memory implementation. Finally, in Section 8 we give our conclusions and some directions for further work.

2. Quickheaps (QHs)

In [$NP10$], the IQS algorithm was introduced and the QH was built on it. Given an unsorted set A, IQS retrieves its k smallest elements in increasing order for any k, that is unknown to the algorithm, in optimal expected time $O(m + k \log k)$. The IQS algorithm can be seen as a nonrecursive variant of QUICKSORT [$Hoa62$].

The work performed the first time we process an array A using IQS is the same as that performed by QUICKSELECT in order to find the smallest element of $A[0, m-1]$ (while this cost is quadratic in the worst case, IQS was focused on average costs, in which case the operation takes linear time). Figure 1 illustrates in the first row an original array and in the second the resulting array after calling QUICKSELECT using the first element of the current partition as the pivot. The key point is that QUICKSELECT generates a sequence of pivots in decreasing order and we can use this pivot sequence in the following operations. In the figure, we have added ovals indicating pivots and an ∞ mark signaling a fictitious pivot in the last array cell. Finally, we have stored all the pivot positions in an auxiliary stack S.

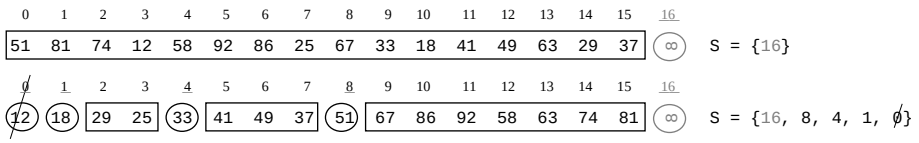

Fig. 1. IQS operation. First row, the original array. Second row, the result after calling QUICKSELECT for the first element. All the pivot positions are stored in stack S. We use this partitioned array in the QH implementation. In the example we use the first element of the current partition as the pivot, but it could be any other element.

If we need to find the second smallest element, we could call QUICKSELECT on $A[1, m-1]$. Instead, IQS checks if we have the position 1 on top of S, that is, if $S.top() = 1$, in which case we are done as $A[1]$ *is* the second smallest element. Else, it calls QUICKSELECT on $A[1, S.top() - 1]$ and pushes the new resulting pivots onto S. We can use IQS to retrieve the next elements, if we need to. In order to preserve the IQS invariant (either the minimum is the element on top of S or it is within

the leftmost chunk) from the first time, we initialize S containing the position of the fictitious pivot ∞. The experimental results [$NP10$] showed that IQS is up to 4 times faster than the classical alternative that uses binary heaps.

If we read the resulting array from right to left (Figure 1), we see a sequence of pivots and chunks starting from the fictitious pivot ∞ (at position 16) with a chunk of elements smaller than it (and greater than the next pivot), until the last pivot 18 (at cell 1) and a last chunk (in this case, without elements). This structure resembles a priority queue and is exploited by the original QH.

Briefly, the QH implements the basic priority queue operations as follows (for further details see [$NP10$]). To find the minimum we simply call IQS. To extract it, after finding it, we remove it from the array and pop its position from the stack. To insert a new element x we need to find the chunk in which we can properly add x. As we have all the pivot positions in the stack, we can directly access array cells in order to shift pivots one cell to the right and move the first element of each chunk to the hole left at the previous pivot position. So, we insert x by moving pivots and elements starting from the fictitious pivot until we reach the first pivot smaller than x, in which case we place x in the empty cell left by the last shifted pivot (which was still $\geq x$).

If we augment the QH with a dictionary managing the current element positions within the array, we can also implement operation *Delete*. To remove an element, we obtain its position from the dictionary and then extract it from the array, placing in the hole the last element of its chunk. So we have an empty cell to shift the next pivot at its right (that is, the one immediately larger than it) one cell to the left, and we go on until reaching the fictitious pivot. Operations *IncreaseKey* and *DecreaseKey* can be implemented by performing a deletion followed by a (re)insertion. We remark that including operation *Delete*, *DecreaseKey* and/or *IncreaseKey* involves $O(m)$ additional space for the dictionary and $O(\log m)$ extra time per operation in order to query and update it. These costs must be added to the analyses that follow if these operations are to be included.

It was shown that the expected amortized complexity of QH operations is logarithmic, when keys inserted and deleted are uniformly distributed over the current set of keys [$NP10$]. In the next section we introduce a novel model that allows us to override the QH key distribution assumption.

3. Leftmost Skeleton Trees (LSTs)

We define a data structure called the *Leftmost Skeleton Tree (LST)* which, depending on the update algorithms we use, represents the leftmost path of a binary search tree (BST) from a given family. We use LSTs for implementing priority queue operations within time bounds that depend on the type of BST represented. In this section we define LSTs and show how to implement the operations of interest on them.

Definition 1. *A* Leftmost Skeleton Tree (LST) *T over a totally ordered domain Z*

is either

- $T = bucket(B)$*, a multiset of* $|B|$ *elements of* Z*; or*
- $T = tree(r, L, B)$*, where* $r \in Z$ *is the root key,* L *is an LST containing only elements* $x \leq r$*, and* B *is a multiset containing only elements* $y \geq r$*.*

We will denote $T_v = (v, L_v, B_v)$ *the subtree of* T *rooted at pivot* v*.*

Definition 2. *The* size *of an LST is*

$$s(bucket(B)) = |B|,$$
$$s(tree(r, L, B)) = 1 + s(L) + |B|.$$

Definition 3. *The* length *of an LST is*

$$\ell(bucket(B)) = 1,$$
$$\ell(tree(r, L, B)) = 1 + \ell(L).$$

Definition 4. *The* depth *of a bucket* B *in an LST is* [a]

$$d(B, bucket(B)) = 0,$$
$$d(B, tree(r, L, B)) = 1,$$
$$d(B, tree(r, L, B')) = 1 + d(B, L), \text{ if } B \neq B'.$$

3.1. *LST structures*

To implement the LST $T = tree(r, L, B)$ we use an array A to store the elements of T and a stack S to manage the pivot positions (i.e., roots of subtrees).

Definition 5. *The representation of an LST in terms of an array* A *and a stack* S *is defined as follows (where ':' is the sequence concatenation operator and the left-to-right order in* S *corresponds to top-to-bottom order):*

$$A(bucket(B)) = B,$$
$$A(tree(r, L, B)) = A(L) : r : B,$$
$$S(bucket(B)) = s(bucket(B)),$$
$$S(tree(r, L, B)) = S(L) : s(tree(r, L, B)).$$

That is, A consists of the sequences of buckets of T, from leftmost to rightmost (which in A will be called *chunks*) separated by the *pivots* that constitute the subtree roots in T. The stack contains the positions of the pivots in A, in left-to-right order as we read S top-to-bottom.

Figure 2 shows an LST example and its memory layout, and will be used to illustrate the LST operations as we describe them next. We use p_v to denote the

[a] By $B \neq B'$ in the third line we mean they are different buckets in the data structure, not that they are different multisets.

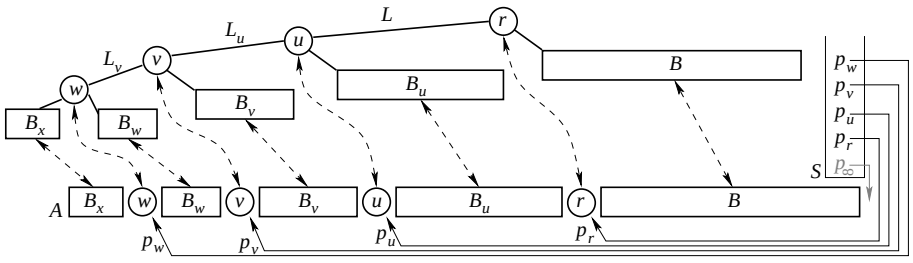

Fig. 2. A Leftmost Skeleton Tree example and its representation as an array A and a stack S. The tree is the logical view and the array plus stack is the physical view.

position of pivot v in array A, starting from cell 0, thus S contains the p_v values of the successive nodes v of T. As in the original QH, we also use an auxiliary integer $p_\infty = size(T)$ at the bottom of S. This ficticious pivot is the only element of S that does not correspond to any LST node. This way, we always have a pivot following a chunk, even when the LST contains no elements.

3.2. *LST Operations*

We now describe a set of operations of interest on LSTs and analyze them in terms of number of basic operations performed. For a more detailed description we refer the reader to the original QH article [$NP10$].

Convert an array B into $T = bucket(B)$. We create stack $S = \langle p_\infty = |B| \rangle$ with one element. If we can use the array B as the array A we are done; this costs $O(1)$. Else, we copy the elements within B onto array A, which costs $O(|B|)$.

Compute $s(T_v)$. It is the position of the pivot at the right of T_v, easily computed in $O(1)$ time if we know where is v in S.

