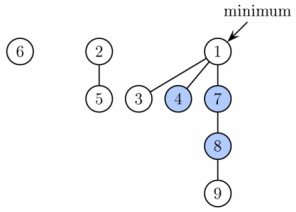
Fibonacci Heap

A **Fibonacci heap** is a specific implementation of the heap data structure that makes use of Fibonacci numbers. Fibo
are used to implement the priority queue element in Dijkstra's algorithm, giving the algorithm a very efficient runnin

Fibonacci heaps have a faster amortized running time than other heap types. Fibonacci heaps are similar to binomial
Fibonacci heaps have a less rigid structure. Binomial heaps merge heaps immediately but Fibonacci heaps wait to me
the `extract-min` function is called. While Fibonacci heaps have very good theoretical complexities, in practice, other h
such as pairing heaps are faster. This is because even in the simplest implementation, Fibonacci heaps require four p
each node, other heaps need two or three.[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-vert

Contents

Structure of Fibonacci Heaps

As the name suggests, Fibonacci heaps use Fibonacci numbers in its structure.

DEFINITION

The Fibonacci numbers are the terms of a sequence of integers in which each term is the sum of the two previous terms with

$$F_1 = F_2 = 1, \quad F_n = F_{n-1} + F_{n-2}. \square$$

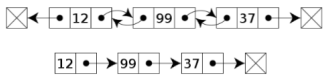
The first few Fibonacci numbers are

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

Like binomial heaps, Fibonacci heaps use doubly linked lists to allow for $O(1)$ time for operations such as splicing of
list, merging two lists, and finding the minimum (or maximum) value. Each node contains a pointer to its parent and
its children. The children are all linked together in a doubly linked list called the child list. Each child in the child list h
for its left sibling and its right sibling.

For each node, the linked list also maintains the number of children a node has, a pointer to the root containing the
key, and whether the node is marked. A node is marked to indicate that it has lost a single child and a node is unmar
loses no children. See the description of the `decrease-key` function in the next section for more details.

Here is an image showing the differences between a singly linked list and a doubly-linked list. The singly linked has c
pointer from each node, while the doubly linked has pointers going both to and from a node.



[3] [4]

In Fibonacci heaps, the degrees of nodes (the number of children) are constrained. Each node in the heap has degre $O(\log n)$, and the size of a subtree rooted in a node of degree k is at least $F_k + 2$, where F_k is the kth Fibonacci nu structure is maintained by having a rule that at most one child can be cut from each non-root node. When a second removed, the node itself needs to be removed from its parent and becomes the root of a new tree. This means that t of trees is decreased in the operation `extract-min`, where trees are linked together.

Fibonacci heaps must satisfy the min-heap property (or max-heap property if making a max-heap) where every nod must have a smaller value than it (the parent), and therefore, the minimum element will always be at the root of one trees.

Minimum Functionalities

Here is how Fibonacci heaps implement the basic functionalities of heaps and the time complexity of each operation children of each node are also related using a linked list. For each node, the linked list maintains the number of child has and whether the node is marked. The linked list also maintains a pointer to the root containing the minimum key marked to indicate if any of its children were removed. This is important so the heap can keep track of how far remov shape is becoming from a binomial heap. If a Fibonacci heap is too different from a binomial heap, it loses many of t time operations that their binomial nature gives it.

These operations are described in terms of a min Fibonacci heap, but they could easily be adapted to be max Fibona operations.

Find Minimum

The linked list has pointers and keeps track of the minimum node, so finding the minimum is simple and can be don constant time.

Merge

In Fibonacci heaps, merging is accomplished by simply concatenating two lists containing the tree roots. Compare th the two heaps to be merged, and whichever is smaller becomes the root of the new combined heap. The other tree is a subtree to this root. This can be done in constant time.

EXAMPLE

Explain why Fibonacci heaps allow for constant time merging. What does this mean for the `insert` operation?

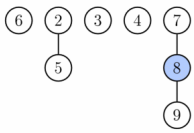
Show Answer

Extract Minimum

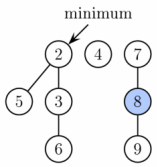
`extract-min` is one of the most important operations regarding Fibonacci heaps. Much of a Fibonacci heap's speed ac comes from the fact that it delays consolidating heaps after operations until `extract-min` is called. Binomial heaps, on hand, consolidate immediately. Consolidation occurs when heap properties are violated, for example, if two heaps ha same order, the heaps must be adjusted to prevent this.

