Algorithms for fast selection on heap data structures

Jerry W Mao George W Tang Michael S Zhang

6.854/18.415J Final Project — Fall 2020

1 Introduction

Heaps are a classical data structure for maintaining ordered sets of data in a comparison-based sorting model. Heaps most commonly support operations such as insertion, delete-min, decrease-key and meld, with some of the most well-known implementations including binary heaps [10], binomial heaps [9] and Fibonacci heaps [5].

It is generally understood that as heaps are tree-like structures, by maintaining order one trades off the right to efficient random access. Specifically, we examine the tasks of selecting or extracting (deleting and returning) the k smallest elements from a heap, which are extensions of the standard operations `find-min` and `delete-min` in a min-heap.

In this paper, we present an algorithm for selection on heap-ordered-trees due to Kaplan et al [7], which makes extensive use of the soft heap data structure [2]. We then present an application of this algorithm to Fibonacci heaps, achieving an optimal algorithm for the selection task [8], which runs in $\mathcal{O}\left(k \log \frac{n}{k}\right)$ time.

We further analyze the algorithm for selection by [7] and [8] and compare it to existing methods such as PICK [1], or naïve repetitive applications of `delete-min`. We show that the improvement offered by the algorithm is bounded by a factor of $\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$, and moreover that for most values of k, existing algorithms are in fact already optimal. Finally, we examine an implementation of these algorithms, to evaluate their element comparison efficiency in a practical setting.

2 Soft heaps

Our algorithm will make extensive use of soft heaps [2], a variant of the classical min-heap data structure. Soft heaps support the same set of basic operations performed by heaps, namely `insert`, `find-min`, `delete-min` and `meld`; however, it achieves all of these operations in amortized $\mathcal{O}(1)$ time.

As a price for this apparent disregard for information-theoretical lower bounds, soft heaps trade a lack in accuracy. Specifically, soft heaps may choose to "corrupt" elements by increasing their key to decrease the overall entropy of the data structure; the user has no control over when corruptions occur. For a customizable parameter ε, soft heaps guarantee only that if n insertions into the heap have occurred, then there may be at most εn corrupted items in the heap at that point in time.

The runtime of these operations is proportional to $\mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$, and for constant ε, this is constant time. As soft heaps are a complex data structure, we provide some details of its functionality in Appendix B, and encourage the reader to consult the original paper by Chazelle [2].

2.1 Intuitions on soft heap applications

We emphasize that soft heaps do not attempt to provide any guarantee on the quantity of elements that are corrupted upon deletion; rather, the only guarantee is on the frequency of corrupted elements currently stored. It is entirely possible for every deleted element to be corrupted, as new corrupted elements may appear during the deletion operation.

Given this, one may question whether soft heaps are even at all useful. Below, we show a classical application of soft heaps to the problem of approximate median finding, as presented by Chazelle [2].

Definition 2.1.1. Given a parameter ε, an **approximate median** of a list of n elements is an element whose rank is between $(1-\varepsilon)\cdot\frac{n}{2}$ and $(1+\varepsilon)\cdot\frac{n}{2}$.

Theorem 2.1.2. *For constant ε, a soft heap can be used to find an approximate median of n items in $\mathcal{O}(n)$ time.*

Proof. Insert all items into a ε-soft heap, and then perform `delete-min` a total of $(1-\varepsilon)\cdot\frac{n}{2}$ times. Find the maximum element e among these deleted items; this takes $\mathcal{O}(n)$ time total.

The rank of e is at least $(1-\varepsilon)\cdot\frac{n}{2}$, because e is the largest among this many items. Additionally, at most εn corrupt elements remain in the soft heap, so at most εn more elements are smaller than e, and hence e has a maximum rank of $(1+\varepsilon)\cdot\frac{n}{2}$. We conclude that e is an approximate median. $\square$

Intuitively, although soft heaps may eventually corrupt arbitrarily many elements, the bound on the quantity of corrupted elements contained in the heap forms a constraint on the extracted elements. This idea forms a basis for selection algorithms on heap data structures, which will use similar concepts.

3 Fast heap selection using soft heaps

We now explore the application of soft heaps to supporting a `select-k` operation on various heap data structures. We begin with binary heaps, and extend the algorithm to d-ary heaps and Fibonacci heaps.

3.1 Selection from binary heaps

We first consider the problem of selecting the k smallest elements from a binary heap. There exists a straightforward $\mathcal{O}(k\log k)$ algorithm for this task, as follows.

Algorithm 3.1.1. Initialize an empty priority queue P, containing the root of the binary heap. Repeatedly extract the minimum element from P, and insert into P its children in the binary heap. Once k elements have been extracted from P, it should be evident that they are the k smallest elements in the heap. As P never contains more than $\mathcal{O}(k)$ elements, the entire algorithm runs in $\mathcal{O}(k\log k)$ time.

As a byproduct, this algorithm also produces these k elements in sorted order; however, this is not required by the `select-k` task. Nevertheless, whether desired or not, producing a sorted list may be a source of extra complexity. To improve upon this naïve solution, we therefore seek to allow unsorted output.

Kaplan et al [7] achieve this by replacing the priority queue with a soft heap H, with error parameter $\varepsilon = \frac{1}{4}$. As soft heaps are inaccurate, we will need select more than k elements; however, the crucial insight is that the soft heap guarantee enables us to select just $\mathcal{O}(k)$ elements, forming a superset of the desired k smallest elements. This improved algorithm, is detailed below.

Algorithm 3.1.2 (SOFT-SELECT)**.** Initialize an empty soft heap Q with error parameter $\varepsilon = \frac{1}{4}$, containing the root of the binary heap. Repeatedly extract the minimum item e from Q. Each time, if e is not corrupt, then insert into Q its children in the binary heap. Also insert the children of any elements corrupted by this operation. Once k elements have been extracted from Q, consider the set of all elements that have ever been inserted into Q. We claim that this set is a superset of the desired output, and is $\mathcal{O}(k)$ in size.

Lemma 3.1.3. *If SOFT-SELECT produces a set of size $\mathcal{O}(k)$ containing all k desired outputs, then the k smallest items can be selected in $\mathcal{O}(k)$ time.*

Proof. This is a simple application of the classical linear-time selection algorithm due to Blum et al [1]. After finding the k-th smallest item e in the set, simply partitioning the set at e produces the desired output. □

We now show that SOFT-SELECT does indeed produce this output as claimed.

Lemma 3.1.4. *SOFT-SELECT inserts the k smallest items from the binary heap into Q.*

Proof. Suppose X is the set of non-corrupted items currently in Q, so that X is initially the root of the binary tree. Firstly, we claim any item that has never been in Q must have an ancestor in X. This is simply because, any time a node x leaves X, whether by corruption or deletion from Q, it is immediately replaced by all of its children. The result then follows by induction.

Moreover, no descendant of X has ever been in Q. We show this also by induction: consider a node y that is inserted into Q. SOFT-SELECT specifies that this occurs only if y is the child of some $x \in X$ that has just been corrupted or deleted from Q. In either case, x is no longer in X, so the invariant is preserved.

Intuitively, we have shown that X forms a barrier between nodes that have and have not been in Q, as shown in Figure 3.1.

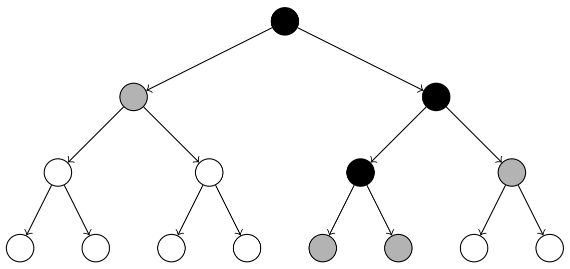


Figure 3.1: An example binary tree, showing the barrier X in grey, other items that have been inserted into Q in black, and items that have not been inserted in white.

Now, we argue that the smallest item in X can never decrease. As elements are never un-corrupted, X only acquires new elements when they are inserted into Q. However, new elements in Q must be descendants of X, and so, by the heap invariant, they are no less than the current smallest item in X.

