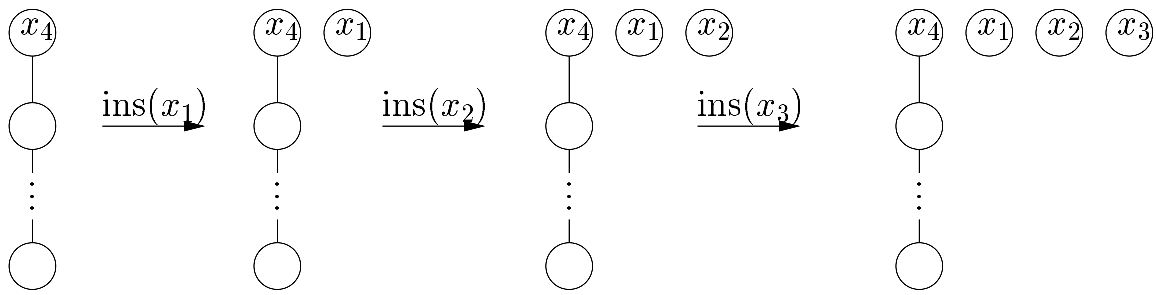
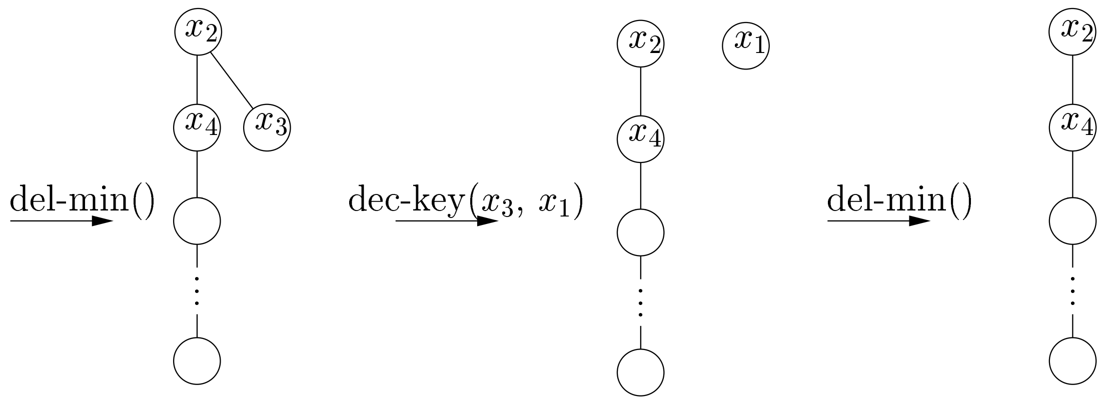
Problem Set 1 Solutions

Problem 1. Suppose that we have a chain of $n-1$ nodes in a Fibonacci heap and we do the following: Insert 3 items $x_1 < x_2 < x_3 < x_4 =$ current minimum. Delete-min. Decrease-key(x_3, x_1). Delete-min.





So the first delete-min removes x_1, links x_2 and x_3, making x_2 the parent of x_3, and links x_2 and the old chain, making x_2 the parent. Now the decrease-key rips x_3 off the tree, and the delete-min removes it. We are left with x_2 at the top of our old chain, giving a chain of length n.

Since we can make a chain of length 1 by inserting an item, we can iterate this procedure to produce a single chain of length n for any n.

Comments from graders: A few students used consolidate as a basic operation. Some did not consider base case. Most students got 10 points.

Problem 2. Let $D(n)$ be the maximum degree of any heap-ordered tree in a fibonacci heap having n elements. Recall that the exponential descendants lemma proves that $D(n) = O(\log n)$. We analyze the general case when a node is cascading cut after more than c of its children are removed. Instead of maintaining a mark bit for each node, a mark counter is maintained. When the mark counter reaches c, a cascading cut is performed and the mark counter is reset to 0. We assume the potential function to be of the form

$$\phi = a \cdot \text{number of roots} + b \cdot \text{sum of mark counters}$$

where a and b are constants we pick later.

We will now analyze the amortized cost of operations. All in-register operations such as operating with local variables, executing instructions, etc are assumed to be free operations. We moreover assume that memory allocation is free.

