Soft Heaps: Explanation and Experimentation

Omer Gul Suvir Mirchandani Charlotte Peale Lucia Zheng
CS166: Data Structures, Spring 2020

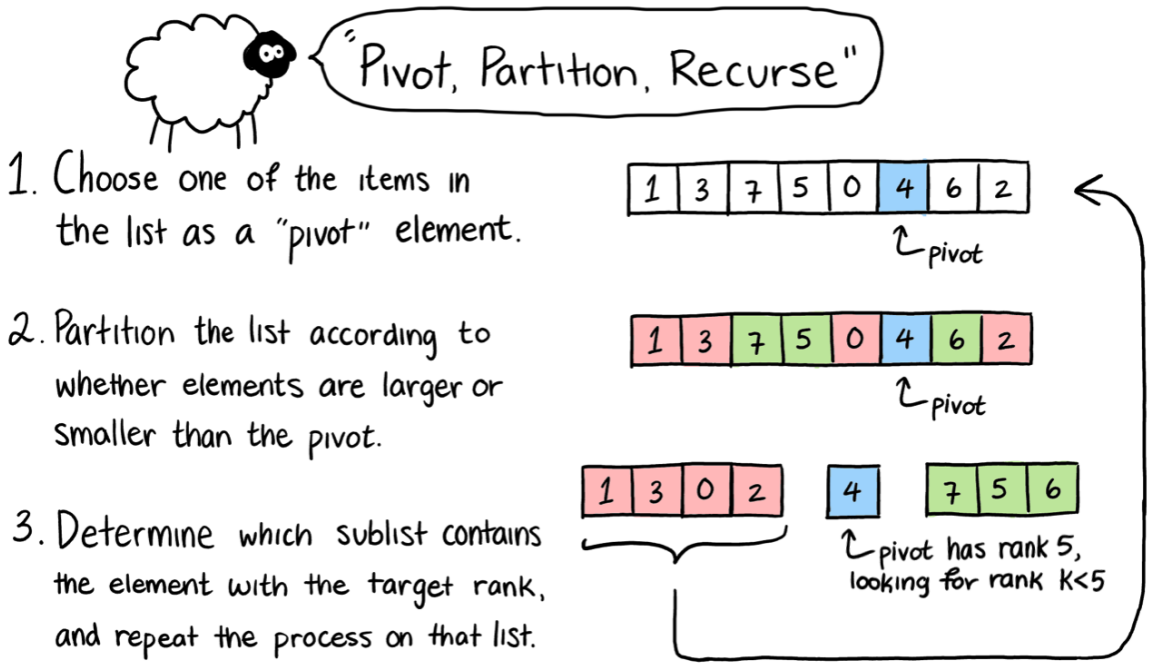
Accompanying Interactive Visualization at http://bit.ly/soft-heaps

1 A Review of the Selection Problem

A *selection algorithm* is a type of algorithm that takes in a list of unsorted elements and some parameter k, and returns the kth smallest element from that list. Such algorithms are incredibly useful in computer science because a fast implementation allows us to find the minimum, median, maximum, or any other arbitrary ranked value from a set of elements.

It is clear that the runtime of any deterministic selection algorithm on a list of n elements must be lower bounded by $\Omega(n)$ because we must see each element at least once before concluding that a particular element is the kth smallest in the list. There are a number of approaches that meet this bound, and the most common strategy of achieving this bound can be summed up by the phrase *"pivot, partition, recurse."*

What does this mean? Let's break down each step:



We note that partitioning can be achieved by a single pass over the list which takes $O(n)$ time, so this approach has the recurrence relation

$$T(n) = P(n) + O(n) + T(\max(n-p, p)),$$

where p is the rank of the chosen pivot and $P(n)$ is the time it takes to choose the pivot from a list of n elements. From this recurrence, it becomes clear that our choice of pivot can greatly affect the runtime of a selection algorithm that uses this strategy.

In the best possible case, we would choose the median every time, giving us a runtime of

$$T(n) = T(n/2) + O(n) + P(n) = O(n) + \sum_{i=0}^{\log n} p(n/2^i)$$

which simplifies to linear time assuming that the time it took to choose the pivot at each step was at most linear. However, we could also imagine a scenario in which we choose the pivot as the minimum or maximum element in the list every time, which would instead give a runtime of

$$T(n) = T(n-1) + O(n) + P(n) = O(n^2) + \sum_{i=1}^{n} P(n).$$

If we choose pivots badly, there's the potential to increase our runtime from $O(n)$ to $O(n^2)$, which is not ideal.

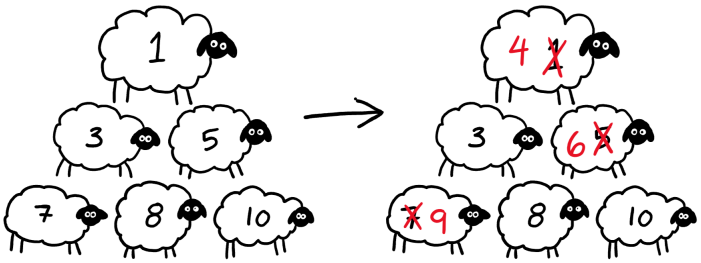
Because this pivot choice is so critical, many of the existing selection algorithms are differentiated by how they approach choosing this pivot so as to minimize the overall runtime. Some choose the pivot randomly, or by partially sorting the list, or by applying clever tricks to guarantee that the pivot is close to the median.

In this article, we'll be exploring an alternate approach that uses a variant of the binomial heap known as the *soft heap* to pick the pivot. Soft heaps were originally created by Chazelle [1] for the purpose of designing a fast deterministic minimum spanning tree algorithm. Chazelle was successful and was able to use soft heaps to design a linear time deterministic algorithm, while the best-known linear time algorithm that had previously existed was randomized. In addition to minimum spanning tree applications, soft heaps can also be applied in algorithms for selection, percentile maintenance, and approximate sorting.

We will explore the soft heap selection algorithm given by Chazelle and then investigate how a simplified version of soft heaps from [2] works behind the scenes. Finally, we will consider some ways we might be able to leverage the structure of soft heaps to speed up Chazelle's original selection algorithm.

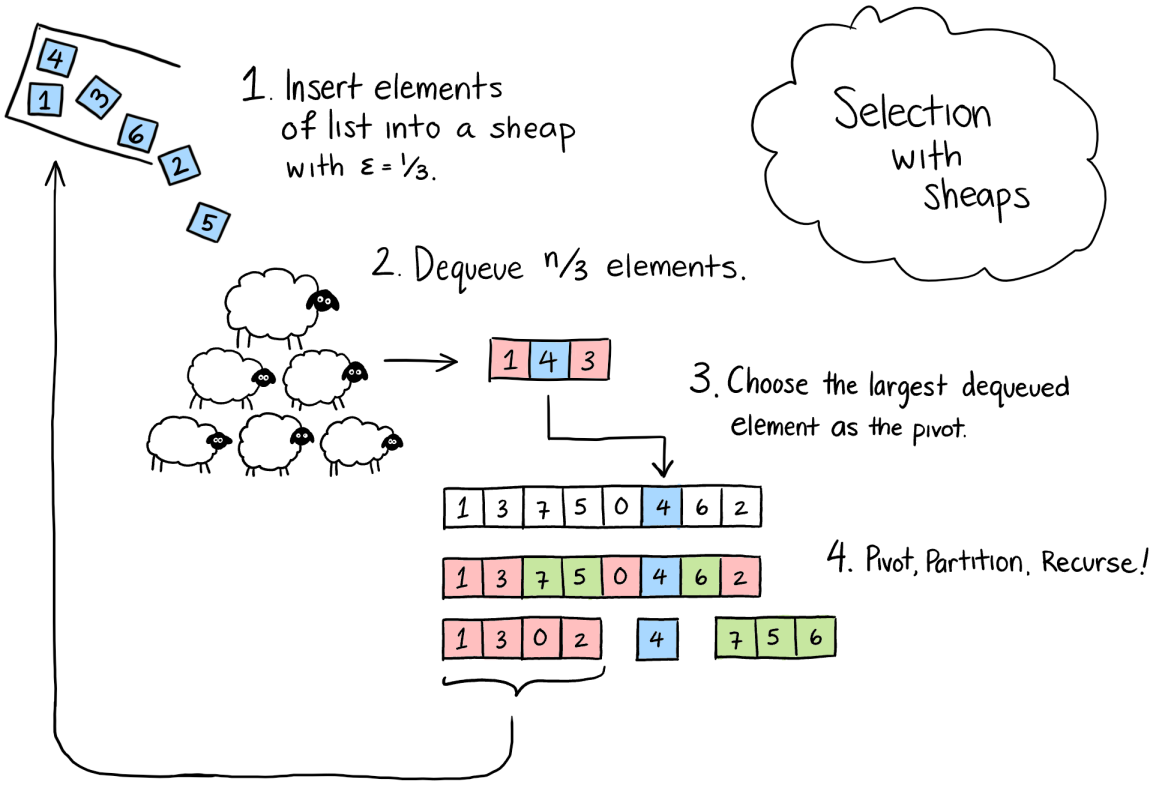
2 Linear Time Selection via Soft Heaps

In terms of functionality, soft heaps (or as we affectionately refer to them, "sheaps") act like normal priority queues—but in order to speed up the heap operations, they trade accuracy for speed and function. During certain operations, elements in the heap will get "corrupted," ending up with a higher (lower priority) key than they were originally inserted with.



In a sheap, some elements may become corrupted.

The corruption parameter ϵ, between 0 and 1, defines the maximum proportion of the n elements that may be corrupted in a soft heap at any given time. Chazelle gives the following approach for a linear time selection algorithm that returns the kth element from a list of n numbers using soft heaps:



Why does this approach to selection work? The key idea is that because of the nature of heap corruption, after we insert n items into the list, there will be at most $n/3$ corrupted elements in the heap with a higher key than what they should really have at any one time.

Therefore, we know with 100% certainty that the largest dequeued element m is larger than $n/3-1$ of the elements in the list because we saw the values of the $n/3-1$ smaller elements

after they were removed. We also know with 100% certainty that m is smaller than all of the remaining *uncorrupted* elements in the heap. There are $2n/3$ total remaining elements, $n/3$ of which could be corrupted, and so we know that the rank of m is between $n/3$ and $2n/3$.

Partitioning the list into a set of elements greater than m and a set of elements less than m allows us to calculate the true rank of m in linear time, and then recurse on whichever of the partitioned lists will contain the sought-after value.

Due to the guarantee that the rank of m is between $n/3$ and $2n/3$, we will eliminate at least $n/3$ elements with each recursive call. The soft heap allows us to perform a `delete-min` operation in amortized constant time. This gives us the recursive relation

$$T(n) \leq T(2n/3) + O(n)$$

which simplifies to an overall $O(n)$ amortized runtime.