Deleting the minimum element is done in three steps. The node is removed from the root list and the node's childre to the root list. Next, the minimum element is updated if needed. Finally, consolidate the trees so that there are no re orders. If any consolidation occurred, make sure to update the minimum element if needed. Delyaing consolidation

The two images below show the `extract-min` function on the Fibonacci heap shown in the introduction.



Fibonacci heap after the first phase of extract minimum. Node with key 1 (the minimum) was deleted and its children were added as separate trees.[6]



Fibonacci 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.[7]

Insert

Insertion to a Fibonacci heap is similar to the `insert` operation of a binomial heap. A heap of one element is created heaps are merged with the `merge` function. The minimum element pointer is updated if necessary. The total number the tree increases by one.

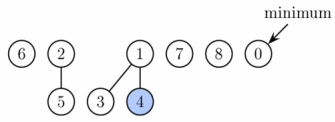
Remove

To delete an element, decrease the key using `decrease key` to negative infinity, and then call `extract-min`. When the no value of negative infinity, since the heap is a min heap, it will become the root of the tree. `extract-min` will remove the element, so doing this deletes the node in question.

Decrease Key

There are two situations that can arise when decreasing the key: the change will cause a heap violation or it will not.

- If the heap properties aren't violated, simply decrease x.
- If a violation does occur, remove the node its parent. If the parent is not a root, mark it. If it has been marked alrea removed as well and its parent is marked, and so on. Continue this process up the tree until either the root or an u node is reached. Next, set the minimum pointer to the decreased value if it is the new minimum.[5]



Fibonacci heap from the introduction 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

The decrease key function marks a node when its child is removed. This allows it to track some history about each no Essentially the marking tracks if[9]:

- The node has had no children removed (unmarked)
- The node has had a single child removed (marked)
- The node is about to have a second child removed (removing a child of a marked node)

Implementation

Here is a pseudocode implementation of Fibonacci heaps[10] [11].

```
1  Make-Fibonacci-Heap()
2  n[H] := 0
3  min[H] := NIL
4  return H
```

```
 5
 6  Fibonacci-Heap-Minimum(H)
 7  return min[H]
 8
 9  Fibonacci-Heap-Link(H,y,x)
10  remove y from the root list of H
11  make y a child of x
12  degree[x] := degree[x] + 1
13  mark[y] := FALSE
14
15  CONSOLIDATE(H)
16  for i:=0 to D(n[H])
17       Do A[i] := NIL
18  for each node w in the root list of H
19      do x:= w
20         d:= degree[x]
21         while A[d] <> NIL
22             do y:=A[d]
23                if key[x]>key[y]
24                  then exchange x<->y
25                Fibonacci-Heap-Link(H, y, x)
26                A[d]:=NIL
27               d:=d+1
28         A[d]:=x
29  min[H]:=NIL
30  for i:=0 to D(n[H])
31      do if A[i]<> NIL
32            then add A[i] to the root list of H
33                 if min[H] = NIL or key[A[i]]<key[min[H]]
34                    then min[H]:= A[i]
35
36  Fibonacci-Heap-Union(H1,H2)
37  H := Make-Fibonacci-Heap()
38  min[H] := min[H1]
39  Concatenate the root list of H2 with the root list of H
40  if (min[H1] = NIL) or (min[H2] <> NIL and min[H2] < min[H1])
41     then min[H] := min[H2]
42  n[H] := n[H1] + n[H2]
43  free the objects H1 and H2
44  return H
45
46
47  Fibonacci-Heap-Insert(H,x)
48  degree[x] := 0
49  p[x] := NIL
50  child[x] := NIL
51  left[x] := x
52  right[x] := x
53  mark[x] := FALSE
54  concatenate the root list containing x with root list H
55  if min[H] = NIL or key[x]<key[min[H]]
56          then min[H] := x
57  n[H]:= n[H]+1
58
59  Fibonacci-Heap-Extract-Min(H)
60  z:= min[H]
61  if x <> NIL
62          then for each child x of z
63               do add x to the root list of H
64                  p[x]:= NIL
65               remove z from the root list of H
66               if z = right[z]
67                  then min[H]:=NIL
68                  else min[H]:=right[z]
69                       CONSOLIDATE(H)
70               n[H] := n[H]-1
71  return z
72
73  Fibonacci-Heap-Decrease-Key(H,x,k)
74  if k > key[x]
```

```
   then error "new key is greater than current key"
key[x] := k
y := p[x]
if y <> NIL and key[x]<key[y]
   then CUT(H, x, y)
        CASCADING-CUT(H,y)
if key[x]<key[min[H]]
   then min[H] := x

CUT(H,x,y)
Remove x from the child list of y, decrementing degree[y]
Add x to the root list of H
p[x]:= NIL
mark[x]:= FALSE

CASCADING-CUT(H,y)
z:= p[y]
if z <> NIL
  then if mark[y] = FALSE
       then mark[y]:= TRUE
       else CUT(H, y, z)
            CASCADING-CUT(H, z)

Fibonacci-Heap-Delete(H,x)
Fibonacci-Heap-Decrease-Key(H,x,-infinity)
Fibonacci-Heap-Extract-Min(H)
```