Insert: Insert is performed by linking a node holding the key to the list of roots. We assume this to be a unit operation. The actual cost of insert is 1 and the change in potential is a. So the amortized cost of insert is $a + 1$.

Decrease-key: The decrease-key operation cuts the specified node, incurring 1 unit cost. Then, for each cascading cut, 2 units of work is done to increment the mark counter and perform the cut. The final mark counter increment costs 1 unit. So the actual cost of decrease-key is $2 + 2k$, where k is the number of cascading cuts. In this process, $k + 1$ new roots are created, k mark counters are reset to 0 and a mark counter is incremented. So the change in potential is $a(k+1) - bck + b = (a - bc)k + a + b$. Therefore the amortized cost is $2 + (a - bc + 2)k + a + b$. For the amortized cost to be $O(1)$, we require

$$bc \geq a + 2 \tag{1}$$

Delete-min: Let r be the number of roots and d be the degree of the current minimum element. The current minimum is held in the data structure. Removing the minimum item and attaching its children as roots will take 1 unit of work (assuming that all lists are maintained as doubly linked lists). Now the fibonacci heap has $r + d - 1$ roots.

The consolidation procedure involves

(a) Creation of at most $D(n)$ buckets: We assume the creation of the bucket array to be free

(b) Finding the new minimum and placing each tree in corresponding bucket: We need to do $r + d - 1$ unit cost node operations.

(c) Merging trees in buckets: Let r_f be the number of roots after consolidation. A single merge involves two read-key operations and one merge. Since each merge reduces the number of trees by one, we have $r + d - 1 - r_f$ merges taking $3(r + d - r_f) - 3$.

(d) Placing the trees in buckets as roots: This takes r_f units of work.

So the actual cost is

$$r + d - 1 + 3(r + d - r_f) - 3 + r_f = 4r + 4d - 2r_f - 4 = 4(r - r_f) + 2d + 2r_f - 4$$

The number of roots reduces from r to r_f. Therefore the change in potential is $a(r - r_f)$. So the amortized cost of delete-min is $(4 - a)(r - r_f) + 2d + 2r_f$. Notice that both d and r_f can be at most $D(n)$. Thus we have a $(4 - a)(r - r_f) + 4D(n)$ bound on the cost of delete-min. We will show later that $D(n) = O(\log n)$. For delete-min to be $O(\log n)$ we require,

$$a \geq 4 \tag{2}$$

The cost of insert and decrease-key can be at most $a+1$ and $a+b+2$ respectively provided (1) and (2) are satisfied. The upper bound on cost of delete-min however remains the same as long as (1) and (2) are satisfied. Therefore, we would like to minimize a and b given the constraints on them and reduce the cost of insert and decrease-key. Solving for this we get $a = 4$, $b = 6/c$ and

$$\phi = 4 \cdot \text{number of roots} + 6/c \cdot \text{sum of mark counters}$$

Now we can upper bound the amortized cost of insertion, decrease-key and deletion by 5, $6 + 6/c$ and $4 + D(n)$ respectively. When $c = 1$, the cost of decrease-key is 12 units. This is improved to 9 units when c is increased to 3.

We will now estimate $D(n)$ for any c. This is similar to the case for $c = 1$. The ith child of a node will have degree at least $i - 1 - c$. Therefore we have the following recurrence on the minimum number of descendants $F(k)$ of a node with degree k:

$$F(k) = F(k-c-1) + F(k-c-2) + \cdots + F(k-c-k)$$

where $F(i) = 1$ for $i \leq 0$. By taking the difference of $F(k)$ and $F(k-1)$ we get

$$F(k) - F(k-1) = F(k-c-1)$$

The solution to this linear recurrence is of the form

$$F(k) = c_1\alpha_1^k + c_2\alpha_2^k + \ldots$$

where each α_i is a root to the equation

$$\alpha^{c+1} - \alpha^c = 1$$

For $c = 1$, the largest root is $(1+\sqrt{5})/2 \approx 1.62$. For $c = 3$, the largest root is 1.38 (computed numerically). So the value of $F(k)$ is $\Theta(1.38^k)$. We can see that $D(n) = 1.38 + o(1)$. Therefore the constant associated with $\lg n$ reduces from $1/\lg 1.62$ to $1/\lg 1.38$, a factor of ≈ 1.5. The additive constant $o(1)$ is ignored in this analysis. For $c = 4$, we get a factor of ≈ 1.74.

