Seminar on Advanced Algorithms and Data Structures - Report

The Soft Heap: An Approximate Priority Queue with Optimal Error Rate [2]

Philippe Blatter

October 2018

1 Introduction

As an introduction we give a short review on the min-heap. A min-heap is a tree-based data structure that stores values in a certain way so that the heap satisfies the heap property. Regarding a min-heap, the heap property says that if v is a parent of u, then the v's key value is either smaller or equal to the key value of u. There is no rule for the children of a node regarding left or right ordering - that means that the key of the left child does not have to be smaller than the key of the right child as this is the case in a binary search tree.
The heap is an efficient implementation of a priority queue in which elements are partially ordered based on their priority value. A binary heap has a depth of $\mathcal{O}(\log(n))$.

Supported heap operations are:

1. findMin - $\mathcal{O}(1)$
2. deleteMin - $\mathcal{O}(\log(n))$
3. insert - $\mathcal{O}(\log(n))$
4. decreaseKey - $\mathcal{O}(\log(n))$ (decreases the key of an item)
5. merge - $\mathcal{O}(\log(n))$ (merging two binomial heaps of size m, n where $m \leq n$)

A soft heap is a simple variant of a priority queue that supports similar functions as a regular min-heap. After deleting the minimum in a regular heap, one has to move up another item to the root which takes $\mathcal{O}(\log n)$ time. The goal is to speed up this operation. Therefore, we allow the data structure to increase the values of certain keys, such items are called corrupted. The *findMin* operation returns the current minimum key, which might or might not be corrupted. The benefit is speed: during heap updates, items travel together in packets in a form of "car pooling", in order to save time.

Theorem 1 *Beginning with no prior data, consider a mixed sequence of operations that includes n inserts. For any $0 < \epsilon \leq 1/2$, a soft heap with error rate ϵ supports each operation in constant amortized time, except for insert, which takes $\mathcal{O}(\log 1/\epsilon)$ time. The data structure never contains more than ϵn corrupted items at any given time. In a comparison-based model, these bounds are optimal.*

Note that this does not mean that only ϵn items are corrupted throughout the entire sequence of operations. This is simply an upper bound for the number of items in the soft heap being corrupted at any given time.

The main reason for the invention of soft heaps was to find a faster algorithm for finding minimal spanning trees. The algorithm that could be built using soft heaps was the fastest at the time. Given a connected graph with n vertices and m weighted edges, the algorithm finds a minimum spanning tree in time $\mathcal{O}(m\alpha(m,n))$, where α is the classical functional inverse of Ackermann's function. [1]

Furthermore, soft heaps are also useful for near sorting, and generally for situations where approximate position (in a sequence) information is sought.

Theorem 2: *For any $0 < \epsilon \leq 1/2$, we can use a soft heap to near-sort n numbers in time $\mathcal{O}(n \log 1/\epsilon)$: this means that the position of any number in the output sequence differs from its true position by at most ϵn.*

Proof of Theorem 2: We only give a short description of the proof. We assume that n is large enough so that $1/\epsilon$ lies between a large constant and $\sqrt{n}$. The idea is to insert n items into a soft heap and then deleting them again until the heap is empty. Then, one can divide the output sequence into distinct intervals in $\mathcal{O}(n)$ time - these intervals are sorted with respect to each other i.e. all items in interval i are smaller than all items in interval $i+1$. We show that there can be at most $6\epsilon n$ items in each interval. Thus, any number can be output in a position at most $6\epsilon n$ off its rank. By replacing ϵ by $\epsilon/6$ we get the upper bound that was claimed in the theorem. Therefore, we have costs of $\mathcal{O}(n \log 1/\epsilon)$ for the heap operations + $\mathcal{O}(n)$ for the postprocessing time which results in a total time of $\mathcal{O}(n \log 1/\epsilon)$.

2 The Data Structure

The soft heap can be seen as a modified binomial heap which is in turn a list of modified binomial trees.

