# CS166 Problem Set 4: Amortized Efficiency

Luis Antonio Perez

## Problem One: Stacking the Deque (3 Points)

A *deque* (***d****ouble-****e****nded* ***que****ue*, pronounced "deck") is a data structure that acts as a hybrid between a stack and a queue. It represents a sequence of elements and supports the following four operations:

- *deque*.`add-to-front`($x$), which adds $x$ to the front of the sequence.
- *deque*.`add-to-back`($x$), which adds $x$ to the back of the sequence.
- *deque*.`remove-front`(), which removes and returns the front element of the sequence.
- *deque*.`remove-back`(), which removes and returns the last element of the sequence.

Typically, you would implement a deque as a doubly-linked list. In functional languages like Haskell, Scheme, or ML, however, this implementation is not possible. In those languages, you would instead implement a deque using three stacks.

Design a deque implemented on top of three stacks. Each operation should run in amortized time $O(1)$. For the purposes of this problem, do not use any auxiliary data structures except for these stacks.

**Solution:**

**Overview:**

The general idea is quite simple. We use a front stack $F$, a back stack $B$, and a temporary stack $T$. $F$ lets us add/remove from the front efficiently, and $B$ lets us add/remove from the back efficiently. $T$ is used only to help shuffle elements from $B$ to $F$ (when $F$ is empty) or vice-versa.

**Operations:**

We now give details for how the operations are performed. $B$, $F$ and $T$ begin empty. We assume that the `size`() of a stack can be computed in $O(1)$. If not, note we can simply track the size explicitly by using a counter which is incremented on `push` and decremented in `pop`.

- *deque*.`add-to-front`($x$): Simply perform $F$.`push`($x$)
- *deque*.`add-to-back`($x$): Simply perform $B$.`push`($x$)

- *deque*.`remove-front`(): If $F$ is not-empty, simply return $F$.`pop`(). If $F$ is empty, `redistribute` the contents of $B$ so that half (rounded up) of the elements end-up in $F$ (in the correct order) and the other half in $B$. Now simply return $F$.`pop`().

- *deque*.`remove-back`(): If $B$ is not-empty, simply return $B$.`pop`(). Otherwise, `redistribute` the elements from $F$ so that half (rounded up) end up in $B$. Now return $B$.`pop`().

The operation `redistribute` can be performed relatively straight-forward. If both stacks are empty, `redistribute` is a no-op. Otherwise, let $A$ be the empty-stack and $B$ the full stack. Then pop half of the elemens (rounded down) from $B$ and push them onto $T$. Now pop the remaining elements in $B$ and push them onto $A$ until $B$ is empty. Lastly, pop the elements from $T$ and push them back onto $B$.

**Correctness:**

Correcness follows almost immediately. The first invariant is that $F$ contains elements in order where the front-most element is at the top with next elements near the bottom. Similarly, $B$ contains elements in order where the back-most element is at the top and the next element towards the bottom.

Note that each operation maintains these invariants, including `redistribute` since the order of half of the elements is reversed when coming out of $B$ (and into $F$), and the other half reversed twice (for a no-op) when out of $B$ and then out of $T$.

**Amortized Runtime:**

The claim is that each of the above operations runs in amortized $O(1)$ time. We can show this by using the *potential method.* We define our potential to be $\Phi = 4|\text{F}.\texttt{size}() - B.\texttt{size}()|$ ($4\times$ the height difference between the front and back stacks). We explicitly note that our potential starts at 0, and cannot end any-lower. As such, this is a valid upper-bound for our analysis. We further note that we pick the constant 4 for ease in our analysis, but a constant closer to 3 should also work.

We now show that each operation has $O(1)$ amortized time.

- *deque*.`add-to-front`($x$): This has an actual work of $O(1)$, with $4|\Delta\Phi| = 4$ ($+4$ if $F.\texttt{size}() \geq B.\texttt{size}()$ and $-4$ if $F.\texttt{size}() < B.\texttt{size}()$) for an amortized cost of $O(1)$.