Comments from graders: The most common error was to find constants in the potential function by directly solving for equations instead of minimizing cost given inequalities as constraints. One point was taken off for this minor error. Some students did not find out exactly how fast decrease-key runs in (b). The constant factor slow-down of decrease-key was not computed in (b). These errors cost 1-2 points.

Problem 3. The goal of this problem is to achieve a constant amortized time lazy insert routine in priority queues.

(a) We can augment the priority queue P with a bucket (implemented as a linked list). An insert operation places the element in the bucket. We define a consolidate operation to incorporate (say) m elements in the bucket in P as follows. A priority queue is constructed on the elements in the bucket using $O(m)$ make-heap and is merged with P in $O(\log n)$ time. The time taken for consolidation is $O(m + \log n)$ where m is the size of the bucket. Delete-min performs a consolidate followed by the standard delete-min for P. Simiarly, merge performs the consolidate operation on both heaps and then performs the merge. If we define the potential function ϕ as the size of the bucket, the amortized cost of insert and delete-min become $O(1)$ and $O(\log n)$ respectively.

We will now extend the potential function to account for merge. Consider a set of initially empty heaps (which may be merged later) on which all operations are performed. The potential associated with these heaps is the sum of their bucket sizes. An insert or delete-min operation on heap h will take $O(1)$ or $O(\log n_h)$ amortized time respectively, where n_h is the number of elements held in the heap h. A merge operation done on two heaps h and h' will incur an amortized cost of $O(\log n_h + \log n_{h'}) = O(\log(n_h + n_{h'}))$.

(b) Although binary heaps do not support $O(\log n)$ merge, it is possible to perform the consolidate operation in $O(m+\log n)$ time. We will now describe two methods of consolidation depending on whether m is more than $n/2$ or not.

Case I: $m \geq n/2$ A make-heap is done on all the n elements. Elements are added level-by-level from the lowest level to the root. Inserting level of height i consisting of $n/2^i$ elements will take $O((1 + i) \cdot n2^{-i})$ time. Therefore the total cost will be $\sum_{i=1}^{\lg n} O((1 + i)n2^{-i}) = O(n) = O(m)$.

Case II: $m < n/2$ The elements in the bucket need to be consolidated into a complete binary heap. Each element is placed in the binary heap ensuring that the heap is complete in each step. Notice that the lowest one/two levels of the binary heap are filled with new unbalanced elements. Now, a level-by-level cascade operation is done on the newly inserted elements. Intutively, the number of elements to be cascaded-up reduces by a factor of 2 in every level.

Lemma 1 *The consolidation operation takes $O(m + \log n)$ time.*

Proof. At most two levels (lowest and next-to-lowest) can be occupied by the newly inserted elements. In each level, the elements are placed next to each other. Let us consider one of these contiguous blocks. Let T be the smallest subtree of the heap having elements in this block as leaves. Subtree T has at most $2m$ leaves and therefore consists of at most $4m$ elements. A cascade can be performed on the elements in T. After reaching the root of T, only $O(\log n)$ cascade-up operations can be performed. Therefore each block requires $O(m + \log n)$ cascade-up operations. There are at most two such blocks. ■

Thus, we have a $O(m + \log n)$-time consolidate. Now we can apply the result shown in (a) to achieve $O(1)$ insert binary heaps.

Aliter (brief outline): We maintain a binary heap of heaps H. Each node x in the binary heap H has a reference *x.heap* to a heap and its key *x.key* as the minimum of *x.heap*. The consolidate operation can be performed by constructing a binary heap in linear time and inserting a reference to it in H. Delete-min can be performed in $O(\log n)$ time in the data structure after consolidation.

Comments from graders: Most solutions to problem (a) were correct. The heap of heaps solution was popular. The rest were more likely to make mistakes in analysis.

Problem 4. We use a F-heap to maintian the bucket data structure.