A binomial tree can be defined recusively:

- A binomial tree of rank 0 is a single node.
- A binomial tree of rank k can be formed by the combination of two binomial trees of rank $k-1$ where one root becomes the child of the other root. An alternative for the second part of the definition could also be: A binomial tree of rank k has a root node whose children are roots of binomial trees of ranks $k-1$, $k-2$, ..., 2, 1, 0 (in this order), as illustrated in figure 1.

In order to understand how the data structure is constructed, it is important to understand the concept of binomial heaps. An important property of a binomial tree is that a tree of rank k consists of 2^k nodes and that the root has k children ($deg(root) = k$). Furthermore, all the trees in the heap must have distinct ranks. Binomial trees that are defined in this way have the special property that we can easily combine two binomial trees of the same rank (this will be discussed in the next chapter). If we want to store n items in a binomial heap, we need one binomial tree for $n = 1$ and at most $\lceil \log_2(n) \rceil$ binomial trees for $n > 1$, where the binary representation of n indicates the existence of the binomial trees of certain ranks.

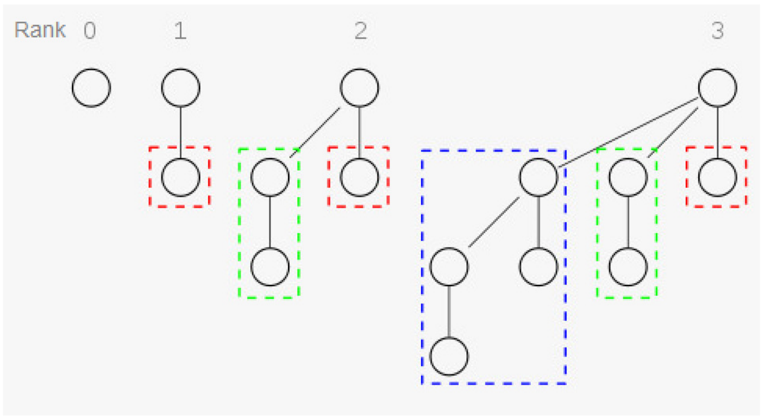

Figure 1: Binomial trees of different ranks [4]

Example
Suppose $n = 12 \Rightarrow \lceil \log_2(12) \rceil = 4$, therefore, we can conclude that we need at most 4 binomial trees. As $12_{10} = 1100_2$ we see that we need a tree of rank 3 and a tree of rank 2 if we enumerate the bits from right to left starting at 0.

Soft Heap: The binomial trees of the soft heap are called soft queues which are slightly modified binomial trees. Initially, a soft queue corresponds to a regular binomial tree which is used as an underlying master-tree. The difference is, that a soft queue allows that nodes are deleted without changing the rank of the parent node - in other words a soft queue may have missing subtrees that are contained in the master-tree. The rank of a node v always corresponds to the number of children in the master-tree. The reason behind this correspondence between a soft queue and its master-tree will be explained in detail when we look at the sift operation. The rank of a queue q' corresponds to the rank of its root. Furthermore we use the following rank invariant: the number of children of the root should be no smaller than $\lfloor rank(root)/2 \rfloor$. This invariant will be useful for the analysis later.

Corruption
Another modification is the following: in a regular heap, every node stores one item with respect to its appropriate priority value. Soft heaps allow us to increase the value of certain keys. In that way, soft heaps can store an entire *item_list* of items per node that are associated with the same priority value. Some of these items might have had another key - their values have been raised. The value with respect to which we order the soft queue is called *ckey*.
Initially, every item's *ckey* matches its key. We will later introduce an operation that allows us to combine the different item lists of the nodes. All items in one *item_list* are associated with the same *ckey* that denotes the priority value. During heap updates, we move entire item lists accross the heap in order to save time. All items in one list are treated equally in terms of priority. We enforce that $ckey(v)$ is an upper bound of all items in $items\_list(v)$. An item x in an $items\_list(v)$ with $key(x) < ckey(v)$ counts as corrupted as its *ckey* is raised. Corruption cannot be reversed.
Furthermore, we introduce an error rate $0 < \epsilon \leq \frac{1}{2}$. In a soft heap storing n items, at most ϵn items have their keys raised and thus count as corrupted. Additionally, we add the constraint that all corrupted items must be stored at nodes of rank greater than $r = r(\epsilon)$.