- *deque*.`add-to-back`($x$): Similar analysis as `add-to-front` (with the role of $F$ and $B$ switched), for amortized cost of $O(1)$.

- *deque*.`remove-front`(): If $F.\texttt{size}() > 0$, the analysis is similar to the previous two for amortized cost of $O(1)$. If $F.\texttt{size}() = 0$, a deeper analysis is required. Note that about half of the elements from $B$ are popped and pushed once (directly into $F$). The other half are popped and pushed twice (one pop and push into $T$, and one pop and push back to $B$). Finally, a final $F$.`pop`() is done. As such, the real cost is at most $(4|B|+1)\Theta(1)$. As for our potential, we have $\Delta\Phi = -4(B.\texttt{size}() - (B.\texttt{size}() \mod 2))$ (the $\mod 2$ accounts for the cases when $|F.\texttt{size}() - B.\texttt{size}()| = 1$) since

$B$.`size`() is odd). Putting these two together, we have an amortized time of $O(1)$ (our saved potential covers our cost).

- *deque*.`remove-back`(): The analysis is similar to `remove-front` (with the role of $F$ and $B$ switched), for an amortized cost of $O(1)$.

## Problem Two: Very Dynamic Prefix Parity (4 Points)

On Problem Set Three, you designed a data structure for the ***dynamic prefix parity problem***. If you'll recall, this data structure supports the following operations:

- `initialize`($n$), which creates a new data structure representing an array of $n$ bits, all initially zero. This takes time $O(n)$.
- $ds$.`flip`($i$), which flips the $i$th bit of the bitstring. This takes time $O(\log n / \log \log n)$.
- $ds$.`prefix-parity`($i$), which returns the parity of the subarray consisting of the first $i$ bits of the array. (the *parity* is 0 if there are an even number of one bits and 1 if there are an odd number of 1 bits). This has the same runtime as the `flip` operation, $O(\log n / \log \log n)$.

Now, let's consider the ***very dynamic prefix parity problem***. In this problem, you don't begin with a fixed array of bits, but instead start with an empty sequence. You then need to support these operations:

- $ds$.`append`($b$), which appends bit $b$ to the bitstring.
- $ds$.`flip`($i$), which flips the $i$th bit of the bitstring.
- $ds$.`prefix-parity`($i$), which returns the parity of the subarray consisting of the first $i$ bits of the array.

Design a data structure for the very dynamic prefix parity problem such that all three operations run in *amortized* time $O(\log n / \log \log n)$, where $n$ is the number of bits in the sequence at the time the operation is performed.

**Solution: Overview:**

The idea is to use a similar strategy as is used when appending to vectors.

On each append, if don't have enough capacity, we double the capacity of our existing data structure. We do this by creating and initializing a new *leaf* bitarray that has the same capacity as our existing data structure. We can immediately see that this leads to bit-arrays that are each double the size of the previous one.

When `flip`ing bits, we just need to quickly located the *leaf* bitarray containing the bit, and flip the corresponding bit.

When computing `prefix-pairty`, we locate *leaf* bitarray containing the bit, compute the corresponding prefix-parity over the array, and then efficiently compute the parity of all smaller arrays combined using a *root* bitarray.

**Detailed Operations and Correctness:**

We now have give the details of how each operation is performed. Let each $D_k$ be the data structure we desiged to solve the ***dynamic prefix parity problem***. Let $ds$ be the new data structure we're designing to solve the ***very dynamic prefix parity problem***. We keep two counters: $c$ which consists of the number of bits that have been appended to $ds$ and $s$, which is the number of bits allocated. Both of these counters begin at 0, and are modified only during $ds$.`append`$(b)$.

