Petar Maymounkov
September 14, 2005
6.854J – Advanced Algorithms
In collaboration with:
Benjamin Rossman, Oren Weimann, and Pouya Kheradpour

Problem 1. We'll partition the necessary sequence of heap operations in supersteps, where each superstep:

1. Consists of a constant number of heap operations,
2. Increases the number of different items in the heap by a constant.

Furthermore, the configuration of the heap between supersteps will consist of one HOT of depth d and one HOT of depth 0, where d is a parameter. The key of the root of the former one will be denoted k_a, and of the latter one as k_*, and $k_a < k_*$ will always hold. For convenience, we denote the heap configuration between supersteps as $\langle d, k_a, k_* \rangle$.

Starting from a configuration $\langle d, k_a, k_* \rangle$, the superstep consists of the following heap operations:

1. Insert item $k_{a'} < k_a$,
2. Insert item $k_d < k_{a'}$,
3. Perform delete-min,
4. Perform decrease-key for k_*, such that new $k_{*'}$ is bigger than $k_{a'}$.

At Step (3): k_d is removed and $k_{a'} - k_*$ form a depth-1 HOT, which is then merged with the HOT rooted at k_a, such that $k_{a'}$ is the new root. The resulting tree has depth $d + 1$. Now $k_{a'}$ has two children, one is a linear subtree of depth d, and one is k_* itself. Step (4) removes k_* and leaves the

HOT rooted at $k_{a'}$ in linear form. Hence, after the superstep, the heap is in configuration $\langle d+1, k_{a'}, k_{*'} \rangle$.

Asymptotically, after n operations the depth of the linear HOT is approximately $n/2$.

Problem 2. The potential function can be viewed as a counter of work that has to be done in the future. And hence, by adding to the potential during a given "amortized" step, we are choosing to count future work now.

In the case of the decrease-key operation for Fibonacci heap, where each node is allowed to lose k children before it cuts off, we modify the potential analysis as follows.

First, each node has to keep a $\lceil \log k \rceil$-bit counter of how many children have been removed so far, still call this the *mark* of the node. Define the potential function as the sum of all marks divided by $k-1$. As a result the amortized (after telescoping) cost a_i of a step becomes:

$$a_i = 1(\text{the first cut}) + 1/(k-1), \ \text{(mark increase at last node of cascade)}$$

Note that because we are discussing a constant change in the running time, it is important to make sure that the dimension of the real as well as the potential costs are the same. In this case they are both measured in number of cut operations (as the basic atomic operation). So the amortized cost of decrease-key is no $1+1/(k-1)$, which is smaller by a constant than $1+1=2$ for the case of 2-cut Fibonacci heaps.

Now, let's examine delete-min. An execution of delete-min comprises the following costs:

1. Remove the minimal entry: 1 real unit,
2. Distribute children to heap's collection: c real units,
3. Find new minimum and consolidate roots: $c + r_{old}$ real units, where r_{old} is the old number of roots. Note that these two operations can be done simultaneously, hence we put them together for simplicity.

4. Added potential cost: r_{new} - r_{old} potential units.

Since we are shooting at exhibiting a constant fraction of slow down, the dilligent thing to do is to have separate potential functions for the different kinds of atomic operations above (1 through 3). However, for simplicity we will assume that they are equally expensive.

Furthermore, the lecture notes assume that the distribution of children to bins takes c real units, because the random-access model is implicitly used. However, this also holds without this assumption if bins are represente by a linked list and each node points to the bin where it would belong (if it is not root), or belongs to (if it is root).

Adding up the real and the potential costs produces:

$$1 + 2c + r_{new}$$

We now address the upper bound on the size of c and r_{new}. Mimicing the notes from class:

Claim: The currently i-th (starting from 1) remaining child (children are ordered historically) has degree at least $i - k$.

Proof: Call this child y. When y was added, its root x had at least $i - 1$ children already present. Hence x had degree at least $i - 1$. Hence, y had degree at least $i - 1$. But since then, up to $k - 1$ children could have been cut away from y, and therefore now y has at least $i - k$ children.