3 How Soft Heaps Work

The soft heap supports the following operations: `insert`, `delete-min`, `meld`, and `find-min`. We will be discussing the "simplified" version given by [2]; the performance difference is summarized by Table 1.

	Lazy Binomial Heap	Soft Heap [1]	Soft Heap Simplified [2]
`insert`	$O(1)$	$O(\log \frac{1}{\epsilon})^*$	$O(1)^*$
`meld`	$O(1)$	$O(1)^*$	$O(1)*$
`delete-min`	$O(\log n)^*$	$O(1)^*$	$O(\log \frac{1}{\epsilon})^*$
`find-min`	$O(1)$	$O(1)$	$O(1)$

Table 1: Runtimes of the lazy binomial heap, the soft heap, and the simplified version of the soft heap. * indicates amortized runtime.

At a high level, a soft heap is a collection of *binary trees*. Each binary tree is heap-ordered (so a child's key is greater than or equal to its parent's key).

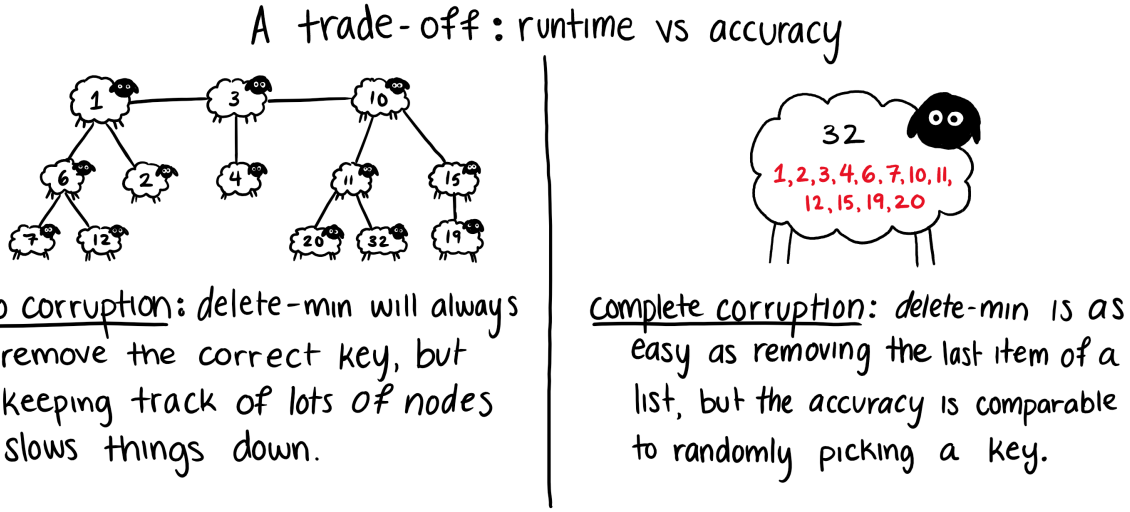
By storing the trees in a particular order, we can make it fast to `find-min`: we keep the tree containing the root of minimum key—call this root M—first in the list. We'll discuss the order we need in more detail shortly, but maintaining it also allows us to `meld` soft heaps together quickly.

We sometimes allow nodes to store items that originally had a larger key. This *corrupts* items of the larger key by associating them with the smaller key. This means, in particular, that M might contain more items than the one that actually has the smallest key. To keep track of all this, it's useful to associate keys with *nodes* in addition to items.

We begin to see the following tradeoff: we know we can `find-min` quickly, but we might end up returning a corrupted item from M because it can contain items from multiple keys.

Further, we can `delete-min` quickly by simply removing an item from M, but we may end up removing a corrupted item from M rather than the true minimum.

Take this to the extreme. If we allowed M to contain all items, we could find and delete "minima" very quickly, but our actions would be wrong more often than not because no information about the sorted order of the items is maintained.



Instead of doing this, we bound how many corrupted items the heap can hold at a given time using the parameter ϵ. ϵ controls how much information about the sorted order of the items in maintained by the soft heap, where low values of ϵ maintain more information about sorted ordering by corrupting less items and high values of ϵ maintain less information about sorted ordering by corrupting more items (allowing for faster deletions).

In a less extreme case, where M holds just a few items, we can still `delete-min` quickly and maintain our order constraints—as long as M has more items. If it runs out, we must perform more work to find the new minimum key and to maintain the order properties.

We will build up to the design of the soft heap by now presenting a few definitions.

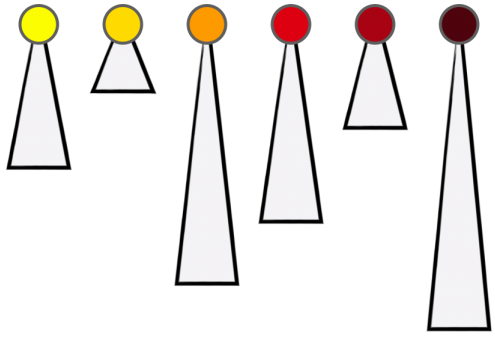
rank: held by every node such that a parent's rank is 1 plus its child's rank. As we will see, no two roots in a soft heap will have the same rank. A tree whose root node has a high rank will be large. Call the tree containing the root of minimum key M. Call the tree with minimum rank R.

findable order: place tree M first; then all the trees who have lesser rank, in order of increasing rank; and then the rest of the trees in findable order.

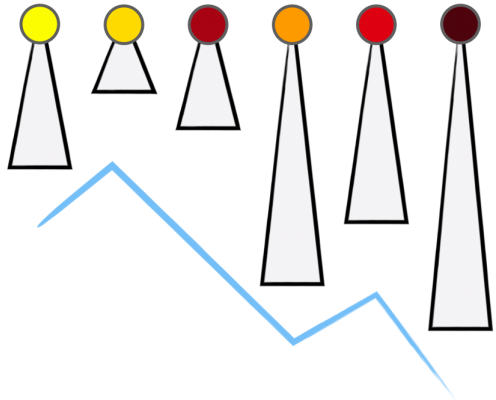
meldable order: place R first, then the rest of the trees in findable order.

Findable order will make find operations fast and meldable order will make meld operations (insert, meld) fast. To make sense of this, let us imagine that we have a collection H of seven trees below. Here, they are in sorted order by key (where the color of the root node corresponds

to its key — yellow corresponds to smallest key, dark red corresponds to largest key). The height of each triangle denotes the rank of the root node.



Let's see what these trees would look like in findable order:



The minimum key (yellow) comes first: then all the trees who have a smaller rank, in ascending order. Then comes the orange key (who is the smallest of the remaining keys), and so on. Note that if we trace the bottoms of the trees, we get a downward zig-zag where the first stroke is upward – we'll call this the $\wedge$ shape.

Now, let's swap the second tree (minimum rank) with the first tree (minimum key) as follows:

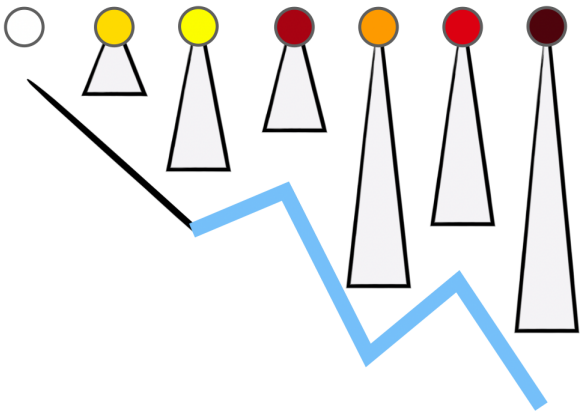
Tracing the bottoms of trees again, we see the $\wedge$ shape again, beginning at the second tree. Thus, nodes from the second node onward are in findable order — and so the overall collection is in meldable order. More generally,

> We can convert trees between findable and meldable order by swapping the first two trees if the second root has a smaller rank (findable $\rightarrow$ meldable) or if the second root has a smaller key (meldable $\rightarrow$ findable).

We'll now discuss why findable and meldable orders make find and meld operations easy. Recall that we defined findable order such that M is in front. We can define our `find-min` operation as follows:

> Assuming the collection of trees is in findable order, return the first item in M.

Now, let's consider the `insert` operation. Say H is in meldable order. If we make a new node E (below, a white node) that forms its own tree, and place it at the front of H, as below...



and note that we are still in meldable order!

To address a special case, say our mustard node had rank 0, like E. In this case, we could temporarily remove the mustard node and link it with E (to make a new node E' of rank 1). We could then make the remaining H into meldable order (which we can do because it's currently findable) and then then insert E'.

More generally, to `insert`(E),

> Assuming H is in findable order, make it meldable. If E has a lesser rank than the first root in H, simply prepend H with E. If E has the same rank as the first root in H, remove that first root, link it with E to make E', and call `insert`(E').

As we might expect, `meld` is a generalization of the `insert` operation for two heaps H_1 and H_2; that is, the functionality is the same as the standard `meld` operation for binomial heaps, but the meldable order makes the implementation efficient.

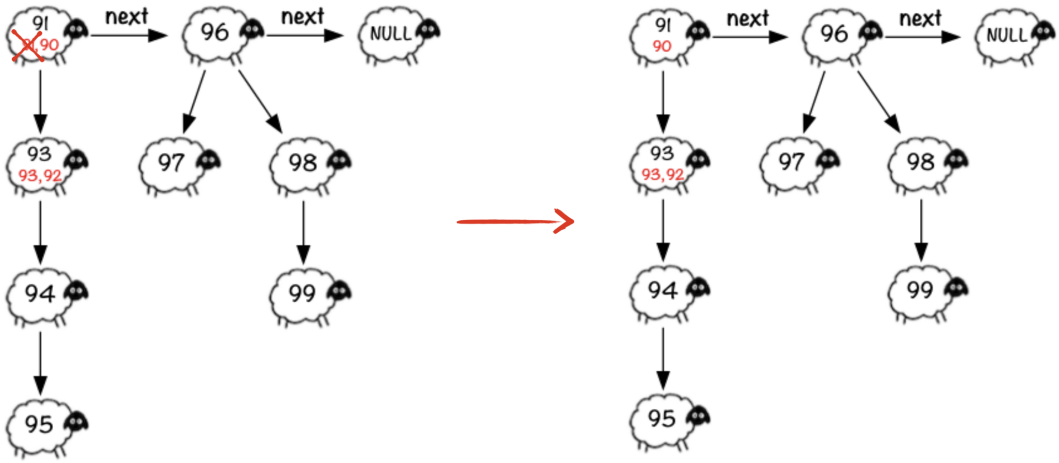
We will now turn our discussion to `delete-min`. In a normal binary heap, we would perform `delete-min` by first deleting the key and value held at the root node. Our goal

after this would be to restructure the binary heap by *filling in* the root node. If the root has no children, then we simply delete the node. Otherwise, the root would absorb the key and value of its child of minimum key. We would then recursively fill in the child of minimum key, terminating when we reach a leaf node.