Insert/Delete x into/from a bucket. We reach the corresponding chunk from the right of A and from the bottom of S. For inserting, we must reach the correct chunk B_v and create an empty cell within it. We start by incrementing p_∞, this way making space in the rightmost chunk B. As long as the current chunk B is not B_v (i.e., B is not the leftmost chunk and x is smaller than the pivot to the left of B), we move the first element of B to the empty space at its end, shift the pivot at the left of B one cell to the right, and move on to the next chunk (to the left of A and toward the top of S), which now has a free space at its end. Once we reach bucket B_v, we place x into its last cell, which is free. This costs $O(d(B_v, T))$.

To delete an element x we need a dictionary, as explained, in order to know its current position p_x in the array A. With p_x we find the chunk B_v containing x by traversing S bottom-to-top. Then, position p_x of B_v is now a free cell. From now on, the extraction is dual to the insertion; that is, we repeat the following until processing the rightmost chunk: we move the last element of the current chunk B to its free cell, move the pivot following B one position to the left, and move on to

the next chunk to the right (whose first position is now a free cell, the one left by the moved pivot). This costs $O(d(B_v, T))$ time (as explained, we are neglecting the cost of operating and updating the dictionary).

Extract the leftmost node (when the leftmost bucket is empty). This could be done just as a special case of deletion, but we optimize this common operation by using the memory allocated for the data in circular fashion, just as the original QHs. Thus, we simply increase by 1 the starting position of the data in the memory array and pop the stack S. Therefore, this extraction needs $O(1)$ time.

Flatten subtree T_v into a bucket. Flattening $T_v = tree(v, L_v, B_v)$ corresponds to removing all of its pivots from S. As all these are on top of the stack, we discard them all at once by updating the stack size to $d(B_v, T)$. As T_v's buckets and pivots are already consecutive in A, we can just consider them all as a single bucket. The total flattening cost is $O(1)$.

Structure bucket B_v with pivot x. We use the traditional QUICKSORT partitioning algorithm on B_v. Thus we carry out $|B_v| - 1$ comparisons. Then, we push p_x, the final position of x after the partitioning, onto S. The total time is $O(|B_v|)$.

4. Randomized Priority Queues using LSTs

We now introduce Randomized QHs, by means of combining LSTs with a scheme inspired in Martínez and Roura's randomized BST [$MR98$]. The idea is that the LST will represent the leftmost path of a random BST (RBST). We explain now the operations, giving their actual time, and later carry out an amortized analysis. Figure 3 shows the relevant pseudocodes.

Heapifying. Given an array of elements, we convert it into an LST by regarding it as a bucket, in linear time.

Insertion. To insert a new element x into T we look for its insertion point. Whenever it belongs to a right child (or left child of the last node), we just add it to the corresponding bucket and finish. Yet, with probability inversely proportional to the current subtree size plus 1, we flatten the current subtree into a single bucket and insert the element in there. The overall time is $O(d(B, T))$, being B the bucket where we insert x.

For simplicity we assume in the pseudocode that all keys are different. Otherwise the solution to maintain an RBST, following Martínez and Roura [$MR98$] is that, whenever we opt for entering a child (instead of reconstructing the tree) and x is equal to the key at the root v of the current subtree $T_v = tree(v, L_v, B_v)$, that is $x = v$, we go left with probability $\frac{s(L_v)+1}{s(T_v)+1}$ and right with probability $\frac{|B_v|+1}{s(T_v)+1}$. All these sizes can be computed in constant time in the LST.

Minimum Finding and Extraction.

To extract the minimum element from T we successively structure the leftmost bucket, using a random bucket element as the pivot, until the leftmost bucket becomes empty. At this point we extract the leftmost node and return its key. Its right

Insert($LST\ T, x \in Z$)
1. **If** $T = bucket(B)$ **Then** Add x to bucket B // $O(depth)$
2. **Else**
3. Let $T = tree(r, L, B)$
4. **If** $random(s(T) + 1) \neq 1$ **Then**
5. **If** $x < r$ **Then Insert**(L, x)
6. **Else** Add x to bucket B // $O(depth)$
7. **Else**
8. Flatten T into $bucket(B')$ // $O(1)$
9. Add x to bucket B' // $O(depth)$

Partition($LST\ T$) // assumes $T = bucket(B)$ and $|B| > 0$
1. Let $T = bucket(B)$
2. $r \leftarrow B[random(|B|)]$
3. Partition $B = B' \cdot r \cdot B''$, such that $x' \in B',\ x'' \in B'' \Rightarrow x' < r < x''$ // $O(|B|)$
4. $T \leftarrow tree(r, bucket(B'), bucket(B''))$ // $O(1)$

ExtractMin($LST\ T$) // assumes $s(T) > 0$
1. **If** $T = bucket(B)$ **Then Partition**(T) // $O(|B|)$
2. Let $T = tree(r, L, B')$
3. **If** $s(L) = 0$ **Then**
4. Flatten T into $bucket(B')$ // $O(1)$
5. Remove r from bucket B' // $O(1)$
6. **Return** r
7. **Else Return ExtractMin**(L)

FindMin($LST\ T$) // assumes $s(T) > 0$
1. **If** $T = bucket(B)$ **Then Partition**(T) // $O(|B|)$
2. Let $T = tree(r, L, B')$
3. **If** $s(L) = 0$ **Then Return** r
4. **Else Return FindMin**(L)

Delete($LST\ T, x \in Z$)
1. **If** $T = bucket(B)$ **Then** Remove x from bucket B // $O(depth)$
2. **Else**
3. Let $T = tree(r, L, B')$
4. **If** $x < r$ **Then Delete**(L, x)
5. **Else If** $x > r$ **Then** Remove x from bucket B' // $O(depth)$
6. **Else**
7. Flatten T into $bucket(B'')$ // $O(1)$
8. Remove x from bucket B'' // $O(depth)$

Fig. 3. Randomized priority queue operations implemented using a Leftmost Skeleton Tree. Operation $random(k)$ returns a uniformly distributed integer in $[1, k]$. The tail recursion of *ExtractMin/FindMin* can be easily removed.

child becomes the new leftmost bucket. Finding the minimum is identical except that the leftmost node is not extracted. The process takes time $O(|B|)$ on average,

where B is the original leftmost bucket.

Deletion. Assume we know the position of an element x to delete from T, that is, we know the position it has in its bucket. We look for its bucket and remove it from there. If, however, x corresponds to a tree node, we flatten the whole subtree into a bucket and then remove x. The overall time is $O(d(B,T))$, where B is the bucket where x belongs.

Again, in the pseudocode we assume for simplicity that all keys are different. If they are not, we can use its bucket position to identify it.

Increasing and Decreasing Keys. We can implement these by means of a deletion followed by a (re)insertion.

4.1. *Properties of RBSTs and LSTs*

An RBST is a classical BST built by inserting the set of keys in random order (each permutation having the same probability). Via randomization one can achieve RBSTs independently of the actual insertion order of the keys [$MR98$].

We first prove a few properties of RBSTs. Some are well known, others are more specific of our purposes. Then we prove a crucial LST property. From now on we use the notation $H_n = \sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1)$.

First, the next term will arise in the amortized cost of minimum extractions.

Lemma 6. *Let v be the leftmost node in an RBST of n nodes (hence it has no left child). Then the expected subtree size of the right child of v is $O(\log n)$.*

Proof. With probability $\frac{1}{n}$ the root is the smallest value, and thus v is the root and its right child is of size $n-1$. Otherwise, the left child of the root is of size i with probability $\frac{1}{n}$ for $i \in [1, n-1]$. Thus the recurrence for the expected size of our subtree is $L(1) = 0$ and $L(n) = \frac{n-1}{n} + \frac{1}{n}\sum_{i=1}^{n-1} L(i)$. The solution is $L(n) = H_n - 1$. □

Most of our operations will have a cost proportional to the leftmost path length. We now show that $E(\ell(T))$, and thus $E(d(B,T))$ for any B, is $O(\log s(T))$.

Lemma 7. *The expected length of the leftmost branch of an RBST of n nodes is $O(\log n)$.*

Proof. This is a well known property, that can be derived from the recurrence $D(0) = 0$ and $D(n) = 1 + \frac{1}{n}\ \sum_{i=0}^{n-1} D(i) = H_n$. □

The following lemma will be useful to show that the expected value of our potential function Φ (to be defined later) is $O(s(T))$.

Lemma 8. *The expected sum of the sizes of all the subtrees of nodes in the leftmost branch of an RBST of n nodes is $O(n)$.*

Proof. This measure corresponds to the expected cost of QUICKSELECT [$Hoa61$] when looking for the first element of an array of size n. It can be derived from $Q(1) = 0$ and $Q(n) = n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} Q(i) = 2n - 2H_n$. □

Finally, we prove that our LSTs maintain the property that the leftmost branch of an LST is a prefix of the leftmost branch of the RBST.

Lemma 9. *If all the buckets of an LST are expanded into RBSTs, then the result is an RBST.*

Proof. We essentially show that our operations correctly mimic those for randomized BSTs [$MR98$] up to the point where we end up in a bucket.

Heapifying. As this creates an LST formed by a single bucket, the property obviously holds.