Observe also that any item e extracted from Q must not be greater than the smallest item of X. This is because items in X are not corrupted, and corruption of e will only increase its key in Q.

Finally, we consider the smallest item $x_{\min} \in X$ at the conclusion of SOFT-SELECT. Any item w not inserted into Q must have an ancestor $x \in X$, so $w \geq x \geq x_{\min}$. On the other hand, since $x_{\min}$ never decreases, all extracted elements are no greater than $x_{\min}$, so the rank of $x_{\min}$ is greater than k. Hence all elements of rank at most k must have been inserted into Q. □

Lemma 3.1.5. *SOFT-SELECT only inserts $\mathcal{O}(k)$ items into H.*

Proof. Let I be the number of insertions into Q, and let m be the total number of corruptions that occur during SOFT-SELECT. There are at most two insertions per deletion and corruption, so $I \leq 2k + 2m$.

Additionally, because there are I insertions, there are at most εI corrupted items in the soft-heap at the conclusion of the algorithm. In the worst case, all k deleted items were corrupted, so $m \leq k + \varepsilon I$.

Combining these two inequalities yields

$$I \leq 2k + 2(k + \varepsilon I) \implies I \leq \frac{4k}{1 - 2\varepsilon} = \mathcal{O}(k)$$

when ε is a constant less than 1/2. □

Theorem 3.1.6. *Using soft heaps, it is possible to perform* `select-k` *on a binary heap in $\mathcal{O}(k)$ time.*

Proof. As soft heap operations are all amortized $\mathcal{O}(1)$ time, Lemmas 3.1.4 and 3.1.5 assert that SOFT-SELECT correctly creates a list of size $\mathcal{O}(k)$ containing the k smallest elements, in $\mathcal{O}(k)$ time. Then, Lemma 3.1.3 produces the desired result. □

3.2 Selection from d-ary heaps

SOFT-SELECT can be extended to general d-ary heaps, at a multiplicative time cost of $\mathcal{O}(d)$. Intuitively, it suffices to find an implicit transformation of any d-ary heap into a binary heap, and then invoke the known solution directly. This is the SOFT-SELECT-HEAPIFY algorithm [7].

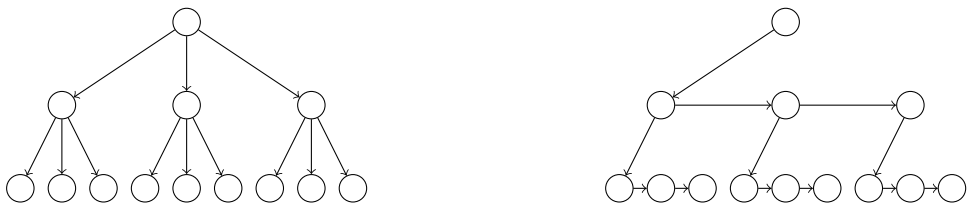


Figure 3.2: Simple binarization example. The left shows the initial heap, while the right shows the heap structure after binarization.

Algorithm 3.2.1 (Binarization)**.** We can convert a heap of maximum degree d into a binary heap as follows. Suppose that the children of each node are sorted in non-decreasing order. Then for any node v, we replace its edges with just two children: one to its smallest child, and one to its next-smallest sibling.

For example, Figure 3.2 shows the effect of a binarization operation on a small heap. However, binarization requires sorting the children of each node into increasing order, which incurs a time cost of $\mathcal{O}(d \log d)$. This is greater than desired, so to remedy this, we first ternarize the heap. This ensures that binarization can operate on a heap with constant $d = 3$.

Algorithm 3.2.2 (Ternarization)**.** For each node v in a d-ary heap, construct a binary heap T_v out of v's children. Then, each node $w \in T_v$ gains an extra child, which is the heap T_w. This produces a heap of maximum degree 3: each v now has at most one child representing its original children, and at most two children corresponding to its original siblings.

Observe that this ternarization process occurs in $\mathcal{O}(d)$ time for each inspected node, due to the process of constructing a binary heap. As such, any d-ary heap can be converted into an equivalent binary heap, by first ternarization to produce constant d, followed by binarization, with the whole process costing a multiplicative factor of $\mathcal{O}(d)$.

Moreover, this binarization process can be performed online; that is, the whole tree need not be binarized at once, and individual nodes and their children can be binarized on an as-needed basis. Using this "on-the-fly" binarization, we only incur the $\mathcal{O}(d)$ cost for each inspected node.

Algorithm 3.2.3 (SOFT-SELECT-HEAPIFY)**.** Binarize an input d-ary heap on the fly, via ternarization. Invoke SOFT-SELECT on this binarized heap.

Theorem 3.2.4. *SOFT-SELECT-HEAPIFY performs* `select-k` *on a d-ary heap in $\mathcal{O}(dk)$ time.*

Proof. We binarize the d-ary heap on-the-fly and then apply SOFT-SELECT to the implicitly constructed binary tree. Due to Lemma 3.1.5, only $\mathcal{O}(k)$ nodes are inserted into Q, so only $\mathcal{O}(k)$ nodes require binarization. As Theorem 3.1.6 asserts SOFT-SELECT spends $\mathcal{O}(k)$ time solving the problem, the overall runtime is therefore $\mathcal{O}(dk)$. □

3.3 Selection from general heap-ordered trees

In the previous section, it was shown that selection from d-ary heaps can be performed in $\mathcal{O}(dk)$ time, as it takes $\mathcal{O}(d)$ time to heapify the children of each node. Consequently, one may expect for the runtime of this algorithm on general heap-ordered trees to depend on the sum of degrees of visited nodes. We formalize this notion below using the results of [7], as this will equip us with the required machinery to demonstrate selection on Fibonacci heaps.

Definition 3.3.1. Let T be a tree rooted at r, and let C be a connected set of nodes containing r. Then, define $F(C)$ to be the sum of the degrees of the nodes in C.

For example, if C is a length-4 path in a binary tree, then $F(C) = 4 \cdot 2 = 8$.

Definition 3.3.2. Let T be a tree rooted at r, and let n be a positive integer. Then, define $D(T, n) = \max\limits_{C \in S} F(C)$, where S is the set of all valid C with $|C| = n$.

For example, if T is a tree in which the root node has 1 child, the child of the root has 2 children, the grandchildren of the root have 3 children, and so on, $D(T, 4) = 1 + 2 + 3 + 4 = 10$.

Theorem 3.3.3. *Let T be a heap-ordered tree. SOFT-SELECT-HEAPIFY can perform* `select-k` *on T in $\mathcal{O}(D(T, 3k))$ time.*

Proof. Use a soft heap with error parameter $\varepsilon = \frac{1}{6}$. The number of deleted nodes is bounded by k, and from the equations in Lemma 3.1.5, the total number of corrupted nodes m is bounded by

$$m \leq k + \varepsilon I \leq k + \frac{4\varepsilon k}{1 - 2\varepsilon} = 2k$$

Therefore, a total of $3k$ nodes are either deleted or corrupted by Q. Observe that in Lemma 3.1.4, we showed that these $3k$ nodes form a connected component of the binarized input tree including the root.

Like in Theorem 3.2.4, we only need to binarize these $3k$ nodes. The cost of binarizing each node is proportional to their degree, and so the total cost is bounded by the sum of the degrees of these $3k$ nodes. It follows by definition that this is bounded by $D(T, 3k)$, where T is the binarized heap.

Aside from binarization, this algorithm must also perform SOFT-SELECT on the resulting binary tree. As such, SOFT-SELECT-HEAPIFY terminates in $\mathcal{O}(k + D(T, 3k)) = \mathcal{O}(D(T, 3k))$ time. □

3.4 Selection and deletion from Fibonacci heaps

In this section, we extend the results of general heap-ordered trees to Fibonacci heaps, to support $\mathcal{O}\left(k \log \frac{n}{k}\right)$ `select-k` and `delete-k`.

The main obstacle to directly applying SOFT-SELECT-HEAPIFY is that Fibonacci heaps can often consist of more than one heap-ordered tree. As such, the solution here seeks to circumvent this by temporarily connecting the forest.