Python implementations of Fibonacci heaps can be quite long, but here is an example Python implementation.

Summary of the Running Times of Fibonacci Heaps

Operation	Amortized Running Time
Insert	$O(1)$
Remove	$O(\log n)$
Find Min	$O(1)$
Extract Min	$O(\log n)$
Decrease Key	$O(1)$
Merge	$O(1)$

According to a paper[12] written by Michael Fredman (one of the inventors of Fibonacci heaps), Fibonacci heaps have downsides: many computer scientists claim that they are difficult to program and their theoretically excellent running aren't always better in practice than theoretically inferior types of heaps.

See Also

- Binary Heaps
- Binomial Heaps
- Pairing Heaps
- Heap Sort

References

WIKIPEDIA
The Free Encyclopedia

Fibonacci heap

Fibonacci heap	
Type	heap
Invented	1984
Invented by	Michael L. Fredman and Robert Endre Tarjan
Time complexity in big O notation	

Algorithm	Average	Worst case
Insert	$\Theta(1)$	
Find-min	$\Theta(1)$	
Delete-min	$O(\log n)$	
Decrease-key	$\Theta(1)$	
Merge	$\Theta(1)$	

In computer science, a **Fibonacci heap** is a data structure for priority queue operations, consisting of a collection of heap-ordered trees. It has a better amortized running time than many other priority queue data structures including the binary heap and binomial heap. Michael L. Fredman and Robert E. Tarjan developed Fibonacci heaps in 1984 and published them in a scientific journal in 1987. Fibonacci heaps are named after the Fibonacci numbers, which are used in their running time analysis.

For the Fibonacci heap, the find-minimum operation takes constant ($O(1)$) amortized time.[1] The insert and decrease key operations also work in constant amortized time.[2] Deleting an element (most often used in the special case of deleting the minimum element) works in $O(\log n)$ amortized time, where n is the size of the heap.[2] This means that starting from an empty data structure, any sequence of a insert and decrease key operations and b delete operations would take $O(a + b \log n)$ worst case time, where n is the maximum heap size. In a binary or binomial heap, such a sequence of operations would take $O((a + b) \log n)$ time. A Fibonacci heap is thus better than a binary or binomial heap when b is smaller than a by a non-constant factor. It is also possible to merge two Fibonacci heaps in constant amortized time, improving on the logarithmic merge time of a binomial heap, and improving on binary heaps which cannot handle merges efficiently.

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, compared to the same algorithm using other slower priority queue data structures.

Structure

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.

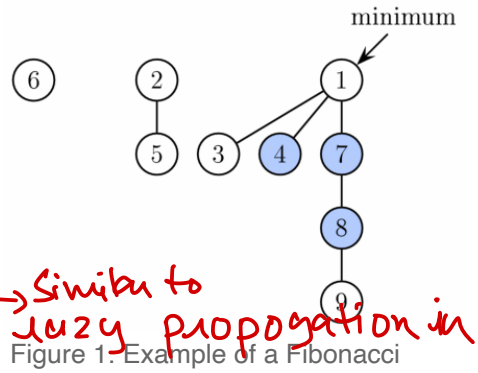


Figure 1: Example of a Fibonacci heap. It has three trees of degrees 0, 1 and 3. Three vertices are marked (shown in blue). Therefore, the potential of the heap is 9 (3 trees + 2 × (3 marked-vertices)).