As we mentioned, soft heaps can store more than one item in a node, so let's define a generalized `fill` operation:

> If the node has no items, absorb the key and item(s) held by the child of its smaller key. If the node has items, absorb the key and *append* the item(s) held by the child of smaller key. Then recurse on the child of smaller key.

If we call `delete-min` and locate M and it has multiple items, we can simply delete one, which can run in $O(1)$ time; we don't need to fill. We can see this in the figure below. (Note that for nodes that have only one item equal to the node's key, we only write the number once for simplicity.)



Now consider the other case when `delete-min` empties the root, which will happen when we perform another `delete-min` in the heap above. We will need the `fill` operation. This is the procedure that induces corruption: we fill *twice* at particular levels of recursion.

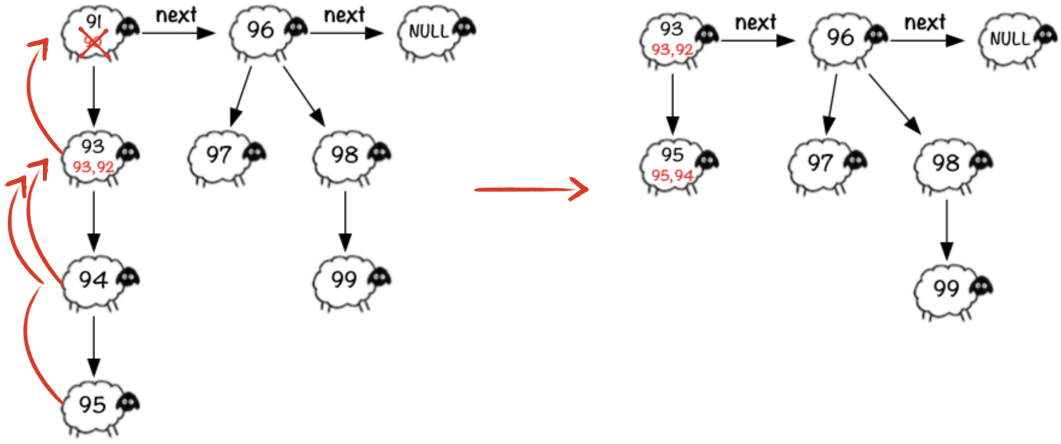
Assume we call `fill` twice at a given point. At the first call, we will place the contents of the child of minimum key inside the empty node and recurse to fill up the child. With the child contents filled up after the first call, we can *append* the contents of the new child of minimum key to our node and update the key to be key of the child during our second call. We then fix the subtree as before.

This double fill operation can cause the keys in a node to be corrupted, since the key of a child can be larger than the original keys of items already inside the node when the second call to `fill` is made.

When exactly do we call `fill` twice rather than once? If we did this everywhere, then a majority of the keys in our tree will be corrupted, making the soft heap near unusable due to a fatal lack of accuracy. Rather, we do this on evenly-ranked layers above a certain threshold t. Intuitively, this means that only nodes of rank greater than t can be corrupted. The even

condition limits how much we branch in the recursion and so we limit the amount of work a single `delete-min` may do.

The example below demonstrates this double fill causing corruption: we fill the node with key 91 up with the node with key 93, and we double fill the node with key 93 to get a node of key 95 containing 95 and 94.



As an additional detail, we use this double fill method to *link* nodes in `insert` and meld, so it is possible to cause corruption by doing a large number of insertions.

We define t to be $\lceil \lg \frac{3}{\epsilon} \rceil$. As we will show in the next section, this allows us to reach the desired accuracy level.

4 Why Soft Heaps Work

4.1 Bounding Corruption

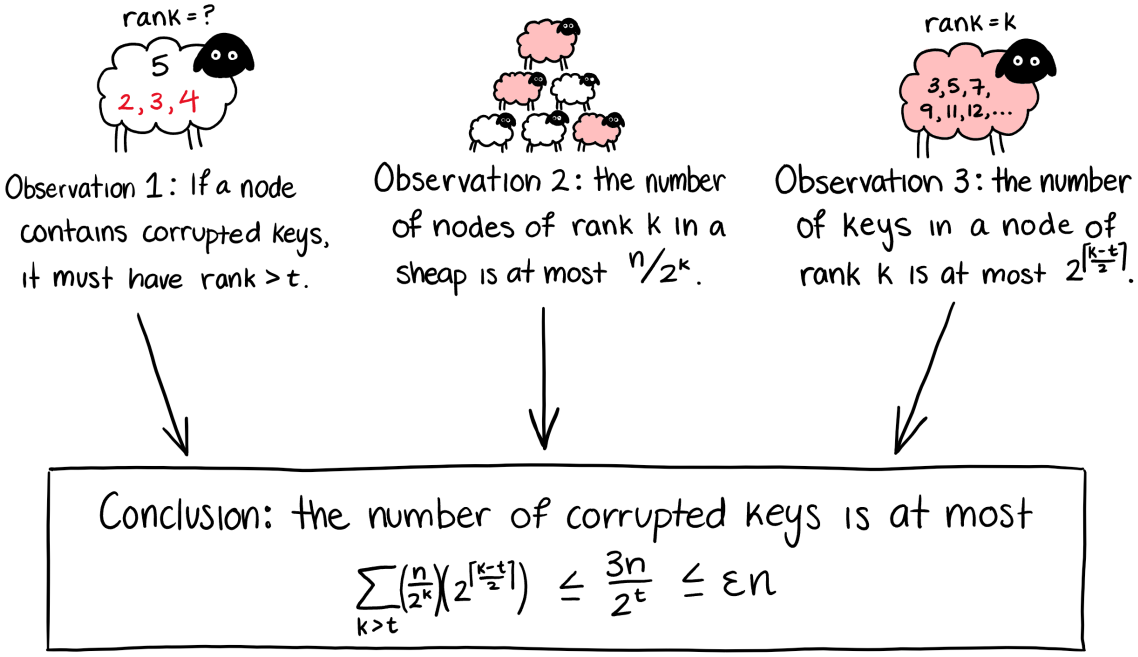
As mentioned in the previous section, we perform double filling on evenly-ranked nodes with ranks above the threshold value t. Not only does this restrict corruption to nodes with ranks higher than t, it also allows the rate of corruption to be overwhelmed by the rate in which tree sizes grow.

Let's focus on what happens when we perform double filling on a node of rank $k > t$. Either k or $k-1$ will be even, which means that we will call `fill` twice on either the node of rank k or its child of minimum key. This in turn means that we will call `fill` twice on two, potentially identical, descendants of rank $k-2$. However, our node of rank k has four descendants of rank $k-2$. If we move below two more layers, we will find that we call `fill` at most four times on descendant nodes of rank $k-4$, while there are at most 16 of them. The rate at which layer sizes grow is twice the rate at which the number of times we call `fill` on a layer grows. Asymptotically, the layer growth will overwhelm the growth of the number of fills, limiting the amount of corruption.

For an application like the selection problem, however, simply stating that the amount of corruption won't be too high isn't enough for us to trust soft heaps as a method. What we

need are concrete guarantees on how many keys we corrupt. The guarantee that soft heaps provide is that after a sequence of operations that contains n insertions,the number of corrupted elements in the tree is upper bounded by ϵn, where ϵ is the same error parameter that we used to define t.

Why is this the case? As we are using Kaplan et al.'s simplified version of the soft heap [2], we will follow their proof of this upper bound. To start, we know that if a key is corrupted, then it must reside in a node of rank greater than t. To get the upper bound, we will then ask two questions: How many nodes of rank greater than t are there and at most how many keys can those nodes contain? The answer to the latter question will give us our upper bound.



Let's start with the first question and look at the number of nodes, not just of some rank k greater than the threshold t, but any arbitrary rank $k \geq 0$. We claim that the number of nodes of rank k is at most $n/2^k$. We will show this using induction. Our base case involves nodes of rank 0. Since we have performed n insertions total, there can be at most n nodes, and therefore, $n = n/2^0$ nodes of rank 0. Since the base case holds, let's assume that the statement is true for some arbitrary $k \geq 0$. The key insight is that when we perform a *link* to combine two trees of rank k during an insertion, we initialize an empty root node of rank $k+1$, assign the two nodes of rank k as children and then perform double even filling on the root node. In other words, any node of rank $k+1$ was constructed using 2 nodes of rank k. Therefore, the number of nodes of rank $k+1$ is at most half the number of nodes of rank k, which, using our inductive hypothesis, is equal to $n/2^{k+1}$. Since the base case was also correct, this statement is true for all k.

We have obtained an upper bound on the number of nodes of rank k, which we can use to obtain an upper bound on the number of nodes of rank greater than t. All that remains for us to do is to quantify how many keys these nodes contain. Let's use $s(k)$ to denote the maximum

number of keys a node of rank k can contain. Since we can't corrupt nodes of rank $k \leq t$, $s(k) \leq 1$ for such nodes. Let's then direct our attention to nodes of rank $k > t$. As discussed before, when we call `fill` on a node of rank $k > t$, we call `fill` twice on descendants of rank $k-2$, which means that the number of elements we can populate a node of rank k with is bounded by twice the number of elements we can populate a node of $k-2$ with. In other words, $s(k) \leq 2s(k-2)$. We can translate this to an explicit numeric value. Consider any $i \geq 1$. Then, we can show that

$$s(t+2i), s(t+2i-1) \leq 2^i$$

using induction, combining the insights that $s(k) \leq 1$ for $k \leq t$ and $s(k) \leq 2s(k-2)$ for $k > t$.

We now have all machinery needed to show the upper bound! We can start working with the initial upper bound of

$$\sum_{k>t} \frac{n}{2^k} s(k).$$

To make the upper bound looser, we can make this into an infinite sum, obtaining:

$$\sum_{k>t} \frac{n}{2^k} s(k) \leq \sum_{i=1}^{\infty} \left(\frac{n}{2^{t+2i}} + \frac{n}{2^{t+2i-1}}\right) 2^i = \frac{n}{2^t} \sum_{i=1}^{\infty} \left(\frac{1}{2^{2i}} + \frac{1}{2^{2i-1}}\right) 2^i = \frac{n}{2^t} \sum_{i=1}^{\infty} \frac{3}{2^i}.$$

Since the infinite sum is a geometric series with $r < 1$, we find that

$$\sum_{k>t} \frac{n}{2^k} s(k) \leq \frac{3n}{2^t} \leq \frac{3n}{2^{\log 3/\epsilon}} = \epsilon n$$

which is the desired upper bound!