For the minimum number of descendants (including yourself) of a node, S_k, we have:

$$\begin{aligned}
&S_0 = 1, \text{(the root itself counts)} \\
&S_1 \geqq 2, \\
&\quad \cdots \\
&S_k \geqq S_0 + kS_0 = k + 1, \text{then for } j > k\text{:} \\
&S_j \geqq S_0 + kS_0 + \sum_{i=1}^{j-k} S_i = k + 1 + \sum_{i=1}^{j-k} S_i,
\end{aligned}$$

Therefore, $S_j \geqq S_{j-1} + S_{j-k}$. We now demonstrate that the roots (both real and complex) of the characteristic polynomial $x^k - x^{k-1} - 1$ of this recurrence are all smaller then φ in absolute value, which would imply that the asymptotics of the recurrence are $O(r^j)$, where $r < \varphi$.

Let $n = k - 1$ and set $re^{i\theta} = x$ for convenience, where $r \geqq 0$ is real. Rewrite the polynomial equation as:

$$r^n e^{in\theta}(re^{i\theta} - 1) = 1, \text{take the absolute value on both sides :}$$
$$r^n |re^{i\theta} - 1| = 1$$

Now, let A be the point 0 (on the complex plane), B be -1, C be $re^{i\theta}$, D be $re^{i\theta} - 1$, E be the x coordinate of D on the x-axis, and F be 1.

From the equation above we have:

$$|AB| = 1, \qquad |DA| = 1/r^n$$
$$|BD| = r, \quad \angle ABD = \angle DAF = \theta$$

Now, using simple trigonometry, we get that:

$$|EA| = 1 - r\cos\theta, \text{ and } |DE| = r\sin\theta$$

Then, we apply the Pythagorian theorem to $\triangle\ ADE$ to get:

$$\cos\theta = \frac{r^2 - 1/r^{2n} + 1}{2r}$$

We observe set $\cos\theta \leqq 1$, and solve for r:

$$r^{2n}(r^2 - 2r + 1) \leqq 1$$

Now, if we set $r = \varphi$ (the Golden ratio) and $n = 1$ we get an equality. At the same time we observe that the right hand side grows with n and r, therefore r must be smaller than φ for $n > 1$ and we are done.

So far we've shown that $S_j = O(r^j)$ for $r < \phi$. To demonstrate that Fibonacci treats with k cuts are truly slower than Fibonacci heaps with 2 cuts, we need to exhibit a case where the worst time bound is reached. This is easily shown, since the equality $S_j = S_{j-1} + S_{j-k}$ is reached when we force the Fibonacci algorithm to construct one complete tree with n elements, and then cut (via decrease-key) the $k - 1$ heaviest elements from each node.

Problem 3a. We are going to build a derivative heap, using the given one as a subroutine. The derivative heap will consist of two data structures: a heap (of the given kind) and a linked list of elements.

Insert-new is implemented simply as an addition on to the linked list.

Delete-min-new does the following: (1) converts the linked list into a temporary heap (of the given kind) using the make-heap-given operation of the given heap; (2) merges the two heaps (with merge-given); and (3) calls delete-min-given on the resulting heap.

Merge-new follows steps (1) and (2) of the delete-min-new above (for both derivative heaps), and then (3) uses the underlying merge-given operation to merge the remaining two heaps into 1.

The key to analysing this new structure is to observe that each insert-new operation incurs 1 unit of real cost, and 1 unit of future cost. Where, the unit of future cost is delayed until the next delete-min-new or merge-new operation, and happens within step (1) of their algorithms. Hence the total cost of insert-new is $O(1)$.

The costs of delete-min-new and merge-new comprise of the $O(\log n)$ real units in steps (2) and (3) (for both of them), and the cost of step (1) is already accounted for by the amortized analysis of the insert-new operation.

Problem 3b. We don't employ the fact that the underlying heap structure is a binary heap, and we just use it as a blackbox heap algorithm as above, which doesn't support merge at all.

Our new data structure consists of two sub-structures: (1) a linked list of items, and (2) a "master" heap (of the blackbox kind) where the items inserted into this heap will be themselves heaps (of the blackbox kind) of regular items.

Insert-new simply adds the new element at the end of the linked list.

Delete-min-new:

1. Run make-heap-blackbox on the items in the linked list

2. Treat the resulting heap as a single item whose key is the key of its smallest element[1], and insert it (using insert-blackbox) into the master heap.

3. Run delete-min-blackbox on the master heap. This operation will remove the subheap with smallest item from the master heap

4. Run delete-min-blackbox on the subheap from the previous step. The item output of this operation will be the output of our delete-min-new.

5. Run insert-blackbox to re-insert the subheap from the previous step back into the master heap, in the case that this subheap is not empty

(Note: decrease-key-new can be implemented identically to delete-min-new.)