Algorithm 3.4.1 (FIBONACCI-SELECT)**.** Create a new node R with key equal to $-\infty$, and temporarily cause all trees in the heap to be children of R. Then, SOFT-SELECT-HEAPIFY on the resulting tree with parameter $k + 1$ will return $-\infty$, along with the k smallest items in the Fibonacci heap. Finally, we remove R and consolidate the heap by repeatedly combining roots of equal rank.

To analyze the runtime of this algorithm, we first recall the following well-known lemmas about Fibonacci heaps, and assert them without proof. Furthermore, let t be the number of heap-ordered trees in the Fibonacci heap.

Lemma 3.4.2. *Let $x_1 \dots x_d$ be the children of a d-degree node. The i^{th} child (in order of being added to the node) has degree at least $\max(i - 2, 0)$.*

Lemma 3.4.3. *A node of rank r has at least $F_{r+1} \geq \phi^r$ descendants, where F_i is the i^{th} Fibonacci number ($F_0 = F_1 = 1$) and ϕ is the golden ratio.*

We now proceed to show a bound on the magnitude of $D(T_f, k)$.

Lemma 3.4.4. *Let T_f denote the tree formed by combining the heap-ordered trees at R. Then for any $k \geq 2$, we have $D(T_f, k) = t + \mathcal{O}\left(k \log \frac{n}{k}\right)$.*

Proof. Consider a connected component $C_f \subseteq T_f$ of k elements, such that the sum of degrees of nodes in C_f is equal to $D(T_f, k)$. Note that since C_f itself is a tree, there are precisely $k - 1$ edges between pairs of nodes in C_f. This leaves $D(T_f, k) - k + 1$ edges from nodes in C_f to their children.

To analyze the worst-case value of $D(T_f, k)$, we seek to maximize the number of edges leaving C_f. Observe that an adversary constructing such a worst-case tree would generally seek to minimize the sizes of subtrees of children of C_f, because if any subtree was larger than necessary, those nodes may be better placed elsewhere to increase the total degree.

Concretely, we consider beginning with the tree C_f and adding new subtrees to it, until all n nodes have been added. Lemma 3.4.2 allows any node to have at most two children of rank 0, and then at most one of each subsequent rank. On the other hand, Lemma 3.4.3 implies that it is optimal to greedily select the smallest available rank; this is because exponentiation by ϕ is a convex function, so Jensen's inequality implies that two trees of rank d can have fewer nodes than trees of rank $d-1$ and $d+1$.

Therefore, we consider the worst-case in which we assign the same quantity d of children to each node. Since T_f contains $k+1$ nodes, the new subtrees contain a total of $n-k+1$ nodes, allowing us to bound d as follows.

$$\begin{aligned}
n-k+1 &\geq (k-1)\sum_{j=1}^{d} F_{j-1} \\
&= (k-1)(F_{d+1}-1) \\
F_{d+1} &\leq \frac{n-k+1}{k-1}+1 \\
&= \frac{n}{k-1} \\
d &= \mathcal{O}\left(\log\frac{n}{k}\right)
\end{aligned}$$

Thus, the maximum number of edges leaving C_f is bounded by $\mathcal{O}(kd) = \mathcal{O}\left(k\log\frac{n}{k}\right)$. We note that the only node allowed to violate Lemma 3.4.2 is the auxiliary root node R, so that the total degree sum is bounded by $D(T_f, k) = \mathcal{O}\left(t + k\log\frac{n}{k}\right)$. □

Theorem 3.4.5. *FIBONACCI-SELECT solves* `select-k` *in amortized* $\mathcal{O}\left(k\log\frac{n}{k}\right)$ *time.*

Proof. Recall from Theorem 3.3.3 that the runtime of this algorithm is $\mathcal{O}(D(T_f, 3(k+1)))$. From Lemma 3.4.4, the algorithm has a real cost of $\mathcal{O}\left(t + k\log\frac{n}{k}\right)$ time.

On the other hand, the consolidation procedure seeks to reduce the number of heaps, causing a change in potential of $\Delta\Phi = \log n - t$. The amortized runtime is therefore $\mathcal{O}\left(k\log\frac{n}{k}\right)$. □

Finally, we remark that the work performed so far allows us to immediately derive a solution for `delete-k`, as follows.

Algorithm 3.4.6 (FIBONACCI-DELETE)**.** Perform FIBONACCI-SELECT, and then delete R as well as all k selected algorithms. This produces at most $\mathcal{O}(D(T_f, 3(k+1)))$ disconnected subtrees; we consolidate trees of equal rank.

This algorithm clearly runs also in $\mathcal{O}\left(k\log\frac{n}{k}\right)$ time.

4 Optimality of algorithms for heap extraction

4.1 Information theoretic lower bound

Under a comparison-based model, information theory dictates a theoretical lower bound on the algorithmic complexity of the `delete-k` operation. Sandlund and Zhang [8] derive this bound by extending upon the multiselection problem, using the results of Dobkin and Munro [3]. We summarize these bounds below.

Definition 4.1.1 (Multiselection)**.** Suppose we are given an unordered collection of n items and k indices $x_1, \dots, x_k$ in ascending order. The **multiselection problem** is to select the x_i^{th} smallest element from the collection, for each i.

Lemma 4.1.2. *Additionally define $x_0 = 0$ and $x_{k+1} = n$. Any algorithm that solves the multiselection problem must incur at least $n \log n - \sum_{i=0}^{k}(x_{i+1} - x_i)\log(x_{i+1} - x_i) - \mathcal{O}(n)$ comparisons.*

Proof. Any algorithm that can identify the desired elements must also determine which elements have ranks between each x_i and x_{i+1}. Otherwise, an adversary can easily provide queries which the algorithm cannot successfully distinguish.

From here, it becomes possible to sort the entire list by sorting each interval individually. It is well-known that comparison-based sorting requires at least $n \log n - \mathcal{O}(n)$ comparisons. On the other hand, sorting each interval can be achieved in at most $(x_{i+1} - x_i)\log(x_{i+1} - x_i)$ comparisons, for a total of

$$\sum_{i=0}^{k}(x_{i+1} - x_i)\log(x_{i+1} - x_i)$$

Subtracting these bounds produces the desired result. □

Theorem 4.1.3. *Any heap structure supporting* `delete-k` *must have $\Omega(\log \frac{n}{k})$ insertion or $\Omega(k \log \frac{n}{k})$* `delete-k` *runtime.*

Proof. We show this via reduction from the multiselection problem in Lemma 4.1.2. Specifically, consider the case where there are $\lfloor \frac{n}{k} \rfloor$ items to be selected, with indices $x_i = ki$. In particular, note that $x_{i+1} - x_i = k$. As a result, the lemma implies a lower bound of

$$n \log n - \left(\left\lfloor \frac{n}{k} \right\rfloor + 1 \right) k \log k - \mathcal{O}(n) = n \log \frac{n}{k} - \mathcal{O}(n)$$

On the other hand, a specific solution to this problem is to insert all n items into a heap, and then repeatedly call `delete-k`. The maximum of each set of k items can be computed in $\mathcal{O}(k)$ time each, and constitute the desired output for the problem.

Suppose there existed a heap that supports insertion in $o(\log \frac{n}{k})$ and `delete-k` in $o(k \log \frac{n}{k})$. Then, by n insertions and $\lfloor n/k \rfloor$ applications of `delete-k`, we obtain the answer to the multiselection problem in $o(n \log \frac{n}{k})$ time, contradicting the established lower bound. □

As a consequence, we conclude that FIBONACCI-DELETE is information-theoretically optimal.

4.2 Comparison to existing algorithms

PICK [1], commonly known as "median of medians", is a classical algorithm for fast selection in an unordered set of items, using recursion to find the k-th smallest item in $\mathcal{O}(n)$ time. It can easily be applied to `select-k` and `delete-k` by finding the k-th smallest item, and then applying a linear-time partitioning algorithm.