Insertion. The original algorithm finds the insertion point, yet for any considered subtree T, with probability $\frac{1}{s(T)+1}$, it converts T into a random tree rooted by x. Our algorithm follows the same process on the leftmost path, and simply inserts the element into the corresponding bucket when it has to go to the right child. When the randomization forces it to rebuild the current tree, it just converts the subtree into a bucket. In all cases, the property is obviously maintained.

Minimum Finding and Extraction. The algorithm just extends the leftmost path into a full random leftmost path (thus preserving the property). In the case of extractions, it also removes the minimum. The result has the same distribution as if the minimum had never been inserted, since the subtree that should be rooted by it is replaced by a bucket.

Deletion. The result is as if the element x had never been inserted: We change nothing until the insertion algorithm decides to use x as the root; then we convert the subtree into a bucket. □

4.2. *Amortized Analysis*

We use the so-called potential debt method [$NP10$].[b] This is a slight variant of the potential method, as follows: Let real costs be c_i and amortized costs be $\tilde{c}_i = c_i - \Delta\Phi_i$, where $\Delta\Phi_i$ is the change in potential caused by the i-th operation. Then, adding over all the operations, we obtain that the overall cost is $\sum c_i = \sum \tilde{c}_i + \Phi_m - \Phi_0$. Hence, Φ represents a debit, not a credit, and the final debt $\Phi_m - \Phi_0$ must be split among all the operations.

[b]This was used instead of the traditional potential method in [$NP10$] because the only successful potential function we found was the sum of pivot positions, which grows when we partition the leftmost chunk (which is the operation whose cost we need to amortize). That is, our potential (credit) function would be negative.

Definition 10. *The* potential debt *function* Φ *of an LST is*

$$\Phi(bucket(B)) = 0,$$
$$\Phi(tree(r, L, B)) = s(tree(r, L, B)) + \Phi(L).$$

Note that, due to Lemma 8, the expected value of $\Phi(T)$ is $E(\Phi(T)) = O(s(T))$. Thus, after carrying out m operations starting with an empty tree and ending with LST T, we have $s(T) \leq m$ and thus $\Phi_m = O(s(T)) = O(m)$, hence the accumulated debt adds just $O(1)$ to each operation.

We now compute the amortized costs of the operations. In most cases we count comparisons, and show this is always proportional to the total work. We will refer to $\Phi(T)$ simply as Φ when T is the global LST (and not a subtree of it), to the Φ value after executing the ith operation as Φ_i, and to the difference between two consecutive Φ values as $\Delta\Phi = \Delta\Phi_i = \Phi_i - \Phi_{i-1}$. We will use $n = s(T)$ sometimes.

Heapifying. This operation costs $O(1)$ if the array can be used directly as the bucket, otherwise it has to copy the n array elements into another location, at cost $O(n)$. It retains $\Phi_1 = 0$. However, it breaks the assumption that $s(T) \leq m$, for example just heapifying followed by a minimum extraction costs time linear in the array size. To include this gracefully in our model, we state that the amortized cost of heapifying an array of size n is $O(n)$.

Insertion. Assume we end up inserting element x in bucket B_v. Then we have carried out $d(B_v, T)$ comparisons, and have to work $O(d(B_v, T))$ to carry out the insertion in the bucket (Section 3.2). Hence the number of comparisons is of the same order of the total complexity[c]. We now analyze the change in the potential debt, once we have inserted x into bucket B_v. Assume $T = tree(r, L, B)$ and $B \neq B_v$, and call $T' = tree(r, L', B)$ the LST after the insertion. Call $\Delta\Phi(T', T) = \Phi(T') - \Phi(T)$, then

$$\begin{aligned}\Delta\Phi(T', T) &= \Phi(tree(r, L', B)) - \Phi(tree(r, L, B)) \\ &= s(tree(r, L', B)) + \Phi(L') - s(tree(r, L, B)) - \Phi(L) \\ &= 1 + \Delta\Phi(L', L).\end{aligned}$$

This continues until $T = bucket(B_v)$ or $T = tree(r, L, B_v)$ and thus x is inserted into bucket B_v. Let B'_v be the bucket after the insertion. In the first case we have $\Delta\Phi(T', T) = \Phi(bucket(B'_v)) - \Phi(bucket(B_v)) = 0$, and in the second

$$\begin{aligned}\Delta\Phi(T', T) &= \Phi(tree(r, L, B'_v)) - \Phi(tree(r, L, B_v)) \\ &= s(tree(r, L, B'_v)) + \Phi(L) - s(tree(r, L, B_v)) - \Phi(L) \quad = \quad 1.\end{aligned}$$

Therefore $\Delta\Phi_i = d(B_v, T)$, and thus $\tilde{c}_i = c_i - \Delta\Phi_i = 0$.

After inserting, however, with probability $\frac{1}{s(T)}$, we choose to flatten T into a bucket. This actually has constant cost (Section 3.2), but the potential Φ decreases abruptly, and this increases $\tilde{c}_i$. Fortunately this occurs with low probability. To

[c]Plus $O(1)$ in case $d(B_v, T) = 0$, but we will omit this for simplicity.

be precise, the decrease in the global Φ is exactly the potential $\Phi(T)$ of the flattened subtree, $\Phi(\mathit{bucket}(B_v)) - \Phi(T) = -\Phi(T)$. Thus the expected value for $-\Delta\Phi_i$ associated to this operation is

$$\begin{aligned} E(-\Delta\Phi_i) &= \left(1 - \frac{1}{s(T)}\right) \cdot 0 \; + \; \frac{1}{s(T)} \cdot E(\Phi(T)) \\ &= \frac{1}{s(T)} \; O(s(T)) \quad = \quad O(1) \end{aligned}$$

and therefore this flattening of T has $O(1)$ expected amortized cost.

Structuring the Leftmost Bucket. When partitioning a bucket B, we carry out $|B|-1$ comparisons and this is of the same order of the total work done (Section 3.2). We then convert $\mathit{bucket}(B)$ into $\mathit{tree}(r, B', B'')$, changing the potential debt as follows:

$$\begin{aligned} \Delta\Phi &= \Phi(\mathit{tree}(r, B', B'')) - \Phi(\mathit{bucket}(B)) \\ &= s(\mathit{tree}(r, B', B'')) + \Phi(\mathit{bucket}(B')) - \Phi(\mathit{bucket}(B)) \\ &= (1 + |B'| + |B''|) + 0 - 0 \quad = \quad |B|. \end{aligned}$$

Thus the amortized cost of the operation is $\tilde{c}_i = c_i - \Delta\Phi_i \leq 0$.

Minimum Finding and Extraction. For minimum finding, we successively structure the leftmost bucket until it becomes empty. Since the structuring operations have zero total amortized cost, the amortized cost of *FindMin* is $O(1)$.

ExtractMin, in addition, flattens the tree rooted at the leftmost node and extracts it from the resulting bucket. This costs $O(1)$, but it also reduces the potential. Using the same $\Delta\Phi(T', T)$ defined for the insertion, we have $\Delta\Phi(T', T) = -1 + \Delta\Phi(L', L)$ when both T and T' are not buckets. When we reach the point where $T = \mathit{tree}(r, B_\emptyset, B)$ (being $B_\emptyset$ an empty bucket) and $T' = \mathit{bucket}(B)$, we have:

$$\begin{aligned} \Delta\Phi(T', T) &= \Phi(\mathit{bucket}(B)) - \Phi(\mathit{tree}(r, B_\emptyset, B)) \\ &= 0 - s(\mathit{tree}(r, B_\emptyset, B)) \\ &= -1 - |B|, \end{aligned}$$

and therefore $-\Delta\Phi_i = \ell(T) + |B|$. This B is the right child of the leftmost node in the tree, and thus its expected size is $O(\log s(T))$ by Lemma 6. Hence the expected amortized cost of minimum extraction is $O(\ell(T) + \log s(T)) = O(\log n)$, by Lemma 7.

Deletion. Once the position p_x of element x is obtained, operation *Delete* traverses the leftmost nodes until finding the bucket B where x lies. Thus it carries out $d(B, T)$ operations and pays also $O(d(B, T))$ to remove x from the bucket, thus the operations are of the same order of the total cost. With an analysis similar to that for insertion and minimum extraction, it can be seen that the potential function decreases by $d(B, T)$, and thus the amortized cost is $O(d(B, T))$, which in expectation is at most $O(\log n)$ (Lemma 7).

If, however, x is the root of a subtree T, then we flatten T into a bucket before removing x. The cost of flattening is constant but its effect on the potential is

$-\Phi(T)$ just as for the insertion. As the root of T has been chosen randomly, the probability that the root of T is equal to the element to delete is $\frac{1}{s(T)}$. Hence this subtracts just $\frac{\Phi(T)}{s(T)} = O(1)$ in expectation from the potential debt, and thus adds just $O(1)$ to the amortized complexity.

Increasing and Decreasing Keys. Obviously the cost of *IncreaseKey* and *DecreaseKey* fits also into the amortized cost of the other operations.