To see an example, the following figure shows a soft heap in which the first 256 natural numbers were inserted, followed by 128 `delete-mins`. ϵ is set to 0.5, so t is $\lceil \lg \frac{3}{\epsilon} \rceil = 3$. Note that none of the nodes with rank 3 or below are corrupted. Note also that less than half of the items are corrupted.

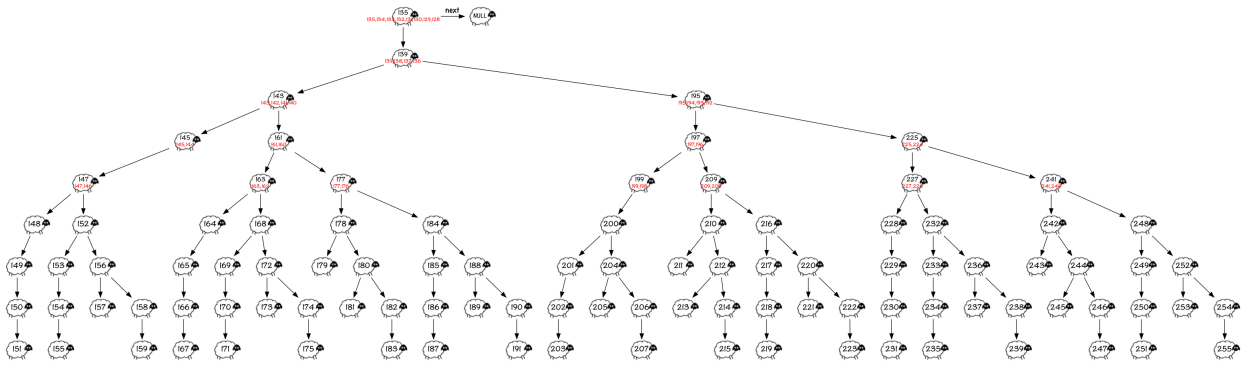


Figure 1: An example of a soft heap with $\epsilon = 0.5$.

4.2 Runtime Analysis of Soft Heaps

The upper bound that we just obtained gives us confidence that the soft heap will work correctly. What it does not tell us is whether it will work *quickly*. Since we're considering soft heaps as an alternate solution to the selection problem, we want them to be as fast as possible in addition to being correct. As such, we must provide runtime bounds to the soft heap operations.

The key strategy that we'll employ is *amortization*. Normally, when we perform runtime analysis, we look at the worst case performance of a given operation, independent of what operations came before. In some data structures, however, expensive operations are always preceded by a long sequence of very cheap operations, allowing us to backcharge the cost of the expensive operations to the cheap operations. In so doing, we reduce the "average" cost of performing operations. With this in mind, we can pretend that each operation o takes $a(o)$ time, such that for any sequence $o_1, \ldots, o_n$ of operations, $\sum_{i=1}^{n} a(o_i)$ takes at least as much time as the true total runtime of the operations. We call $a(o)$ the amortized runtime of operation o. This is a fitting framework for soft heaps, since deletions either empty a node and cause a cascade of fill operations or take $O(1)$ time, making use of the large number of keys held in a corrupted node.

From Table 1, we saw that all operations take $O(1)$ amortized time apart from `delete-min`, which takes $O(\log \frac{1}{\epsilon})$ amortized time. This is what allows the selection algorithm to work so fast. By setting ϵ to be a constant, we make each individual `delete-min` be $O(1)$ amortized time, which, in turn, makes the partition selection process take $O(n)$ time.

To derive amortized runtimes for each operation, we will have to reason about how many times the `fill` operation is called. This is an intricate process, so let's start with a simpler analysis and show that `meld` and `insert` take $O(1)$ amortized time without the fillings.

4.2.1 Runtime of `meld` and `insert`

We will use what is called the potential method, which is a way of formalizing the intuition of backcharging expensive operations to preceding cheap operations. We will define a potential function acting on the entire data structure outputting a non-negative real value. If an operation causes the potential to increase, we can imagine the data structure getting charged up, with the change in potential added to the runtime. If an operation causes the potential to decrease, the data structure loses charge. The change in potential is then subtracted from the runtime. The former corresponds to cheap operations while the latter corresponds to expensive operations.

Let's then define the potential of a soft heap as the number of trees plus the maximum node rank. Assume that we want to meld two soft heaps H_1 and H_2 which contain h_1 and h_2 trees and have maximal rank r_1 and $r_2 \geq r_1$ respectively. We can first focus on the change in potential. The heaps initially have potentials of $h_1 + r_1$ and $h_2 + r_2$ respectively. If we perform k links during the meld process, the total number of trees decreases by k. As a result of the linking, the maximal rank of the melded heap can be at most $r_2 + 1$. The change in potential

then is

$$(h_1 + h_2 - k + r_2 + 1) - (h_1 + r_1) - (h_2 + r_2) = 1 - k - r_1.$$

The `meld` operation itself takes $O(r_1)$ time to insert the heaps contained in H_1 and $O(k)$ time to perform k links without the fillings. The amortized time of meld then becomes

$$O(r_1 + k) + O(1) \cdot (1 - r_1 - k) = O(1).$$

Since `insert` was a specific case of `meld`, it too takes $O(1)$ amortized time.

4.2.2 Runtime of `delete-min`

We now direct our attention to `delete-min`. Since a `delete-min` operation cannot increase the number of trees or the maximal rank, it can only decrease the potential. For this reason, we will ignore the potential that we just defined for `meld` and `insert` and focus instead on the number of times `fill` is called. In so doing, we can provide upper bounds on how much time we spend doing fillings and how much time we spend on deletion outside of `fill` operations. Combining these, we can get our worst case and then amortized runtimes for deletion. In these analyses, we will assume that we have performed d deletion and n insertion operations.

We will start by showing that the total number of `fill` operations is $O(td+n)$ by analyzing two separate cases. We call a `fill` operation a "low" filling if it is performed on a node of rank $k \leq t$ and a "high" filling if it is performed on a node of rank $k > t$. We will provide upper bounds to both types of fillings and combine these to get the overall bound.

Let's then focus first on low fillings. We claim that the total number of low fillings is $O(td + m)$. To show this, we will make use of a trick similar to the potential function we used in our amortized analysis of `meld` and `insert`. We can imagine each item stored in a node having some charge and assume that this charge increases only when the item is picked up in a low filling. The charge of a given item then represents how many low fillings that item was involved in. If we quantify the total charge of a soft heap, we quantify the total number of low fillings. Since a low filling can only be performed on nodes of rank at most t and since an item can only increase ranks, the total charge of any item is at most t. Then, we have two possibilities: an item is either deleted or it is not. The total charge that comes from deleted items is at most td. For the total charge coming from items that are not deleted, we can make use of our bounds on the maximum number of items held in a node of rank k. We know that a node of rank $k \leq t$ can hold at most 1 item and that there are at most ϵn items total held in nodes of rank $k > t$. As a result, the total charge from these items is at most

$$\sum_{k=1}^{t} k\frac{n}{2^k} + t(\epsilon n) = n\sum_{k=1}^{t} \frac{k}{2^k} + \lceil \lg \frac{3}{\epsilon} \rceil \epsilon n = O(n) + O(n) = O(n).$$

Adding these two results, we find that the total number of low fillings is $O(td + n)$.

Our next step is quantifying the number of high fillings, which we claim is at most $3n$. To get this bound, we will have to reason directly about the maximum number of times double even filling can be called on a given node of rank k, which we denote $f(k)$. Since a node of

rank 0 can only be filled when it is created, $f(0) = 1$. With the base case defined, let's consider any arbitrary node x of rank $k \geq 1$. We can perform filling on x as long as its subtrees store items, since if x is a leaf node, a `fill` call will lead to its destruction. Given that x has at most two children of rank $k-1$, we can call `fill` on its children at most $2f(k-1)$ times before destroying them and making x a leaf node. As such, $f(k) \leq 2f(k-1)$. This bound is tighter when $k > t$ and k is even, since we will call `fill` twice in this case, making $f(k) \leq f(k-1)$. We can make use of these inequalities to get the following bounds: If $k \leq t$, then $f(k) \leq 2^k$ and if $k > t$, then $f(k) \leq 2^{\lceil (k+t)/2 \rceil}$. We are almost done! All we need to do is use $f(k)$ to get a bound on the total number of times double even filling is performed on all nodes of rank $k > t$. We can yet again use our upper bound on the number of nodes of a given rank to find that the number of high fillings is bounded by

$$\sum_{k>t} \frac{n}{2^k} f(k) \leq \frac{n}{2^t} \sum_{i=1}^{\infty} \left(\frac{1}{2^{2i-1}} + \frac{1}{2^{2i}}\right) 2^{t+i} = n \sum_{i=1}^{\infty} \frac{3}{2^i} = 3n$$

as desired.

In conclusion, since the number of low fillings is $O(td+n)$ and since the number of high fillings is $O(n)$, the total number of fillings is $O(td+n)$. We will make use of this fact to get worst case bounds on the runtime of `delete-min`. Firstly, since each `fill` operation, ignoring the recursive calls, takes $O(1)$ time, we spend $O(td+n)$ time performing `fill`. All that we need to ask now is how much time we spend on deletion outside of filling. Let's assume that `delete-min` removes an item from a node of rank k. If this does not empty the node, then deletion takes $O(1)$ time. Otherwise, we need to do $O(k+1)$ work to destroy the node in the event that it has no children and to restore findable order for roots of rank at most k.

We perform d deletions. In the worst case, each of these deletions empty a node of rank $k > t$. We can shave off $O(td)$ time and focus on how the remaining $O(k-t)$ time is spent on each deletion from a node of rank k. Let's assume that we empty each node of rank $k > t$ as many times as we can to get a very loose upper bound. In order to empty a node, we must first fill it. Therefore, the remaining $O(k-t)$ time from each node is at most

$$\sum_{k>t} \frac{n}{2^k} f(k)(k-t) \leq n \sum_{i=1}^{\infty} \left(\frac{2i-1}{2^{2i-1}} + \frac{2i}{2^{2i}}\right) \frac{2^{t+i}}{2^t} = O(n).$$

The total time spent on deletions ignoring filling is thus $O(td+n)$, which makes the total time spent on deletions overall $O(td+n)$.

