The Logarithmic Funnel Heap: A Statistically Self-Similar Priority Queue

Christian Löffeld

May 31, 2017

Abstract

The present work contains the design and analysis of a statistically self-similar data structure using linear space and supporting the operations, insert, search, remove, increase-key and decrease-key for a deterministic priority queue in expected $O(1)$ time. Extract-max runs in $O(\log N)$ time. The depth of the data structure is at most $\log^* N$. On the highest level, each element acts as the entrance of a discrete, $\log^* N$-level funnel with a logarithmically decreasing stem diameter, where the stem diameter denotes a metric for the expected number of items maintained on a given level.

1 Introduction

A priority queue is an abstract data structure based on the concept of a heap [1]. It maintains a collection of objects such that the largest key, for some predefined key type, is accessible in $O(1)$ time. It is one of the most ubiquitously utilized data structures in computer science. Priority queues are employed in a variety of applications, such as scheduling, event simulation, and graph analysis [2]. A great number of ideas and approaches have shaped the development of priority queues [3]. A very efficient example of a priority queue is based on van Emde Boas Trees [4] and widely used in network routers. Various priority queues have been devised that exhibit optimal running times for multiple supported operations. Among them, the Pairing-Heap, invented by Fredman et al. [5], the Fibonacci heap [6], Brodal Queues [7], and the Rank-Pairing Heap by Haeupler et al. [8]. Cherkassky et al. [9, 10] introduced the notion of multi-level bucket heaps which are efficient data structures for shortest-path algorithms. Raman [10] and Thorup [11] have designed efficient monotone priority queues for shortest path algorithms.

The present work contains the design and analysis of a dynamically adjusted and deterministic multi-level data structure supporting the operations of a priority queue. The depth of the data structure is at most $\log^* N$. Each level of the data structure is an implicit binary heap. On the highest level of the data structure, each element of the heap acts as the entrance to a discrete logarithmic funnel. In fact, the data structure acts like an array of discrete logarithmic funnels. The number of items in a level increases exponentially when ascending from a level to the next higher one.

The data structure exhibits statistical self-similarity, i.e. it is mathematically identical, for instance, to coastlines and fern leaves. Statistical self-similarity may be interpreted as an engineering tool. Buades et al. have used the concept to construct image-denoising algorithms.

---

*Email: christian.loffeld@gmail.com

1 Let $\log N$ denote the binary logarithm of $N$. The total number of items maintained by the priority queue is represented by $N$. 

1
For each level of the data structure, a quantitative closed-form relationships that facilitates implicit internal item load balancing can be derived. These relationships contain the LambertW-function [14] which is utilized, among others, in the analysis of quantum-mechanical and general-relativistic problems [15], as well as biochemical kinetics [16].

The data structure supports the following priority queue operations,

- **make-heap():** set up an empty priority queue
- **max-item():** return reference to item with the largest key.
- **extract-max():** remove item with largest key.
- **insert(k):** insert item with key k (data is optional). Return unique identifier id.
- **remove(id):** remove item with identifier id.
- **increase-key(id, k):** increase key of item with identifier id to value k.
- **decrease-key(id, k):** decrease key of item with identifier id to value k.
- **search(id, f):** apply function f to data of item with identifier id.

**Organization.** The contents of is report is laid out into four parts. Initially, basic structural concepts regarding design and implementation of the data structure supporting the priority queue are outlined. In the second part, the general mathematical setting for the data structure is derived. The functions supported by the priority queue system are discussed and analyzed in the third part, which is then followed by some concluding remarks.

2 Preliminaries

Here, we outline a statistically self-similar multi-level priority queue system that is based on the notion of the implicit binary heap.

A priority queue with i levels is referred to as $pQ_i$. It is constructed from a hash table, a stack, and two additional structural concepts, namely the meta-heap and the common-heap, both of which are implicit binary max-heaps. Each element of the priority queue $pQ_i$ is a common-heap that contains a priority queue of type $pQ_{i-1}$. The total number of items $n_i$ maintained by $pQ_i$ determines its composition in terms of the number $k_i$ and expected sizes $n_{i-1}$ of its common-heaps.

On a local level, the hash table in $pQ_i$ is used to store the item location, i.e. the heap-ID of the common-heap that maintains a particular item. Globally, it facilitates the sequential referral to the particular $pQ_0$ binary heap that ultimately stores the item in an array.

The stack maintains the IDs of common-heaps that are temporarily suspended during a size reduction procedure of $pQ_i$. This mechanism facilitates the recycling of common-heap allocations if $pQ_i$ needs to add a common-heap during a growth procedure.

The meta-heap of level i is the backbone of the priority queue $pQ_i$. It is an array of pointers to all common-heaps in $pQ_i$, and maintains the local max keys of all common-heaps in heap-order, i.e. the first element of the meta-heap points to the common-heap that contains the global max key, the second element points to the second largest key, etc. This arrangement facilitates look-up of the global max element at the head of the meta-heap in $O(1)$ time. This arrangement also means that the size of the meta-heap is always the same as the number of common-heaps (see Figure 1).

Like the pointers in the meta-heap, the common-heaps are also maintained by an array. A common-heap is a structure that contains members to store (i) the common-heap ID, which is
Figure 1: Graphical representation of the bidirectional linking between the meta-heap and the common-heaps in the priority queue \( pQ_i \) on level \( i \) s.t. \( \forall \; i > 0 \).