If the heap structure also supports constant-time insertion (perhaps amortized time), then `delete-k` can also be achieved in $\mathcal{O}(n)$ time, by simply constructing a new heap out of the remaining elements in the partition.

Lemma 4.2.1. *If $k = \Theta(n)$, then PICK is asymptotically information-theoretically optimal.*

Proof. Suppose that c_1 and c_2 are constants such that $c_1 n \le k \le c_2 n$. Then,

$$n \le \frac{1}{c_1} \cdot k \le \frac{1}{c_1 \log \frac{1}{c_2}} \cdot k \log \frac{n}{k}$$

As PICK runs in $\mathcal{O}(n)$ time, in this case it also runs in $\mathcal{O}\left(k \log \frac{n}{k}\right)$ time, matching the information-theoretic lower bound presented in Theorem 4.1.3. □

Another straightforward algorithm for `delete-k` is REPEATED-DELETE, in which `delete-min` is simply called on the heap a total of k times.

Lemma 4.2.2. *If $k = \mathcal{O}\left(n^{1-\varepsilon}\right)$ for any $\varepsilon > 0$, then REPEATED-DELETE on any heap supporting $\mathcal{O}(\log n)$ time `delete-min` is asymptotically information-theoretically optimal.*

Proof. Suppose c is a constant such that $k \le c \cdot n^{1-\varepsilon}$. Then,

$$k \log n = \frac{1}{\varepsilon} \cdot k \log n^{\varepsilon} = \frac{1}{\varepsilon} \cdot k \log \frac{n}{n^{1-\varepsilon}} \le \frac{1}{\varepsilon} \cdot k \log \frac{c \cdot n}{k}$$

Calling `delete-min` k times would spend a total time of $\mathcal{O}(k \log n)$, which is also $\mathcal{O}\left(k \log \frac{n}{k}\right)$, matching the information-theoretic lower bound presented in Theorem 4.1.3. □

Thus, the FIBONACCI-DELETE can only outperform these naïve algorithms when k is neither $\Theta(n)$ nor $\mathcal{O}\left(n^{1-\varepsilon}\right)$ for any $\varepsilon > 0$.

Theorem 4.2.3. *If $k = \Theta(n)$ or $k = \mathcal{O}\left(n^{1-\varepsilon}\right)$ for some $\varepsilon > 0$, then FIBONACCI-DELETE is asymptotically equivalent to one of PICK or REPEATED-DELETE. Otherwise, it can outperform these solutions by a factor of up to $\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$.*

Proof. From Lemmas 4.2.1 and 4.2.2, FIBONACCI-DELETE is asymptotically equivalent to one of the two naïve algorithms when $k = \Theta(n)$ or $k = \mathcal{O}\left(n^{1-\varepsilon}\right)$.

Otherwise, we consider its factor of improvement. PICK runs in $\mathcal{O}(n)$, so the improvement is $\mathcal{O}\left(\frac{n}{k \log \frac{n}{k}}\right)$. REPEATED-DELETE runs in $\mathcal{O}(k \log n)$, so the improvement is $\mathcal{O}\left(\frac{k \log n}{k \log \frac{n}{k}}\right)$. Therefore, the overall factor is

$$\mathcal{O}\left(\min\left(\frac{n}{k \log \frac{n}{k}}, \frac{k \log n}{k \log \frac{n}{k}}\right)\right)$$

Balancing terms in this expression yields a maximum bound at $k = \Theta\left(\frac{n}{\log n}\right)$, which corresponds to an improvement factor of $\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$. □

5 Selection algorithms in practice

While FIBONACCI-DELETE is information-theoretically optimal, Theorem 4.2.3 suggests that in practical applications, there are very few values of k for which it outperforms both PICK and REPEATED-DELETE asymptotically. Moreover, even for such values of k, the small improvement factor of $\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$ may be dominated by constant terms. In this section, we evaluate implementations of these algorithms on synthetic datasets and various query patterns.

5.1 Experimental setup

To deterministically evaluate the efficiency of each algorithm, we measure the quantity of performed element comparisons in each test case. In practical applications in which a total ordering of elements is expensive to compute, optimizing the number of comparisons may be an important design goal. Throughout all tests, we use the `insert`, `decrease-key`, and `delete-k` operations, as these are sufficient to generate interesting tree structures.

5.2 Test data generation

Each algorithm was tested on ten representative problem families produced using a robust generator for an input size of 10^6 elements and varying values of k, as shown in Table 5.1.

The generator allows for the customization of three important aspects of test cases: the randomness of insertion sequences, the order of operations performed, and the item on which `decrease-key` acts. The parameters for these aspects are a constant $0 \leq \alpha \leq 1$, a 3×3 stochastic matrix T, and an element sampler, respectively. These parameters suffice to produce a diverse suite of test data examining a wide range of query and data patterns.

Insertion Stream First, the stream is generated as a sorted (either monotonically increasing or decreasing) sequence. Then, a permutation is produced using a Fisher-Yates shuffle [4]. Each swap requested by Fisher-Yates is performed with probability α, controlling the extent of the shuffle; for instance, $\alpha = 0$ would not allow any shuffling, while $\alpha = 1$ produces an unbiased permutation.

Operation Stream We generate operations from a Markov chain defined over the three operation types. The stochastic matrix T defines a transition space on which a random walk can be used to generate the sequence of operations, according to $T_{i,j} = Pr(j|i)$. To avoid invalid operations such as deleting more elements than the heap contains, illegal transitions are discarded and the probability distribution is normalized. Finally, the quantity of `insert` and `decrease-key` operations is capped at n, the size of the data stream.

Element Sampler Each `decrease-key` operation must also have an associated target element. The element sampler records the time when each element was inserted, where time is defined as the number of previous insertions. Then among all elements still in the heap, the sampler determines the first and last inserted items, samples a time t uniformly at random within that

Formula	$\mathcal{O}(1)$	$\log n$	$n^{0.25}$	$n^{0.5}$	$0.01n$	$n^{0.75}$	$\frac{n}{\log n}$	$0.1n$	$n^{0.9}$	$0.5n$
k **value**	5	20	32	1000	10 000	31 622	50 172	100 000	251 189	500 000

Table 5.1: k values tested during experimentation, with their corresponding algebraic expression in terms of $n = 10^6$.

time window, and returns the first element inserted after t. This strategy has the benefit that recently inserted items are more likely to be sampled, while a few items remain present for long periods of time, reflecting many real-world applications. It was found that this scheme was sufficiently robust to produce interesting results.

5.3 Problem families

Ten representative problem families were used to benchmark PICK, REPEATED-DELETE, and FIBONACCI-DELETE. Within each family, varying values of k were tested for `delete-k`, to search for k where FIBONACCI-DELETE may outperform the naïve algorithms. These families attempt to test both extreme cases, to measure the robustness of our approach, as well as datasets that attempt to simulate real-world applications.

Ordered-Ordered The data stream is monotonically increasing ($\alpha = 0$). Identical operations are adjacent to each other, and the ordering is `insert`, `decrease-key`, `delete-k`. Specifically, for a small value of ε,

$$T = \begin{pmatrix} 1-\varepsilon & \varepsilon & 0 \\ 0 & 1-\varepsilon & \varepsilon \\ 0 & 0 & 1 \end{pmatrix}$$

Ordered-Uniform-Random The insertions are arranged in increasing order, but the operations are uniformly random (i.e. T has all entries $\frac{1}{3}$).

Uniform-Random-Ordered The data stream is an unbiased permutation ($\alpha = 1$), and the T from **Ordered-Ordered** is used.

Uniform-Random-Uniform-Random The data stream is an unbiased permutation ($\alpha = 1$), and the T from **Ordered-Uniform-Random** is used.

Cliffs-Uniform-Random The data stream is formed by ten equally-sized "cliffs", where a "cliff" is a series of increasing elements. The T from **Ordered-Uniform-Random** is used.

Hills-Uniform-Random The data stream is formed by five equally-sized "hills", where a "hill" is a series of increasing elements followed by a series of decreasing elements. The T from **Ordered-Uniform-Random** is used.

