Homework Assignment 1

Submission Guidelines.

- Latex submissions are preferred but not mandatory. However, please write your solutions on unruled paper and upload a clearly scanned copy. Please do not use poorly lit photographs. Keep your solutions brief and to the point. Meaningless rambles fetch negative credits.
- Collaboration is allowed in teams of up to two students, but each student must write and submit their own individual solutions in their own words. Also mention the name of your collaborator.
- "Bibles" such as textbooks, lecture notes from other courses, or online forums (e.g., Stack Exchange) may not be used as references for your write-up.
- Limited use of large language models (LLMs) is permitted, but you may not directly copy-paste text from them; use them only for clarification and then produce your own reasoning and explanations.

Problem 1. (10 points) Recall that during **Decrease-Key** operation in a Fibonacci heap, once a node loses a second child, we cut away the root along with its remaining subtree and cascade this process upward in the original tree until we find a node which has lost only one children. Suppose instead of waiting for loss of the second child, we wait till a node loses k children (for constant $k \geq 3$). Analyze the amortized runtime of n_1 inserts, n_2 deletes and n_3 decrease-key operations in such a heap. Your runtimes will clearly depend on k. You need to show the tightest possible bounds for full credit. In other words, saying $O(1)$ is not enough. You need to specify exact constants.

Solution.

The main (and possibly only) non-trivial part of this problem is to figure out the right potential function which helps us analyze the effect of waiting till k children are removed before cutting a node away from its parents. Here is one way of doing that (might not be the only way). For each node we store a counter of how many of its children were removed (call count_i the counter of node i). To analyze the running time of the operations we use the following potential function:

$$\Phi \;=\; \#\text{roots} \;+\; \frac{2}{k-1}\sum_i \text{count}_i.$$

Insert. As in the original Fibonacci heap, INSERT increases the number of roots by one and does not affect the counters; hence INSERT has $O(1)$ amortized cost.

Decrease-Key. Suppose a DECREASE-KEY triggers c cascading cuts. The real cost is $1 + c$ (one decrease plus c cuts). The potential change has two parts:

- the number of roots increases by c, contributing $+c$ to $\Delta\Phi$;
- counters change: each time a child is cut-away, the counter for that node is reset to 0 - since we have c cuts and each one must have its count exactly $k-1$ at the moment it was cut away, the potential changes by $\frac{-2c(k-1)}{k-1} = -2c$. However, note that at the node where the cascading stops, the count not increases by 1 - since it loses one more child but is not reset. This countributes $2/(k-1)$ to the potential.

Hence, the total change $\Delta\Phi = c - 2c + 2/(k-1) = -c + 2/(k-1)$

Thus, amortized cost of Decrease-key is $1 + c - c + 2/(k-1) = 1 + 2/(k-1)$. Note that this is stricly better than Fibonacci Heaps for $k \geq 3$

Delete-Min. The amortized cost of DELETE-MIN is bounded by the maximum degree M of a root. Let S_m be the minimum number of nodes in a heap-tree whose root has degree m. Because a node is cut only after losing k children, the degree of its i-th child is at least $\max\{0, i-k\}$ (the proof is very similar to the one done in class for $k = 2$). Also $S_m = m + 1$ for $m = 0, \ldots, k-2$ (base cases). For $m \geq k-1$ we have

$$S_m = (k-1) + \sum_{i=k-1}^{m} S_{i-k)},$$

hence

$$S_m - S_{m-1} = S_{m-k}.$$

This solution to this recurrence, φ_k, is the largest root of the equation of

$$\varphi_k^k - \varphi_k^{k-1} = 1.$$

If M is the maximum possible degree in an n-node heap, then $S_M \leq n$, so

$$M = O\left(\log_{\varphi_k} n\right) = O\left(\frac{\log_{\varphi_2} n}{\log_{\varphi_2} \varphi_k}\right).$$

The point is that for Fibonacci Heap, the runtime only had $\log_{\varphi_2} n)$ whereas now we have another term in the denominator which is a *decreasing* function of k. Hence, the amortized runtime of Extract-Min actually goes up compared to Fibonacci Heaps

Problem 2. (5 + 5 points) Let P be a priority queue that supports the following operations:

- `insert`, `delete-min`, and `merge` in $O(\log n)$ time, and
- `make-heap` in $O(n)$ time, where n is the size of the resulting priority queue.

