Part VI
Fibonacci Heaps

1. Let S be a string composed of n bits (S represents a number between 0 and 2^{n-1}). Suppose we store S using a circular list with a pointer to the least significant bit. Define the operations $inc(S) = S + 1 \mod 2^n$ and $div2(S) = \lfloor S/2 \rfloor$. How can we implement $inc(S)$ and $div2(S)$ such that their amortized cost is $O(1)$. Use a potential function (you may assume that initially $S = 0$). ☆

   Solution: For implementing $div2(S)$ we set the least significant bit to zero and move the pointer to the second least significant bit. For $inc(S)$ we flip all ones to zeroes starting from the least significant bit. If we go through the whole list S we stop (integer overflow), if we encounter a zero before going through the whole list we change it to 1 and then stop. We can set $\Phi(S)$ equal to the number of ones in S. The $div2(S)$ operation either decreases $\Phi(S)$ by one or leaves it unchanged, therefore the amortized cost of $div2(S)$ is $O(1)$. If an integer overflow happens the actual cost of $inc(S)$ is $O(n)$, however the potential is decreased by n, otherwise if $inc(S)$ changes c bits the actual cost is $O(c)$ and the potential difference is $-c + 2$ ($c - 1$ ones are changed to 0 and one 0 is changed to 1). Therefore the amortized cost of $inc(S)$ is also $O(1)$.

2. Consider an array T of size $T.size$ and let $T.num$ be the number of elements present in T. We consider a single operation that adds one element to T. When T is full (i.e., $T.num = T.size$) and we need to add another element, we allocate a new array T_{new} of size $2T.size$ and copy the elements from T to T_{new}, remove T and rename T_{new} as T. Define a potential function such that the amortized cost of this operation is $O(1)$. [Hint: Let $\Phi(T)$ depend on $T.num$ and $T.size$. Make sure $\Phi(T)$ cannot be negative.] ☆

   Solution: We set $\Phi(T) = 2T.num - T.size$ and we start from T full of size 1. By the construction of T, we always have $\Phi(T) \geq 0$. The amortised cost of adding an element to T when T is not full is clearly $O(1)$: $\Phi(T)$ increases by 2 and the

actual cost is $O(1)$. Adding an element to T when T is full has the actual cost of $O(T.size) = O(T.num)$ and doubles $T.size$. Therefore the amortized cost is $O(T.size) + (2(T.num+1) - 2T.size) - (2T.num - T.size) = O(T.size) + 2 - T.size = O(1)$.
Extra: Does the following potential work: $\Phi(T) = \frac{2}{T.size} T.num^2$?

3. How can you implement an insert operation in a binary heap with n nodes in $O(\log n)$ time? Define a potential function H such that the amortized cost of the Insert operation equals $O(\log n)$ and that of a Delete-Min only $O(1)$. ☆

3.1 Amortized analysis of the *delete-min* and *decrease-key* operation (p. 70)

4 Bounding the maximum degree (p. 71)

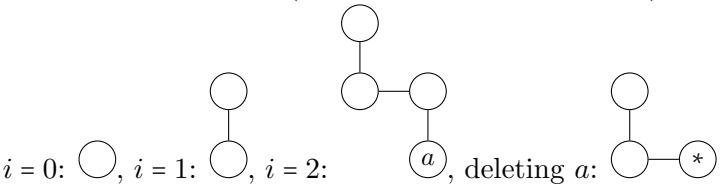
4.1 On Fibonacci heaps (p. 72)

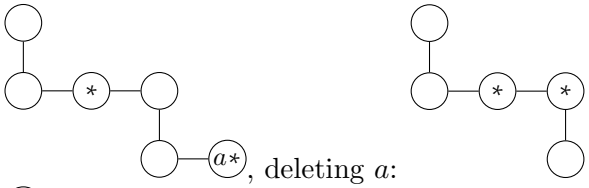
1. Suppose we also wish to support a UNION and DELETE operation for Fibonacci heaps. How would you implement these operations? What is the (amortized) cost of your solution? ☆

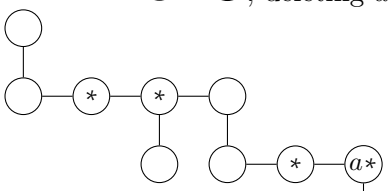
Solution: UNION can be implemented in $O(1)$: we link the root lists to each other and determine the minimum key of the union of two heaps by picking the smaller minimum key of the two heaps. Clearly, the potential Φ isn't changed by a UNION operation.
The DELETE operation can be implemented as a *decrease-key* operation (for example to minimum key-1 or to $-\infty$) followed by a *delete-min* operation. The amortized cost is then $O(D(n)) \leq O(\log n)$.

2. What is the maximum number of vertices in the root-list of a Fibonacci heap consisting of 40 vertices given that we just executed a DELETE-MIN operation? ☆

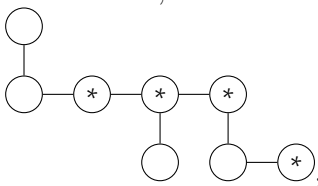
Solution: Minimal tree of degree i can be constructed from two minimal trees of degree $i-1$ as follows ($*$ means a vertex is marked):



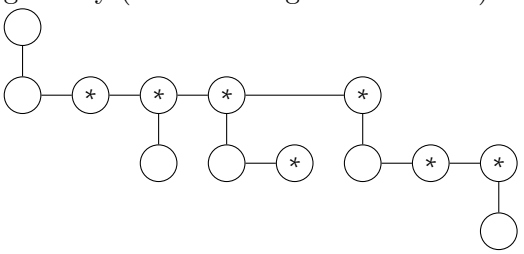
$i = 3$:  , deleting a:



$i = 4$:  ,



deleting b (or a and b): , the minimal tree of degree $i = 5$ is given by (after deleting some vertices):

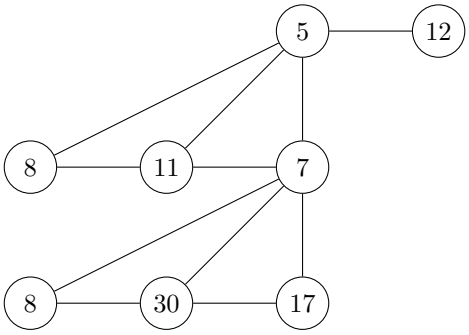


Explanation on how to make these trees (if allowed to use the UNION operation): Make a two heaps each containing a minimal tree of degree k. We will refer to the root with the greater key as vertex v. Perform a union operation on these two heaps, and insert a node of key $-\infty$. Delete that node, after the clean-up you are left with a tree of degree $k+1$ that isn't necessarily minimal. Now delete a child of v together with the descendants of this child. You should choose the child with the most descendants.
The simplest solution without the UNION operation is adding $2^0+2^1+2^2+\cdots+2^i+1 = 2^{i+1}$ nodes followed by delete min, this creates trees of degree $0,\ldots,i$. Delete all extra nodes to obtain minimal trees of degree $0,\ldots,i$.

In general, one can show that the smallest tree where the root has rank i consists of F_{i+2} nodes (see section 4). After a *delete-min* operation no two roots have the same rank (number of children of the root). As $1+2+3+5+8+13 = 32 \leq 40$ and $32+21 > 40$, the maximum number of roots after the *delete-min* is 6: we take the smallest trees discussed above up to and including $i = 5$ where we don't delete 8 of the extra vertices.

3. Redo the amortized analysis of the DELETE-MIN and DECREASE-KEY operations if we use $t(H) + m(H)$ as a potential function. ☆

4. Give a sequence of operations (starting from an empty heap) to obtain the following Fibonacci heap or explain why this is impossible. ☆



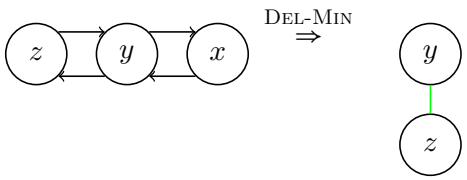
Solution: This is impossible as at least one of the lowest nodes has to have a child (see Theorem 4.1), i.e. the sub-tree rooted in the node with key 7 is not a Fibonacci tree.

5. Let $n(H)$ be the number of nodes in a Fibonacci heap H. Suppose we use the potential function: $\hat{\Phi}(H) = \log(n(H)!) + t(H) + 2m(H)$. Argue that both the DELETE-MIN and DECREASE-KEY operations have an $O(1)$ amortized cost. Why is the overall amortized cost when using $\hat{\Phi}$ still $O(|E| + |V| \log |V|)$ when executing Prim's algorithm? ☆

6. What is the maximum depth of a node in a Fibonacci heap consisting of n nodes? Prove your answer and provide a sequence of operations that creates such a node.

Solution: The depth of a node in a tree is the number of edges from the node to the root of the tree. In a Fibonacci heap containing n nodes, the depth will always be less than or equal to $n - 1$.

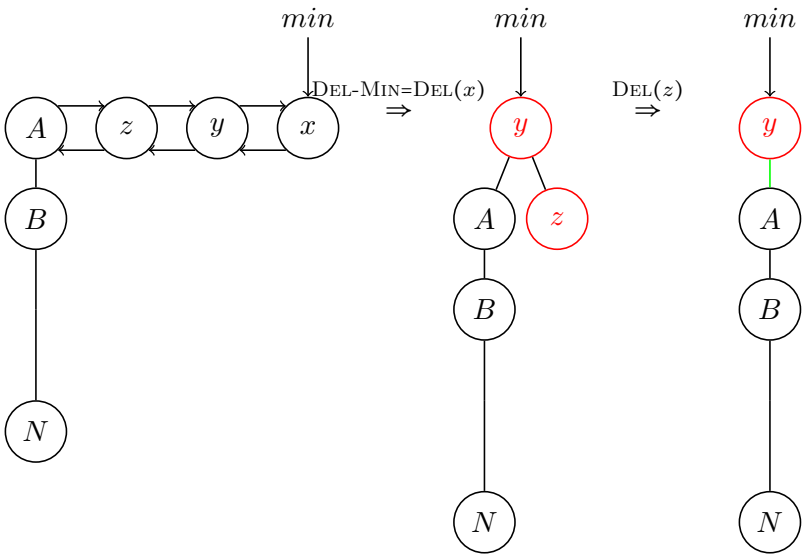
We now prove (by induction) that a Fibonacci heap of size n has a maximum depth of $n - 1$ (depth n is mathematically impossible due to the definition of depth). The base case is $n = 2$ ($n = 1$ is also possible, but trivial). We insert 3 items and delete the minimum element (supposing $x < y < z$).



The leaf node clearly has depth 1 (as there is 1 edge from one of the roots to the deepest leaf). This is $n - 1$ and can not grow further with only $n = 2$.

We now prove the induction step. We assume the nodes of an n-node Fibonacci heap have maximum depth $n - 1$ (organized as one path).

We insert 3 new nodes x, y and z, where we again assume $x < y < z$ and $z < A$ where A is the root node of an existing n-node Fibonacci heap.



We now have $n+1$ nodes (the previous n-node Fibonacci heap and one new node y). The maximum depth grows by one and is now n.

7. Suppose we allow a node to lose $s > 1$ children before promoting it to the root list. Would this alter the overall runtime complexity? Redo the parts of the (amortized) analysis that require changes to support your answer.