Uniform-Random-Sequential-DecreaseKey Since each `decrease-key` operation modifies the shape of the heap, this case considers calling longer sequences of `decrease-key`, so that the structure is significantly altered between `delete-k` operations. Specifically,

$$T = \begin{pmatrix} 1/3 & 1/3 & 1/3 \\ 1/10 & 4/5 & 1/10 \\ 1/3 & 1/3 & 1/3 \end{pmatrix}$$

Less-Random-Sequential-Operations The sequence is scrambled with $\alpha = 0.25$, which corresponds to a real-world scenario in which the data stream is generated with increasing time, but the arrival of each element is noisy. Furthermore, operations are clustered together, which may correspond to different phases of a data-processing algorithm. Specifically,

$$T = \begin{pmatrix} 0.8 & 0.1 & 0.1 \\ 0.1 & 0.8 & 0.1 \\ 0.1 & 0.1 & 0.8 \end{pmatrix}$$

More-Random-Sequential-Operations This family is the same as **Less-Random-Sequential-Operations**, except $\alpha = 0.8$, so each element's arrival is more noisy.

Uniform-Random-One-Delete The data stream is an unbiased permutation ($\alpha = 1$), and there are many `insert` and `decrease-key` operations with few `delete-k` operations. Specifically,

$$T = \begin{pmatrix} 0.495 & 0.495 & 0.01 \\ 0.495 & 0.495 & 0.01 \\ 0.495 & 0.495 & 0.01 \end{pmatrix}.$$

5.4 Experimental results

Overall, the results of the experiment support the hypothesis that for small values of k, REPEATED-DELETE uses the fewest comparisons, while for large values of k, PICK uses the fewest. We refer the reader to Appendix A for the full tabulated results of the experiment, and also make available our implementation at `https://github.com/gtangg12/6.854-final-project`.

The results also suggest that there is almost no k for which FIBONACCI-DELETE is superior to both naïve algorithms. In our experiment, it is always outperformed by at least one of the two, confirming the hypothesis that constant factors may render it undesirable for practical applications.

Notably, we run REPEATED-DELETE on both Fibonacci heaps and binary heaps, and find that the Fibonacci heap variant is consistently more efficient. On the other hand, it is a general trend in our results that as k increases to between 30 000 and 100 000, these algorithms begin to be outperformed by FIBONACCI-DELETE; however, this is in turn rapidly outperformed by PICK as k approaches n.

We recall that Theorem 4.2.3 suggests best relative performance of FIBONACCI-DELETE for values of k around $\Theta(\frac{n}{\log n})$. Empirically, we find that this does indeed appear to be the case when comparing with REPEATED-DELETE: while the algorithm is overall less efficient than the naïve algorithms, the algorithms achieve closest performance when k is in this range. For PICK, however, we point towards problem families such as **Uniform-Random-Uniform-Random**, on which the performance is comparable (see Figure A.4), and problem families such as **Uniform-Random-One-Delete**, on which PICK appears to perform far better (see Figure A.10).

Nevertheless, even at this value of k, the results suggest that constant factors prevent FIBONACCI-DELETE from outperforming PICK and REPEATED-DELETE at this best value of k. As such, we conclude that for $n = 10^6$, these constant factors may be a similar order of magnitude to the value of $\frac{\log n}{\log \log n}$, as found in Theorem 4.2.3.

We also examine the effects of the data stream and operation sequence. Observe that in Figures A.1 (**Ordered-Ordered**) and A.3 (**Uniform-Random-Ordered**), the graph contours are similar, albeit with the latter requiring more comparisons overall for all algorithms. Similarly, Figures A.2 (**Ordered-Uniform-Random**) and A.4 (**Uniform-Random-Uniform-Random**) also show comparable results. This suggests that the relative performance of these algorithms is influenced more heavily by the sequence of operations than by the data stream. In particular, in both cases a random insert insertion sequence only negligibly increases the comparison count compared to an ordered sequence, suggesting that the operation sequence is a dominant factor in determining comparison efficiency.

6 Conclusions

In this paper we present and implement FIBONACCI-DELETE, a soft-heap based algorithm for the `delete-k` operation on heap data structures. We show that from an information-theory perspective, it is asymptotically optimal comparison-wise, given that it supports constant-time insertion. It is able to bring an improvement of up to $\mathcal{O}\left(\frac{\log n}{\log\log n}\right)$ over other algorithms such as PICK and REPEATED-DELETE.

We further note that PICK and REPEATED-DELETE are theoretically optimal for a subset of values of k. To gain further perspective, we implement these algorithms and evaluate them on synthetic datasets. Indeed, we find that FIBONACCI-DELETE outperforms PICK for small values of k, and REPEATED-DELETE for large values of k, but is never able to simultaneously outperform both.

As such, we conclude that while FIBONACCI-DELETE is theoretically optimal, in practice it may be more desirable to consider alternate algorithms for applications where $n \approx 10^6$. We remark that there is potential for further investigation with $n \gg 10^6$, which were not examined due to compute constraints.

References

[1] Manuel Blum et al. "Time Bounds for Selection". In: *J. Comput. Syst. Sci.* 7.4 (Aug. 1973), pp. 448–461. ISSN: 0022-0000. DOI: `10.1016/S0022-0000(73)80033-9`.

[2] Bernard Chazelle. "The Soft Heap: An Approximate Priority Queue with Optimal Error Rate". In: *J. ACM* 47.6 (Nov. 2000), pp. 1012–1027. ISSN: 0004-5411. DOI: `10.1145/355541.355554`.

[3] David Dobkin and J. Ian Munro. "Optimal Time Minimal Space Selection Algorithms". In: *J. ACM* 28.3 (July 1981), pp. 454–461. ISSN: 0004-5411. DOI: `10.1145/322261.322264`.

[4] Ronald A. Fisher and Frank Yates. "Statistical Tables for Biological Agricultural and Medical Research". In: *Nature* 144.3647 (Sept. 1939), pp. 533–533. ISSN: 1476-4687. DOI: `10.1038/144533a0`.

[5] Michael L. Fredman and Robert Endre Tarjan. "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms". In: *J. ACM* 34.3 (July 1987), pp. 596–615. ISSN: 0004-5411. DOI: `10.1145/28869.28874`.

[6] Haim Kaplan, Robert E. Tarjan, and Uri Zwick. "Soft Heaps Simplified". In: *SIAM Journal on Computing* 42.4 (2013), pp. 1660–1673. DOI: `10.1137/120880185`.

[7] Haim Kaplan et al. *Selection from heaps, row-sorted matrices and $X + Y$ using soft heaps.* 2018. arXiv: `1802.07041 [cs.DS]`.

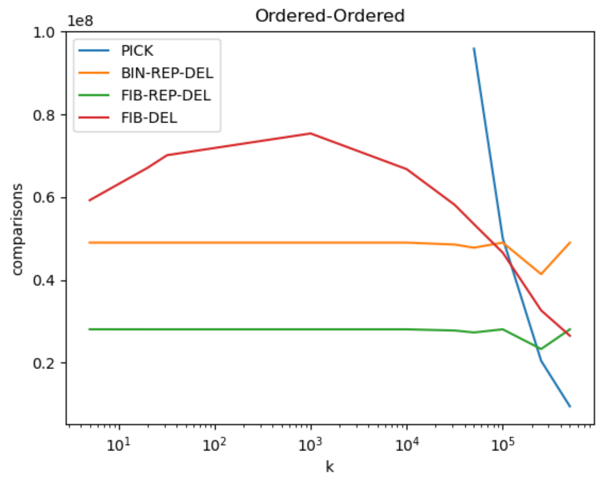
[8] Bryce Sandlund and Lingyi Zhang. *Selectable Heaps and Optimal Lazy Search Trees.* 2020. arXiv: `2011.11772 [cs.DS]`.

[9] Jean Vuillemin. "A Data Structure for Manipulating Priority Queues". In: *Commun. ACM* 21.4 (Apr. 1978), pp. 309–315. ISSN: 0001-0782. DOI: `10.1145/359460.359478`.