Figure 2: Graphical representation for an \( \alpha \)-level priority queue system with \( \alpha = 3 \). The large boxes contain the meta-heap on level \( i \). Each element points to a common-heap of expected size \( n_{i-1} \), which itself contains a meta-heap for level \( i-1 \). The expected size of a common-heap from level \( i \) is \( \log n_i \) \( \forall \; i > 1 \) and \( \sqrt{\log n_i} \) if \( i = 1 \). The structure of the priority queue system is reminiscent of self-similar entities such as fractals, fern leaves or coastlines.

its immutable index location in the array, (ii) the temporary address of the meta-heap pointer that currently points to it, and (iii) an implicit binary max-heap. In the priority queue \( pQ_i \), this binary max-heap is implemented as the priority queue \( pQ_{i-1} \), while in the base priority queue \( pQ_1 \), it is implemented as an one-dimensional array, or more specifically, vectors of items (see Figures 3 and 4, respectively). The statistically self-similar nature of an \( \alpha \)-level priority queue system is illustrated in Figure 2. It indicates that each element pointed to by the level \( i \) meta-heap is a common-heap containing \( k_{i-1} \) common-heaps of expected size \( n_{i-1} \).

The common-heaps store the address of the meta-heap element that points to it in order to compute in \( O(1) \) time the location in the meta-heap where a heap-property violation may have occurred as the result of the removal or modification of the key of an item. Further, the heapID is maintained so that the heap location of an item can be easily updated in the hash
An empty priority queue is initialized by setting up a bidirectional pointer link between a common-heap and a position in the meta-heap, and by also assigning to the common-heap its ID, which is zero, assuming array indexing starts from zero.

Also, since dynamic memory allocation typically leads to pointer invalidation, the meta-heap array, as well as the array containing the common-heap structures, is initialized to be of fixed-size. This choice is not a technical necessity, but it simplifies the algorithms and thus the subsequent analysis of the main ideas. Further, this choice sets an upper limit on the number of items the priority queue can maintain in order to guarantee the worst-case time bounds for its supported operations.

Properties of some Fundamental Data Structures and Notation

Throughout the following discussion, frequent use is made of the following properties of some fundamental data structures. We also introduce some notation that will be helpful.

- **binary heap as an array of size n**: Insertion and max-extraction of items are $O(\log n)$ time operations. Making a heap requires $O(n)$ operations.
  - *push the heap*: insert item into heap of size $n$, and restore heap order in the new range $[0, n+1]$
  - *pop the heap*: extract max item in heap of size $n$, and restore heap order in the new range $[0, n-1]$
  - *update the heap*: perform *push heap*, *pop heap* or both
  - *make a heap*: generate a heap from $n$ elements

- **hash table**: insertion and look-up are $O(1)$ expected time operations.

- **stack**: insertion and extraction are $O(1)$ time operations.
3 The α-Level Logarithmic Funnel System

On the highest level, α, the priority queue maintains the total number of items, \( n_\alpha = N \). An item type must hold a key of a comparable type, such as an integer or a word, as well as a unique numeric ID. The latter is assigned to it on level \( \alpha \). We define the system equation for level \( i \), such that the expected \( n_i \) items are arranged into \( k_i \) common-heaps of expected size \( n_i - 1 \),

\[
n_i = k_i n_{i-1}, \tag{3.1}
\]

and where we define the equilibrium relations on level \( i \) as

\[
n_{i-1} = \begin{cases} \log k_i & \text{if } i > 1 \\ \sqrt{\log k_i} & \text{if } i = 1 \end{cases} \tag{3.2}
\]

Onto this recursive system of common-heaps, we want to inscribe the operations of a priority queue such that the work performed by a function on level \( i \) is the sum of the work performed on level \( i - 1 \) and a constant contribution from level \( i \) itself. Formally, we require functions that can be described by the first-order recurrence relation with constant coefficients,

\[
T_i = c_1 T_{i-1} + c_0. \tag{3.3}
\]

The general solution of the recurrence relation Eq. 3.3 for an α-level system is then,

\[
T_\alpha = c_1^{\alpha-1} T_1 + c_0 \sum_{j=0}^{\alpha-2} c_1^j \tag{3.4}
\]

\[
= c_1^{\alpha-1} T_1 + \frac{c_0}{c_1 - 1} (c_1^{\alpha-1} - 1) \tag{3.5}
\]

\[
= O(T_1) \tag{3.6}
\]

With a base case, \( T_1 = O(\sqrt{\log n_1}) = O(n_0) \), the associated time complexity of the general solution for an α-level system is

\[
T_\alpha = O(n_\alpha). \tag{3.7}
\]

From Eqns. 3.1 and 3.2, we may derive the relation between the number of items managed by common-heaps on adjacent levels as follows, \( n_{i-1} \leq \log n_i \forall i > 1 \) and \( n_{i-1} \leq \sqrt{\log n_i} \) for \( i = 1 \). Thus, a common-heap on level \( i - 1 \) is logarithmically smaller than a common-heap on level \( i \). We can state the general solution of the system in terms of the total number of items \( n_\alpha \), maintained by a α-level priority queue system as

\[
T_\alpha = O((\log \log ... \log n_\alpha)^\frac{1}{\alpha}) \tag{3.8}
\]

Thus, in a system with \( \log^* n_\alpha \) levels, the functions adhering to Eq. 3.3 can be considered to have \( O(1) \) expected time complexity.

Combining equations 3.1 and 3.2, the relationship between the expected number of items \( n_i \) and the number of common-heaps \( k_i \) on level \( i \) may be stated as follows,

\[
n_i = \begin{cases} k_i \log k_i & \text{if } i > 1 \\ k_i \sqrt{\log k_i} & \text{if } i = 1 \end{cases} \tag{3.9}
\]

From equations 3.9, we derive load balancing functions that are used to implicitly facilitate the realization of relations 3.1 and 3.2. Specifically, we obtain

\[
k_i = \begin{cases} \exp(W_0(n_i)) & \text{if } i > 1 \\ \exp(\frac{1}{2}W_0(n_i^2 \ln 4)) & \text{if } i = 1 \end{cases} \tag{3.10}
\]
\( W_0 \) represents the LambertW-function on the main branch \( W_0 \). The control is considered implicit, because not the expected sizes \( n_{i-1} \), but the number \( k_i \) of common-heaps is controlled using the total size \( n_i \) of items maintained by level \( i \).