Theorem 11. *The expected amortized cost of operations* Insert*,* FindMin*,* ExtractMin*,* Delete*,* IncreaseKey *and* DecreaseKey *over an initially empty LST simulating an RBST is* $O(\log n)$*, where* n *is the number of elements in the LST at the moment of executing the operation. An LST initialized by heapifying an array of* n *elements adds* $O(n)$ *to the overall cost of the sequence of operations.*

5. Experimental Results

We compare the performance of Randomized QHs with the original QHs on sequences of operations of the form $(\textit{ins}\text{-}(\textit{del}\text{-}\textit{ins})^k)^m(\textit{del}\text{-}(\textit{ins}\text{-}\textit{del})^k)^m$, where *ins* stands for inserting a random number and *del* for extracting the minimum [$San00$]. These sequences start and end with an empty heap. For $k = 0$ this is equivalent to HEAPSORT algorithm [$Wil64$], that is, we insert m random elements to the heap and then extract the minimum m times. The uniformity assumption holds in this sequence, so this is a good case for the original QHs. For larger k values, however, the inserted elements tend to be larger than the current average of the QH, thus breaking the uniformity assumption.

Figure 4 shows the performance for $k = 0$ to 4. We include the original QHs (QH), our new Randomized QHs (RQH), and the clasical binary heap (BH) implementing Wegener's speedup [$Weg93$]. Our machine is a 2.4 GHz Intel Core i5, with 4 GB of RAM and local disk, running Mac OS X Snow Leopard version 10.6.4. The algorithms were coded in C++, and compiled with `g++` version 4.2.1 optimized with `-O3 -DNDEBUG -m32 -ffast-math`. For each experimental datum shown, we averaged over at least 10 repetitions. We measure user times.

On the left we can see that BHs behave independently of k for small m values, but they worsen as m grows. This is a consequence of their poor locality of reference (the effect diminishes as k grows because more operations are carried out in the same area of the bottom of the heap). QHs perform logarithmically (straight line) for $k = 0$, exhibiting high locality of reference. However, as we move to $k = 1$ the cost increases because the *del* operation creates buckets and thus insertions are not all as cheap as with $k = 0$ (where they all fall in a single bucket). Apart from this, $k > 0$ deviates from the uniformity assumption and QH times become superlogarithmic, more markedly as k grows. Instead, RQHs are clearly logarithmic for any k. The only effect of k is, just as explained for QHs, the sudden increase from $k = 0$ to 1.

On the right side we compare the different techniques for $k = 0$ and $k = 2$. For $k = 0$ the first place is disputed between BHs and QHs for small m, and QHs take

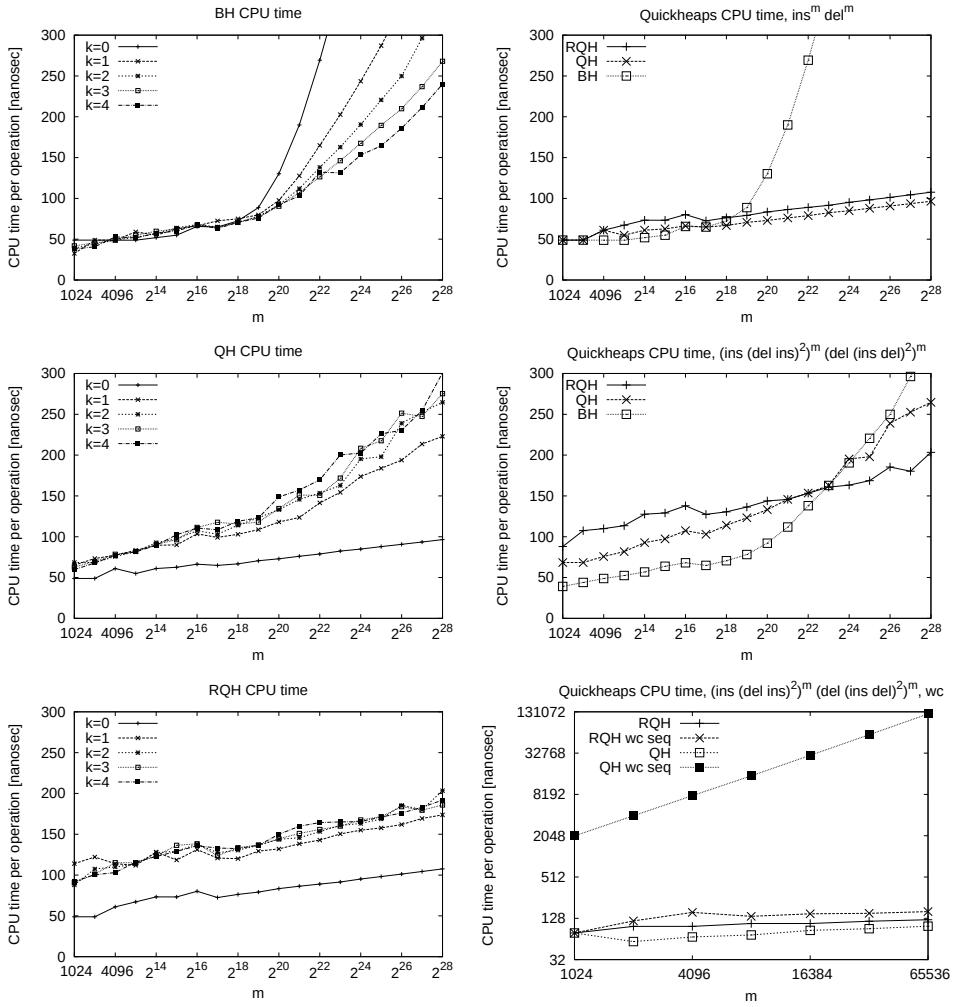

Fig. 4. Time performance of QHs, RQHs, and BHs, for the sequences of operations corresponding to $k = 0$ to 3. The x axis is m (logscale) and the y axis is the user time per operation. On the left we separate per data structure, on the right per k value. We use logscale in y for the last plot.

over for already small $m > 2^{16}$ (RQHs are always pretty close, at most 12% slower). For larger k, BHs are the best for moderate m values, but RQHs take over for larger $m > 2^{22} \ldots 2^{24}$, where they become significantly faster than QHs for $k = 1$ (up to 28%), $k = 2$ (up to 30%), $k = 3$ (up to 48%), and $k = 4$ (up to 56%).

In the last right plot we give the results for a variant of sequence $k = 2$ (wc) where the *ins* instructions insert the values in decreasing order. This produces a quadratic behavior on QHs, whereas RQHs stay logarithmic, their times just increased by at most 57% for $k = 1$, 32% for $k = 2$, 21% for $k = 3$, and 17% for $k = 4$.

6. Providing Worst-Case Amortized Guarantees

We now use LSTs to implement QHs in a way that guarantees logarithmic amortized cost for all the operations in the worst, rather than expected, case while still maintaining simplicity and practical efficiency. The idea is, as before, to maintain the property that the LST contains a leftmost path of a tree whose properties are known. This time our solution, α-Balanced QH, is inspired in Andersson's *General Balanced Trees (GBTs)* [*And*99] instead of RBSTs.

We assume the buckets of our LST represent *α-balanced* BSTs, that is, the size of the smallest subtree is at least α times the total, for some parameter $0 < \alpha < 1/2$. Moreover, we assume that the nodes in the LST contain *redundant* elements, which are created and used only for delimiting the buckets (and later might not be present in the QH). That is, the set of elements is formed by those contained in the buckets. We will also enforce that no bucket contains less than $\frac{1-\alpha}{\alpha}$ elements (except if the whole T is just a bucket); otherwise it will be merged with another bucket.

With these changes in mind, we (re)define the size $s(v)$ and the height $h(v)$ of an LST node.

Definition 12. *The* size *of an LST is*

$$s(bucket(B)) = |B|,$$
$$s(tree(r, L, B)) = s(L) + |B|.$$

Definition 13. *Let* $\beta = \frac{1}{1-\alpha}$*, thus* $1 < \beta < 2$*. The* height *of an LST is*

$$h(bucket(B)) = 1 + \lceil \log_\beta |B| \rceil,$$
$$h(tree(r, L, B)) = 1 + \max(h(L), 1 + \lceil \log_\beta |B| \rceil).$$

Note that $\ell(v) \le h(v)$, and that $h(bucket(B))$ is the maximum possible height of an α-balanced LST built from the elements in B, considering that nodes will be redundant: The recurrence $H(n) \le 1 + H((1-\alpha)n)$, $H(\frac{1-\alpha}{\alpha}) = 0$, has solution $H(n) \le 1 + \log_\beta n - \frac{\log \alpha}{\log(1-\alpha)}$, thus our h value is the minimum safe integer bound.

While $s(v)$ can be computed in constant time, all the $\ell(v)$ and $h(v)$ values for all the nodes v can be computed in $O(\ell)$ overall time. The number of elements in our LST T will be $n = s(T)$ (that is, we do not count the redundant elements). We will also maintain a global number d, the number of deletions since the last global reconstruction, and enforce the following two conditions, for some constants $c > 1$ and $b > 1$:

Condition 1. $h(T) \le 1 + \lceil c \log_\beta(n + d) \rceil$, otherwise we shorten the longest tree path somehow.

