WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Čeština
Deutsch
Español
فارسی
Français
עברית
日本語
Русский
Српски / srpski
Svenska
ไทย
Українська
中文
Edit links

Fibonacci heap

From Wikipedia, the free encyclopedia

In computer science, a **Fibonacci heap** is a heap data structure consisting of a collection of trees. It has a better amortized running time than a binomial heap. Fibonacci heaps were developed by Michael L. Fredman and Robert E. Tarjan in 1984 and first published in a scientific journal in 1987. The name of Fibonacci heap comes from Fibonacci numbers which are used in the running time analysis.

Find-minimum is $O(1)$ amortized time.[1] Operations insert, decrease key, and merge (union) work in constant amortized time.[2] Operations delete and delete minimum work in $O(\log n)$ amortized time.[2] This means that starting from an empty data structure, any sequence of *a* operations from the first group and *b* operations from the second group would take $O(a + b \log n)$ time. In a binomial heap such a sequence of operations would take $O((a + b) \log n)$ time. A Fibonacci heap is thus better than a binomial heap when *b* is asymptotically smaller than *a*.

Using Fibonacci heaps for priority queues improves the asymptotic running time of important algorithms, such as Dijkstra's algorithm for computing the shortest path between two nodes in a graph.

Contents [hide]

Structure [edit]

A Fibonacci heap is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree. This flexibility allows some operations to be executed in a "lazy" manner, postponing the work for later operations. For example merging heaps is done simply by concatenating the two lists of trees, and operation *decrease key* sometimes cuts a node from its parent and forms a new tree.

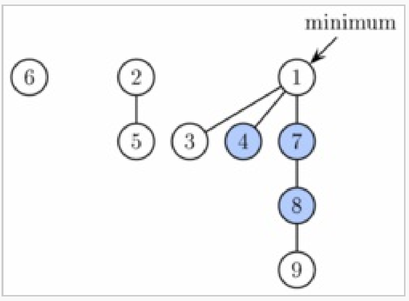


Figure 1. Example of a Fibonacci heap. It has three trees of degrees 0, 1 and 3. Three vertices are marked (shown in blue). Therefore the potential of the heap is 9 (3 trees + 2 × (3 marked-vertices)).

However at some point some order needs to be introduced to the heap to achieve the desired running time. In particular, degrees of nodes (here degree means the number of children) are kept quite low: every node has degree at most $O(\log n)$ and the size of a subtree rooted in a node of degree *k* is at least F_{k+2}, where F_k is the *k*th Fibonacci number. This is achieved by the rule that we can cut at most one child of each non-root node. When a second child is cut, the node itself needs to be cut from its parent and becomes the root of a new tree (see Proof of degree bounds, below). The number of trees is decreased in the operation *delete minimum*, where trees are linked together.

As a result of a relaxed structure, some operations can take a long time while others are done very quickly. For the amortized running time analysis we use the potential method, in that we pretend that very fast operations take a little bit longer than they actually do. This additional time is then later combined and subtracted from the actual running time of slow operations. The amount of time saved for later use is measured at any given moment by a potential function. The potential of a Fibonacci heap is given by

Potential = $t + 2m$

where t is the number of trees in the Fibonacci heap, and m is the number of marked nodes. A node is marked if at least one of its children was cut since this node was made a child of another node (all roots are unmarked). The amortized time for an operation is given by the sum of the actual time and c times the difference in potential, where c is a constant (chosen to match the constant factors in the O notation for the actual time).

Thus, the root of each tree in a heap has one unit of time stored. This unit of time can be used later to link this tree with another tree at amortized time 0. Also, each marked node has two units of time stored. One can be used to cut the node from its parent. If this happens, the node becomes a root and the second unit of time will remain stored in it as in any other root.

Implementation of operations [edit]

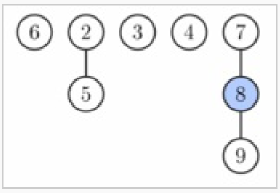
To allow fast deletion and concatenation, the roots of all trees are linked using a circular, doubly linked list. The children of each node are also linked using such a list. For each node, we maintain its number of children and whether the node is marked. Moreover we maintain a pointer to the root containing the minimum key.