With this upper bound, we fill in the details of the aggregate method amortized analysis briefly described in [2] to show the amortized runtime of deletion operations. Using the aggregate method, $a(\texttt{delete-min}) = \frac{T^*(c)}{c}$ where $T^*(c)$ is the maximum amount of work done by any series of c operations. Consider any arbitrary $c = d+n$, where the series of c operations consists of d deletions and n insertions and the work done is maximal for c operations. Applying the deletion operation time bound result above, and the fact that insertions are $O(1)$

amortized time, $T^*(c) = O(td+n) + O(n) = O(td+n)$. Then,

$$\begin{aligned} a(\texttt{delete-min}) &= \frac{T^*(c)}{c} \\ &= \frac{O(td+n)}{d+n} \\ &= \frac{O(t(d+n))}{d+n} \\ &= O(t) \\ &= O(\lceil \log \frac{3}{\epsilon} \rceil) \\ &= O(\log \frac{1}{\epsilon}). \end{aligned}$$

Since our choice of the series of c operations was arbitrary, this yields the amortized time bound for `delete-min` in a soft heap as $O(\log \frac{1}{\epsilon})$.

5 Improving on Linear Selection with Soft Heaps

5.1 Motivation

We'll now return to the selection problem. As discussed in Section 1, picking a good pivot is an important part of the linear time selection algorithm. In that section, we established the intuition that the best pivot choice is one that reduces the size of the list for the next recursive call as much as possible.

In Section 2, we introduced Chazelle's [1] approach for a linear time selection algorithm that uses soft heaps to choose the pivot. The rank of the pivot chosen in the manner described by Chazelle is always deterministically lower bounded by

$$\textit{number of } \texttt{delete-min} \textit{ calls} \cdot n$$

and upper bounded by

$$\textit{number of } \texttt{delete-min} \textit{ calls} \cdot n + \epsilon n,$$

where ϵ is the corruption parameter of the soft heap.

This is because the pivot is always chosen to be the maximum of the elements dequeued from the heap by `delete-min` calls, which is found by comparing the elements as they are dequeued and seen. Since a soft heap controls corruption based on the corruption parameter ϵ, it's also guaranteed that there are at most ϵn elements still in the heap which originally had a lesser key than the chosen pivot but were corrupted.

We observe that the process of choosing a pivot using a soft heap in the selection algorithm permits control over the bounds on the rank of the pivot (both in terms of what the bounds are and how tight the bounds are) in each recursive `select` call, based on the choice of the number of `delete-min` (dequeue) calls and ϵ.

Chazelle's selection algorithm sets the fraction of elements dequeued to be a constant fraction of n ($\frac{1}{3}$), and also sets ϵ for the soft heap built in each select call to be constant at $\frac{1}{3}$. Thus, the lower bound on the rank of the pivot is *number of* `delete-min` *calls* $\cdot\, n = \frac{n}{3}$ and the upper bound on the pivot is *number of* `delete-min` *calls* $\cdot\, n + \epsilon n = \frac{n}{3} + \frac{1}{3}n = \frac{2n}{3}$ for every recursive `select` call.

Since the potential for fine-grain control was not leveraged in Chazelle's version of linear selection, we wanted to explore what might happen if we made different choices about the number of `delete-min` calls and ϵ to exercise greater control over the bounds on the rank of the pivot chosen using the soft heap in each `select` call, and in particular, control the bounds in such a way as to pick a better pivot to partition around.

This idea motivates our central question:

> How can we utilize the controls over the bounds on the rank of the pivot to choose a better pivot and reduce the number of recursive `select` calls needed, in order to increase the overall efficiency of the linear time selection algorithm?

5.2 Implementation

5.2.1 Tuning ϵ

We start by looking at the ϵ corruption parameter. In Chazelle's version, $\epsilon = \frac{1}{3}$, so the upper bound on the rank of the pivot is $\frac{2n}{3}$.

What would happen if we held the fraction of elements dequeued constant, but set ϵ to be larger than $\frac{1}{3}$? This would increase the upper bound on the rank of the pivot to be greater than $\frac{2n}{3}$.

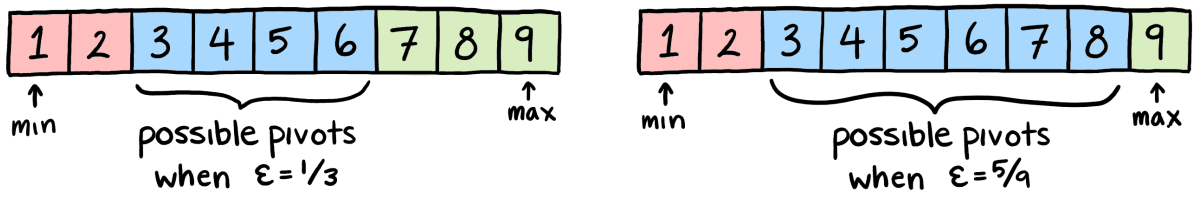
But doing so would be disastrous if the `select`(k, lst) call was attempting to select an element with rank k, where k is small relative to the size of the list. Consider the following example. Let $lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]$. Say we are trying to find the rank 1 element, element 1. In Chazelle's version, the pivot would be between rank $\frac{9}{3} = 3$ and rank $\frac{2 \cdot 9}{3} = 6$, elements 3 and 6. If we increased ϵ, the upper bound on the rank of the pivot would be greater, so the pivot chosen could be of higher rank, perhaps element 7 or element 8. But this would be a poorer choice than a pivot between elements 3 and 6, since it would reduce less of the list for the next recursive `select` call.

On the other hand, doing so might be good for when k is relatively large compared to the size of the list, particularly when $\frac{k}{n} > \frac{2}{3}$. Let's see this through an example using the same lst as before. Say we are trying to find the rank 8 element, element 8. If we increased ϵ, the upper bound on the rank of the pivot would be greater, so the pivot chosen could be of higher rank, perhaps element 7 or element 8. In this case, element 7 or 8 would be a better choice for pivot than elements between 3 and 6, since it would either reduce more of the list for the next recursive `select` call or find us the exact element we are looking for. But what if we increased ϵ so much that the upper bound on the rank of the pivot was element 9 and it was chosen as the pivot? This would be a worse pivot choice than an element between 3 and 6 because then the pivot only reduces the size of the list by 1 for the next recursive `select` call.

Intuitively, these toy examples tell us two things:

(1) Increasing ϵ for `select`(k, lst) call for low rank k elements can lead to worse pivots being chosen.

(2) Increasing ϵ for `select`(k, lst) call for high rank k elements can lead to better pivots being chosen, but the rank upper bound should not increase over rank k, since this can lead to worse pivots being chosen.



A toy example: increasing ε can improve performance when selecting the maximum, but makes selecting the minimum less efficient.

Our first optimized implementation of the original linear time selection algorithm leverages this insight by setting $\epsilon = \frac{1}{10}$ for `select`(k, lst) calls where the rank k is relatively small, $\frac{k}{n} < \frac{1}{3}$, and $\epsilon = \frac{k}{n} - \frac{1}{3}$ for `select`(k, lst) calls where the rank k is larger, $\frac{1}{3} \leq \frac{k}{n}$. Note that ϵ in this implementation is set to be lower than in the Chazelle version for rank k where $\frac{1}{3} < \frac{k}{n} < \frac{2}{3}$, making dequeues slower for these cases, but we hypothesized this would be counterbalanced by the gains in the other cases, which held true when the implementation was tested in practice.

5.2.2 Tuning `delete-min` calls

Next, we observed that Chazelle's original version of the selection algorithm has particularly inefficient behavior for `select`(k, lst) call for low rank k elements, where $k < \frac{n}{3}$. Specifically, `select` calls `delete-min` $\frac{n}{3}$ times to dequeue $\frac{n}{3}$ keys to find a pivot lower bounded by rank $\frac{n}{3}$ and upper bounded by rank $\frac{2n}{3}$. But since $k < \frac{n}{3}$, the call wastes time dequeue-ing elements that are larger than the element we are interested in and potentially overshoots so much that the list only partitions off the $\frac{n}{3}$ largest elements, when it could have done much less work by simply dequeue-ing around k elements by setting the number of `delete-min` calls to k and ϵ to a small value, which would cause the `select` call to either reduce the size of the list by much more or find the exact element we are interested in.

Another observation is relates to the interaction between the number of `delete-min` calls and ϵ. Recall that each `delete-min` call, the most expensive operation in a soft heap, runs in amortized $O(\log \frac{1}{\epsilon})$, so increasing the number of `delete-min` calls done in a `select` call increases the runtime of a `select` call. For the low rank $k < \frac{n}{3}$ case, a good way to deal with this would be to set ϵ to $\frac{k}{n}$. This increases ϵ as k increases, offsetting the added cost of the `delete-min` call needed to dequeue a greater k number of elements with faster `delete-min` operations by allowing for more corruption. Intuitively, this seems like a good approach because if k is very small, like $\frac{k}{n} = \frac{1}{1000}$, having a comparatively high upper bound on the rank

of the pivot can lead to a pivot that overshoots the element of interest, which is very wasteful since the actual work required to dequeue elements to find the exact element of interest is low; and if k is slightly larger, like $\frac{k}{n} = \frac{1}{4}$, it's more permissible for the pivot to overshoot the element of interest.

Our second optimized implementation of the linear time selection algorithm leverages these additional insights by setting tuning the *number of* `delete-min` *calls* (d) as follows:

Values of k	d	ϵ
$\frac{k}{n} < \frac{1}{3}$	k	$\frac{k}{n}$
$\frac{1}{3} \leq \frac{k}{n} < \frac{2}{3}$	$\frac{n}{3}$	$\frac{k}{n} - \frac{1}{3}$
$\frac{2}{3} \leq \frac{k}{n}$	$\frac{2n}{3}$	$\frac{k}{n} - \frac{2}{3}$

Table 2: Tuned values of d and ϵ for various values of k.

This implementation applies the observations discussed in this section directly for the low rank k case, where $\frac{k}{n} < \frac{1}{3}$. For the higher rank k cases, it uses the method from the first implementation for higher rank k: set the number of elements to dequeue for rank k's within a certain range to a constant fraction f of n, to set a lower bound on the rank of the pivot, and allow for corruption up to rank k by setting $\epsilon = \frac{k}{n} - f$, to set a higher bound on the rank of the pivot. The reason the higher rank range is divided into two ranges in this implementation is because we observed that in the first implementation, when ϵ was greater than $\frac{1}{3}$, the pivots chosen became too unpredictable and at times, extremely undesirable, due to the increased number of corrupted elements in the heap. By splitting the higher rank range into two ranges of size $\frac{n}{3}$, we ensure that ϵ is never greater than $\frac{1}{3}$ to ameliorate this problem.