(a) Using one or more structures like P as a black box, show that you can create a data structure that performs `insert` in $O(1)$ amortized time, without affecting the cost of `delete-min` or `merge` (i.e., $O(\log n)$ amortized time). Do not rely on the availability of an efficient `decrease-key` operation in your solution.

Solution. One standard way to do this is to augment the priority queue with a linked list. At any point in time, out data-structure will keep both the priority queue $\mathcal{P}$ and the list $\mathcal{L}$ and the elements will belong to *either* of them.

Whenever we do an insertion, just add the element to $\mathcal{L}$ - clearly this is $O(1)$ worst case time (even better than amortized).

Now we define a consolidation operation which will be required for both delete-min and merge. The consolidation operation has two steps - (a) Call make-heap on the elements in $\mathcal{L}$ - call this temporary heap $\mathcal{P}'$ (b) Call merge on $\mathcal{P}, \mathcal{P}'$.

With the above definition, now we can define delete-min. First consolidate and then perform the usual delete-min. A merge operation can also be defined in a similar way.

Analysis. Although defining a potential for this analysis is kind of an overkill, but it keeps the thing clean. So we define the potential of the data-structure as $|\mathcal{L}|$. Then the amortized cost of insert is still $O(1)$. To see the amortized cost of say a delete-min, the real cost is $\log n + |\mathcal{L}|$. But notice that the change in potential will be $-|\mathcal{L}|$. Hence, the amortized cost of delete still remain $\log n$.

(Rubric : Any correct solution is fine. +3 for defining the insert, delete-min and merge properly, +2 for analysis where even an semi-formal but correct argument will be rewarded)

(b) Using the above technique, show that even binary heaps can be modified to support `insert` in $O(1)$ amortized time while maintaining an $O(\log n)$ bound for `delete-min`. Note that binary heaps do not support `merge` in $O(\log n)$ time.

Solution. Our data-structure will be maintaining not one but several heaps $\mathcal{P}_1, \mathcal{P}_2, \cdots \mathcal{P}_k$ at any point of time and a linked list $\mathcal{L}$ as above . There would be an auxillary master heap $\mathcal{P}_m$ which will maintain the roots of the main heaps. Insertion is as in part (a) - just add to $\mathcal{L}$.

Consolidation will now consist of creating a new heap $\mathcal{P}_{k+1}$ with the elements in $\mathcal{L}$ and inserting the root in the master heap. Hence, delete-min will now involve two things - consolidation and then doing a delete-min on the master heap.

The analysis is very similar to (a).

We want a precise description of your data structure and all the operations along with their runtime (either amortized or worst case). Further, Fibonacci heaps already achieve the above bounds. But clearly you cannot use such a thing as your solution.

Problem 3. (5+5+5 points) In class, we saw how to build a perfect hash table for a set of keys given in advance. In this problem, we will generalize this to allow insertion and deletion of keys in the table *dynamically*.

Recall that perfect hashing involves a top-level table hashing to multiple second-level tables using 2-universal hash functions. We initially consider only insertions. The basic approach is to rebuild any table (top-level or second-tier) when it violates the invariants on which perfect hashing relies.

(a) Consider a second-level table that, in the static case, uses $f(s) = O(s^2)$ space to store s items. We gave a construction of such a table that succeeds with probability 1/2. Suppose that we take the following approach to insertions: if inserting the s-th item violates perfection of the table, repeatedly build a table of size $f(2s)$ until you get a perfect one. Show that when s items are inserted (one at a time, with rebuilds as necessary) the expected total cost of all the rebuilds is $O(s)$. You may assume that array initialization takes constant time (or figure out how to do that if you wish).