As the number of children of a node varies in a binomial tree depending on the node's rank, it is convenient to use a binary tree where each node v stores two pointers to its children ($v \to child$ and $v \to next$) in order to implement a binomial tree. We enforce the following *next-invariant*: we always store the child with the smaller *ckey* as $v \to next$. We define the following rule: A binomial tree of rank k will be modelled as a binary tree of height $k+1$ with 2^k leaves. In this way, all the nodes of the binomial tree end up in the leaves of the binary tree. The benefit of representing binomial heaps in this way, is that the internal nodes of the binary tree represent the melding of two binomial trees of the same rank and have the same *ckey* as their smaller child. Figure 2 shows how to represent a binomial tree as a binary tree.

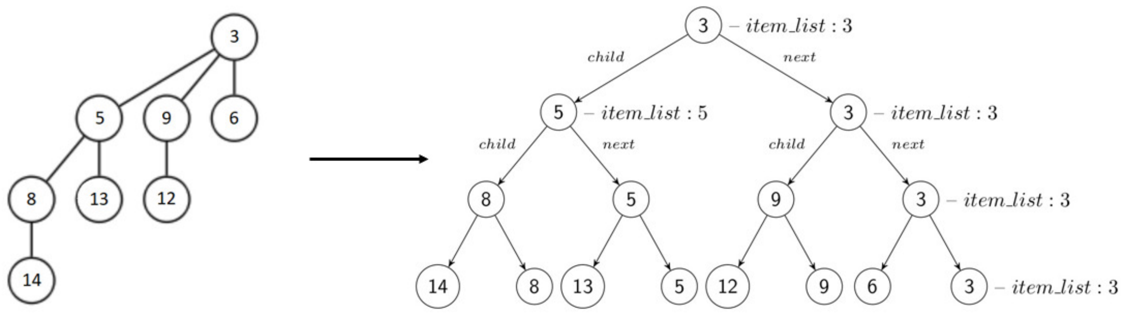

Figure 2: Conversion of a binomial tree into a binary tree

Furthermore, we store a pointer to the root of minimum *ckey* among all queues.

Summary
Summarizing the explanation above: A soft heap is a list of soft queues. The goal of data structure is to be faster than a regular heap. We therefore allow the soft heap to increase keys and thus corrupt items. These corrupted items are stored in item lists at nodes v with $rank(v) > r$. If an item x is stored in an item list of a node v that is associated with a $ckey(v) > key(x)$, x counts as corrupted.

3 The Heap Operations

We limit ourselves to the essential operations.

Insert

When inserting a new item into the soft heap, we build a new queue consisting of a single node v and meld it into the heap. The initial item list of v solely contains $key(v) = ckey(v)$. As v has no children, the rank of the queue is 0.

Melding two heaps

When melding two soft heaps S_1 and S_2, we consider the heap that consists of fewer queues e.g. S_1. We meld every queue q of S_1 into S_2. As previously mentioned, a soft heap cannot contain more than one soft queue of rank k. Therefore, in order to meld a queue q of S_1 with rank k into S_2, we go through the ordered list of queues in S_2 until we find the first queue q' with $rank(q') >= k$.

1. If $rank(q') > k$, then we simply insert q into the list of queues in S_2 right before q'.
2. If $rank(q') = k = rank(q)$ we have a conflict as the different queues are supposed to have distinct ranks. We make use of the property that two binomial trees of rank k can easily be combined to a new binomial tree of rank $k+1$ by simply attaching the root with the larger *ckey* as a new child of the other root. This is shown in figure 3.