Condition 2. $d < (\beta^{\frac{b-1}{c}} - 1)\, n$, else we rebuild T to α-balance and reset $d = 0$.

As a consequence, an LST T of n nodes will ensure that $h(T) \le \lceil c \log_\beta(n) + b \rceil$. It will achieve $O(\ell(T)) = O(\log n)$ amortized time for any sequence of operations. For technical reasons that will be clear soon, our results will hold for any constant $c \ge 2$.

6.1. *Adapting LST Operations*

As we now manage redundant elements, we need to adapt the LST structures and operations to this case. Array A is as in the original LST (Section 3.2). However, in the stack S we store pairs (b_x, x), where b_x is the position of the first cell of the bucket (not the pivot position, which as mentioned may no longer exist in the bucket), starting from cell 0, and x is a copy of the pivot value. So integer p_∞ is now a pair $(p_\infty, \perp)$. Additionally, we add a new operation to merge buckets.

Convert an array B into $T = bucket(B)$. Remains with no change.

Compute $s(T_v)$: It is the next b_x value at the right of T_v, computed in constant time.

Insert/Delete x into/from bucket B_v: For the insertions/deletions, we have to move the first/last bucket element in order to create an empty cell. However, as we have copies of pivots in S, instead of moving a pivot we just increase/decrease its bucket bound. The rest is as in the original LST.

Extract the leftmost node (when the leftmost bucket is empty). Remains with no change.

Flatten subtree T_v into $bucket(B'_v)$. Assuming that T_v is a tree, we need to remove all the pairs (b_x, x) in T_v from the stack S, in time $O(1)$. The rest is as in the original LST.

Structure bucket B_v with pivot x: The partitioning is as in the original LST, where p_x is the final position of x after the partitioning. Then we push pair (p_x, x) into S in constant time.

Merge two consecutive buckets: Consider a tree $T_u = tree(u, T_v = tree(v, L_v, B_v), B_u)$. To merge B_v with B_u we have to delete the pair (b_u, u) from S, obtaining $T'_v = tree(v, L_v, B_v \cdot B_u)$. This costs $O(\ell(T_u))$.

6.2. *Priority Queue Operations*

Heapifying. We simply convert the array into a bucket. As $h(bucket(B)) = 1 + \lceil \log_\beta |B| \rceil \leq 1 + \lceil c \log_\beta |B| \rceil$, the invariant is established.

Insertion. We follow the normal insertion procedure, until we have to insert into a bucket B. At this point we simply use the LST insertion operation (which takes $O(d(B, T))$ time).

Next, we get back to the root computing $s(v)$ and $h(v)$ for all the nodes v in the leftmost path to B, recalling only the current values and the lowest v for which $h(v) > 1 + \lceil c \log_\beta s(v) \rceil$, in $O(\ell(T))$ overall time. If $h(root) = h(T) > 1 + \lceil c \log_\beta (n + d) \rceil$, then Condition 1 is violated and we have to restore it by doing some rebalancing at v (note that there must exist such a v).

As we see in Section 6.3, B cannot be a direct child of v, but rather must descend by v from the left. Let $T_v = tree(v, T_u = tree(u, L_u, B_u), B_v)$. We convert the subtree into $T'_u = tree(u, L_u, B_u \cdot B_v)$. This is carried out via the LST operation

of merging two buckets and it costs time $O(d(B,T))$. This reduces the height of the subtree by one (as shown in Section 6.3), and it is enough to restore the condition $h(T) \leq 1 + \lceil c \log_\beta(n+d) \rceil$, which was lost due to a height growth by 1 inside the left child of v (we will also show that h can only increase by 1).

Finding and Extracting Minimum. We structure the leftmost bucket essentially as in the randomized version, with some differences. (1) We structure until the leftmost bucket becomes of size $\leq \frac{1-\alpha}{\alpha^2}$, not zero. This guarantees that a further α-partitioning will not create blocks under the minimum size. We then look sequentially for the minimum within the leftmost block. (2) The pivot is copied, not moved, to the LST node we create, that is, the new node contains a redundant element. (3) We do not use a random pivot, but one that guarantees an α-partitioning, that is, so that the smallest of the two blocks is of size at least $\alpha|B|$. If we have to extract the minimum, and if after this the leftmost bucket gets below the minimum size, we concatenate the two leftmost buckets, in $O(1)$ time.

We also increase counter d during an extraction, and if Condition 2 does not hold anymore, we flatten the whole tree into a bucket in constant time and reset $d = 0$.

For producing an α-partition, we can for example use the idea of *introspective selection* [$Mus97, Val00$]: we try a constant number κ of random pivots and, if none produces an α-partition, we run a classical linear worst-case time median selection algorithm [$BFP+73$]. This one is rather slow, but will be invoked with low probability, $(2\alpha)^\kappa$. On average, $\frac{1}{1-2\alpha}$ attempts would be sufficient to find an α-partition. The process should behave in practice similarly to the randomized version, yet with a worst-case time guarantee.

Deletion. Using the dictionary, we find the node and, if it is in bucket B, we remove it in time $O(d(B,T))$. If B becomes below the minimum size, we remove it from the LST, together with its corresponding node (recall that nodes contain redundant keys, which might have been deleted), by merging with a neighbor bucket, in $O(d(B,T))$ time. We also increase d, and if Condition 2 does not hold anymore, we flatten the whole tree into a bucket in constant time, resetting $d = 0$.

6.3. *Amortized Analysis*

We will combine the potential debt function Φ used in Section 4.2 (which now uses our modified $s(v)$ definition) with a classical (credit) function Ψ inspired in the analysis by Andersson [$And99$]. Let v be an LST node and let v_H and v_L its higher and shorter children, respectively, that is, $h(v_H) \geq h(v_L)$ (note that one of the two is indeed a $bucket(B)$). In case of ties, v_H should be the left child. Now, we define

$$\delta(v) \quad = \quad \max(0,\ s(v_H) - \frac{1-\alpha}{\alpha} s(v_L)),$$

where we note that $\delta(v) = 0$ on a tree built by α-partitioning a bucket.

Definition 14. *The potential function* Ψ *of an LST is*

$$\Psi(bucket(B)) = 0,$$
$$\Psi(tree(r, L, B)) = \frac{1}{\gamma} \cdot \delta(tree(r, L, B)) + \Psi(L), \text{ where } \gamma = \frac{\beta^{1-1/c} - 1}{\beta - 1}.$$

We note that $0 < \gamma < 1$, thus $\frac{1}{\gamma} > 1$. The next lemma is essential for the analysis.

Lemma 15 ([*And*99]) *Let* T *be such that* $h(T) > 1+\lceil c \log_\beta n \rceil$*, and* v *be the lowest node on a longest path of* T *such that* $h(v) > 1 + \lceil c \log_\beta s(v) \rceil$*. Then* $\delta(v) > \gamma \cdot s(v)$*.*

Proof. The proof is similar to that of Andersson's Lemma 1 [*And*99]. Since v is the lowest node violating the balancing condition, v_H does not, that is, $h(v_H) \leq 1 + \lceil c \log_\beta s(v_H) \rceil$. Since $h(v) > 1 + \lceil c \log_\beta s(v) \rceil$ and $h(v) = 1 + h(v_H) \leq 2 + \lceil \log_\beta s(v_H) \rceil$, we have $\lceil c \log_\beta s(v) \rceil < 1 + \lceil c \log_\beta s(v_H) \rceil$. Thus the ceilings can be removed to obtain $s(v_H) > \beta^{-1/c} s(v)$. On the other hand, $\delta(v) \geq s(v_H) - \frac{1-\alpha}{\alpha} s(v_L) = s(v_H) - \frac{1-\alpha}{\alpha}(s(v) - s(v_H)) = \frac{1}{\alpha} s(v_H) - \frac{1-\alpha}{\alpha} s(v)$. By substituting $s(v_H)$ with the lower bound just derived we get $\delta(v) > (\frac{\beta^{-1/c}}{\alpha} - \frac{1-\alpha}{\alpha}) s(v) = \gamma \cdot s(v)$. □

We now prove some further technical lemmas that are needed. The first gives a lower bound to $h(v)$, the second establishes that an insertion can make a tree height grow by at most one, and the third shows that the insertion cannot occur too close to the lowest node that gets unbalanced.

