0.1 Fibonacci heaps

- *Algorithm:* Fibonacci heaps (algo. **??**)
- *Input:*
- *Complexity:* $O(logn)$ for EXTRACT-MIN and DELETE, $O(1)$ for other operations
- *Data structure compatibility:* Fibonacci heap as a data structure, implemented through linked list
- *Common applications:* Dijkstra's shortest path algorithm

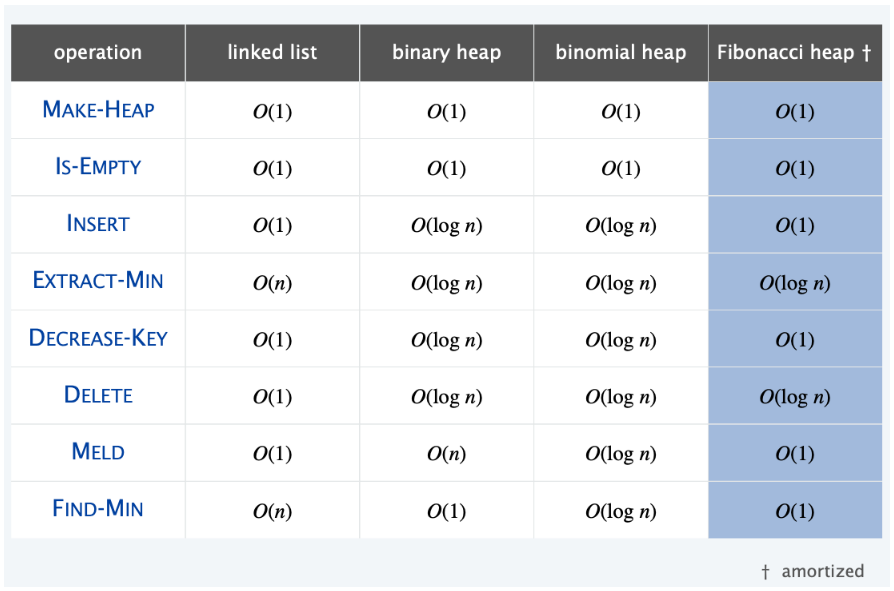
Problem. Fibonacci heaps

The original motivation is to improve the Dijkstra's shortest path algorithm. First, it supports a set of operations that constitutes what is known as a "mergeable heap." Second, several Fibonacci-heap operations run in constant amortized time, which makes this data structure well suited for applications that invoke these operations frequently [**1**].

Description

As a data structure that constructs a mergeable heap, it supports operation *MAKE-HEAP(), INSERT(H,x), MINIMUM(H), EXTRACT-MIN(H), UNION(H_1, H_2), DECREASE-KEY(H,x,k), DELETE(H,x)*, where H is a fibonacci heap and x is an element, k is a new key value.

As a implementation of priority queue, its has good amortized time complexity, compare to linked list, binary heap or binomial heap (*MELD* stands for *UNION*) [**2**].



operation	linked list	binary heap	binomial heap	Fibonacci heap †
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IS-EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
EXTRACT-MIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
FIND-MIN	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$

† amortized

Figure 1: Comparison of Priority Queue Performance

Structure:

- A set of heap ordered trees, such that in each tree, $child.key \geq parent.key$.
- Heap store a pointer of the minimum node. Root list is maintained through circular, doubly linked-list.
- Node has a pointer to its parent; a pointer to any of its children; a pointer to its left and right siblings; its degree (number of children); whether marked.

Analyze of Time complexity:

For a given Fibonacci heap, potential function: $\Phi(H) = trees(H) + 2 \cdot marks(H)$. For each operation, analyze the actual cost, change in potential and amortized cost.

- *Insert:* $c_i = O(1), \quad \Delta\Phi = \Phi(H_i) - \Phi(H_{i-1}) = +1, \quad \hat{c}_i = c_i + \Delta\Phi = O(1)$
- *Extract Min:* $c_i = O(rank(H)) + O(trees(H)), \quad \Delta\Phi \leq rank(H') + 1 - trees(H), \quad \hat{c}_i = c_i + \Delta\Phi = O(logn)$
- *Decrease Key:* Denote the number of cuts as c, $O(1)$ for each cut and meld it into root list, $c_i = O(c), \quad \Delta\Phi = O(1) - c, \quad \hat{c}_i = c_i + \Delta\Phi = O(1)$
- *Union:* $c_i = O(1), \quad \Delta\Phi = 0, \quad \hat{c}_i = c_i + \Delta\Phi = O(1)$
- *Delete:* Achieved by *Decrease Key* and *Extract Min*, so $\hat{c}_i = O(rank(H))$

How to analyze rank(H)

A Fibonacci heap with n elements, then $rank(H) \leq log_\Phi m$, where Φ is the golden ratio $= (1/\sqrt{5})/2 \approx 1.618$.

Discussion:

The difficulty of implementation and the constant factor make it not so practical in real life. It will be useful for large data size. The $\Theta(1)$ amortized time of *Decrease Key, Extract Min, Delete* make it desirable when number of such operations needed. So computing minimum spanning tree, finding single-sourse shortest paths make essential use of it.

It is named as “Fibonacci Heap” because Fibonacci numbers are used in running time analysis. Every node in Fibonacci Heap has at most degree at $O(logn)$. The size of a subtree rooted in a node of degree k is at least F_{k+1}, where F_k is the $k-$th Fibonacci number.