Furthermore, let *leafs* be an array of pointers where the $i$-th index points to the $D_p$ for the *leaf* bit-array containing the $i$-th bit, where $p$ is such that $2^p$ is the largest power of 2 such that $2^p \leq (i+1)$. Let $D_{\text{root}}$ be the root data structure for a bit-array where the $p$-th index represents the parity of the entire bit-array corresponding to $D_p$. Note that all of these data structures start as empty.

- $ds$.`flip`$(i)$:

  Check $i < c$ (throw an error if not). Take $D_p = \textit{leafs}[i]$ and perform $D_p$.`flip`$(i-p)$, followed by $D_{\text{root}}$.`flip`$(p)$. In English, we flip the bit in the corresponding *leaf* bitarray, and also flip the parity-bit in the root bitarray corresponding to the *leaf* bitarray since the parity of the *leaf* bitarray as a whole has now changed. This maintains our invariants.

- $ds$.`prefix-parity`$(i)$:

  Check $i < c$ (throw an error if not). Take $D_p = \textit{leafs}[i]$, and compute $b_p = D_p$.`prefix-parity`$(i-p)$. This is the prefix-parity of the *leaf* bit-array containing $i$. To compute the complete *prefix-parity*, we must compute the parity of all preceeding *leaf* bit-arrays. Thankfully, we can do this directly by simply computing $b_{\text{root}} = D_{\text{root}}$.`prefix-parity`$(p-1)$ if $p > 1$, else let $b_{\text{root}} = 0$. Finally, simply return $b_p$ XOR $b_{\text{root}}$, which is the complete `prefix-parity` for $i$ and is correct given our invariants.

- $ds$.`append`$(b)$:

  If $c < s$, we first increment $c$. If $b == 0$, we're done. If $b == 1$, we perform $ds$.`flip`$(c-1)$.

  Now for the interesting case where $c == s$ ($c > s$ cannot occur). We first increment $c$ and double $s$ (if $s == 0$, set $s$ to 1). Then, `initialize`$(\max\{1, \frac{s}{2}\})$ $D_p$, where $p$ is such that $2^p$ is the largest power of 2 where $2^p \leq c$. We then append $\max\{1, \frac{s}{2}\}$ ($O(n)$) pointers to *leafs* each pointing to $D_p$. We then construct a new $D'_{\text{root}}$ by calling $D'_{\text{root}}$.`initialize`$(p+1)$ and then repeatedly calling $D'_{\text{root}}$.`flip`$(r)$ for each bit in the *root* bit-array which is 1. If $b == 0$, we are done. Otherwise, we simply call $ds$.`flip`$(c-1)$.

## Amortized Runtime Analysis

The claim is that each of the above operations run in $O(\log n / \log\log n)$ time, where $n$ is the number of bits at the time the operation is performed.

We show this by using the potential methond. First, define $n' = c - 2^p$ where $2^p$ is the largest power of 2 less than or equal ot $c$. This is just counting number of `append`s which have been performed into the largest $D_p$. Then the potential for our data structure is defined as:

$$\Phi = \begin{cases} 2n' & n' = 0 \\ 2n' + c_{\texttt{initialize}}(2n' + \log 2n') + c_{\texttt{flip}} \log 2n' \log\log 2n' & n' \ in \{1, 2\} \\ 2n' + c_{\texttt{initialize}}(2n' + \log 2n') + c_{\texttt{flip}} \log 2n' \log\log 2n' / \log\log\log 2n' & n' > 2 \end{cases}$$

We start with $\Phi_0 = 0$. Also note that $\Phi$ is never negative, so our amortized run-time will be an upper-bound. Furthermore, we note that $c_{\texttt{initialize}}$ corresponds to the big-o constant associated with the $O(n)$ run-time of `initialize` and similarly $c_{\texttt{flip}}$ corresponds to the big-o constant associated with the $O(\log n / \log\log n)$ run-time of the non-amortized `flip`.

We now show that each of the operations have amortized runtimes of $O(\log n / \log\log n)$.