Lemma 16. *For any node or bucket* v*,* $h(v) \geq 1 + \lceil \log_\beta s(v) \rceil$*.*

Proof. This is immediate if v is a bucket, by definition. Now, if $v = tree(r, L_v, B_v)$, assume inductively that $h(L_v) \geq 1 + \lceil \log_\beta s(L_v) \rceil$, whereas by definition $h(B_v) = 1 + \lceil \log_\beta |B_v| \rceil$. Assume $s(L_v) \geq |B_v|$, thus $s(v) \leq 2s(L_v)$. Then $h(v) \geq 1 + h(L_v) \geq 2 + \lceil \log_\beta s(L_v) \rceil \geq 2 - \log_\beta 2 + \lceil \log_\beta s(v) \rceil \geq 1 + \lceil \log_\beta s(v) \rceil$. The case $|B_v| \geq s(L_v)$ is symmetric. □

Lemma 17. *After a single insertion converting node or bucket* v *into* v'*, it holds* $h(v') \leq 1 + h(v)$*.*

Proof. If $v = B$ is a bucket converted into $v' = B'$, then $h(B) = 1 + \lceil \log_\beta |B| \rceil$ and $h(B') = 1 + \lceil \log_\beta (|B| + 1) \rceil$. If $h(B') > 1 + h(B)$, then $\log_\beta (|B| + 1) > 1 + \log_\beta |B|$, thus $|B| + 1 > \beta |B|$, which implies $|B| < \frac{1}{\beta - 1} = \frac{1-\alpha}{\alpha}$. This size is below the minimum permitted for any bucket B. Once the base case is proved, the inductive case where $v = tree(r, L_v, B_v)$ is trivial. □

Lemma 18. *Assume* v *is the lowest node such that* $h(v) > 1 + \lceil c \log_\beta s(v) \rceil$ *after an insertion. Then the insertion cannot have occurred in the child nor grandchild buckets of* v *if* $c \geq 2$*.*

Proof. Let $v = tree(r_v, T_u = (r_u, L_u, B_u), B_v)$, and let B be the bucket where the insertion occurred. We first show that $B \neq B_v$ if $c \geq 2$: If that were true, then if $h(v)$ had grown due to an insertion in $B = B_v$, then $h(v) = 2 + \lceil \log_\beta |B| \rceil$. On the other hand, $s(v) \geq |B|$. Hence $h(v) > 1 + \lceil c \log_\beta s(v) \rceil$ can hold only if $c < \frac{1+\log_\beta |B|}{\log_\beta |B|}$. Note $|B| \geq 1 + \frac{1-\alpha}{\alpha} = 1 + \frac{1}{\beta-1}$ as we have just inserted there. Thus $\frac{1+\log_\beta |B|}{\log_\beta |B|} \leq \frac{1+\log_\beta (1+\frac{1}{\beta-1})}{\log_\beta (1+\frac{1}{\beta-1})}$, which can be proven analytically to be < 2 whenever $1 < \beta < 2$. Thus we get $c < 2$, a contradiction with the assumption $c \geq 2$.

Using a similar argument, we can show that $B \neq B_u$ either: If that were the case, $h(v) = 3 + \lceil \log_\beta |B| \rceil$, and $s(v) \geq |B| + |B_u|$, thus it must be $c < \frac{2+\log_\beta |B|}{\log_\beta (|B|+|B_u|)}$. Since $|B| + |B_u| \geq 1 + \frac{2}{\beta-1}$, we have $c < \frac{2+\log_\beta (1+\frac{1}{\beta-1})}{\log_\beta (1+\frac{2}{\beta-1})}$, which can be shown analytically to be < 2, contradicting again the assumption.

(Interestingly, the argument holds for one more level; two more levels require a larger c.) □

We will define the amortized cost of operation i as $\tilde{c_i} = c_i - \Delta\Phi_i + \Delta\Psi_i$. Our final amortized cost will be $\sum c_i = \sum \tilde{c}_i + (\Phi_m - \Phi_0) - (\Psi_m - \Psi_0) = \sum \tilde{c}_i + \Phi_m - \Psi_m$. We consider now each operation in turn.

The final debt. The final debt, $\Phi_m - \Psi_m$, must be split among all the operations. We show that these two values cancel each other so that this final debt is $O(m)$ and thus adds a constant amortized cost to each operation. Let $v = tree(r, L, B)$, then $\Phi(v) = s(v) + \Phi(L) = |B| + s(L) + \Phi(L)$. Let us consider two cases: (1) $s(L) > |B|$ and (2) $s(L) \leq |B|$. In the first case, as $h(B) = 1 + \lceil \log_\beta |B| \rceil$ and $h(L) \geq 1 + \lceil \log_\beta s(L) \rceil$ (Lemma 16), it follows that $h(L) \geq h(B)$ and thus $\delta(v) \geq s(L) - \frac{1-\alpha}{\alpha}|B|$. Thus $\Phi(v) - \Psi(v) = (1 + \frac{1-\alpha}{\alpha\gamma})|B| - (\frac{1}{\gamma} - 1)s(L) + \Phi(L) - \Psi(L) < (1 + \frac{1-\alpha}{\alpha\gamma})|B| + \Phi(L) - \Psi(L)$. In case (2), it can be that $\Phi(v) - \Psi(v) \leq s(v) + \Phi(L) - \Psi(L)$, but we note that $s(L) \leq s(v)/2$. For the final bucket both Φ and Ψ are zero. Hence, as we go down the leftmost path, there are two types of nodes. Those of type (1) add $1 + \frac{1-\alpha}{\alpha\gamma}$ times their bucket size; those of type (2) form a recurrence that is never larger than the form $R(n) = n + R(n/2)$. Thus, we have a recurrence of the form $R(n) \leq \max\{(1 + \frac{1-\alpha}{\alpha\gamma})k + R(n-k), n + R(n/2)\}$, for arbitrary k, which can be proved to be $R(n) \leq (1 + \frac{1-\alpha}{\alpha\gamma})n$ by induction.

Heapifying. The analysis is identical to Section 4.2. The operation keeps $\Phi_1 = \Psi_1 = 0$.

Insertion. As in Section 4.2, if inserting into a bucket B, we carry out $c_i = d(B, T)$ comparisons, and also $\Delta\Phi = d(B, T)$. As for Ψ, we can at most increase by 1 each of the $\delta(v)$ values we go through, hence $\Delta\Psi \leq \frac{1}{\gamma} \cdot d(B, T)$. Overall, the amortized cost of an insertion is $\tilde{c_i} = c_i - \Delta\Phi + \Delta\Psi \leq \frac{1}{\gamma} \cdot d(B, T) = O(\frac{c}{\gamma} \log_\beta n)$.

In addition, we might have to merge the child and grandchild buckets of a node v in the insertion path after the insertion. By Lemma 18 we can assume that the subtree rooted at v is of the form $T_v = tree(v, T_u = tree(u, L_u, B_u), B_v)$

and the insertion did not occur in B_v but within T_u, thus the merging produces $T'_u = tree(u, L_u, B_u \cdot B_v)$.

As promised when we described insertion, we prove now that this merging reduces the height of the subtree by 1. Again by Lemma 18, the insertion must have occurred inside L_u. As $h(L_u)$ grew by 1 (Lemma 17) and made $h(v)$ grow, the height of v after the insertion and before merging the blocks was $h(v) = 2 + h(L_u)$. The new height of the subtree is $h(u') = 1 + \max(h(L_u), 1 + \lceil \log_\beta(|B_u| + |B_v|)\rceil)$. So we have to prove that $2 + h(L_u) > 2 + \lceil \log_\beta(|B_u| + |B_v|)\rceil$. If this were false, then $2 + \lceil \log_\beta(|B_u| + |B_v|)\rceil \geq 2 + h(L_u) > 1 + \lceil c \log_\beta s(v)\rceil = 1 + \lceil c \log_\beta(|B_u| + |B_v| + s(L_u))\rceil$. Hence $2 + \log_\beta(|B_u| + |B_v|) > 1 + c \log_\beta(|B_u| + |B_v|)$ and thus $\beta(|B_u|+|B_v|) > (|B_u|+|B_v|)^c$, which cannot happen if $c \geq 2$ since $|B_u|, |B_v| \geq \frac{1}{\beta-1}$ and $1 < \beta < 2$.

Now that we established correctness, we focus on the change of potential caused by this merging. We note that node v satisfies the condition of Lemma 15, and thus $\delta(v) > \gamma \cdot s(v)$. To see this, notice that the insertion path made $h(T)$ grow, and therefore v is the lowest node of a longest path, and its v_H is strictly higher than its v_L. Nodes lower than v within the insertion bucket B are sufficiently balanced and thus they cannot be candidates lower than v.

The merging of $B_u \cdot B_v$ produces $\Delta\Phi = -|B_u| - s(L_u)$. On the other hand, we have $\Delta\Psi = \frac{1}{\gamma}(\delta(u') - \delta(v) - \delta(u))$. Observe that, because $h(L_u) > 1 + \lceil \log_\beta |B_u|\rceil$, $\delta(u) = \max(0, s(L_u) - \frac{1-\alpha}{\alpha}|B_u|)$, and because $2 + h(L_u) > 2 + \lceil \log_\beta(|B_u| + |B_v|)\rceil$ (see above), it follows that $h(L_u) \geq 1 + \lceil \log_\beta(|B_u| + |B_v|)\rceil$, and thus $\delta(u') = \max(0, s(L_u) - \frac{1-\alpha}{\alpha}|B_u| - \frac{1-\alpha}{\alpha}|B_v|)$. Thus, $\delta(u') - \delta(u) \leq 0$, and therefore $\Delta\Psi < -\frac{1}{\gamma}(\gamma \cdot (|B_u| + |B_v| + s(L_u))) = -|B_u| - |B_v| - s(L_u)$. Thus $-\Delta\Phi + \Delta\Psi < -|B_v|$, and therefore $\tilde{c_i} = c_i - \Delta\Phi + \Delta\Psi < c_i = \ell(T) + O(1) = O(c \log_\beta n)$.