In the following sections, we develop the full priority queue system with functions for item insertion, search, removal, increase-key and decrease-key that can be described by the recurrence relation Eq. 3.3, and such that the associated time complexities on an \( \alpha \)-level system can be described by Eq. 3.8.

4 Analysis
Let \( \Omega \) be the set of functions supported by the \( \alpha \)-level priority queue system as described in Section 3. The set contains the functions \( \text{insert()} \), \( \text{search()} \), \( \text{remove()} \), \( \text{increase-key()} \), and \( \text{decrease-key()} \). Pseudo-code of all functions supported by the priority queue system can be found in the supplementary material.

Theorem 4.0.1. Let \( N \) be the total number of items maintained by the \( \alpha \)-level priority queue system, and let \( \phi \) be a function from the set \( \Omega \), then the operational cost of \( \phi \) on level \( \alpha \), \( T_{\phi,\alpha} \) can be described by Eq. 3.8, i.e. \( T_{\phi,\alpha} = O((\log \log \ldots \log N)^{\frac{1}{2}}) \).

We prove this statement by induction, i.e. showing for each of the functions \( \phi \) in \( \Omega \) the following two lemmas,

Lemma 4.0.2. Let \( n_1 \) be the total number of items maintained by a level-1 priority queue \( pQ_1 \), then the operational cost of \( \phi \) on level 1 is \( T_{\phi,1} = O(n_0) \).

Lemma 4.0.3. The operational cost of \( \phi \) on level \( i > 1 \) is described by a linear first-order recurrence relation with constant coefficients, \( T_{\phi,i} = c_{\phi,1} T_{\phi,i-1} + c_{\phi,0} \).

4.1 The insert-function
The insert() function contains a mechanism referred to as tunneling. It is loosely derived from the quantum-mechanical tunneling phenomenon which facilitates overcoming a potential energy barrier by "tunneling" through it. The mechanism guarantees constant cost insertion to items whose key is larger than the local max key of the common-heap, to which the item has been uniformly assigned to at first. The tunneling mechanism facilitates a passage to common-heaps that are pointed to by meta-heap pointers with indices in the half-open interval \( [0,t_B) \), where we denote the index \( t_B \) as the tunneling barrier at position \( 2^c - 1 \), and where \( c \) is a constant. In the domain \( [0,t_B) \), meta-heap updates will incur an operational cost of \( O(\log 2^c) \equiv O(1) \). Next, we prove Theorem 4.0.1 for insert().

Lemma 4.1.1. Lemma 4.0.3 with \( \phi = \text{insert()} \)

Proof. Insertion of an item \( x \) into the priority queue \( pQ_1 \) is performed as follows. The item is assumed to hold an integer key, and a unique numeric ID provided by the caller of insert(), namely the priority queue \( pQ_2 \). Firstly, the queue size is incremented. Then, a continuously looping index variable \( i \) visiting all meta-heap locations, selects a pointer in the meta-heap and thus, implicitly a common-heap, \( A \) of expected size \( n_0 \). These operations all require \( O(1) \) time. Selection is performed implicitly in order to know by construction, the index location \( i \) within the meta-heap where a heap property violation as a result of item insertion may occur, and so that the heap-property in the meta-heap can be restored in the interval \( [0,i] \).

\(^2\)For convenience, functions are referred to in the text by their name suffixed with closed brackets, e.g. insert()
Next we will distinguish two cases. Firstly, if common-heap \( A \) is empty or \( x.key \) is smaller or equal than the current local max key of \( A \), the algorithm proceeds to insert item \( x \) into \( A \) by (1) adding a \((x.itemID, A.heapID)\)-pair to the hash table, (2) adding the item itself to the array of common-heap \( A \), and (3) restoring its heap-property. This sequence of steps incurs a cost of \( O(\log n_0) \) time. Now, only if the common-heap \( A \) was empty before the insertion, a meta-heap update is required. Thus, this update occurs exactly once for each of the \( k_1 \) common-heaps in \( pQ_1 \). The total cost \( C_T \) for these operations is therefore, \( O(k_1 \log k_1) = O(k_1 n_0^2 \beta_1) \). Recall the system equation 3.1 for level 1, \( n_1 = k_1 n_0 \), and distribute the total cost \( C_T \) among the \( n_1 \) items in \( pQ_1 \). The amortized cost for the \( k_1 \) meta-heap updates is then \( C_T/n_1 = O(n_0) \). Aggregating all partial costs for this branch then yields an amortized branch cost of \( O(n_0) \) time.

In the second case, \( x.key \) is larger than the current local max key of common-heap \( A \), and inserting \( x.key \) into \( A \) would thus require a meta-heap update of cost \( O(\log k_1) \) time. Instead, item \( x \) is tunnelled to and inserted into one of the common-heaps pointed to by meta-heap pointers with indices in the half-open interval \([0, t_B)\). In Lemma 4.1.2 below, the amortized cost \( T_{\text{tunnel},1} \) of tunnel() on level 1 is established to be \( O(\log n_0) \) time.

Finally, an \( O(1) \) time check is performed that ensures that not too few common-heaps are available in order to keep the expected size of the common-heaps at the level required by the system equation, i.e. near equilibrium defined by Eq. 3.2. The check is optionally lenient, i.e. it allows for an adjustable deviation from optimality before facilitating the addition of a common-heap by the function \( \text{grow}() \). In Lemma 4.1.4 the expected amortized cost \( T_{\text{grow},1} \) of \( \text{grow}() \) is determined to be \( O(n_0) \) time.

By aggregating the operational costs for all branches, the expected amortized cost of \text{insert}() on level 1 may be expressed in the form,

\[
T_{\text{insert},1} = \max\{O(n_0), T_{\text{tunnel},1} \} + T_{\text{grow},1} = \max\{O(n_0), O(\log n_0)\} + O(n_0)
\]

Thus, the operational cost for \text{insert}() on level 1 is \( T_{\text{insert},1} = O(n_0) \).