- *ds*.`flip`($i$):

  The actual run-time is $O(1)$ (lookup in `leafs`) + $O(\log(n/2)/\log\log(n/2))$ (flip of the corresponding leaf structure, which has at most $n/2$ bits) + $O(\log\log n / \log\log\log n)$ (flip of the root structure, which has $\log n$ bits). This gives a total actual cost of $O(\log n / \log\log n)$. We note that $\Delta\Phi = 0$, so this is also the amortized cost.

- *ds*.`prefix-parity`($i$):

  The analysis is idential to `flip` where we replace `flip` with `prefix-parity`. Similary, the potential doesn't change. As such, the amortized cost is $O(\log n / \log\log n)$.

- *ds*.`append`($b$):

  If $c < s$ and $b == 0$, there is no actual work done and the change in potential $\Delta\Phi$ is at most $2 + 3c_{\texttt{initialize}} + c_{\texttt{flip}}$ (constant), giving an amortized run-time of $O(1) = O(\log n / \log\log n)$.

  If $c < s$ and $b == 1$, the actual work done is a single bit-flip with amortized run-time of $O(\log n / \log\log n)$. The change in potential $\Delta\Phi$ is at most $2 + 3c_{\texttt{initialize}} + c_{\texttt{flip}}$ (constant), giving a total amortized run-time of $O(\log n / \log\log n)$.

  In the worst-case, we must initialize a new $D_p$ which takes at most $c_{\texttt{initialize}} n * O(1)$ time. We then append $n$ pointers to *leafs*, taking $n * O(1)$. Next, initializing the new $D_{\text{root}}$ takes $c_{\texttt{initialize}} \log n * O(1)$ time, and making it consistent with the previous structure will take $\log n c_{\texttt{flip}} \log\log n / \log\log\log n * O(1)$. Finally, in the worst case, we must still call `flip` which has amortized run-time of $O(\log n / \log\log n)$.

  The above gives the amount of actual work performed (in the worst case) as:

  $$[n + c_{\texttt{initialize}}(n + \log n) + c_{\texttt{flip}} \log n \log\log n / \log\log\log n] * O(1) + O(\log n / \log\log n)$$

We must now compute $\Delta\Phi$. The easiest way to do this is to consider $\Phi_{m-1}$ (before the append) and $\Phi_m$ (after the append). Immediately before the append, the largest $D_p$ is completely full and has size $\frac{n}{2}$, so our potential is:

$$\Phi_{m-1} = n + c_{\texttt{initialize}}(n + \log n) + c_{\texttt{flip}} \log n \log\log n / \log\log\log n$$

Once we've completed the operation, the largest $D_p$ is now completely empty except for the first element. Our new potential is therefore:

$$\Phi_m = 2 + 2(c_{\texttt{initialize}} + 1)$$

Our difference in potential is therefore $\Delta\Phi = \Phi_m - \Phi_{m-1}$.

We note that $-\Phi_{m-1}$ exactly cancels all of the work-performed except for the final `flip`. As such, our total amortized run-time is $O\left(\log n / \log\log n\right)$ (plus $\Phi_m$, which we ignore since it is constant).

With the above, we have show that our data-structure has amortized run-time of $O\left(\log n / \log\log n\right)$ for all the given operations.

## Problem Three: Palos Altos (3 Points)

The order of a tree in a Fibonacci heap establishes a lower bound on the number of nodes in that tree (a tree of order $k$ must have at least $F_{k+2}$ nodes in it). Surprisingly, the order of a tree in a Fibonacci heap does *not* provide an upper bound on how many nodes are in that tree.

Prove that for any natural numbers $k > 0$ and $m$, there's a sequence of operations that can be performed on an empty Fibonacci heap that results in the heap containing a tree of order $k$ with at least $m$ nodes.

**Solution:** We first note that the problem statement is incomplete, and that we must have $m \geq F_{k+2}$. We now prove this modified statement.