Finding and Extracting Minimum. Similarly to Section 4.2, the cost of structuring the leftmost bucket cancels out with the increase of $\Delta\Phi$. As our partitioning is α-balanced, $\delta(v) = \delta(tree(r, B, B')) = 0$ and thus $\Delta\Psi = 0$. Hence $\tilde{c_i} \leq 0$. On top of this we have a cost of at most $\frac{1-\alpha}{\alpha^2} = O(1)$, the maximum size of the leftmost bucket after structuring it, to find the minimum sequentially.

Removing the minimum also costs $O(1)$ because we use circular arrays. It causes $\Delta\Phi = -d(B, T)$ and may cause each $\delta(v)$ in the path grow by $\frac{1-\alpha}{\alpha}$ (if all the left children of nodes v happen to be v_L), thus $\Delta\Psi \leq \frac{1-\alpha}{\alpha\gamma} d(B, T)$. Thus the amortized cost is $\tilde{c_i} = c_i - \Delta\Phi + \Delta\Psi \leq (1 + \frac{1-\alpha}{\alpha\gamma})\ell(T) + O(1) = O((1 + \frac{1-\alpha}{\alpha\gamma}) c \log_\beta n)$.

In case we have to remove the leftmost bucket because its size becomes $< \frac{1-\alpha}{\alpha}$, the merging causes $\Delta\Phi = -|B'| - |B|$, but also $\delta(v) \geq |B'| - \frac{1-\alpha}{\alpha}|B|$ disappears from Ψ, thus $-\Delta\Phi + \Delta\Psi \leq -(\frac{1}{\gamma} - 1)|B'| + (1 + \frac{1-\alpha}{\alpha\gamma})|B|$. Thus the amortized cost is $\tilde{c_i} < c_i + (1 + \frac{1-\alpha}{\alpha\gamma})\frac{1-\alpha}{\alpha} = O(1)$. Further effects of the deletion of r are considered next.

Deletion. Removing a node at bucket B costs $d(B, T)$ comparisons and $\Delta\Phi = -d(B, T)$. It can also increase each $\delta(v)$ in the path by $\frac{1-\alpha}{\alpha}$, thus $\Delta\Psi \leq \frac{1-\alpha}{\alpha\gamma} \cdot d(B, T)$. Overall, $\tilde{c_i} \leq (2 + \frac{1-\alpha}{\alpha\gamma})\, d(B, T) = O((2 + \frac{1-\alpha}{\alpha\gamma}) c \log_\beta n)$.

Let $v = tree(r, L, B)$, where the deletion occurs in B. If B becomes smaller than $\frac{1-\alpha}{\alpha}$, it is removed (via merging with the bucket to its right) together with r. This changes the potential debt, $\Delta\Phi > -s(L) - \frac{1-\alpha}{\alpha}$, and also the credit, $\Delta\Psi = -\frac{1}{\gamma}\delta(v) < -\frac{1}{\gamma}(s(L) - \frac{(1-\alpha)^2}{\alpha^2})$. Hence, apart from the extra cost $d(B,T) = O(c\log_\beta n)$ of the merging, we have $-\Delta\Phi+\Delta\Psi < -(\frac{1}{\gamma}-1)s(L)+(1+\frac{1-\alpha}{\alpha\gamma})\frac{1-\alpha}{\alpha} = O(1)$ extra amortized cost. The situation is similar if B is the leftmost bucket, as seen in the previous operation.

If, however, the bucket that gets too small is the right child of the root, then we must merge it with the bucket to the left. Here $\Delta\Phi = -n$ and $\Delta\Psi \le -\frac{1}{\gamma}(s(root_H) - \frac{1-\alpha}{\alpha}|B|) = -\frac{1}{\gamma}(n - \frac{1}{\alpha}|B|) < -\frac{1}{\gamma}(n - \frac{1-\alpha}{\alpha^2})$. Thus the amortized cost is $\tilde{c_i} < c_i - (\frac{1}{\gamma} - 1)n + O(1) < c_i + O(1) = \ell(T) + O(1) = O(c\log_\beta n)$.

We note that a deletion decreases n, increases d, and cannot increase $h(T)$, thus if $h(T) \le 1 + \lceil c\log_\beta(n+d)\rceil$ held before the operation, it holds afterwards. We only carry out a global flattening of T when d reaches $(\beta^{\frac{b-1}{c}} - 1)\, n$. This guarantees that $1 + \lceil c\log_\beta(n+d)\rceil \le 1 + \lceil c\log_\beta(\beta^{\frac{b-1}{c}}n)\rceil = \lceil c\log_\beta(n) + b\rceil$ as promised. The global reconstruction costs $O(1)$ but it sets $\Phi = \Psi = 0$. The same analysis of the final debt shows that the amortized cost of the flattening is at most $(1 + \frac{1-\alpha}{\alpha\gamma})n$, and therefore we can split this cost among $(\beta^{\frac{b-1}{c}} - 1)n$ deletions. Thus we must charge deletions with an additional amortized cost of $\frac{1+(1-\alpha)/(\alpha\gamma)}{\beta^{\frac{b-1}{c}}-1} = O(1)$.

Just as in Section 4.2, we can now state a theorem with worst-case guarantees.

Theorem 19. *The amortized cost of operations* Insert*,* FindMin*,* ExtractMin*,* Delete*,* IncreaseKey *and* DecreaseKey *over an initially empty LST that simulates a GBT is* $O(\log n)$*, where* n *is the number of elements in the LST at the moment of executing the operation. An LST initialized by heapifying an array of* n *elements adds* $O(n)$ *to the overall cost of the sequence of operations.*

7. Going Cache-Oblivious

Both the original QHs and the variants introduced in this paper display high locality of reference, which explains their empirical success. This makes them good candidates for secondary memory implementations as well. Indeed, it was shown that the average amortized I/O cost of the original QHs is $O((1/B)\log(n/M))$, where B is the disk block size and M the amount of main memory available, assuming $M = \Omega(B\log n)$ for a QH of n elements [$NP10$] (this is not a standard assumption, but a pretty realistic one).

This result was obtained for the *cache-oblivious* model [$FLPR99, BF03$], where the algorithm is designed without knowledge of B or M, and then analyzed assuming an optimal paging strategy. This is not unrealistic for the asymptotics because, for example, LRU is 2-competitive if it uses twice the memory of the optimal algorithm [$ST85$]. The result is not yet the optimal $O((1/B)\log_{M/B}(n/B))$ [$BF03$], indeed

achieved by cache-oblivious priority queues [$San00, DKS07$], but rather close, and with the added value of the simplicity of the data structure.

We show now that that analysis [$NP10$] can be extended to our new variants. That is, without any modification, our QHs achieve $O((1/B)\log(n/M))$ amortized I/O cost per operation (in the expected or the worst case, depending on the variant). Just as in the previous article [$NP10$], we consider only operations *Heapify*, *Insert*, *FindMin*, and *ExtractMin*. The others require a dictionary for locating the elements in the array, which in secondary memory does not achieve the stated I/O bounds.

A first simple result arises from noticing that all our accesses to memory are either sequential traversals over a prefix of A or the top of S, or in the positions of A corresponding to block beginnings. Those positions move slowly, each of them at most one position forward per operation. It was shown [$NP10$] that, if the page scheduler caches the first two blocks of A, the whole S, and the page containing the beginnings of all the blocks, then the amortized I/O cost of each atomic operation is $O(1/B)$, and thus the amortized cost of the QH operations is $O(\log(n)/B)$. This analysis translates verbatim to our new variants. Note that we need $M = \Omega(B \log n)$ in order to maintain the pages where each block starts.

To improve the result, the analysis assumes that the scheduler splits M into two equal parts, leaving $M/2 = \Theta(M)$ memory for the above, and other $M' = M/2 = \Theta(M) \geq 2$ for caching a longer prefix of A. Therefore, any operation that occurs within $A[1, M']$ is I/O-free. Using a slightly modified potential function, it is shown that the amortized number of operations performed out of the prefix is $O(\log(n/M))$. Then one applies the argument of $O(1/B)$ amortized cost per such operation to obtain the final bound. We state now the theorem and then prove the details within each variant.

Theorem 20. *The amortized I/O-cost of operations* Insert*,* FindMin*, and* ExtractMin *over an initially empty QH, under the cache-oblivious model with disk page size B and memory size $M = \Omega(B \log n)$, is $O((1/B)\log(n/M))$, where n is the number of elements in the QH at the moment of executing the operation. A QH initialized by heapifying an array of n elements adds $O(n/B)$ to the overall cost of the sequence of operations. These bounds hold in the expected case for Randomized QHs and in the worst case for α-Balanced QHs.*

7.1. *Randomized QHs*

We follow the proof of the original article [$NP10$], showing that it applies to our Randomized QHs as well. It is first shown that only $O(\log(n/M))$ pivots point outside the "free" area, using an exponential-decreasing property that is a consequence of the uniform-key-distribution assumption. We can easily show that this is also the case in our structure, without any assumption.