Note that operations (2),(3),(4) and (5) in delete-min-new all take $O(\log n)$ time, because neither the master heap nor the subheaps can have more than n elements in them, and hence these operations take $O(\log n)$ time each.

On the other hand, identically to part (3a) of the problem, the $O(n)$ cost of the make-heap-blackbox operation is cancelled by paying for it in terms of potential credit when inserting new items with the insert-new function.

Problem 4a. We are given (as input) a tree of nodes, where each node points to its children (or to its first child, and there is a sibling list among the children). We are also given pairs of queries.

We observe that in the offline algorithm the algorithm's state during the traversal of the tree is independent of the query answering. In particular, for each step of the tree traversal (identified by the currently visited node) we define the "current configuration" to consist of (a) the currently visited node, and the (b) assignment of union names to previously visited nodes. The post-order traversal of the tree defines a natural timeline on the evolution

[1]The key of the smallest element of a heap can be computed by running delete-min-blackbox on it, and then re-inserting the just-deleted item with insert-blackbox.

of the configuration. The entire history of configurations is independent of the particular queries being answered on the way. Hence, our goal will be to create a persistent data structure that can query the configuration for any point in time.

Part (b) of the configuration is embodied by the state of the union-find (union-by-rank) data structure. Each node of the original union-find data structure corresponds to an item of the input tree together with a parent pointer and a rank counter. In the persistent-union-find, we shall modify the nodes to consist of:

1. parent-ptr: Parent pointer, initially NULL
2. parent-time: Time when parent pointer was assigned a non-NULL value
3. name-tree: Pointer to a balanced binary tree, which contains pairs of the form $\langle$time, new-name$\rangle$. This tree will be used to resolve the "name" of the node for a given time. The tree is initially empty
4. rank: Rank counter, initially set to 0

In the persistent-union-find, the union operations are implemented as follows:

Persistent-find-set is implemented as the regular version of it, except for a parent pointer is not followed, if the current time is prior to the parent-time timestamp. When the set root is reached, the name of the set is looked up in the name-tree by timestamp. The first portion of the algorithm takes $O(\log n)$ steps due to the properties of the union-by-rank structure. The second portion also takes $O(\log n)$ steps as there can be at most n elements (names) in the name-tree. Hence, persistent-find-set takes $O(\log n)$ steps.

Persistent-union proceeds as follows:

1. Do a persistent-find-set to find the root node of the set
2. Use the rank of the item being added and the root item to figure out which one will be the new root.

3. Set the parent-ptr and the parent-time of the non-root item to point to the root item

4. Insert a $\langle$time-now, name$\rangle$ pair in the name-tree of the root item, to reflect the desired root name. The desired root name (in the case of our application) is the root name of the existing set we wanted to add the new set to.

It is easy to observe that persistent-union takes at most $O(\log n)$ steps.

This completes the definition of the persistent-union-find structure.

We preprocess the input tree by traversing it post-order, and updating the persistent-union-find structure. This takes $O(n \log n)$ operations. Also, while preprocessing we label each tree node with the time step that it corresponds to. Then during the online query answering, when we get a pair of tree nodes, we know what to set the current time stamp to.

Problem 4b. In the preprocessing step, we first convert the given tree into a binary tree. To do this, we replace each node in the tree that has $m > 2$ children by a binary tree (with dummy inner nodes) and m leaves. This step takes $O(m)$ time per original node, and hence the entire conversion takes $O(n)$ time.

Then we traverse the tree to compute the number of descendant for each node. And then we order the 2 children of each node in ascending order by number of descendants. Also $O(n)$ time.

Then we do another traversal of the tree, where we put back pointers from each node to the highest node that it can reach (by walking up the binary tree's edges) without taking any turns in the binary tree, and we also record the direction of walking (up-left or up-right).

This completes the preprocessing. It is easy to show that the length of each path from a node to the root along the back pointers is at most $\log n$ steps, because the back pointers represent the Huffman compression of the binary encoding of the path from the root to the node (along the binary tree's regular edges). The result follows since the entropy of a random variable with n distinct values cannot be more than $\log n$.

To answer queries. For each query, we retrace the path to the root from the query nodes, and invert it. So we have the paths from the root to the query nodes (along the back edges). Both paths are at most $\log n$ steps. We find the first node (in each path) at which they differ, and using the direction of walking (up-left or up-right) to this node, we choose which of the two is the common parent.