[10] J. W. J. Williams. "Heapsort". In: *Commun. ACM* 7.6 (June 1964). Ed. by G. E. Forsythe, pp. 347–349. ISSN: 0001-0782. DOI: `10.1145/512274.512284`.

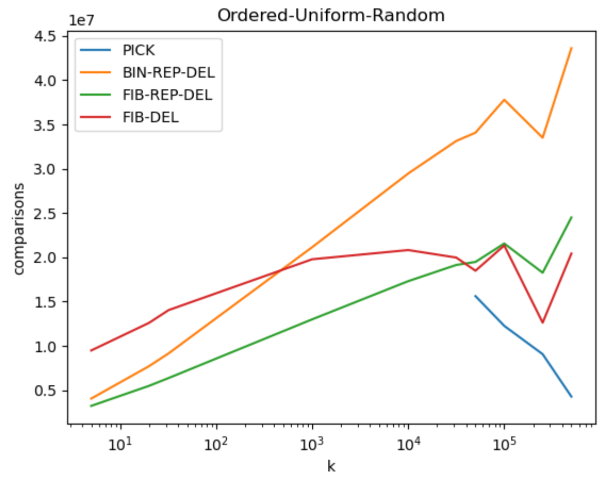
Appendix A Results

Included below are the results of the experiment, showing the quantity of comparisons used by each algorithm, for various values of k in each test family. FIB-REP-DEL refers to REPEATED-DELETE, while BIN-REP-DEL refers to the same REPEATED-DELETE algorithm, but benchmarked on a binary heap, rather than a Fibonacci heap. Note that a number of tests were not run for PICK, as the algorithm is very computationally inefficient for large volumes of `delete-k` operations.



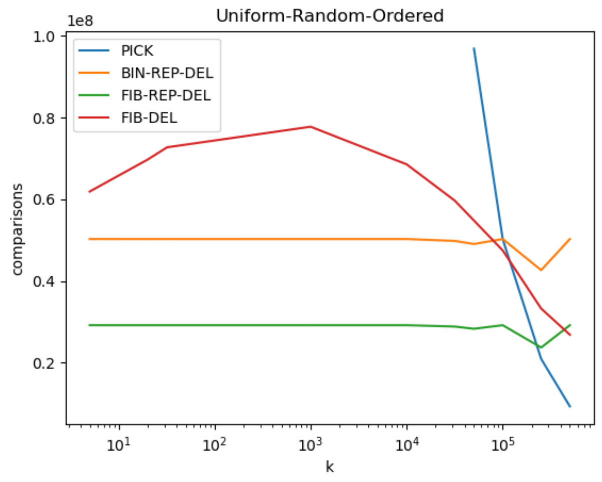
k	PICK	BIN-REP-DEL	FIB-REP-DEL	FIB-DEL
5	not run	49021012	28081404	59260027
20	not run	49017249	28076538	67086585
32	not run	49017810	28075406	70142523
1000	not run	49017853	28074971	75387099
10000	467401235	49018636	28075682	66765719
31622	not run	48552985	27784956	58137996
50172	95862157	47798206	27315318	53479690
100000	50070045	49018581	28074750	46603110
251189	20459744	41406042	23319601	32648801
500000	9455067	49018412	28067518	26519992

Figure A.1: Ordered-Ordered



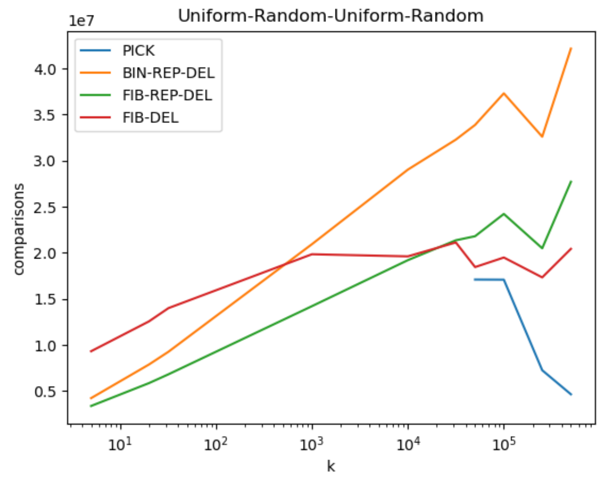
k	PICK	BIN-REP-DEL	FIB-REP-DEL	FIB-DEL
5	not run	4054784	3230845	9488031
20	not run	7714745	5495149	12604175
32	not run	9160860	6376341	14045124
1000	not run	21147698	13003674	19784622
10000	17346771	29467152	17312243	20823733
31622	not run	33140300	19141657	19979015
50172	15623774	34074833	19482496	18491601
100000	12266466	37793081	21564517	21339696
251189	9071549	33503532	18265331	12627981
500000	4280007	43611304	24504681	20429044

Figure A.2: Ordered-Uniform-Random



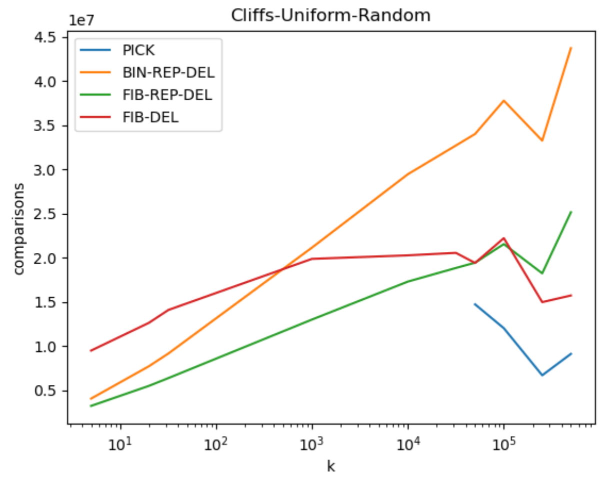
k	PICK	BIN-REP-DEL	FIB-REP-DEL	FIB-DEL
5	not run	50294030	29210326	61907060
20	not run	50291358	29211295	69708837
32	not run	50291847	29211547	72729938
1000	not run	50293527	29210022	77764308
10000	469681428	50292943	29209987	68556468
31622	not run	49827272	28880548	59690008
50172	96851067	49072874	28339208	54806135
100000	50244336	50293666	29210261	47493232
251189	20954125	42678666	23744815	33279244
500000	9346404	50294045	29208346	26884373

Figure A.3: Uniform-Random-Ordered



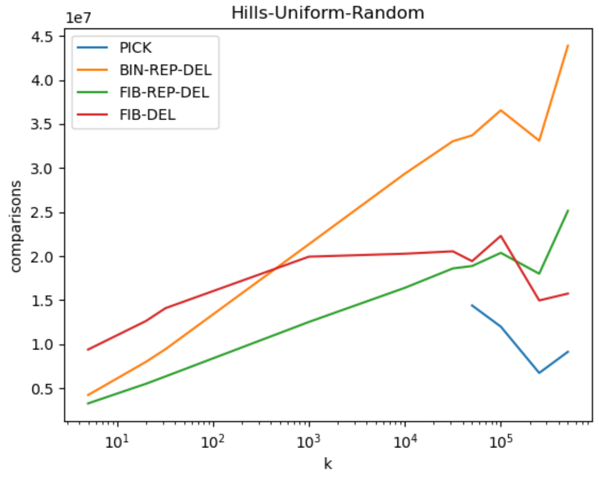
k	PICK	BIN-REP-DEL	FIB-REP-DEL	FIB-DEL
5	not run	4238395	3387738	9325009
20	not run	7879510	5862240	12558188
32	not run	9270388	6823883	14002289
1000	not run	20934096	14211862	19844465
10000	18504042	29024970	19203515	19614731
31622	not run	32280058	21359493	21125298
50172	17099795	33872009	21800673	18453178
100000	17086597	37298441	24217005	19491215
251189	7262645	32594557	20492668	17334154
500000	4655683	42147697	27705736	20429384