**Lemma 4.1.2.** On level 1, the operational cost of \text{tunnel}() is \( O(\log n_0) \).

**Proof.** As stated above, the \text{tunnel}() function facilitates insertion of an item \( x \) into one of the \( t_B = 2^c \) common-heaps that are pointed to by meta-heap pointers with indices in the half-open interval \([0, t_B - 1)\). A specific common-heap \( A \) is selected by an index variable that is continuously looping this interval. Identically to the operations performed in the first branch of \text{insert}(), a \((x.itemID, A.heapID)\)-pair is added to the hash table, the item itself is appended to the array of common-heap \( A \), and its heap-property is subsequently restored. The amortized cost for these operations, is \( O(\log n_0) \) time. In contrast to the insert function, it is then tested whether item \( x \), after inclusion into \( A \), now constitutes the new local max item of \( A \). If true, a meta-heap update is required. Since the common-heap \( A \) is guaranteed to be located within the meta-heaps interval \([0, t_B - 1)\), this update is \( O(e) \equiv O(1) \) time. The expected cost of \text{tunnel}() on level 1 is therefore \( O(\log n_0) \) time.

**Lemma 4.1.3.** In \( pQ_1 \), the relative frequency of calls to \text{grow}() from \text{insert}(), \( \beta_1 \) is expected \( O(1/n_0) \).

**Proof.** Let \( n_1 \) be the current number of elements maintained by \( pQ_1 \), and let \( k_1^* \) be the associated optimal number of common-heaps. We want to know how many additional elements \( \Delta n_1 \) can be added to \( pQ_1 \) before \( k_1^* + 1 \) is the optimal number of common-heaps for \( pQ_1 \). The number of common-heaps \( k_1 \) can be expressed as a function of the number of elements \( n_1 \) as shown in Eq. 3.10.

\[
k_1 = \exp(W_0(n_1^2 \ln(4))/2),
\]
where $W_0$ represents the LambertW-function on the main branch $W_0$. Upon the substitutions, \( k_1 := k_1^* + 1 \), and \( n_1 := n_1 + \Delta n_1 \), the expression is solved for \( \Delta n_1 \), and we obtain the relation,

\[
\Delta n_1 = \sqrt{\log k_1^* + 1} \approx n_0.
\]  

(4.4)

It shows that the required increase \( \Delta n_1 \) in the number of elements is approximately equal to the average number of elements \( \sqrt{\log k_1^*} = n_0 \) in a common-heap on level 1. This result is very intuitive. It says that a new common-heap is required only if enough items have been added in order to fill up a common-heap. Hence, only after the number of elements \( n_1 \) in \( pQ_1 \) has increased by \( \Delta n_1 \approx n_0 \) elements, an additional common-heap is recruited to restore the balance required by the system equation. Only \( n_0 \) consecutive calls to \( \text{insert}() \) can accomplish this worst case, every other combination of operations supported by \( pQ_1 \) will lead to fewer calls to \( \text{grow}() \). This means that the grow function will be called at most on every \( n_0 \) call to \( \text{insert}() \), and therefore, the relative frequency of calls to \( \text{grow}() \), \( \beta_1 = O(1/n_0) \).

Lemma 4.1.4. On level 1, the amortized cost of \( \text{grow}() \) is \( O(n_0) \) time.

Proof. We established in Lemma 4.1.3 that the function \( \text{grow}() \) is called on at most every \( n_0 \) call to \( \text{insert}() \). The function is concerned only with setting up and adding a new common-heap \( B \) to \( pQ_1 \). Growing the priority queue is performed by selecting the common-heap \( A \) that is currently being pointed to by a pointer at \( \text{metaHeap}[j] \), where \( j \) is the index variable that is continuously looping the constant-cost interval \([0, t_B - 1]\). Then, a sequence of \( O(1) \) time book-keeping steps are executed. These operations set up and bidirectionally link \( B \) with the meta-heap. In case the selected common-heap \( A \) contains less than two items, \( \text{grow}() \) is done. Otherwise the lower half of \( A \)'s heap elements is transferred to the new common-heap \( B \). The associated memory of \( A \) is subsequently cleared, and thus, two operations with expected \( O(n_0) \) time cost are incurred. After the item transfer is completed, \( B \) has to be transformed into an actual heap. This operation also has expected \( O(n_0) \) time cost associated with it. Finally, a meta-heap update is required and incurs a cost \( O(\log k_1) \) time. The probability of selecting a common-heap \( A \) with fewer than two items depends on the nature of the sequence of operations performed by \( pQ_1 \). Without further analysis, we assume this probability to be small and thus, the total expected cost of the grow function is \( O(\log k_1) \). With \( \beta_1 = O(1/n_0) \), the expected amortized cost for \( \text{grow}() \) is \( O(n_0) \).

Lemma 4.1.5. Lemma 4.0.3 with \( \phi = \text{insert}() \)

Proof. The procedural characteristics of this function are effectively identical to the insert function described for \( pQ_1 \). At first, a common-heap \( A \) on level \( i \) is uniformly selected by a continuously iterating index variable. Then, the same two cases are considered before the algorithm proceeds with the insertion of item \( x \).

If the common-heap is empty or \( x\text{.key} \) is less than or equal to the local max key of \( A \), the item is inserted directly using the level \( i - 1 \) method \( \text{insert}() \), incurring a cost \( T_{\text{insert},i-1} \).

Alternatively, \( x\text{.key} \) is larger than the current local max key. In this case, item \( x \) is tunneled to one of the constant-cost common-heaps in the level-\( i \) meta-heap interval \([0, t_B]\). In Lemma 4.1.6 the operational cost for level-\( i \) tunnel() is established to be \( O(T_{\text{insert},i-1}) \).