5.2.3 Tuning the min/max heap property of the soft heap

The second optimized implementation worked quite well for `select` on low rank k's, producing significant improvement over both Chazelle's version with no tuning and the first improved implementation with tuning of ϵ. For `select` on higher rank k's however, performance reaches a plateau because the number of `delete-min` calls used in the case where $\frac{2}{3} \leq \frac{k}{n}$, $\frac{2n}{3}$, increases the runtime of the `select` call, counterbalancing the time savings from choosing better pivots.

To address this shortcoming, we thought about how the pivot choosing procedure uses a soft heap that maintains the min-heap property, where keys higher in the soft heap's trees have smaller values than those lower in the soft heap's trees. This allows for quick access to the lower rank elements of the list inserted into the soft heap, but slower access to the higher rank elements of the list inserted into the soft heap. Thinking about this property inspired the idea to uses a soft heap that maintains the max-heap property in `select`(k, lst) call for high rank k elements.

Our third optimized implementation of the linear time selection algorithm used the ideas of the second implementation as a foundation, but altered the behavior for the `select`(k, lst)

call, where rank $k > \frac{n}{2}$ (greater than median element). In that case, we use a soft heap that maintains the max-heap property, instead of the min-heap property, by negating the values of the elements in the list before inserting them into the soft heap (negating the pivot once it is chosen back to the original element value before partitioning and the original list of element for partitioning). The rest of the tuning is done similarly to the tuning from the second implementation, with symmetric settings for the `select`(k, lst) call for high rank k using max-heap soft heaps.

Specifically, for $\frac{k}{n} \leq \frac{1}{2}$, the pivot choosing procedure uses a min-heap soft heap as before, setting number of `delete-min` calls $= k$ and $\epsilon = \frac{k}{n}$ where $\frac{k}{n} < \frac{1}{3}$ and number of `delete-min` calls to be $\frac{n}{3}$ and $\epsilon = \frac{k}{n} - \frac{1}{3}$ where $\frac{1}{3} \leq \frac{k}{n}$. For $\frac{1}{2} < \frac{k}{n}$, the pivot choosing procedure uses a max-heap soft heap, setting number of `delete-min` calls $= n - k + 1$ and $\epsilon = \frac{n-k+1}{n}$ where $\frac{n-k+1}{n} < \frac{1}{3}$ and number of `delete-min` calls $= \frac{1}{3}$ and $\epsilon = \frac{n-k+1}{n} - \frac{1}{3}$ where $\frac{1}{3} \leq \frac{n-k+1}{n}$.

5.3 Testing Performance

We implemented Chazelle's version of the linear time select algorithm using soft heaps and the three versions as described above. We based our Python implementation of the soft heap on the implementation by Kaplan et al. [2].

As expected, the runtime for all versions appears linear in the size of the list, $O(n)$, when tested on lists of different sizes with random ordering permutations, for `select`(k, lst) call on all values of k. But, we see clearly from Figure 2, Figure 3, and Figure 4 that our optimized implementations, in particular the second and third implementation, performed much better than the original version by a noticeable constant factor (less steep slope), for `select`(k, lst) call on all values of k.

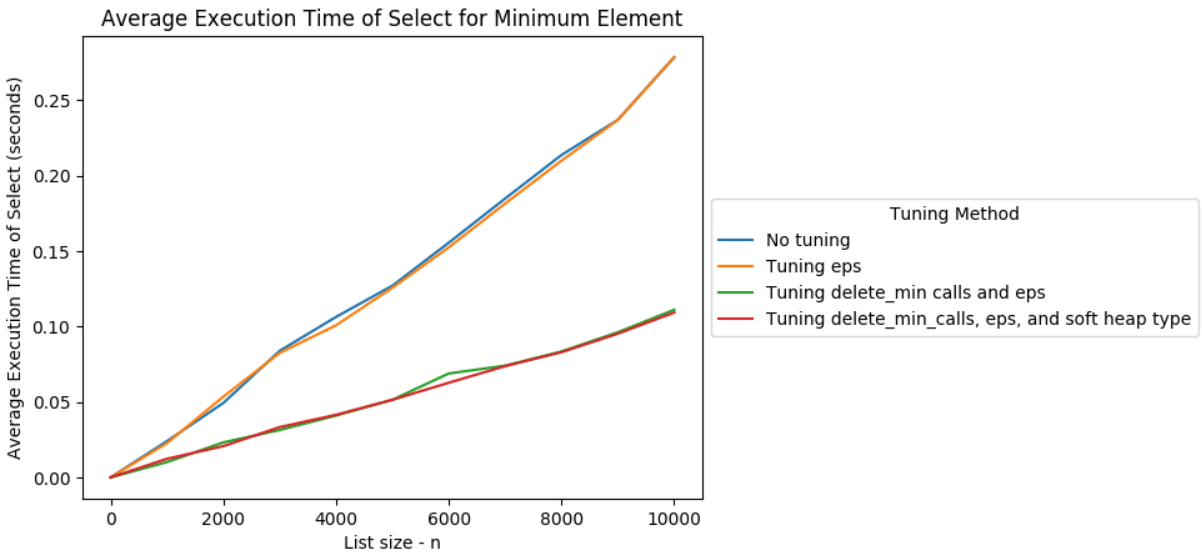


Figure 2: Average Execution Time of `select` for Minimum Element vs. List Size

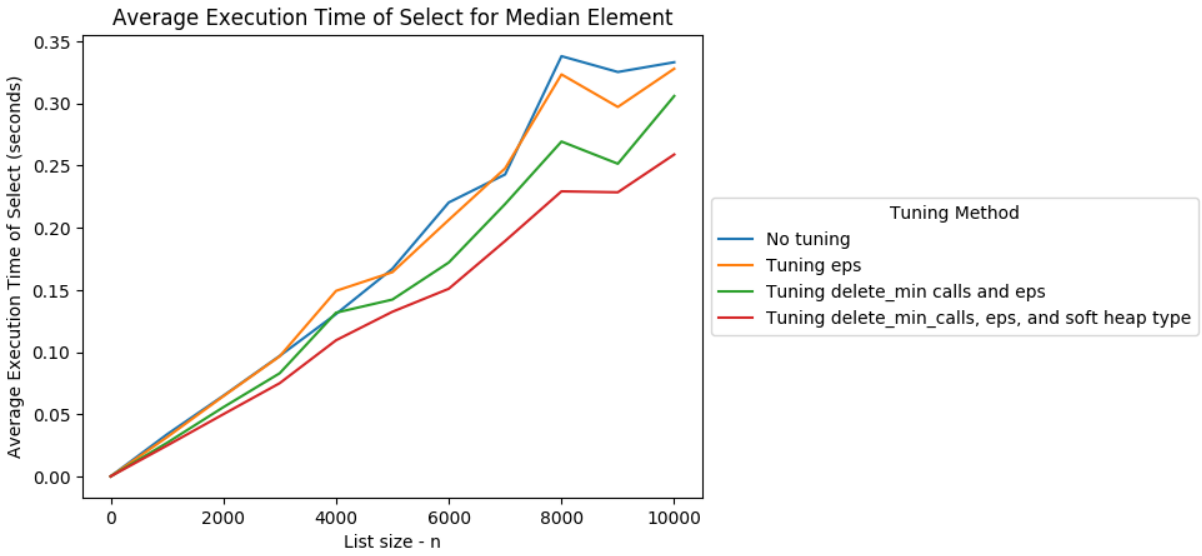


Figure 3: Average Execution Time of `select` for Median Element vs. List Size

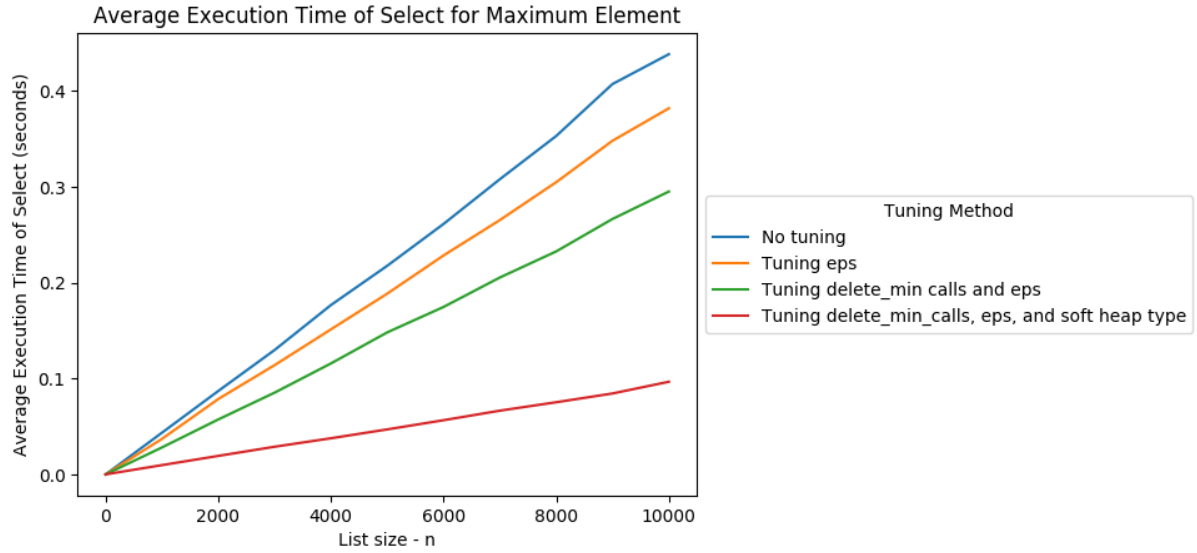


Figure 4: Average Execution Time of `select` for Maximum Element vs. List Size

We also performed some tests on `select`(k, lst) calls on one lst of size $n = 10^5$, varying k to see how the optimized versions compare to the original version for different values of k, shown in Figure 5. Here, we noticed that the optimized version that used both min-heap soft heaps and max-heap soft heaps replicated the improved runtime behavior for low rank queries in the optimized version that used only min-heap soft heaps. We saw that the selection algorithm for median rank in the optimized version had the most similar execution time to that of the original version, which was to be expected, since our optimizations focused on reducing