We show this using a similar strategy to that outlined in lecture. Take $k > 0$ and $m \geq F_{k+2}$.

1. Find $p$ such that $m \leq 2^p$ and $k < p$ (e.g, find some power of 2 larger than $m$ where the exponent is larger than $k$).
2. `enqueu` $2^{p+1} + 1$ elements into the heap. This leads to $2^{p+1} + 1$ trees of order 0 in the heap.
3. Perform a single `extract-min`. This leads to a single tree of order $p + 1$ containing $2^{p+1}$ nodes. In fact, this single tree is a binomial tree. This is by construction, as shown in lecture. Recall that a binomial tree of order $p + 1$ is a single node whose children are binomial trees of order $0, 1, 2, \cdots, p$.
4. Prune aways all children of our single tree until only the $k$ children with the $k$-largest orders remain (note that we picked $p > k$, so we must always prune at least one child-tree). The pruning process for a given child tree can be done as follows. Beginning with the leaf-nodes, perform a `decrease-key` to cut the leaf from the tree and to make it the new mimimum. Then perform an `extract-min` to get rid of the single node. Repeat this until only the $k$-children with the highest orders remain.

At the end of the above procedure, we can immediately see that we are left with a Fibonacci heap containing a single tree. This tree is, by construction, of order $k$ (it only has $k$ children, since all others were pruned away). All that is left is to show that the number of nodes in the tree is at least $m$.

First, we note that the number of nodes in the tree must be greater than the number of nodes in the child with the largest order (we must have at least one child, since $k > 0$). By our procedure above, this child is actually a binomial tree of order $p$. As such, the child contains exactly $2^p \geq m$ (by construction) nodes. Therfore, our tree has at least $m$ nodes.

## Problem Four: Meldable Heaps with Addition (12 Points)

Meldable priority queues support the following operations:

- `new-pq()`, which constructs a new, empty priority queue;
- *pq*.`insert`($v$, $k$), which inserts element $v$ with key $k$;
- *pq*.`find-min()`, which returns an element with the least key;
- *pq*.`extract-min()`, which removes and returns an element with the least key; and
- `meld`($pq_1, pq_2$), which destructively modifies priority queues $pq_1$ and $pq_2$ and produces a single priority queue containing all the elements and keys from $pq_1$ and $pq_2$.

Some graph algorithms, such as the Chu-Liu-Edmonds algorithm for finding the equivalent of a minimum spanning tree in directed graphs (an *optimum branching*), also require the following operation:

- *pq*.`add-to-all`($\Delta k$), which adds $\Delta k$ to the keys of each element in the priority queue.

Using lazy binomial heaps as a starting point, design a data structure that supports all `new-pq`, `insert`, `find-min`, `meld`, and `add-to-all` in amortized time $O(1)$ and `extract-min` in amortized time $O(\log n)$.

Some hints:

1. You may find it useful, as a warmup, to get all these operations to run in time $O(\log n)$ by starting with an *eager* binomial heap and making appropriate modifications. You may end up using some of the techniques you develop in your overall structure.
2. Try to make all operations have worst-case runtime $O(1)$ except for `extract-min`. Your implementation of `extract-min` will probably do a lot of work, but if you've set it up correctly the amortized cost will only be $O(\log n)$. This means, in particular, that you will only propagate the $\Delta k$'s through the data structure in `extract-min`.
3. If you only propagate $\Delta k$'s during an `extract-min` as we suggest, you'll run into some challenges trying to `meld` two lazy binomial heaps with different $\Delta k$'s. To address this, we recommend that you change how `meld` is done to be even lazier than the lazy approach we discussed in class. You might find it useful to construct a separate data structure tracking the `meld`s that have been done and then only actually combining together the heaps during an `extract-min`.
4. Depending on how you've set things up, to get the proper amortized time bound for `extract-min`, you may need to define a potential function both in terms of the structure of the lazy binomial heaps and in terms of the auxiliary data structure hinted at by the previous point.