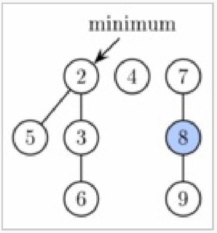
Operation **find minimum** is now trivial because we keep the pointer to the node containing it. It does not change the potential of the heap, therefore both actual and amortized cost are constant.

As mentioned above, **merge** is implemented simply by concatenating the lists of tree roots of the two heaps. This can be done in constant time and the potential does not change, leading again to constant amortized time.

Operation **insert** works by creating a new heap with one element and doing merge. This takes constant time, and the potential increases by one, because the number of trees increases. The amortized cost is thus still constant.

Operation **extract minimum** (same as *delete minimum*) operates in three phases. First we take the root containing the minimum element and remove it. Its children will become roots of new trees. If the number of children was d, it takes time $O(d)$ to process all new roots and the potential increases by $d-1$. Therefore the amortized running time of this phase is $O(d) = O(\log n)$.


Fibonacci heap from Figure 1 after first phase of extract minimum. Node with key 1 (the minimum) was deleted and its children were added as separate trees.

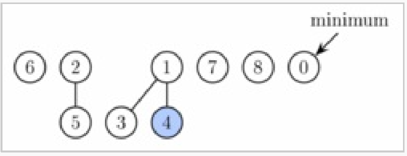


Fibonacci heap from Figure 1 after extract minimum is completed. First, nodes 3 and 6 are linked together. Then the result is linked with tree rooted at node 2. Finally, the new minimum is found.

However to complete the extract minimum operation, we need to update the pointer to the root with minimum key. Unfortunately there may be up to n roots we need to check. In the second phase we therefore decrease the number of roots by successively linking together roots of the same degree. When two roots u and v have the same degree, we make one of them a child of the other so that the one with the smaller key remains the root. Its degree will increase by one. This is repeated until every root has a different degree. To find trees of the same degree efficiently we use an array of length $O(\log n)$ in which we keep a pointer to one root of each degree. When a second root is found of the same degree, the two are linked and the array is updated. The actual running time is $O(\log n + m)$ where m is the number of roots at the beginning of the second phase. At the end we will have at most $O(\log n)$ roots (because each has a different degree). Therefore the difference in the potential function from before this phase to after it is: $O(\log n) - m$, and the amortized running time is then at most $O(\log n + m) + c(O(\log n) - m)$. With a sufficiently large choice of c, this simplifies to $O(\log n)$.

In the third phase we check each of the remaining roots and find the minimum. This takes $O(\log n)$ time and the potential does not change. The overall amortized running time of extract minimum is therefore $O(\log n)$.

Operation **decrease key** will take the node, decrease the key and if the heap property becomes violated (the new key is smaller than the key of the parent), the node is cut from its parent. If the parent is not a root, it is marked. If it has been marked already, it is cut as well and its parent is marked. We continue upwards until we reach either the root or an unmarked node. Now we set the minimum pointer to the decreased value if it is the new minimum. In the process we create some number, say k, of new trees. Each of these new trees except possibly the first one was marked originally but as a root it will become unmarked. One node can become marked. Therefore the number of marked nodes changes by $-(k-1) + 1 = -k + 2$. Combining these 2



Fibonacci heap from Figure 1 after decreasing key of node 9 to 0. This node as well as its two marked ancestors are cut from the tree rooted at 1 and placed as new roots.

changes, the potential changes by $2(-k + 2) + k = -k + 4$. The actual time to perform the cutting was $O(k)$, therefore (again with a sufficiently large choice of c) the amortized running time is constant.

Finally, operation **delete** can be implemented simply by decreasing the key of the element to be deleted to minus infinity, thus turning it into the minimum of the whole heap. Then we call extract minimum to remove it. The amortized running time of this operation is $O(\log n)$.