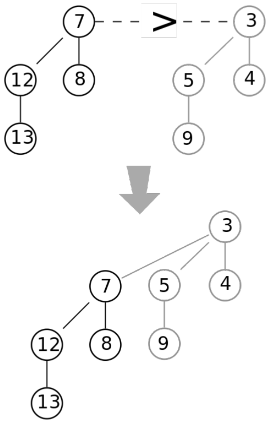

Figure 3: Two binomial trees of rank 2 combined to a new one

In the second case, the combination of both trees results in a tree of rank $k+1$. We revert to the analogy of binary representation of the heaps. If S_2 already contains a queue of rank $k+1$, we have to repeat the operation and meld the new queue into the heap S_2, similar to a carry propagation in binary addition.
If there is no queue q' in S_2 with $rank(q') >= rank(q)$, we simply insert q at the end of the list of queues in S_2.

After melding two heaps, one has to make sure that *min_pointer* still points to the root with the smallest *ckey*.

DeleteMin

In order to delete the min, we look at the root v of the queue to which *min_pointer* is pointing to.

- If $items\_list(v)$ contains only one key x, we simply return x and delete it from the list.
- If $items\_list(v)$ contains more than one key, we return the first key in the list and delete it.

In both cases, the key that is returned might be corrupted. It might occur that the root's item list is empty. In this case, we have to refill it by pushing items from nodes further down in the tree up towards the root. We do this by using the sift operation, which is the key operation of the soft heap.

Sift

Instead of only storing one item at each node, the soft heap corrupts some values and stores them in the item lists of nodes of rank greater than r. In that way, we do not have to bring up a new item to the root every single time after deleting the minimum. We only have to move items up the tree towards the root when the item list of the root is empty. The pseudocode for the sift operation is at the end of this sub-chapter and can be used as a reference. It makes sense to show how the algorithm works using binary trees instead of binomial trees as it is nearer to the actual implementation. As mentioned before, in the binary tree representation, all the nodes from a binomial tree are stored in the leaves.

Explanation:
The goal of the operation is to move items up the tree towards the root. The basic idea is to start at the root of the binary tree and recursively go down to a leaf and collect the items of the lower nodes on the way back up. If the nodes on the recursion path fulfill certain conditions, we recursively perform the operation again so that the recursion tree is branching and we move up even more items. First we empty the $items\_list(v)$ as it will be refilled by new items from further down the tree. Then we check if the node we are currently looking at is a leaf - if this is the case, we set its $ckey(v)$ to ∞, indicating that we visited this leaf. Nodes whose *ckeys* are set to ∞ will be deleted during the clean-up procedure as they are not needed anymore. The leaf's initial *ckey* is not lost when we set it to ∞ - its initial value is stored in its parent node.
If the node we are looking at is not a leaf, we call sift recursively on the child with the smaller *ckey* which is $v \to next$. After $sift(v \to next)$, $v \to next$ might have a new *ckey* value so we might have to swap the children because of the *next-invariant*. An example for such a swap can be seen in figure 4 where the two subtrees of the root need to be swapped. After that, $items\_list(v)$ is set to $items\_list(v \to next)$ and $ckey(v)$ is replaced by $ckey(v \to next)$. Figure 4 illustrates the deletion of the last element in the item list followed by a sift operation going from the root down to the leaf and back up: the crossed out values are being replaced by the values right next to it. Horizontal arrows indicate that the subtrees have to be exchanged.