As a courtesy to your TAs, please start off your writeup by giving a high-level overview of how your data structure works before diving into the details.

**Solution:**

**Overview**

The general idea is to be lazy (lazier than binomial heaps), delaying almost all work to the $pq$.`extract-min` function.

To support `add-to-all` efficiently, we make some modifications. Each tree will also store a $\Delta k_T$ corresponding to already propoaged $\Delta k$ which should be added to keys of nodes in $T$.

Additionally, to make sure we can support `meld` efficiently, we keep an auxilary data structure, $N$, to track the melds that have occurred. This data structure will be a binary tree. The root, $N_{pq}$, corresponds to the current priority queue. The children of a node $N_{pq_1}, N_{pq_2}$, correspond to two melded priority queues which resulted in $N_{pq}$. Each leaf node has a pointer to the linked-list of roots of its priority queue (intermediate nodes don't point to anything). All nodes store $\Delta k_{pq_i}$, the unpropogated amount to be added to the keys contained in the priority queue represented by the node.

Each operation except `extract-min` does the minimum amoung of work possible, maintaining the properties described above.

Lastly, we also maintain a pointer such that each node $T$ in a priority queue $pq$ can immediately access $N_{pq}$ (and thereby access $\Delta k_{pq}$ in constant time).

**Operations** We now present how each operation is performed in detail.

- $pq$.`new-pq`():

  Simply initialize an empty doubly-linked list (for the roots, of which there are none). Create the node $N_{pq}$ and set $\Delta k_{pq} = 0$. Set its pointer to the head of the empty doubly-linked list.

- $pq$.`insert`$(v, k)$:

  Add a new single-node tree, $T$, to the forest (append to the linked list of roots). The node contains the element $v$ with key $k' = k - \Delta k_{pq}$ and with $\Delta k_T = 0$. Make this $T$ have a pointer to $N_{pq}$. If the *min* pointer is not set, set it to point to $T$. Otherwise, let $T'$ be the tree pointed to by *min* and $k_{\min}$ the corresponding key of the node. Then compare $k$ to $k_{\min} + \Delta k_{T'} + \Delta k_{pq}$ (the min modifed by any propagated and unpropoaged changes), and update the *min* pointer accordingly.

- $pq$.`find-min`():

  Return the element pointed to by the *min* pointer.

- $pq$.`extract-min`():

  We use the *min* pointer to find the tree with the minimum key, $T$. We remove this node, and add the children, $C_i$, to our heap (adding to the list of trees). As we're adding the children, we update $\Delta k_{C_i} += \Delta k_T$, thereby propogating the key deltas to each child to maintain our invariants.

Now, we compact our auxilary data-structure by propogating from parent node $N_p$ to children nodes $N_{C_1}, N_{C_2}$ the key delta such that $\Delta k_{C_i} + = \Delta k_P$. Eventually we'll be left only with the leaf-nodes (all other nodes destroyed). Each leaf-node $N_L$ maps directly to one of the original priority queues which we were supposed to meld, with $\Delta k_L$ fully updated to reflect all calls to `add-to-all`. The only remaining task is to destroy these nodes my fully propogating the key deltas to their corresponding trees. We do this during the tree compaction stage.

First, create $N_{pq'}$ and set $\Delta k_{pq'} = 0$. At the end, this will be the root of our auxilary data structure.

For the tree-compaction stage, we follow a modified-version of the tree-compaction algorithm for binary heaps. The difference is that when fusing two trees of the same order, $T_1$ and $T_2$, we have to be a bit careful about the keys. First, we propogate $\Delta k_{pq_1}$ and $\Delta k_{pq_2}$ down to the trees themselves (eg, set $\Delta k_{T_i} + = \Delta k_{pq_i}$). Once this is done, we should compare the values $(k_i + \Delta k_{T_1})$ and $(k_2 + \Delta k_{T_2})$, and select the appropriate minimum tree using these values. WLOG, let $T_1$ be the tree with the smaller node. Then we modify the pointer in $T_1$ so as to point to $N_{pq'}$ (this is to prevent double-counting, since this tree already has all the key deltas propogated).