(a) The amortized cost of insert is the cost of searching down k levels and then moving up k level during its deletion. Therefore insert takes $O(k)$ time if a F-heap is used to organize elements in a bucket. The amortized cost of decrease-key is the cost of a delete-min and an insert. So decrease-key takes $O(\log \Delta)$ time. The cost of delete-min is the cost of one delete-min on a bucket, which takes $O(\log \Delta)$ time. Recall that $\Delta = C^{1/k}$. We set $k = \log \Delta = k^{-1} \log C$ to minimize the cost of a priority queue operation. Therefore each priority queue operation takes $O(\sqrt{\log C})$ when $k = \sqrt{\log C}$.

   Aliter (brief outline): An alternate solution is to "make the queue" circular. That is, insert the values in the queue mod C. Since the range of values used at any time is only C, the only possible resulting confusion is that the minimum value in the queue may not be the minimum value mod C. This is easily remedied by adding the successor operation and using successor and delete to get the next minimum instead of find-min and delete.

(b) Let $G = (V, E)$ be an undirected graph with edge weights $c(e) \in 1, 2, \ldots C$. Let $n = |V|$ and $m = |E|$. There are m decrease-keys and n inserts and deletes. Finding shortest paths on G using the above data structure will take $O((m+n)\sqrt{\log C})$ time.

   We can also tune the data structure to optimize for the cost of finding shortest paths, which is $O(n(t_{insert} + t_{delete}) + mt_{dkey})$. Now, t_{insert} is $O(1+k)$ and $t_{delete} = t_{dkey} = O(1 + k^{-1} \log C)$. The cost of finding shortest paths when $nk = mk^{-1} \log C$ is $O(m + \sqrt{mn \log C})$.

Comments from graders: Some solutions did not account for the difficulty in determining the next minimum in the second approach. Two points were taken off for incomplete description. Each part is worth 2 points in part (a).

Problem 5. The Van Emde Boas data structure comprises of a recursive VEB data structure H on high-half words and a VEB data structure $L(h)$ for each high-half word in H. It also has a hash table that holds `(high-half-word, present/not)` pairs. A query on the presence of a high half word takes constant time. Finally, the VEB data structure stores the current minimum item. We augment the data structure to hold the current maximum item too.

If a VEB data structure has only one element, we just keep the element, we do not create the recursive structures. Let $b = \log U$. Assume for simplicity that b is a power of 2 and that we will not store two elements with the same value. Notice that we can use a linked list to store items with same value. Insert and delete operations need to maintain a consistent current maximum. This is similar to the maintenance of current minimum.

We will now describe the operations $find(v)$, $succ(v)$ and $pred(v)$. We denote an item v as (v_h, v_l) if v_h and v_l are the high and low half words respectively. Recall that the operations are efficient only if one recursive call happens to a VEB data structure, halving the number of bits operated on.

find(v): If the structure has only one item, see if v is it. Otherwise, do a find on v_h in the hash table and a recursive find on v_l in $L(v_h)$ and report

$$find(v) = (v_h \in H) \wedge L(v_h).find(v_l).$$

pred(v) If v is the minimum item in its bucket, do a pred on v_h in H and return the maximum of that bucket. Otherwise do a pred on v_l in the recursive structure at the bucket corresponding to v_h. In short

$$pred(v) = \begin{cases} (H.pred(v_h), L(\underline{H.pred(v_h)}).max) & \text{if } (v_h \notin H) \vee (v_h = L(v_h).min) \\ (v_h, L(v_h).pred(v_l)) & \text{otherwise} \end{cases}$$

The underlined expression is a common sub-expression that is evaluated only once. So we perform only one recursive call.

succ(v) Successor can be done similarly.

$$succ(v) = \begin{cases} (H.succ(v_h), L(\underline{H.succ(v_h)}).min) & \text{if } (v_h \notin H) \vee (v_l = L(v_h).max) \\ (v_h, L(v_h).succ(v_l)) & \text{otherwise} \end{cases}$$

Again, the underlined expression is a common sub-expression that is evaluated only once. So we perform only one recursive call.

The number of bits operated on is reduced by 1/2 with each iteration. Each iteration performs $O(1)$ hash table lookups and $O(1)$ minimum/maximum field lookups, taking $O(1)$ time. So all the above operations take $O(\log b)$ time, where b is the maximum number of bits in an item.

Comments from graders: Some solutions did not handle the absence of the item on which prev/succ were called. The most common major error was to use an auxiliary data structure with $\omega(1)$-time per operation instead of maintaining max.