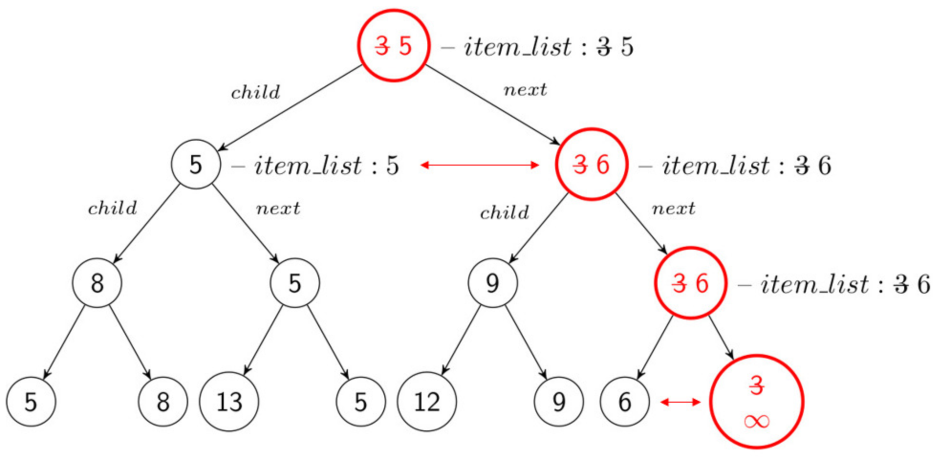

Figure 4: Single sift operation

After replacing the item list of the current node, we check if the loop-condition holds. The loop-condition consists of two terms:

i sift has only been called recursively once (max. binary branching of recursion tree at every node)

ii the rank of v exceeds the threshold r and either is odd or it exceeds the rank of the highest-ranked child of v by at least two.

The first part (i) makes sure that the recursion tree is branching at most one time at this level. As mentioned in the introduction to the data structure, we enforce that corrupted items can only be stored at nodes with rank greater than r. This is checked by the first part of ii). The second part of ii) makes sure that the recursion tree branches the right amount time. If both conditions are fulfilled, we again call $sift(v \rightarrow next)$ - that means that we again want to start moving items up the tree towards the root starting at that node. The eventual item list of v will consist of its current item list appended to the newly computed item list of $v \rightarrow next$. Again, $ckey(v)$ is replaced by $ckey(v \rightarrow next)$. In this case all the items x in $items\_list(v)$ with $key(x) < ckey(v)$ are corrupted as their key has been raised. After the second call of sift, we might again have to swap the children so that the *next-invariant* is not violated.

In the end we get rid of the nodes v with $ckey(v) = \infty$ i.e. the nodes that are not needed anymore as their initial keys have been moved further up in the tree.
Recall that each queue has an underlying master-tree. The leaves that we are deleting from the soft queue in this clean-up process will not be deleted in the master-tree. Thus, the ranks of the nodes are not changed during this process. Our goal is to ensure that the ranks of both children of a node in the binary tree are equal. Enforcing this might cause a discrepancy between the rank (number of children in the master-tree) and the actual number of children (in the binomial tree). Nevertheless, this allows us to easily swap the children in the binary tree. Recall that an inner node in the binary tree represents the melding of two binomial trees of the same rank. Thus, swapping the children in the binary tree is equivalent to combining the two binomial trees the opposite way (exchanging which root is the child of the other root) which is in turn possible because of the nice property of binomial trees.
If both children of a node v are not used anymore, we just delete both and v becomes a leaf. If only one child is not used anymore, we restore the pointers in a way that every node in the binary tree still has two children that represent binomial trees of the same rank. Technically speaking: If both, $ckey(v \rightarrow child)$ and $ckey(v \rightarrow next)$ are ∞, then we delete both and v becomes a leaf. If only $ckey(v \rightarrow child)$ is ∞, we set $v \rightarrow child = v \rightarrow next \rightarrow child$ and $v \rightarrow next = v \rightarrow next \rightarrow next$. As the rank of a node is defined with respect to the master-tree, a node of rank k can now have two children of rank $k-2$ which is one of the conditions checked in the loop-condition. In this way, both children of a node in the binary (!) tree will always have the same rank. Figure 5 illustrates the clean-up process if only one child is leaf.