However, at some point 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 direct

children) are kept quite low: every node has degree at most $\log n$ and the size of a subtree rooted in a node of degree k is at least F_{k+2}, where F_k is the kth 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

$$\text{Potential} = t + 2m$$

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

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 $O(d) = O(\log n)$.

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 $O(\log n)$.

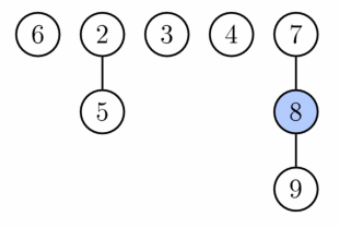


Figure 2. Fibonacci heap from Figure 1 after first phase of extract minimum. Node with key 1 (the minimum) was deleted and its children were added as separate trees.

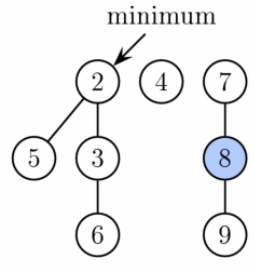


Figure 3. Fibonacci heap from Figure 1 after extract minimum is completed. First, nodes 3 and 6 are linked together. Then the result is linked with tree rooted at node 2. Finally, the new minimum is found.

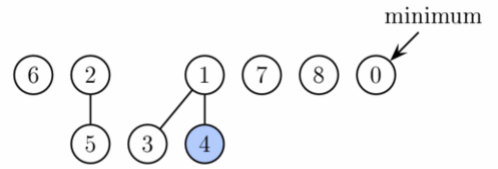


Figure 4. Fibonacci heap from Figure 1 after decreasing key of node 9 to 0. This node as well as its two marked ancestors are cut from the tree rooted at 1 and placed as new roots.

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 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

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

Although Fibonacci heaps look very efficient, they have the following two drawbacks:[3]

1. They are complicated to implement.
2. They are not as efficient in practice when compared with theoretically less efficient forms of heaps. In their simplest version they require storage and manipulation of four pointers per node, whereas only two or three pointers per node are needed in other structures, such as the binary heap, binomial heap, pairing heap, Brodal queue and rank-pairing heap.

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. One such structure, the Brodal

queue,[4] is, in the words of the creator, "quite complicated" and "[not] applicable in practice." Created in 2012, the strict Fibonacci heap[5] is a simpler (compared to Brodal's) structure with the same worst-case bounds. Despite having simpler structure, experiments show that in practice the strict Fibonacci heap performs slower than more complicated Brodal queue and also slower than basic Fibonacci heap.[6][7] The run-relaxed heaps of Driscoll et al. give good worst-case performance for all Fibonacci heap operations except merge.[8]

Summary of running times

Here are time complexities[9] of various heap data structures. Function names assume a min-heap. For the meaning of "$O(f)$" and "$\Theta(f)$" see Big O notation.

Operation	find-min	delete-min	insert	decrease-key	meld
Binary[9]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[9][10]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[a]	$\Theta(\log n)$	$O(\log n)$
Skew binomial[11]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$[b]
Pairing[12]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$o(\log n)$[a][c]	$\Theta(1)$
Rank-pairing[15]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$\Theta(1)$[a]	$\Theta(1)$
Fibonacci[9][2]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$\Theta(1)$[a]	$\Theta(1)$
Strict Fibonacci[16]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Brodal[17][d]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2–3 heap[19]	$O(\log n)$	$O(\log n)$[a]	$O(\log n)$[a]	$\Theta(1)$	*?*

a. Amortized time.
b. Brodal and Okasaki describe a technique to reduce the worst-case complexity of *meld* to $\Theta(1)$; this technique applies to any heap datastructure that has *insert* in $\Theta(1)$ and *find-min*, *delete-min*, *meld* in $O(\log n)$.
c. Lower bound of $\Omega(\log \log n)$,[13] upper bound of $O(2^{2\sqrt{\log \log n}})$.[14]
d. 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 $O(n)$.[18]

Practical considerations

Fibonacci heaps have a reputation for being slow in practice[20] due to large memory consumption per node and high constant factors on all operations. Recent experimental results suggest that Fibonacci heaps are more efficient in practice than most of its later derivatives, including quake heaps, violation heaps, strict Fibonacci heaps, rank pairing heaps, but less efficient than either pairing heaps or array-based heaps.[7]

References

1. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990]. "Chapter 20: Fibonacci Heaps". *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. pp. 476–497. ISBN 0-262-03293-7. Third edition p. 518.