Lemma 21. *Let T be an LST of size larger than M'. The average number of subtrees with size exceeding M' is $1 + H_n - H_{M'} = O(\log(n/M))$.*

Proof. The 1 comes from the whole LST; then we have to count how many times we choose pivots until falling within the first M' cells of A. We choose the first pivot at random. With probability M'/n it is already within that area; otherwise the pivot falls in the area $A[M'+1,n]$, where we count 1 and continue recursively. The recurrence is $p(M') = 0$ and $p(n) = \frac{1}{n}\sum_{i=M'+1}^{n}(1+p(i-1))$, whose solution is $p(n) = H_n - H_{M'}$. □

A new potential debt function is defined. We adapt it slightly because we do not include the leftmost bucket in our potential functions:

$$\Phi'(\mathit{bucket}(B)) = 0,$$
$$\Phi'(\mathit{tree}(r,L,B)) = \max(s(\mathit{tree}(r,L,B)) - M', 0) + \Phi'(L).$$

It can then be shown that the partitioning operation has zero amortized cost under this function: If we partition a leftmost block of size s, we work on $\max(s - M', 0)$ non-free cells, and this is precisely the amount by which $\Delta\Phi'$ increases when structuring the bucket, thus cancelling out with the real cost.

We now reconsider the analysis of the operations in a Randomized QH (Section 4.2), focusing only in the parts where the amortized cost is $O(\log n)$, to show that now it is $O(\log(n/M))$ (in particular, *Heapify* and *FindMin* are already $O(1)$ amortized time). It is convenient to redefine $d(B,T)$ as the depth of B in T, yet limited to counting only the blocks that start out of the prefix $A[1,M']$. Hence $d(B,T) = O(\log(n/M))$ by Lemma 21.

Insertion Assume we end up inserting element x in bucket B_v. Then we carry out $d(B_v,T)$ accesses (those within $A[1,M']$ are free). The change in the potential debt Φ' can be adapted from that of Section 4.2 as follows:

$$\begin{aligned}\Delta\Phi'(T',T) &= \Phi'(\mathit{tree}(r,L',B)) - \Phi'(\mathit{tree}(r,L,B))\\ &= \max(s(T') - M', 0) + \Phi'(L') - \max(s(T) - M', 0) - \Phi'(L)\\ &= 1 + \Delta\Phi'(L',L).\end{aligned}$$

This continues until $T = \mathit{bucket}(B_v)$ or $T = \mathit{tree}(r,L,B_v)$ and thus x is inserted into bucket B_v, where the difference is also 1. It also finishes once $s(T') \le M'$, that is, T' falls within the free prefix. This happens after $d(B_v,T)$ steps, thus $\Delta\Phi_i = d(B_v,T)$, and therefore $\tilde{c}_i = c_i - \Delta\Phi_i = 0$. Flattening is easily seen to retain constant amortized cost.

Extraction. We must reanalyze the effect of flattening the leftmost node using the new potential function. We have $\Delta\Phi'(T',T) = -1 + \Delta\Phi'(L',L)$ when both T and T' are not buckets and are outside of the free prefix (thus the -1 is added through $O(\log(n/M))$ recursive steps). If we reach the case $T = \mathit{tree}(r,B_\emptyset,B)$ and $T' = \mathit{bucket}(B)$, and T is still outside the free prefix, we have:

$$\begin{aligned}\Delta\Phi'(T',T) &= \Phi'(\mathit{bucket}(B)) - \Phi'(\mathit{tree}(r,B_\emptyset,B))\\ &= 0 - \max(s(\mathit{tree}(r,B_\emptyset,B)) - M', 0)\\ &= -1 - |B| + M',\end{aligned}$$

and therefore $-\Delta\Phi'_i = O(\log(n/M)) + |B| - M'$. This B is the right child of the leftmost node in the tree, and thus its expected size is H_n by Lemma 6. Hence we require $M' \geq H_n$ in order to cancel it and retain $O(\log(n/M))$ expected amortized cost. This is not a problem because we are already assuming $M' = \Omega(B \log n)$.

7.2. *α-Balanced QHs*

The α-balancing implies that there are at most $\log_\beta(n/M')$ block beginnings out of the free prefix, and therefore again $d(B,T) = O(\log(n/M))$ under our redefinition of $d(B,T)$. We use the same Φ' debt function, and a new Ψ' credit function that differs from Ψ in that $\Psi'(T) = 0$ if $s(T) \leq M'$. The final debt and operation *Heapify* are trivial because they already add $O(1)$ cost per operation.

Insertion As in Section 6.3, if inserting into a bucket B, we carry out $c_i = d(B,T)$ accesses (recall that $d(B,T)$ does not count the accesses within the free prefix), and also $\Delta\Phi' = d(B,T)$. On the other hand, Ψ' increases by 1 each of the $\delta(v)$ values we go through, yet it accounts only for those outside the free prefix, hence $\Delta\Psi' \leq \frac{1}{\gamma} \cdot d(B,T)$. Thus $\tilde{c_i} = c_i - \Delta\Phi' + \Delta\Psi' = O(\log(n/M))$.

Another point to analyze is the merging of blocks B_u and B_v. This does not change the potential unless $s(T_v) > M'$, in which case $\Delta\Phi' \geq -|B_u| - s(L_u) + M'$. On the other hand, $\Delta\Psi' \leq \frac{1}{\gamma}\delta(v)$ (which is counted because T_v is larger than M'). Hence $-\Delta\Phi' + \Delta\Psi' < |B_v| - M'$ and therefore the amortized cost stays $O(\log(n/M))$.

Extracting Minimum The change in the potentials is easily seen to be $O(d(B,T)) = O(\log(n/M))$. Possible removal of the empty leftmost bucket causes only a constant change in the potential.

8. Conclusions

We have presented a randomized variant of Quickheaps (QHs), a recent priority queue implementation that is simple to code, efficient in practice, and offers logarithmic expected time for all the basic operations when some uniformity distribution properties hold on the sequence of operations. We proved that Randomized QHs offer the same time guarantees without any assumption on the distribution of operations, which makes them more robust than the original QHs, and useful in applications where the distribution assumptions do not hold. Our experimental results show that Randomized QHs are almost as fast as the original ones when the operations follow the uniformity assumption, and retain their good performance on sequences that make the original QHs much slower.

We have also introduced a variant of QHs that offers worst-case amortized logarithmic time bounds for all the operations. It retains locality of access and is only slightly more complicated than randomized QHs.

The original QHs are also cache-efficient, and offer a cache-oblivious implementation on secondary memory [$NP10$]. We have also shown that these properties are

inherited in our stronger variants.

Future work involves implementing the worst-case variant and comparing it with the original and randomized QHs, especially in concrete applications where the uniform distribution properties assumed in the analysis of the original QHs do not hold. We expect the worst-case variant not to be much slower than the randomized one, in exchange for guaranteeing worst-case amortized performance.

References

[And99] A. Andersson. General balanced trees. *J. of Algorithms*, 30(1):1–18, 1999.

[BF03] G. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. 35th ACM Symp. on Theory of Computing (STOC)*, pages 307–315, 2003.

[BFP+73] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. of Computer and System Sciences*, 7(4):448–461, 1973.

[DKS07] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard Template Library for XXL data sets. *Software Practice & Experience*, 38(6):589–637, 2007.

[FLPR99] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th IEEE Symp. on Foundations on Computer Science (FOCS)*, pages 285–297, 1999.

[FSST86] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[FT87] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. of the ACM*, 34(3):596–615, 1987.

[Hoa61] C. Hoare. Algorithm 65 (FIND). *Comm. ACM*, 4(7):321–322, 1961.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[MR98] C. Martínez and S. Roura. Randomized binary search trees. *J. of the ACM*, 45(2):288–323, 1998.

[Mus97] D. R. Musser. Introspective sorting and selection algorithms. *Software Practice & Experience*, 27(8):983–993, 1997.

[NP10] G. Navarro and R. Paredes. On sorting, heaps, and minimum spanning trees. *Algorithmica*, 57(4):585–620, 2010.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[San00] P. Sanders. Fast priority queues for cached memory. *ACM J. of Experimental Algorithmics*, 5, 2000.

[ST85] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. ACM*, 28(2):202–208, 1985.

[ST86] D. D. Sleator and R. E. Tarjan. Self adjusting heaps. *SIAM J. on Computing*, 15(1):52–69, 1986.

[Val00] J. Valois. Introspective sorting and selection revisited. *Software Practice & Experience*, 30(6):617–638, 2000.

[vEBKZ77] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[Vui78] J. Vuillemin. A data structure for manipulating priority queues. *Comm. ACM*, 21(4):309–315, 1978.

[Weg93] I. Wegener. BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small). *Theoretical Computer Science*, 118(1):81–98, 1993.

[Wil64] J. Williams. Algorithm 232 (HEAPSORT). *Comm. ACM*, 7(6):347–348, 1964.