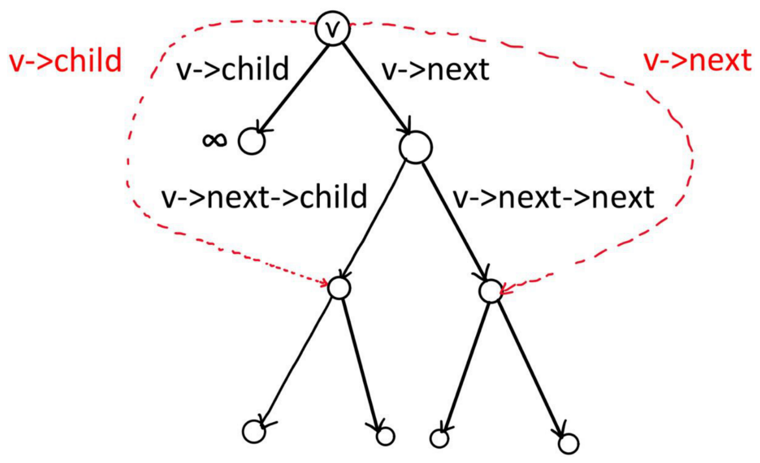

Figure 5: Clean-up operation

```
 sift(v)
$item\_list(v) \leftarrow T \leftarrow \emptyset$
if v has no child then
   set $ckey(v)$ to $\infty$ and return;
end if
1. sift($v \rightarrow next$);
if $ckey(v \rightarrow next > ckey(v \rightarrow child))$ then
   exchange $v \rightarrow next$ and $v \rightarrow child$
end if
$T \leftarrow T \cup item\_list(v \rightarrow next)$
if loop-condition holds then
   goto 1
end if
$item\_list(v) \leftarrow T$;
clean-up =0
```

DeleteMin II
In the previous explanation, we skipped an important part of the operation. If the list is empty, we first have to check, whether the rank invariant holds that was mentioned in the introductory section to the data structure (the number of children of the root should not be less than $\lfloor rank(root)/2 \rfloor$), before refilling the list using sift. Previous sifting may have caused the loss of too many children of the root and hence a violation of the invariant. In this case, we dismantle the queue and re-meld its children into the heap.

4 Complexity

4.1 The Error Rate

We claimed that at most ϵn items were corrupted in the soft heap. Note that this does not mean that only ϵn items are corrupted in total during all operations. ϵn only limits the amount of corrupted items that are contained in the heap at a certain point in time. In order to prove our claim, we need two lemmata and one helper lemma. To achieve the desired error rate, we set

$$r \overset{\text{def.}}{=} 2 + 2\lceil \log 1/\epsilon \rceil$$

Lemma 4.1

$$|item\_list(v)| \leq max\{1, 2^{\lceil rank(v)/2 \rceil - r/2}\}$$

Proof: Recall that the item lists can only be enlarged during a sift call. Therefore, we again favor the binary tree representation over the binomial tree representation. If the recursion tree of sift is only a path, meaning that sift is only called recursively once (no branching of recursion tree), the item list of a node simply moves up to a node of higher rank and the lemma holds by induction. For the case that the loop condition is fulfilled and sift is called twice, *items_list*(v) will be composed of the concatenation of the two item lists associated with $v \to next$ after each call of $sift(v)$. As the second part of the loop condition contains a disjunction, we have to do a case distinction. In both cases, let k be $v \to rank$ and let k' be $v \to next \to rank$. We know that $k > r$:

i First, we look at the case where k is odd: we know that $k' < k$ and assume that $rank(v \to next) > r$. By induction, either of the two computed item lists of $v \to next$ will have the size of at most $2^{\lceil (rank(v)-1)/2 \rceil - r/2} = 2^{\lceil rank(v)/2 \rceil - 1 - r/2}$ which is true for an odd k.

ii Next, we look at the case where $k > k' + 1 \geq r + 2$. Either of the two computed lists of $v \to next$ will have the size of at most $2^{\lceil (rank(v)-2)/2 \rceil - r/2} = 2^{\lceil rank(v)/2 \rceil - 1 - r/2}$ which is true for every k.