Lastly, the algorithm may call \( \text{grow}() \) in case a new common-heap has to be added to \( pQ_1 \). As shown in Lemma 4.1.8, the amortized cost for level-\( i \) \( \text{grow}() \) is \( O(1) \).

Thus, the expected amortized cost incurred by the insert function on level \( i \) can be expressed as follows

\[
T_{\text{insert},i} = \max\{ T_{\text{insert},i-1}, T_{\text{tunnel},i} \} + T_{\text{grow},i}
\]

(4.5)

\[
= \max\{ T_{\text{insert},i-1}, O(T_{\text{insert},i-1}) \} + O(1)
\]

(4.6)

This shows that the operational cost of \( \text{insert}() \) on level \( i \) is indeed the aggregate sum of \( O(1) \) work performed on level \( i \) and contributions of work accomplished on level \( i - 1 \), i.e. \( T_{\text{insert},i} = c_1 T_{\text{insert},i-1} + c_0 \).
Lemma 4.1.6. The operational cost $T_{\text{tunnel},i}$ of tunnel() on level $i$ is $O(T_{\text{insert},i-1})$.

Proof. On level $i$, a common-heap $A$ is selected uniformly from the level-$i$ meta-heap interval $[0, t_B)$. The heap location of item $x$ is updated in the level $i$ hash table. Both operations are $O(1)$ time. Then, the $i-1$-level method insert() is invoked on the common-heap $A$. This operation incurs operations cost of $O(T_{\text{insert},i-1})$ time. Finally, we perform an $O(1)$ time operation to check whether item $x$ is the new local max item of common-heap $A$. In case, the test returns true, a meta-heap update is required. Since the common-heap is guaranteed to be situated in the meta-heap interval $[0, t_B)$, the update incurs an operational cost of $O(1)$. Thus, the amortized operational cost of tunnel() on level $i$ can be expressed in terms of the operational cost of insert() on level $i-1$, and a constant contribution on level $i$, more formally $T_{\text{tunnel},i} = O(T_{\text{insert},i-1})$. ■

Lemma 4.1.7. On level $i$, the relative frequency of calls to grow(), $\beta_i$ is expected $O(1/n_i-1)$.

Proof. Let $k^*_i$ be the optimal number of common-heaps for the current number $n_i$ of items maintained by $pQ_i$. We want to know how many additional $\Delta n_i$ items can be added to $pQ_i$ before $k^*_i + 1$ is the optimal number of common-heaps for $pQ_i$. From the system equation, Eq. 3.9, we can derive the load balancing function, Eq. 3.10, which states the number of common-heaps $k_i$ as a function of the total number of items $n_i$ in $pQ_i$. We perform two substitutions in the load balancing function, namely $k_i := k^*_i + 1$, and $n_i := n_i + \Delta n_i$, solve the expression for $\Delta n_i$, and obtain the relation,

$$\Delta n_i = k^*_i \log(k^*_i + 1) - n_i + \log(k^*_i + 1). \quad (4.7)$$

Using the system equation, Eq. 3.9 we infer that $k^*_i \log(k^*_i + 1) - n_i \approx 0$. Further, invoking the equilibrium relation of $pQ_i$, Eq. 3.2, we can see that the change in the number of items required to recruit an additional common-heap is approximately equal to the expected number of items $n_{i-1}$ in a level-$i$ common-heap, i.e.

$$\Delta n_i \approx \log(k^*_i + 1) \approx n_{i-1}. \quad (4.8)$$

Thus, identically as for $pQ_1$, only after $O(n_{i-1})$ items are added to $pQ_i$, an additional common-heap is required to move the system back towards its equilibrium state. Equivalently to the situation for $pQ_1$, this means that on level $i$, the grow function will be invoked on at most every $n_{i-1}^{th}$ call to insert(). Therefore, the relative calling frequency $\beta_i$ is $O(1/n_{i-1})$. ■

Lemma 4.1.8. The amortized cost of grow() on level $i$ is expected $O(1)$ time.

Proof. On level $i$, let $A$ be a common-heap of expected size $O(n_{i-1})$, selected with uniform probability from the constant-cost tunneling domain, $[0, t_B)$ of the meta-heap. Further, recruit the memory for a new empty common-heap $B$ and initialize it. The set of these operations requires $O(1)$ time. In case, common-heap $A$ contains fewer than two items, the process is completed, otherwise $\frac{k_{i-1}}{2}$ common-heaps, equivalent to $O(n_{i-1})$ items, of common-heap $A$ are transferred to common-heap $B$ as intact heaps. Since the $k_{i-1}/2$ common-heaps are already in heap-order, only the cost of copying and the associated book-keeping of $O(n_{i-1})$ items is incurred. In order to include the new common-heap $B$ into the level-$i$ meta-heap without violating the meta-heap’s heap-order, an updated using $O(\log k_i) = O(n_{i-1})$ operations is required. Aggregating all partial operational costs then yields a total cost for grow() of $O(n_{i-1})$ time. Considering Lemma 4.1.7 stating the relative calling frequency of grow() as $O(1/n_{i-1})$, the amortized operational cost of grow() on level $i$ is expected $O(1)$ time. ■

4.2 The search() function.

The collection of functions, search(), remove(), increase-key(), and decrease-key() are similar in character due to their reliance on hash tables to identify in expected $O(1)$ time, the level-0 common-heap ($pQ_0$) of expected size $O(n_0)$, i.e. the array of items that stores a required item. In the array, the item is then linearly tracked down in expected $O(n_0)$ time. As described
in previous sections, maintaining a record of an item location in a hash table requires \( O(1) \) operations.

We will first prove Theorem 4.0.1 for \( \phi() = \text{search}() \), by showing Lemma 4.0.2 and Lemma 4.0.3. This is the most basic function of the family. It is designed to modify the data that is associated with a given item. Additionally it provides intuition for the mechanism underlying its related functions.