Figure A.4: Uniform-Random-Uniform-Random



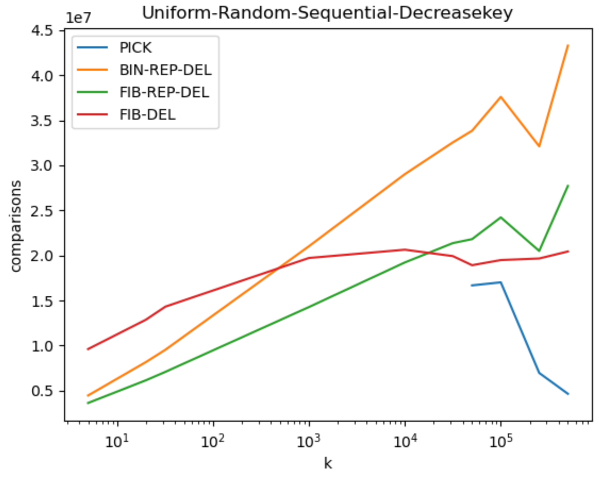
k	PICK	BIN-REP-DEL	FIB-REP-DEL	FIB-DEL
5	not run	4057668	3231974	9493976
20	not run	7717625	5497095	12633370
32	not run	9168953	6382917	14095181
1000	not run	21143740	13003169	19875052
10000	16561815	29458829	17303992	20274818
31622	not run	32706380	18842326	20556964
50172	14727682	34013914	19443681	19412888
100000	12025619	37786457	21550357	22222606
251189	6694109	33262332	18235505	14976639
500000	9119830	43725753	25153713	15731291

Figure A.5: Cliffs-Uniform-Random



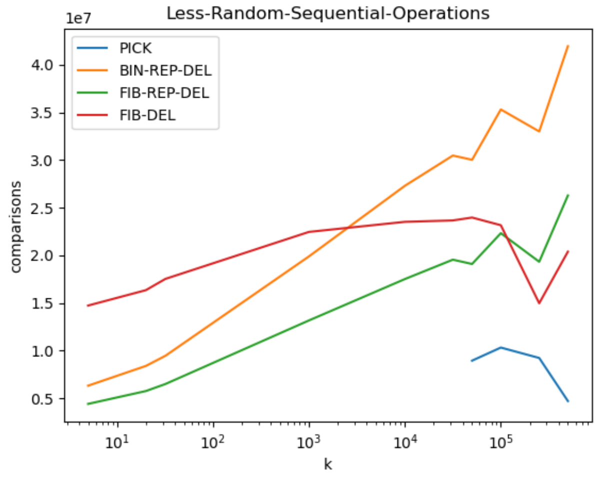
k	PICK	BIN-REP-DEL	FIB-REP-DEL	FIB-DEL
5	not run	4228279	3276289	9390306
20	not run	7986935	5501456	12625764
32	not run	9429151	6338289	14085551
1000	not run	21347093	12522687	19933036
10000	16439518	29354571	16396772	20268272
31622	not run	33038934	18597878	20541473
50172	14394458	33698423	18872301	19422044
100000	11989225	36549337	20366293	22292918
251189	6733131	33100420	17998160	14963229
500000	9134573	43881721	25139371	15742833

Figure A.6: Hills-Uniform-Random



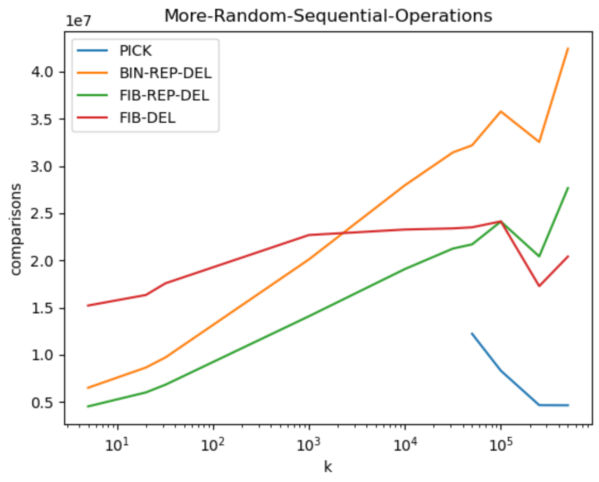
k	PICK	BIN-REP-DEL	FIB-REP-DEL	FIB-DEL
5	not run	4475904	3642168	9621442
20	not run	8176974	6156402	12875019
32	not run	9551557	7096744	14328237
1000	not run	21020827	14277479	19721273
10000	18359699	29018560	19225719	20638100
31622	not run	32541608	21369793	19929380
50172	16682163	33833557	21810950	18918275
100000	17019049	37585323	24223915	19490012
251189	6972759	32100922	20498228	19664387
500000	4661056	43269899	27708225	20442210

Figure A.7: Uniform-Random-Sequential-DecreaseKey



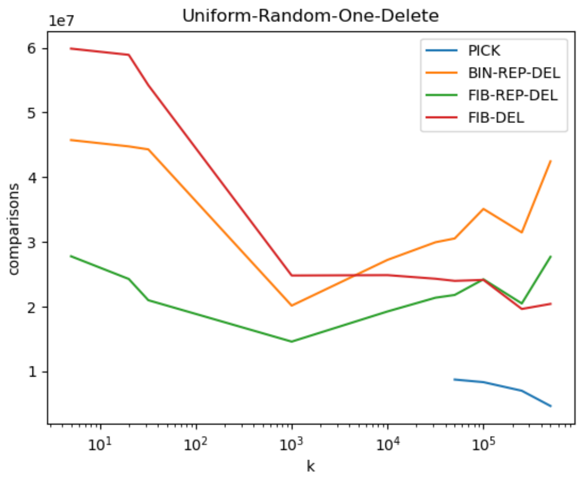
k	PICK	BIN-REP-DEL	FIB-REP-DEL	FIB-DEL
5	not run	6309729	4398120	14724536
20	not run	8384480	5747036	16342428
32	not run	9453505	6487883	17527096
1000	not run	19884383	13160458	22457815
10000	17658377	27301513	17506196	23517162
31622	not run	30478318	19542858	23663599
50172	8939153	30033785	19089663	23968242
100000	10313624	35317815	22325669	23161329
251189	9220303	33009022	19323250	14964352
500000	4695472	41964593	26284733	20386857

Figure A.8: Less-Random-Sequential-Operations



k	PICK	BIN-REP-DEL	FIB-REP-DEL	FIB-DEL
5	not run	6518612	4545576	15223251
20	not run	8659130	6023882	16338854
32	not run	9730618	6844282	17576516
1000	not run	20104160	14086615	22686446
10000	18384910	27952569	19084003	23267566
31622	not run	31427912	21249983	23387252
50172	12233471	32182433	21697854	23501811
100000	8326074	35762969	24114816	24129095
251189	4677014	32535795	20421491	17274447
500000	4663850	42408338	27657554	20408472

Figure A.9: More-Random-Sequential-Operations



k	PICK	BIN-REP-DEL	FIB-REP-DEL	FIB-DEL
5	-1	45744498	27789902	59877359
20	-1	44768690	24280799	58916508
32	-1	44308322	20997295	54225523
1000	-1	20163561	14600152	24818257
10000	8929274	27237159	19263459	24873962
31622	-1	29956104	21385207	24315739
50172	8736518	30543047	21812739	23987188
100000	8335571	35110430	24228291	24137629
251189	7007701	31468097	20494689	19646502
500000	4655202	42463011	27705518	20427789

Figure A.10: Uniform-Random-One-Delete

Appendix B Soft heaps

Here we describe the soft heap data structure. Recall that for constant ε, soft heaps support `insert`, `find-min`, `delete-min` and `meld` all in amortized $\mathcal{O}(1)$ time, at the cost of an accuracy sacrifice due to corruption. We survey the implementation by Kaplan, Tarjan and Zwick [6], which is based on binary heaps, rather than binomial heaps as originally described by Chazelle [2].

B.1 Overview

Like Fibonacci heaps, soft heaps are based on a collection of trees. Each node is associated with a *key* satisfying the heap invariant; moreover, we maintain that the left child of any node has key no greater than the right child. If a node has only one child, then that child is its left child.