Proof of degree bounds [edit]

The amortized performance of a Fibonacci heap depends on the degree (number of children) of any tree root being $O(\log n)$, where n is the size of the heap. Here we show that the size of the (sub)tree rooted at any node x of degree d in the heap must have size at least F_{d+2}, where F_k is the kth Fibonacci number. The degree bound follows from this and the fact (easily proved by induction) that $F_{d+2} \geq \varphi^d$ for all integers $d \geq 0$, where $\varphi = (1 + \sqrt{5})/2 \doteq 1.618$. (We then have $n \geq F_{d+2} \geq \varphi^d$, and taking the log to base φ of both sides gives $d \leq \log_\varphi n$ as required.)

Consider any node x somewhere in the heap (x need not be the root of one of the main trees). Define **size**(x) to be the size of the tree rooted at x (the number of descendants of x, including x itself). We prove by induction on the height of x (the length of a longest simple path from x to a descendant leaf), that **size**(x) ≥ F_{d+2}, where d is the degree of x.

Base case: If x has height 0, then $d = 0$, and **size**(x) = 1 = F_2.

Inductive case: Suppose x has positive height and degree $d>0$. Let $y_1, y_2, ..., y_d$ be the children of x, indexed in order of the times they were most recently made children of x (y_1 being the earliest and y_d the latest), and let $c_1, c_2, ..., c_d$ be their respective degrees. We **claim** that $c_i \geq i-2$ for each i with $2 \leq i \leq d$: Just before y_i was made a child of x, $y_1,...,y_{i-1}$ were already children of x, and so x had degree at least $i-1$ at that time. Since trees are combined only when the degrees of their roots are equal, it must have been that y_i also had degree at least $i-1$ at the time it became a child of x. From that time to the present, y_i can only have lost at most one child (as guaranteed by the marking process), and so its current degree c_i is at least $i-2$. This proves the **claim**.

Since the heights of all the y_i are strictly less than that of x, we can apply the inductive hypothesis to them to get **size**(y_i) ≥ $F_{c_i+2} \geq F_{(i-2)+2} = F_i$. The nodes x and y_1 each contribute at least 1 to **size**(x), and so we have

$$\mathbf{size}(x) \geq 2 + \sum_{i=2}^{d} \mathbf{size}(y_i) \geq 2 + \sum_{i=2}^{d} F_i = 1 + \sum_{i=0}^{d} F_i.$$

A routine induction proves that $1 + \sum_{i=0}^{d} F_i = F_{d+2}$ for any $d \geq 0$, which gives the desired lower bound on **size**(x).

Worst case [edit]

Although the total running time of a sequence of operations starting with an empty structure is bounded by the bounds given above, some (very few) operations in the sequence can take very long to complete (in particular delete and delete minimum have linear running time in the worst case). For this reason Fibonacci heaps and other amortized data structures may not be appropriate for real-time systems. It is possible to create a data structure which has the same worst-case performance as the Fibonacci heap has amortized performance.[3][4] One such structure, the Brodal queue, is, in the words of the creator, "quite complicated" and "[not] applicable in practice." Created in 2012, the strict Fibonacci heap is a simpler (compared to Brodal's) structure with the same worst-case bounds. It is unknown whether the strict Fibonacci heap is efficient in practice. The run-relaxed heaps of Driscoll et al. give good worst-case performance for all Fibonacci heap operations except merge.

Summary of running times [edit]

In the following time complexities[1] $O(f)$ is an asymptotic upper bound and $\Theta(f)$ is an asymptotically tight bound (see Big O notation). Function names assume a min-heap.

Operation	Binary[1]	Binomial[1]	Fibonacci[1]	Pairing[5]	Brodal[6][a]	Rank-pairing[8]	Strict Fibonacci[9]
find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
delete-min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$[b]	$O(\log n)$[b]	$O(\log n)$	$O(\log n)$[b]	$O(\log n)$

insert	$\Theta(\log n)$	$\Theta(1)$[b]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$[b]	$o(\log n)$[b][c]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
merge	$\Theta(m \log n)$[d]	$O(\log n)$[e]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

a. ^ Brodal and Okasaki later describe a persistent variant with the same bounds except for decrease-key, which is not supported. Heaps with n elements can be constructed bottom-up in $O(n)$.[7]
b. ^ *a b c d e f g* Amortized time.
c. ^ Bounded by $\Omega(\log \log n), O(2^{2\sqrt{\log \log n}})$[2][10]
d. ^ n is the size of the larger heap and m is the size of the smaller heap.
e. ^ n is the size of the larger heap.