Once the fusing process is complete, we are done.

- $pq$.`meld`$(pq_1, pq_2)$:

  Let $T_{pq_i}$ be the tree pointed to by the *min* pointer in $pq_i$ and $k_{pq_i}$ the corresponding key in the node for $i \in \{1, 2\}$. Now compare the value $k_{pq_1} + \Delta k_{T_{pq_1}} + \Delta k_{pq_1}$ to $k_{pq_2} + \Delta k_{T_{pq_2}} + \Delta k_{pq_2}$ to determine the new minimum element (we're comparing the true value of the keys if all updates had been made to both priority queues). Set the *min* pointer of our priority queue accordingly.

  Take the two auxilary data structures $N_{pq_1}$ and $N_{pq_2}$ and merge them with a new parent $N_{pq}$. Set $\Delta k_{pq} = 0$. Stop. The melding will be performed in `extract-min`.

- $pq$.`add-to-all`$(\Delta k)$:

  Simply update $\Delta k_{pq} + = \Delta k$.

**Amortized Runtime Analysis:**

We claim that each operation has an amortized runtime of $O(1)$ except for `extract-min`, which has an amortized run-time of $O(\log n)$. We do this using the *potental method* using the potential function $\Phi$ that's equal to the total number of trees in our priority queue (even if we haven't quite finished `meld`ing them) plus the total number of nodes in our auxilary data structure. We note that $\Phi_0 = 0$, and that $\Phi$ is never negative. As such, this potential function is a valid upper-bound on our run-time.

- $pq$.`new-pq`():

The actual time spent in this function is $O(1)$ (we just allocate a constant number of objects), with $\Delta\Phi = 1$ (we add one node to our auxilary data structure). This gives an amortized run-time of $O(1)$.

- *pq*.`insert`$(v, k)$:

  The actual time spent in this function is $O(1)$ (add to a linked list and update values), with $\Delta\Phi = 1$ (we add one new tree). This gives an amortized run-time of $O(1)$.

- *pq*.`find-min`():

  The actual time spent is $O(1)$ (follow a single pointer), with $\Delta\Phi = 0$. The amortized run-time is therefore $O(1)$.

- *pq*.`add-to-all`$(\Delta k)$:

  The actual time spent is $O(1)$ (update a value), with $\Delta\Phi = 0$ to give an amortized run-time of $O(1)$

- *pq*.`meld`$(pq_1, pq_2)$:

  The actual time spent is $O(1)$ (update the *min*). The change in potential is $\Delta\Phi = 1$, since our auxilary data structure gains a node. However, the total number of trees remains the same, so there is no change there.

- *pq*.`extract-min`():

  Removing the min takes time $O(1)$ (updating a few values). Compacting the auxilary data structure is $\Theta(M)$, where $M$ is the number of nodes in the auxilary data structure at the time `extra-min` is called. The tree-compaction step takes $O(T + \log n)$, where $T$ is the number of trees at the start. This gives the total amount of work done as $\Theta(M) + O(T + \log n)$.

  Now we focus on potential. We begin with $T$ trees and end with at most $\log n$ trees, for an overall change in the number of tress if $-T + \log n$. As far as our auxilary data structure, we begin with $M$ nodes and end with a single node, for an overall change of $-M + 1$. This gives the total potential change $\Delta\Phi = -T + \log n - M + 1$.

  This means that for a large enough multiplicative constant applied to our potential, we'll have have an amortized time of $O(\log n)$, as desired.

## Problem Five: Splay Trees in Practice (8 Points)

This one is all coding! Make sure to submit your code on myth.