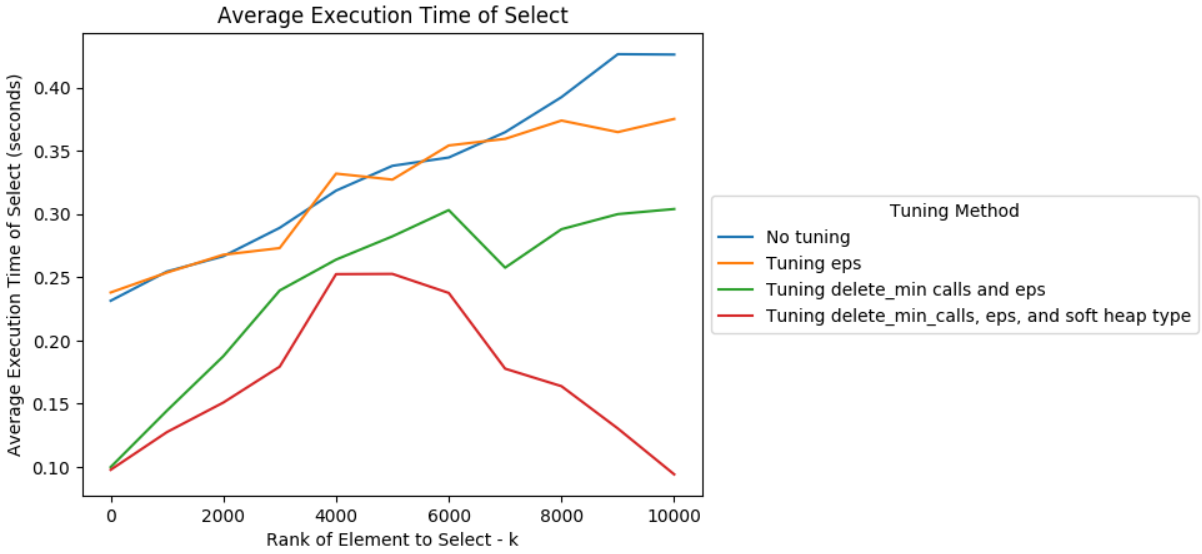


Figure 5: Average Execution Time of `select` vs. Rank of Element to Select

the inefficiencies of the original version for low and high rank queries.

Lastly, we created a visualization to highlight the reduction in recursive `select` calls in the optimized versions of the selection algorithm compared to the original version of the selection algorithm. This reduction in recursion is the key result of the better pivot choices in each `select` call and the source of the faster execution time of the selection algorithm in the optimized versions.

The visualizations are displayed in Figure 6, 7, and 8. Each visualization shows the recursive `select` call stack, with the partition of the list and the pivot chosen in each `select` call color-coded. The three visualizations correspond to a `select`(k, lst) for $k = 100$ (maximum element) on the same list of size 100 using three versions of the selection algorithm: (1) the original version, (2) the version tuning the number of `delete-min` calls and ϵ (3) the version tuning the number of `delete-min` calls, ϵ, and soft heap type (min/max heap property).

The visualization highlights how versions (2) and (3) produce time savings that result from choosing better pivots and reducing the depth of the recursive `select` call stack.

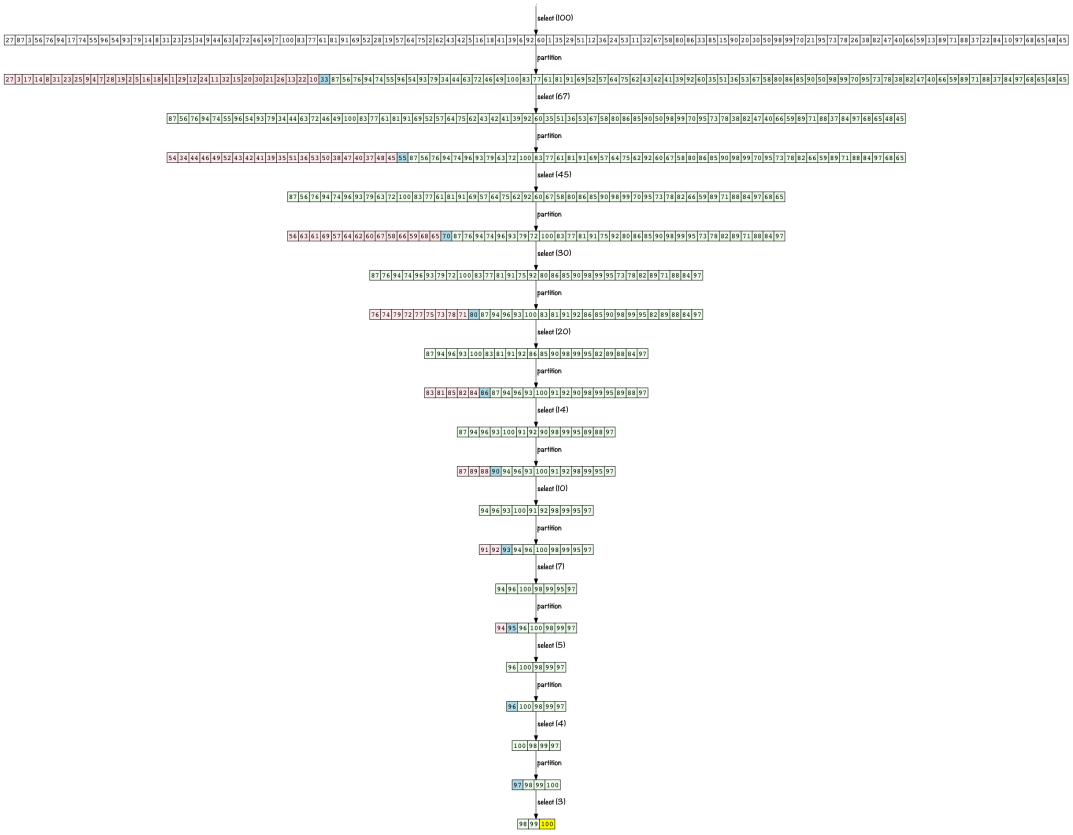


Figure 6: Original version

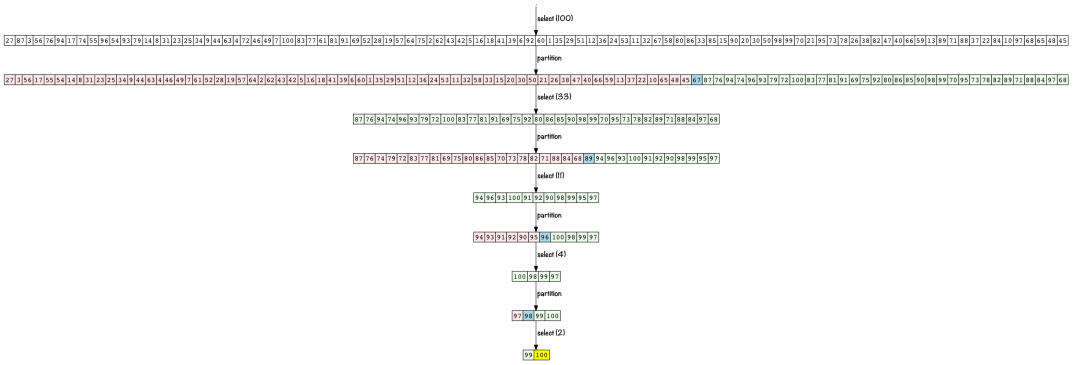


Figure 7: Optimized version tuning the number of `delete-min` calls and ϵ

Figure 8: Optimized version tuning the number of `delete-min` calls, ϵ, and soft heap type

5.4 Further Improvements for Intermediate Rank Selection

From testing the above versions of linear selection with soft heaps, we observed that intermediate rank selection had poor performance relative to low rank or high rank selection. This is because the most optimal version of the algorithm (depicted by the red line in Figure 5) focuses on increasing time savings for high rank selection, using the tuning method that we found worked well for low rank selection on a soft heap that maintains the min-heap property (depicted by the green line in Figure 5), on a soft heap that maintains the max-heap property instead. But this adaptation did not change the algorithm's behavior for intermediate rank selection much, since regardless of whether the algorithm builds a min-heap or a max-heap for an intermediate rank selection, the algorithm still performs many `delete-min` calls to get to the intermediate rank elements at the bottom of the min-heap or max-heap. In particular, when the initial selection call is for an intermediate rank element, the time spent on a relatively large number of `delete-min` calls on the soft heap in the initial `select` call overshadows later savings, leading to the parabolic shape of the red line in Figure 5.

This observation motivated us to further extend the versions of linear selection with soft heaps presented above, with a focus on improving the performance of intermediate rank selection queries. We implemented two additional versions of linear selection with soft heaps, using the best performing version from above, with tuning on `delete-min` calls, ϵ, and soft heap type (min/max heap property), as a foundation.

The first version takes inspiration from the median of medians algorithm often used to find an approximate median pivot for linear selection. The median of medians algorithm finds an approximate median pivot, instead of an exact median pivot, but still works quite well as the pivot choosing subroutine within a linear-time selection algorithm. We looked into other approximate median algorithms and found several others that use random sampling [3]. At a high level, these algorithms deterministically find an approximate median of a set of elements within certain error bounds by choosing a random subset of suitable size and finding the median of the subset as an approximation to the full set. This approach can be generalized to find an approximate k rank element as well. We used this approach to create a subroutine that finds an approximate intermediate rank element with random sampling and called this subroutine for intermediate rank `select` calls ($\frac{1}{3} \leq \frac{k}{n} \leq \frac{2n}{3}$) in this new version. (This is instead of defaulting to the behavior of the foundational version of making $\frac{n}{3}$ `delete-min` calls on a soft heap with either the min-heap or max-heap property.)

In each random sample, we sample $\frac{n}{5}$ elements from lst and make a soft heap containing only the subset. Then, $\frac{n}{15}$ `delete-min` calls reduce the subset by $\frac{1}{3}$ of its size to find an approximate intermediate rank pivot. This reduces the time bottleneck of `delete-min` calls for intermediate rank queries present in the best performing version from above without compromising the quality of the pivot by too much.

The second version takes an even lazier approach than the first version. For intermediate rank `select` calls ($\frac{1}{3} \leq \frac{k}{n} \leq \frac{2n}{3}$), it simply selects a pivot from the list of elements at random, without bothering to build a soft heap and make `delete-min` calls at all. For low rank `select` calls ($\frac{k}{n} < \frac{1}{3}$) and high rank `select` calls ($\frac{k}{n} > \frac{2n}{3}$), it defaults to the behavior of the foundational version, building a soft heap with the min-heap or max-heap property as appropriate

and tuning `delete-min` calls and ϵ. Unlike the first version described, where the approximate rank of the chosen pivot is bounded based on the size of the randomly sampled subset of elements and the ϵ corruption parameter of the soft heap used to sort the subset, in the second version, the approximate rank of the chosen pivot is unbounded and could be in worst-case, the minimum or maximum element. In that case, a `select` call for an intermediate rank query might potentially only reduce the size of the list by one. But this version proves to be incredibly fast on intermediate rank queries, since it avoids building and deleting from a soft heap at all in the `select` call for a certain range of intermediate ranks.