**Lemma 4.2.1.** Lemma 4.0.2 with \( \phi = \text{search}() \)

**Proof.** On level 1, the required item is tracked down in expected \( O(n_0) \) time. Subsequently, the operations on the data may be performed requiring \( O(1) \) operations. Then, the operational cost of search on level 1 is expected \( O(n_0) \) time.

**Lemma 4.2.2.** Lemma 4.0.3 with \( \phi = \text{search}() \)

**Proof.** The search function uses the hash table of \( pQ_1 \) to look up in expected \( O(1) \) time the common-heap \( A \) that holds the required item \( x \) in a priority queue of type \( pQ_{i-1} \). By invoking the method \( A.\text{search}() \), it then refers the \( O(1) \) time work to be performed on the data of item \( x \), to a \( pQ_{i-1} \) in which the item is maintained. The operational cost of \( \text{search}() \) on level \( i \) can thus be summarized by a constant amount of work performed on level \( i \) in addition to work performed on level \( i - 1 \). More formally, the operational cost of search on level \( i \) may be stated in terms of the operational cost of search on level \( i - 1 \), i.e. as \( T_{\text{search},i} = c_1 T_{\text{search},i-1} + c_0 \).

### 4.3 The remove() function.

**Lemma 4.3.1.** Lemma 4.0.3 with \( \phi = \text{remove}() \)

**Proof.** On level 1, i.e. for a priority queue \( pQ_1 \), an item \( x \) is tracked down in its home common-heap \( A \) in expected \( O(n_0) \) operations. It is established whether \( x \) constitutes the local max item of \( A \), which occurs with expected probability of \( 1/n_0 \). Next, the item is removed from \( A \), causing common-heap updates requiring expected \( O(\log n_0) \) operations. Then, the associated hash table entry is removed and the size of \( pQ_1 \) is updated.

In case, item \( x \) was holding the local max key before its removal from common-heap \( A \), we may regard the updated common-heap \( A' \) as the root of the sub-heap \( \Psi_{A'} \) of the meta-heap. Before \( A' \) can be popped off \( \Psi_{A'} \), the position of \( A' \) in the meta-heap as well as the last index of the sub-heap \( \Psi_{A'} \) must be determined. The former operation is \( O(1) \) time while the latter as well as the associated meta-heap updates are \( O(\log k_1) \) time. Lastly, the common-heap \( A' \) must be re-introduced into the meta-heap also using \( O(\log k_1) \) operations. Since this case occurs with expected probability \( 1/n_0 \), the amortized cost for this case is \( O(\log k_1/n_0) \). Since by definition, \( n_0 = \sqrt{\log k_1} \), the amortized case cost can be simplified to \( O(n_0) \).

Lastly, it is determined whether the number of available common-heaps in \( pQ_1 \) exceeds the optimal size including a predefined tolerance, analogously as described for insert(). If the number of common-heaps has to be reduced by a single unit, the function trim-and redistribute() is invoked with a relative calling frequency, as established in Lemma 4.1.3 for grow() being called from insert(). In particular, as shown in Lemma 4.3.2, the amortized operational cost of trim-and redistribute() on level 1 is \( O(n_0) \). Aggregation of all partial costs then shows that the expected amortized cost for remove() on level 1 is \( O(n_0) \).

**Lemma 4.3.2.** The amortized cost of \( \text{trim-and-redistribute}() \) on level 1 is \( O(n_0) \).

**Proof.** This function redistributes each of the expected \( O(n_0) \) elements in the common-heap pointed to by the last position in the meta-heap, to expected \( O(n_0) \) other common-heaps from the entire data structure. Choosing the last element in the meta-heap makes reduction of the meta-heap size an \( O(1) \) operation. Only a size decrement and suspension of the associated heapID to the stack are required. The \( O(n_0) \) items are redistributed within \( pQ_1 \) using a slimmed down version of insert(), named reinsert(), which requires expected \( O(n_0) \) operations.
(Lemma 4.1.1). In particular, the slimmed down version does not update the number of items, and does not check whether an additional common-heap is required because the number of items in $pQ_1$ has not changed. Finally, the content of the last common-heap is cleared, and its size set to zero. Consequently, the work performed by trim-and-redistribute() is $O(n_0)$ time. Given that trim-and-redistribute() is executed in the worst-case only on every $n_0^{th}$ call (Lemma 4.1.3), its fully amortized cost is $O(n_0)$. ■

Lemma 4.3.3. Lemma 4.0.3 with $\phi = \text{remove}()$

Proof. On level $i$, this function operates almost identically to its counterpart in $pQ_1$, with a single but critical exception. It first uses the hash table to locate the common-heap $A$ in which the item $x$ marked for removal is stored. This operation incurs $O(1)$ expected time cost, but then instead of linearly searching the common-heap of expected size $n_{i-1}$, it invokes the $pQ_{i-1}$-method remove() on $A$. This method requires $T_{\text{remove},i-1}$ expected time to remove an item including all related common-heap update operations from level $i - 1$.

In case, the item $x$ was holding the local max key before its removal, which occurs with expected probability of $1/n_{i-1}$, then analogously to the remove() procedure in $pQ_1$, the sub-heap of the meta-heap that has $A'$ as its root has its heap-order restored such that the updated common-heap $A'$ is placed into a correct meta-heap location. These meta-heap update operations incur costs of $O(\log k_i)$ time. Given relation 3.2, in particular, $n_{i-1} = \log k_i \forall i > 1$, the amortized cost for this case is $O(1)$ time.

Since an item was removed, the system may now be sufficiently out of balance (see Eq. 3.2), so that the suspension of a common-heap is warranted. The associated test is $O(1)$ time. The function trim-and-redistribute() deals with the suspension as well as all other operations required to maintain heap-order. The calling characteristics are established in Lemma 4.1.7.

In particular, trim-and-redistribute() is invoked with expected probability $O(1/n_{i-1})$, and as shown in Lemma 4.3.4, thus exhibits an amortized running time of $O(T_{\text{reinsert},i-1})$.