Solution. Let s - the total number of items inserted be an integer in $(2^k, 2^{k+1}]$ for some non-negative integer k (any integer $s > 0$ can be expressed like this). Now observe that by our protocol, we started with a table of size say 1 and then doubled all the way up to 2^{k+1}. Now, remember that every time the table size is doubled from 2^i to 2^{i+1}, the total amount of work done is $O(2^{i+1}$ in expectation - this is simply because we may need to rehash till we get a perfect hash function. We showed in class that the expected number of times is 2 and each rehashing takes time proportional to the number of elements currently being stored. Note that if you try to argue with only one second level table, you will not be able to pull-off the geometric sum.

Further, (and this is crucial), although the size of one second level table is quadratic in the number of items that are being hashed, the expected *total* size of all second level tables is always $O(s)$ (the current number of items). Hence, each time you rebuild a second level table, you will be spending time proportional to the current number of items s.

Now we apply the usual geometric sum. The total work done in rebuilding the hash table is roughly $O(1 + 2 + 4 + \cdots 2^{k+1}) = O(2^{k+1}) = O(s)$.

Rubric : +2 for realizing what the doubling trick is really doing, +1 for stating that total size of second level tables is till $O(s)$ and +2 for the final calculation using geometric sum. Again, this is sample rubric, evaluation is more subjective. any correct solution gets full reward.

(b) Let s_i be the number of items in bucket i in the second-level table. Make a similar modification for the top level: any time the table on n items has bucket sizes s_i violating the requirement that $\sum_i s_i^2 = O(n)$, we rebuild the top table and all the second-level tables. Show that the expected total time is $O(n)$ when n items are inserted.

Solution sketch. The solution idea is exactly same as before. We double the size of the table every time the insertion of a new item makes the ratio n and size of the table exceed say a constant fraction 1/4. When we double the top level table, recall that we will need to not only rebuild the top level table but also the second level tables. Now we repeat the rebuild process for the top level until the $\sum_i s_i^2$ becomes $O(n)$. After that, we rebuild the second level tables. As has been discussed several times, in order to achieve a perfect hashing, in expectation we need to repeat each of these a constant number of times and hence the total work done is $O(n)$. We can finally do the same geometric sum argument as above to finish the argument.

(c) Now consider deletions. When an item is deleted, we can simply mark it as deleted without removing it (and if it is reinserted, unmark the deletion). This makes deletion cheap. But if there are many deletions, the table might end up using much more space than it needs, which could invalidate the amortized bounds of the previous parts. To fix this, we can rebuild the hash table any time the number of "holes" (items marked deleted) exceeds half the total items in the table. Show that this cleanup work does not change (i.e., can be charged against) the amortized $O(1)$ cost of insertions.

Solution sketch. Let us identify the main issue here. In general, deletion of items cannot violate perfection property (this is obvious). However, the trouble is the following - suppose your total current tables sizes is $4n$ and you initially had n items. Now after a lot of deletion, the number of items may become very small compared to n. Now you can no longer argue that the table size is a constant times the number of current items. Hence we take the approach suggested above.

Suppose for a moment that we had n insertions followed by deletions only. By (a) and (b), insertion takes $O(1)$ expected time without deletions. Suppose we now consider deletions. Every time we make a deletion, we mark a node and hence deletion time is $O(1)$. Every time we rebuild the entire hash table due to more than $n/2$ holes getting created, we take expected time proportional to $n/2$ to do so. However, if this happens only once, we are still ok ! We can simply charge this to the $O(n)$ cost we had paid anyway during insertion.

However, what if we have more deletions ? Note that the next time we rebuild will only be when number of items drops to $n/4$ and hence the time taken will be proportional to $n/4$. This can go on. However, all of these deletion will accumulate cost that is something like $O(n + n/2 + n/4 + \cdots)$ - all of which can be charged to the $O(n)$ time for deletion.

Now what if insertions and deletions are interleaved. One can essentially formalize the above by using a potential function that is equal to the number of marked nodes. I am not expecting you to do this precisely. If you have written the above idea, you get full credit.