As *items_list*(v) will be the concatenation of both refilled lists, its size will be at most $2 \cdot 2^{\lceil rank(v)/2 \rceil - 1 - r/2} = 2^{\lceil rank(v)/2 \rceil - r/2}$.□

Lemma 4.2 *The soft heap contains at most $n/2^{r-3}$ corrupted items at any given time.*

In order to prove this lemma, we first prove another lemma that helps us with the actual proof.

Lemma 4.3 *If S is the node set of a binomial tree then*

$$\sum_{v \in S} 2^{rank(v)/2} \leq 4|S|$$

Proof of Lemma 4.3: Lemma 4.3 follows from the inequality

$$\sum_{v \in S} 2^{rank(v)/2} \leq 2^{k+2} - 3 \cdot 2^{k/2} < 4|S|,$$

where k is the rank of the binomial tree.

We prove our claim by induction over k:
Base case ($k = 0$): As $k = 0$, there is only one node in the tree $\Rightarrow 2^0 = 1 \leq 2^2 - 3 \cdot 2^0 = 1$

Step case ($k \to k+1$): We know that a tree of rank $k+1$ consists of two subtrees of rank k. Let's denote the set of nodes of the tree of rank $k+1$ as S'.

$$\sum_{v \in S'} 2^{rank(v)/2} = 2 \sum_{v \in S} 2^{rank(v)/2} \overset{\text{I.H.}}{\leq} 2(2^{k+2} - 3 \cdot 2^{k/2}) = 2^{k+3} - 3 \cdot 2^{k/2+1} \leq 2^{(k+1)+2} - 3 \cdot 2^{(k+1)/2} \square$$

Proof of Lemma 4.2: We look at the nodes whose ranks are greater than r. Let's denote the set of nodes of rank greater that r in the respective master-tree q' as R'. Within a binomial tree, the set of nodes with a rank greater than r number a fraction of $1/2^r$ of all the leaves. Recall that the number of leaves in a binomial tree of rank k is 2^{k-1}, resulting in 2^{k-r-1} nodes of rank greater than r. Summing over all master-trees q', we get

$$\sum_{q'} |R'| \leq n/2^r$$

as an upper bound. At this point we have everything that we need for our proof: we can use Lemma 4.1 that says something about the length of the item list, we have a statement regarding the node set of a binomial tree and a statement on the set of nodes of rank greater than r. The goal is to combine all of them.

Using Lemma 4.1, we can show that $\forall v \in R' \quad items\_list(v) \leq 2^{\lceil (rank(v)+1-r)/2 \rceil}$. Summing over all master-trees and all nodes in R' we get

$$\sum_{q'} \sum_{v \in R'} 2^{(rank(v)+1-r)/2} = 2 \sum_{q'} \sum_{v \in R'} 2^{(rank(v)-r-1)/2}$$

The first sum represents the sum of the lengths of all item lists from nodes in R'. This represents the number of items that are potentially corrupted based on the fact that the number of corrupted items is bounded by the sum of items being stored in the item lists of nodes with rank larger than r. As mentioned above, there are 2^{k-r-1} nodes of greater rank than r in every q'. These nodes build a binomial tree of rank $k - r - 1$ themselves. Now we can apply Lemma 4.2 and Lemma 4.3 and conclude that $2 \sum_{q'} \sum_{v \in R'} 2^{(rank(v)-r-1)/2} \leq 2 \cdot 4|R'| = 8|R'|$ and therefore $\sum_{q'} 8|R'| = \sum_{q'} 2^3 \cdot |R'| \leq 2^3 \cdot n/2^r = n/2^{r-3}$.$\square$

4.2 The Running Time

DeleteMin has trivially a constant running time. We only need to look at sift and meld.