The results of the tests from Section 5.3 on these two additional versions of linear selection with soft heaps are displayed below, in Figure 9, 10, 11, and 12. As expected, we found that these two versions (depicted by the purple and brown line) performed better on intermediate rank selection queries than the previous best performing version (depicted by the red line). In particular, one can see that the first and second version perform visibly better than the original version and the three versions presented in Section 5.2 on intermediate rank selection queries (like median selection) and perform no worse than the other versions on low and high rank selection queries, in all of the tests.

We noticed that the second version actually outperforms the first version on average, even though it does not guarantee a rank bound on pivots chosen for intermediate rank selection queries. So, in an application where we are concerned about adversarial or pathological inputs and worst-case performance, we'd prefer the first version to the second version, but in an application where we care more about average case performance, we'd prefer the second version to the first version.

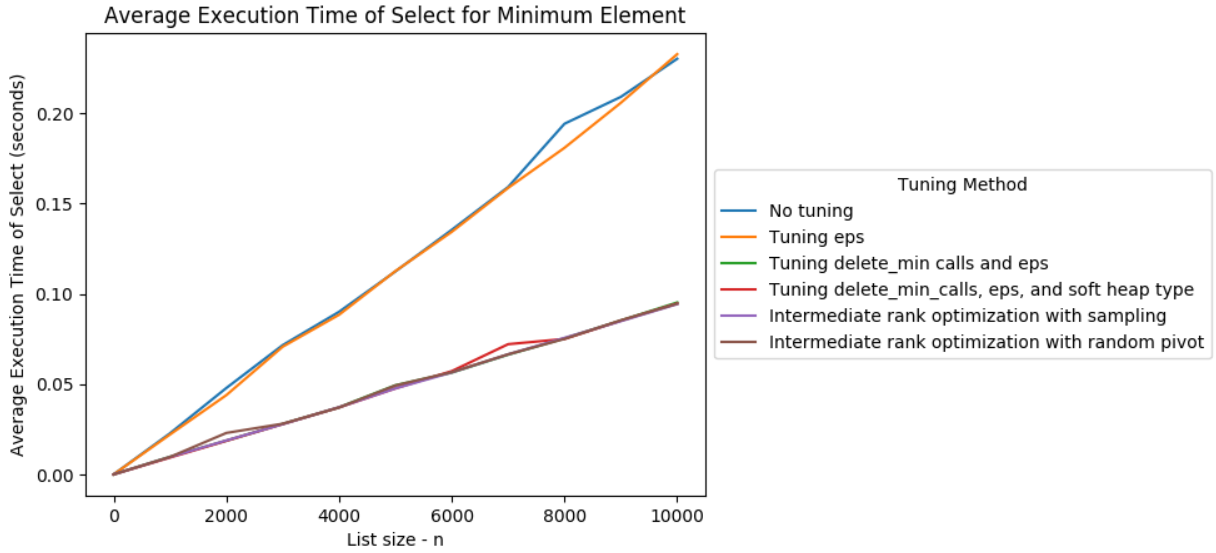


Figure 9: Average Execution Time of `select` for Minimum Element vs. List Size

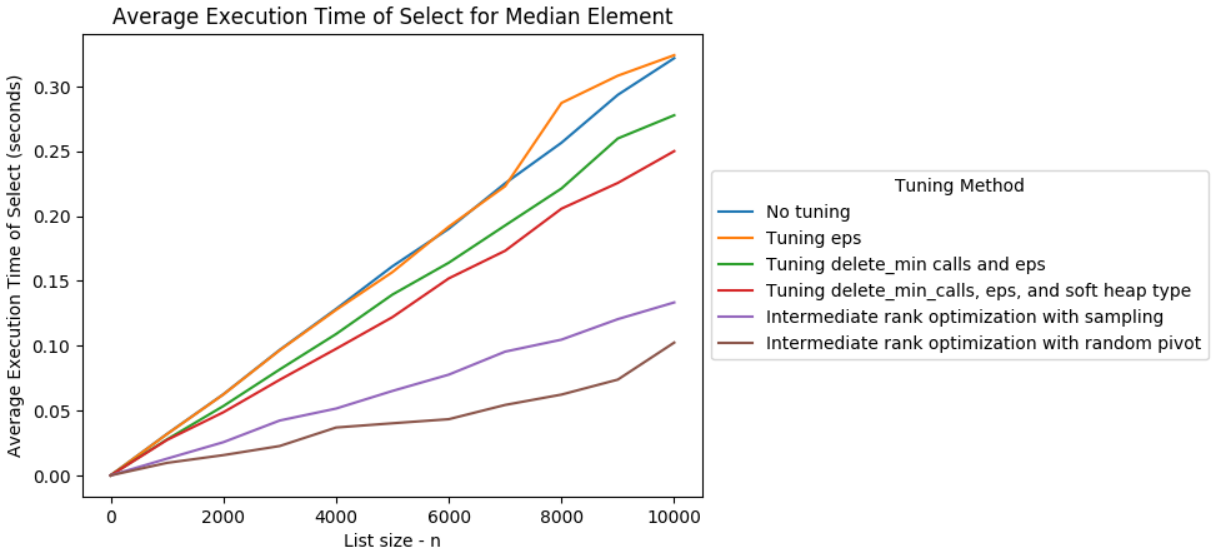


Figure 10: Average Execution Time of `select` for Median Element vs. List Size

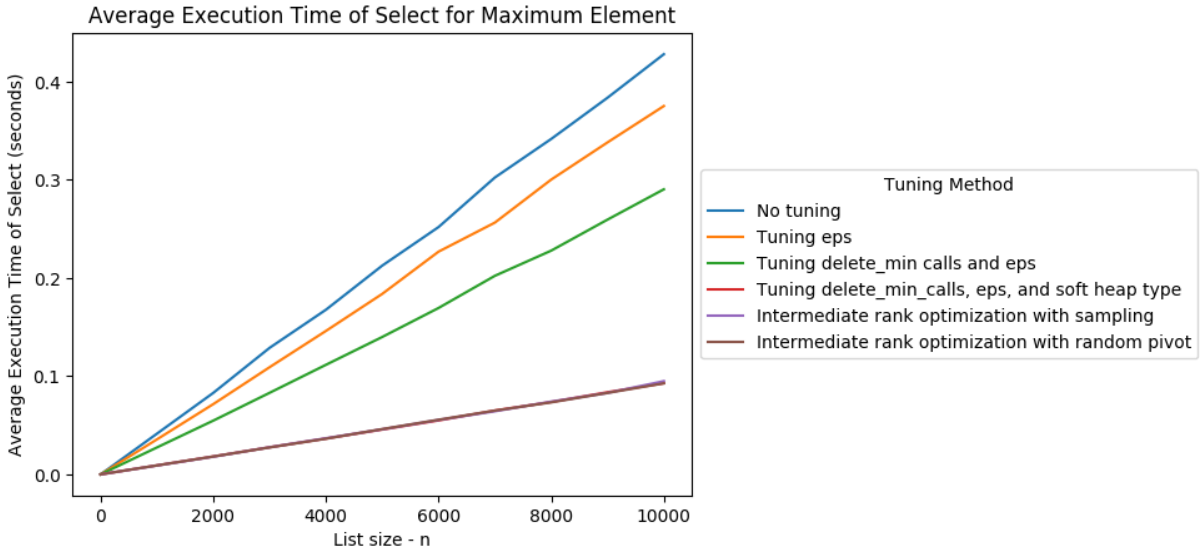


Figure 11: Average Execution Time of `select` for Maximum Element vs. List Size

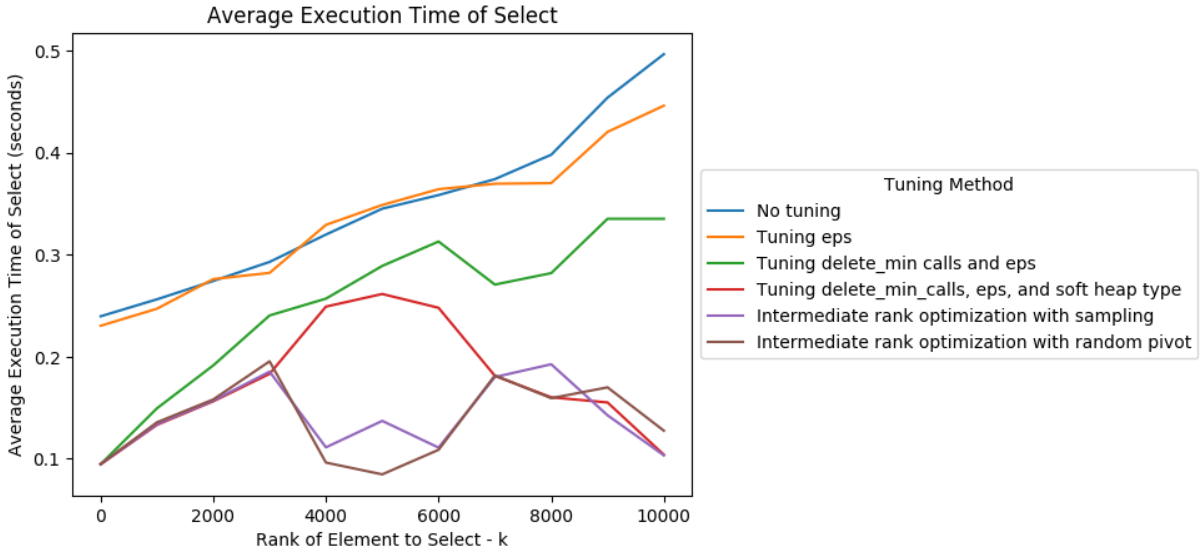


Figure 12: Average Execution Time of `select` vs. Rank of Element to Select

6 Conclusion

Chazelle's selection algorithm using the soft heap captures the fascinating idea that an intricate yet imperfect data structure can be used as the basis of a deterministically correct algorithm.[1] In the effort to make the behavior of the soft heap more transparent, we created an interactive visualization at `http://bit.ly/soft-heaps`.

References

[1] Bernard Chazelle. "The soft heap: an approximate priority queue with optimal error rate". In: *Journal of the ACM (JACM)* 47.6 (2000), pp. 1012–1027.

[2] Haim Kaplan, Robert E Tarjan, and Uri Zwick. "Soft heaps simplified". In: *SIAM Journal on Computing* 42.4 (2013), pp. 1660–1673.

[3] J. Ian Munro and Mike Paterson. "Selection and sorting with limited storage". In: *Theoretical Computer Science* 12 (1980), pp. 315–323.

[1] Beauty behind imperfection characterizes another of Chazelle's loves, jazz music—which, as it happens, inspired the creation of the Academy-Award winning film *La La Land*, directed by his son.