Unlike typical heaps, soft heap nodes may store multiple items; this is crucial to decreasing the entropy of the structure and achieving a runtime faster than standard heap structures. We store this set as a linked list associated with each node.

Definition B.1.1. An element e is **corrupted** if it is stored in a node whose key is not equal to e. Its corrupted value is equal to that key.

Each node also has a *rank*, always equal to one less than the rank of its parent. The rank of a node is never changed, even if its children are destroyed.

Definition B.1.2. The **rank** of a tree is equal to the rank of its root.

Soft heaps consist of a collection of trees of distinct rank; these ranks determine which trees are merged during various heap operations to keep the trees balanced. This collection of trees is also maintained as an (ordered) linked list. To support $\mathcal{O}(1)$ `find-min` and `delete-min`, we maintain a pointer to the root of minimum key among all trees in the soft heap. We will need to define a notion called a *findable order* in order to efficiently maintain this pointer, and we maintain that the soft heap is always in findable order. Why this is important shall become evident later.

Definition B.1.3. An ordered collection of trees is in **findable order** if the first tree T is the one with minimum root key, and it is followed by all trees of rank less than T in increasing order of rank, followed by the remaining trees in findable order.

B.2 Implementation of soft heap operations

Below we provide a high-level overview of `insert`, `find-min` and `delete-min`, along with necessary auxiliary functions. We omit `meld` as it is not required in our algorithm; however, it can be performed very similarly to `insert`. The full details of these operations can be found in [6].

Algorithm B.2.1 (`find-min`)**.** As soft heaps are always in findable order, return any element from the set associated with root of the first tree.

Typically, binary heaps handle element deletion by emptying the root, and recursively filling empty nodes with their smallest child. The key insight for soft heaps is that the fill operation need not be limited to empty nodes. Filling a non-empty node will increase its key and corrupt all elements in its set. Kaplan, Tarjan and Zwick [6] demonstrate that filling nodes an extra time when they have even rank at least $t := \lceil \log_2 \frac{3}{\varepsilon} \rceil$ suffices to maintain both accuracy and runtime guarantees.

Algorithm B.2.2 (`fill`)**.** Set the target node's key to its left child, and acquire the left child's items by merging linked lists. Call `de-fill` on the left child, or delete it if it is a leaf.

Algorithm B.2.3 (`de-fill`)**.** "Double-even fill" simply delegates to `fill`. Then, if the target node's rank is even and at least t, call `fill` again.

Now, while findable order is useful for `find-min`, it is not as convenient for operations such as `insert` where tree ranks may be more important, so we define another ordering as follows.

Definition B.2.4. An ordered collection of trees is in **meldable order** if the first tree T is the one with minimal rank, followed by all remaining trees in findable order.

We observe that findable order and meldable order differ by at most swapping the first two trees. Therefore, we have trivial algorithms for converting between the two. This then allows us to define the `insert` operation, as well as `delete-min`.

Algorithm B.2.5 (`key-swap`)**.** Convert from meldable to findable order by swapping the first two trees if necessary.

Algorithm B.2.6 (`rank-swap`)**.** Convert from findable to meldable order by swapping the first two trees if necessary.

Algorithm B.2.7 (`insert`)**.** Create a rank-0 tree with the element to insert. Then `rank-swap` the heap into meldable order, execute a `meldable-insert`, and `key-swap` back into findable order.

Algorithm B.2.8 (`meldable-insert`)**.** Here we show how to add a tree T to a heap H in meldable order, provided the rank of T is at most the minimum rank in H. If the rank of T is strictly less than H, we simply `key-swap` H into findable order and prepend T. Otherwise, we link T and the first tree of H by joining them to a new empty node and invoking `de-fill`. Then `rank-swap` the remaining trees of H into meldable order and recursively apply `meldable-insert`.

Algorithm B.2.9 (`delete-min`)**.** As soft heaps are always in findable order, remove an element from the set associated with the root of the first tree. If this node still contains elements, we are done. Otherwise, call `de-fill` on this node if it has children, or else delete it. To restore the heap into findable order, suppose T is the first tree whose rank is greater than the tree from which the delete occurred. It suffices to perform a single forward pass of `rank-swap` up to T to sort trees by rank, followed by a backward pass of `key-swap` to move small keys forward.

B.3 Analysis

We defer to [6] for a rigorous analysis, and instead provide brief intuitions for proofs. Let m and d be the total number of insertions and deletions performed, respectively.

Theorem B.3.1. *There are at most εm corrupted items.*

Proof. Nodes of rank at most t are never double-filled, so they are only filled when empty. Therefore corruption occurs only at nodes of rank greater than t, so it suffices to bound the quantity of these high-rank nodes.

Now, there are at most $m/2^k$ nodes of rank k, since 2^k insertions (rank 0 nodes) are needed to create a rank k node. In addition, there are at most $2^{\lceil (k-t)/2 \rceil}$ elements at each node of rank k, as a factor of two is incurred at every second rank above t due to double fillings. Combining these facts yields the desired bound. $\square$

Next, we analyze the number of times `fill` is called. To do this, consider fills at or below rank t, or "low fillings", and fills above rank t, or "high fillings", separately.

Lemma B.3.2. *The number of low fillings is at most $td + \mathcal{O}(m)$.*

Proof. Charge each `fill` operation to the item moved by the operation. Deleted items have been moved at most t times under rank t, incurring a td charge. Also, only $\mathcal{O}(m)$ fills are charged to items currently in the heap, since items at or below rank t have only been filled a small number of times, and there are few items above rank t. □

Lemma B.3.3. *The number of high fillings is at most $3m$.*

Proof. Consider developing a recurrence for the maximum possible number of fillings $f(k)$ at a node of rank k. The key idea is that for odd k or $k \leq t$, each fill triggers one child fill, so $f(k) \leq 2f(k-1)$ because each node has two children. Likewise, for even $k > t$, $f(k) \leq f(k-1)$ because each fill triggers two child fills. This gives $f(k) \leq 2^{\lceil (k+t)/2 \rceil}$, which gives the desired bound when combined with the previously mentioned fact that there are only $m/2^k$ nodes of rank k. □

We are now prepared to analyze the runtime of the five heap operations. Note that by definition `find-min` is worst-case $\mathcal{O}(1)$, so it remains to prove the bounds for `insert` and `delete-min`.

Lemma B.3.4. *The time spent doing fillings is $\mathcal{O}(td + m)$.*

Proof. Excluding recursive calls, the time spent in a `fill` operation is $\mathcal{O}(1)$, so this follows directly from Lemmas B.3.2 and B.3.3. □

Lemma B.3.5. *Excluding filling, the time doing deletions is $\mathcal{O}(td + m)$.*

Proof. When the first root is not emptied, deletion is $\mathcal{O}(1)$ worst-case. Otherwise, it requires $\mathcal{O}(k+1)$ time excluding filling, where k is the rank of the first root, due to calls to `rank-swap` and `key-swap` in the reordering phase. For each deletion, suppose we charge $t+1$ time to the deletion. Then, for root-emptying deletions where $k > t$, there is $k-t$ uncharged time. Note that the number of root-emptying deletions at a node is bounded by the number of fillings at that node. Using the $f(k)$ from Lemma B.3.3 and the $m/2^k$ bound shows that this uncharged time totals $\mathcal{O}(m)$. □

Lemma B.3.6. *Excluding filling, the time doing insertions is amortized $\mathcal{O}(1)$.*

Proof. Define a heap potential function equal to the number of roots plus the maximum rank of a tree in the heap. Each recursive iteration of `meldable-insert` reduces the number of trees, so the operation amortizes to $\mathcal{O}(1)$. □

Theorem B.3.7. *Soft heaps implement deletion in amortized $\mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$ and all other operations in amortized $\mathcal{O}(1)$.*

Proof. Combining Lemmas B.3.4, B.3.5, and B.3.6, and recalling that $t = \mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$ suffices to prove this theorem. □

Corollary B.3.7.1. *For constant ε, soft heaps implement all operations in amortized $\mathcal{O}(1)$.*