Practical considerations [edit]

This section requires expansion.
(February 2015)

Fibonacci heaps have a reputation for being slow in practice[11] due to large memory consumption per node and high constant factors on all operations.[12]

References [edit]

1. ^ *a b c d e* Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 20: Fibonacci Heaps, pp.476–497. Third edition p518.
2. ^ *a b c* Fredman, Michael Lawrence; Tarjan, Robert E. (1987). "Fibonacci heaps and their uses in improved network optimization algorithms" (PDF). *Journal of the Association for Computing Machinery* **34** (3): 596–615. doi:10.1145/28869.28874.
3. ^ Gerth Stølting Brodal (1996), "Worst-Case Efficient Priority Queues", *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms* (Society for Industrial and Applied Mathematics): 52–58, doi:10.1145/313852.313883, ISBN 0-89871-366-8, CiteSeerX: 10.1.1.43.8133
4. ^ Brodal, G. S. L.; Lagogiannis, G.; Tarjan, R. E. (2012). *Strict Fibonacci heaps* (PDF). Proceedings of the 44th symposium on Theory of Computing - STOC '12. p. 1177. doi:10.1145/2213977.2214082. ISBN 9781450312455.
5. ^ Iacono, John (2000), "Improved upper bounds for pairing heaps", *Proc. 7th Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science **1851**, Springer-Verlag, pp. 63–77, doi:10.1007/3-540-44985-X_5
6. ^ Brodal, Gerth S. (1996), "Worst-Case Efficient Priority Queues", *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms* (PDF), pp. 52–58
7. ^ Goodrich, Michael T.; Tamassia, Roberto (2004). "7.3.6. Bottom-Up Heap Construction". *Data Structures and Algorithms in Java* (3rd ed.). pp. 338–341.
8. ^ Haeupler, Bernhard; Sen, Siddhartha; Tarjan, Robert E. (2009). "Rank-pairing heaps" (PDF). *SIAM J. Computing*: 1463–1485.
9. ^ Brodal, G. S. L.; Lagogiannis, G.; Tarjan, R. E. (2012). *Strict Fibonacci heaps* (PDF). Proceedings of the 44th symposium on Theory of Computing - STOC '12. p. 1177. doi:10.1145/2213977.2214082. ISBN 9781450312455.
edit
10. ^ Pettie, Seth (2005). "Towards a Final Analysis of Pairing Heaps" (PDF). *Max Planck Institut für Informatik*.
11. ^ http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/FibonacciHeaps.pdf, p. 79
12. ^ http://web.stanford.edu/class/cs166/lectures/07/Small07.pdf, p. 72

External links [edit]

- Java applet simulation of a Fibonacci heap
- MATLAB implementation of Fibonacci heap
- De-recursived and memory efficient C implementation of Fibonacci heap (free/libre software, CeCILL-B license)
- Ruby implementation of the Fibonacci heap (with tests)
- Pseudocode of the Fibonacci heap algorithm
- Various Java Implementations for Fibonacci heap

v · t · e **Data structures** [hide]

Types	Collection · Container
Abstract	Associative array · Double-ended priority queue · Double-ended queue · List · Map · Multimap · Priority queue · Queue · Set (multiset) · Disjoint Sets · Stack

Arrays	Bit array · Circular buffer · Dynamic array · Hash table · Hashed array tree · Sparse array
Linked	Association list · Linked list · Skip list · Unrolled linked list · XOR linked list
Trees	B-tree · Binary search tree (AA · AVL · red-black · self-balancing · splay) · Heap (binary · binomial · **Fibonacci**) · R-tree (R* · R+ · Hilbert) · Trie (Hash tree)
Graphs	Binary decision diagram · Directed acyclic graph · Directed acyclic word graph
List of data structures	

Categories: Fibonacci numbers | Heaps (data structures)