Let $T_i$ be the amortized operational cost of remove() in $pQ_i$, and let $T_{i-1}$ be the accumulated operational cost depending only on work performed on level $i - 1$. All other operations are specific to the current instance of $pQ_i$, and accumulate to an amortized cost, $c_0 = O(1)$. Thus, the aggregate of all partial costs of remove() on level $i$ can be described by the recurrence relation, $T_i = c_1 T_{i-1} + c_0$. ■

Lemma 4.3.4. On level $i > 1$, the amortized running time of trim-and-redistribute() is $O(T_{\text{reinsert},i-1})$.

Proof. The mechanisms of trim-and-redistribute() on level $i$ and 1 are virtually identical. The level $i$ method reinsert() is called on each of the $O(n_{i-1})$ items stored in the common-heap pointed to by the last element of the level $i$ meta-heap. Previously, we established Lemma 4.1.5 for insert(). It then implicitly holds for its slimmed down version, reinsert(). Thus, the $O(n_{i-1})$ items are each redistributed with an individual cost $T_{\text{reinsert},i} = T_{\text{reinsert},i-1} + O(1) = O(T_{\text{reinsert},i-1})$, and thus aggregating to a cost $O(n_{i-1} T_{\text{reinsert},i-1})$. Further, the memory associated with the $O(n_{i-1})$ items must be cleared, requiring another $O(n_{i-1})$ time operation. Given its relative calling frequency of $O(1/n_{i-1})$, the amortized operational cost for trim-and-redistribute() on level $i$ is $O(T_{\text{reinsert},i-1})$. ■

4.4 The increase-key() function.

Lemma 4.4.1. Lemma 4.0.2 with $\phi = \text{increase-key}()$

Proof. In order to increase a key of an item $x$, the common-heap $A$ in which it is stored is located in $O(1)$ expected time using the hash table. Then, three cases will be considered.

Firstly, the new key of item $x$ is not larger than the current local max key of $A$. In this case, the item is tracked down in $A$ using $O(n_0)$ expected operations, and subsequently its key is updated to the new key. It follows a common-heap update that has $O(\log n_0)$ operations associated with it.
The second case is realized if the item \( x \) is not the current local max item of \( A \), but the new key of \( x \) is larger than the current local max key. In this case, simply updating the item in its common-heap \( A \) would require a subsequent meta-heap update of cost \( O(\log k_1) \) time. Instead, we prepare to tunnel() the item to a constant-cost common-heap in the interval \([0, t_n]\) of the meta-heap. In order to do this, we first track down the item in \( A \) requiring \( O(n_0) \) expected operations, and then remove the item from common-heap \( A \) using \( O(\log n_0) \) operations. Then \( x \) is tunneled with the new key replacing the old. As established in Lemma 4.1.2, the expected cost for tunnel() on level 1 is \( O(\log n_0) \).

In the final case, which occurs with expected probability \( 1/n_0 \), item \( x \) is indeed the local max item, and a meta-heap update, after the key has been changed to the new key value, is inevitable. This is always the case since we require that the new key is larger than the old one, and thus the meta-max position of the common-heap may change. The amortized cost for this case is thus, \( O(n_0) \). As a result, the amortized expected cost for increase-key() on level 1 is \( O(n_0) \) time.

**Lemma 4.4.2.** Lemma 4.0.3 with \( \phi = \text{increase-key()} \)

**Proof.** On level \( i \), the common-heap \( A \) holding the item \( x \) with itemID is located in \( O(1) \) expected time. Analogously to the case on level 1, the same three cases are considered. In the first case, the \( pQ_{i-1} \) method increase-key() is invoked on \( A \), incurring a cost \( T\text{increase-key,i−1} \) time. In the second case, the \( pQ_{i-1} \) method remove() is followed by a call to the \( pQ_{i} \) method tunnel(), which is shown in Lemma 4.1.6 to incur an operational cost of \( O(T\text{insert,i−1}) \). The last case occurs with expected probability \( 1/n_{i-1} \), and requires \( O(\log k_i) \) operations, which is equivalent to an amortized cost of \( O(1) \) time. The aggregated operational cost for increase-key() on level \( i \) is, the sum of \( O(1) \) time work performed on level \( i \) and operations executed by methods on level \( i−1 \). As a result, the cost of the increase-key() function on level \( i \) can be expressed by the recurrence relation, \( T_i = c_i T_{i−1} + c_0 \).

4.5 The decrease-key() function.

**Lemma 4.5.1.** Lemma 4.0.2 with \( \phi = \text{decrease-key()} \)

**Proof.** After the common-heap \( A \) that holds the item \( x \) is located in expected \( O(1) \) operations, it is determined whether the item constitutes the local max item.

In case, it is not the local max item, it is tracked down in \( A \) using expected \( O(n_0) \) operations, and subsequently the key value is updated. Then, we establish the last index of the sub-heap in \( A \) that holds \( x \) as its root, and use it to pop item \( x \) off this sub-heap, and then reintroduce it from this last index, so that heap order is re-established in the entire common-heap \( A \). This set of operations requires \( O(\log n_0) \) time.

With an expected probability of \( 1/n_0 \), item \( x \) holds the local max key, and reducing its value will likely require a meta-heap update. Thus, firstly the x.key is decreased and subsequently the common-heap is updated to restore heap-order using requires expected \( O(\log n_0) \) operations. Then, the position of \( A \) in the meta-heap, as well as the last index of its sub-heap is established. The two indices are used to update the heap-order in the sub-heap, and also in the entire meta-heap using \( O(\log k_i) \) operations. Given its relative frequency of occurrence, the amortized cost of this case is \( O(n_0) \) time. Aggregation of all partial costs, then yields a level-1 running time for decrease-key() of \( O(n_0) \).