4.2.1 Meld

Imagine that all n nodes from both heaps together were inserted into a new one and that $n = 2^k$ for $k \in \mathbb{N}$. This scenario would be the worst case in terms of melds / combinations. Every one-node queue has cost 1 and gets a credit in case of a carry operation. First of all, all one-node heaps will be combined into heaps of rank 1 which releases one credit to pay for the work. Then all heaps are combined to heaps of rank 2, again releasing one credit to pay for the work. We keep combining the trees until we have one big binomial tree. In this example, we represent the binomial heap as a binary tree. A leaf denotes a one-item queue, whereas an internal node indicates the melding of two heaps. We know that we can build one single binomial tree with $n = 2^k$ nodes. Still using the binary tree representation, in order to get from one layer to the next layer, always two queues of the same rank are combined and release one credit. There are 2^k leaves of costs 1 and we have costs of 2^{k-1} in order to get to the layer above etc. (costs are always halved). If we keep going up to the root and sum up all the costs, we get

$$\sum_{i=0}^{k} 2^{k-i} = 2^{k+1} - 1 = 2n - 1 \in \mathcal{O}(n).$$

In the case that the rank invariant is violated and we have to dismantle the root, we can use the fact that the root v has lost more than $\lceil rank(v)/2 \rceil$ children. This means that one child of rank at least $rank(v)/2$ must be missing. This **missing** subtree has at least $2^{\lceil rank(v)/2 \rceil - 1}$ leaves that are missing in the queue and make up for the costs of re-melding the other children of v.

4.2.2 Sift

The earliest stage where the recursion tree of sift can start branching is at a node of rank $r + 1$. Along such a path of length at least $r + 1$, the recursion tree branches at least once. We show that the depth of the recursion tree is in $\mathcal{O}(r)$. We assume that n is large enough so that $1/\epsilon$ lies between a large constant c and $\sqrt{n}$. Let's assume that c is in $\Omega(n^{1/4})$. We know that the entire tree has a depth that is logarithmic in n and that $r \in \mathcal{O}(\log 1/\epsilon)$. By comparing the lower bound of r with the height of the entire tree, we see that the recursion path is in $\mathcal{O}(r)$ as $\frac{\log n}{\log n^{1/4}} = \frac{\log n}{\frac{1}{4}\log n} = 4$.
If the recursion tree is branching, we always merge two item-lists. There can at most be $n-1$ merges of item lists in a soft heap storing n items. Therefore, we conclude that sift has a running time of $\mathcal{O}(rn)$. By setting $r = 2 + \lceil \log(1/\epsilon) \rceil$ we get $\mathcal{O}(n \log 1/\epsilon)$.

The costs of updating the *min_pointer* that points to the queue with the minimum *ckey* are covered either by the costs of the sift operation or are made up for by the missing leaves as mentioned above. This will not be elaborated further in this report.

Proof of Theorem 1: We proved the running time in the section above. By lemma 4.2, the number of corrupted items is bounded by $n/2^{r-3}$. We choose $r(\epsilon) = 2 + 2\lceil \log 1/\epsilon \rceil$ which proves the upper bound for the number of corrupted items. Using a standard counting argument, one can show that the claimed bounds are optimal in a comparison-based model.

5 Conclusion

The paper from Kaplan and Zwick [3] was a good introduction into the topic as Chazelle's paper was not completely clear from the beginning.
Finally, there was no clear explanation, why the amortized time of *insert* is $\mathcal{O}(\log 1/\epsilon)$. As proven in 4.2 the meld operation has costs of $\mathcal{O}(n)$ and the sift operation has costs of $\mathcal{O}(n \log 1/\epsilon)$. Therefore, it seems unclear, why the insert operation has amortized costs of $\mathcal{O}(\log 1/\epsilon)$ as the insert operation can only cause a further meld operation and no sift operation.

References

[1] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. J. ACM, 47(6):1028–1047, 2000.

[2] Bernard Chazelle. The soft heap: an approximate priority queue with optimal error rate. J. ACM, 47(6):1012–1027, 2000.

[3] Haim Kaplan and Uri Zwick. A simpler implementation and analysis of chazelle's soft heaps. In Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009, pages 477–485, 2009.

[4] Wikipedia. Binomial heap — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Binomial%20heapoldid=863114906, 2018. [Online; accessed 12-October-2018].