**Lemma 4.5.2.** Lemma 4.0.2 with \( \phi = \text{decrease-key()} \)

**Proof.** On level \( i \), the common-heap \( A \) that contains the \( pQ_{i−1} \) maintaining the item \( x \) associated with itemID, is located in expected \( O(1) \) operations. For later use, a note is made whether \( x \) is the current local max item of \( A \). Then, the \( pQ_{i−1} \) method decrease-key() is invoked on \( A \), and consequently incurs an expected cost of \( T\text{decrease-key,i−1} \) time. Next, we check whether item \( x \) was the local max item before the key value reduction, and in case it was, the sub-heap of the meta-heap that has the common-heap \( A' \) as its root is updated using a sequence of \( O(\log k_i) \) operations. This case occurs with expected probability \( 1/n_{i−1} \), and thus requires an
amortized $O(1)$ operations. The amortized cost for decrease-key() is then expressible in terms of the work incurred on level $i - 1$, in addition to $O(1)$ operations on level $i$, or formally in terms of the recurrence relation Eq. 3.3, $T_i = c_1 T_{i-1} + c_0$. □

5 Concluding Remarks

The design and implementation of a log*$^* N$-level priority queue data structure with expected $O(1)$ running time for the functions insert(), search(), remove(), increase-key() and decrease-key() is presented. The priority queue is composed from elementary data structure building blocks such as binary heaps, hash tables and stacks. The internal structure of the priority queue system is statistically self-similar. Load balancing on a given level $i$ of the priority queue system is implicitly controlled using functions that are derived from the quantitative relationships between the number of common-heaps $k_i$ and their required expected size $n_{i-1}$ on that level.

Computing the optimal number of common-heaps $k_i$ as a function of the current number of items $n_i$ maintained by a level $i$, requires the evaluation of functions of the general form

$$k_i = \exp(\lambda_i W_0(f_i(n_i)))$$ (5.1)

where, $W_0$ denotes the LambertW-function on the main branch $W_0$, $f_i(n_i)$ represents a function that takes the number of items $n_i$ as its only argument, and $\lambda_i$ is a constant pre-factor. A parts depend on the level $i$ where the function is invoked.

Evaluation of this expression is an $O(1)$ time operation. However, depending on the implementation the constant factor variations are non-negligible. The fastest options is probably, to load all required values from a pre-computed list or similarly, compute all values upon initialization of the priority queue. The space required for both options is $O(n)$. Alternatively, an on-the-go scheme could be considered, where a value is computed during operation only if it has not been computed and stored previously. This procedure will lead to a full table for a priority queue that is operated for long times, and thus the computational cost converges to zero,. The space requirement is also $O(n)$. Yet another possibility would be to always compute values as required, then the space requirement is $O(1)$.

Since we are here only interested in positive integer solutions of Equation 5.1, the demand on accuracy is very low with respect to typical numerical evaluations of such expressions. The exponential function could probably be sufficiently well approximated by a second-order Taylor expansion, rather than by the full precision implementation of a numerics library.

The practicality and robustness of this particular multi-level approach is yet to be determined. The system may be too involved compared to leaner implementations of other priority queues. However, it cannot be excluded that interesting or useful applications for this system exist. In order to start evaluating the practical value of this priority queue system, we are currently in the process of exposing the system to various input scenarios and analyzing its respective response. We anticipate that with a set of appropriate modifications, the system may have some utility in concurrent operations.

References

[1] Williams, J.W.J. Algorithm 232: Heapsort, Commun. ACM, 7 (6), 347–348, 1964

[2] Daniel H. Larkin, Siddhartha Sen, Robert E. Tarjan. A Back-To-Basics Empirical Study of Priority Queues, Proceedings of the Meeting on Algorithm Engineering & Experiments, pp 61–72, 2014
[3] Gerth Stølting Brodal. A Survey on Priority Queues, in Space-Efficient Data Structures, Streams, and Algorithms, Lecture Notes in Computer Science, Vol 8066, pp 150-163, 2013

[4] van Emde Boas, P., Kaas, R. & Zijlstra, E. Design and implementation of an efficient priority queue, Math. Systems Theory, Vol 10 (1), pp 99–127, 1976

[5] G. S. Brodal. Worst-case efficient priority queues. in Proceedings of the Seventh ACM-SIAM Symposium on Discrete Algorithms (SODA), pp 52–58, 1996

[6] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. Algorithmica, 1, pp 111–129, 1986

[7] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM, 34, pp 596–615, 1987

[8] Boris V. Cherkassky, Andrew V. Goldberg, Tomasz Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation, Mathematical Programming, 73 (2), pp 129–174, 1996

[9] Boris V. Cherkassky, Andrew V. Goldberg, Craig Silverstein. Buckets, Heaps, Lists, and Monotone Priority Queues, SIAM J. Comput., 28 (4), pp 1326–1346, 1999

[10] Rajeev Raman. Fast Algorithms for Shortest Paths and Sorting, Technical Report TR 96-06, King’s College, London, 1996

[11] Mikkel Thorup. On RAM Priority Queues, SIAM J. Comput., 30 (1), 86–109., 2000

[12] Bernhard Haeupler, Siddhartha Sen, Robert E. Tarjan. Rank-Pairing Heaps, SIAM J. Comput., 40 (6), pp 1463–1485, 2011

[13] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. Self-similarity-based image denoising. Commun. ACM, 54(5), pp 109–117, 2011

[14] R.M. Corless, G.H. Gonnet, D.E.G. Hare, D.J. Jeffrey, D.E. Knuth. On the Lambert W Function. Advances in Computational Mathematics, 5, pp 329-359, 1996

[15] Tony C. Scott, Robert Mann, and Roberto E. Martinez II. General Relativity and Quantum Mechanics: Towards a Generalization of the Lambert W Function. Appl. Algebra Eng., Commun. Comput. 17, 1, pp 41–47, 2006

[16] Brian Wesley Williams The Utility of the Lambert Function in Chemical Kinetics. J. Chem. Educ., 87, 6, pp 647–651, 2010