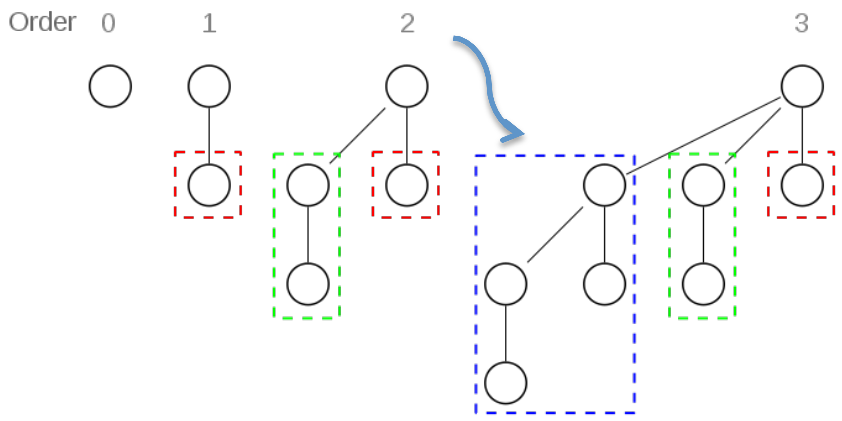
Binomial heap and Fibonacci heap (Cormen et. al. Chapter 19-20; also from Wikipedia https://en.wikipedia.org/wiki/Binomial_heap and https://en.wikipedia.org/wiki/Fibonacci_heap)

Binomial heap

A heap (*priority queue*) supporting quick merging of two heaps. It is implemented as a collection of binomial trees, defined recursively: 1) a binomial tree of order 0 is a single node; 2) a binomial tree of order *k* has a root node whose children are roots of binomial trees of orders *k*−1, *k*−2, ..., 2, 1, 0 (in this order).

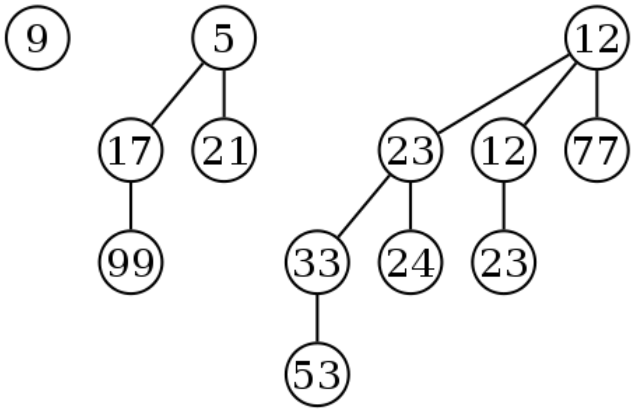


A binomial tree of order *k* has 2^k nodes, height *k*.
Because of its unique structure, a binomial tree of order *k* can be constructed from two trees of order *k*−1 trivially by attaching one of them as the leftmost child of the root of the other tree. This feature is a major advantage over other conventional heaps for merging two heaps.

A binomial tree of order n has $\binom{n}{d}$ nodes at depth d.

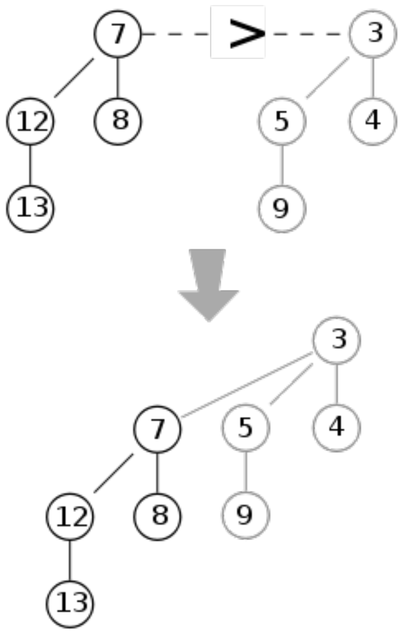
Levels / Order	1	2	3	4	5	6
1	1					
2	1	1				
3	1	2	1			
4	1	3	3	1		
5	1	4	6	4	1	
…						

In a binomial heap, 1) each binomial tree is implemented as a min-heap (with the *min-heap property*: the key of a node is greater than or equal to the key of its parent), and 2) there can only be either *one* or *zero* binomial trees for each order, including the order 0. The first property ensures that the root of each binomial tree contains the smallest key in the tree, which applies to the entire heap. The second property implies that a binomial heap with *n* nodes consists of at most (log n) +1 binomial trees. In fact, the number and orders of these trees are uniquely determined by the number of nodes *n*: each binomial tree corresponds to one digit in the binary representation of number *n*. For example number 13 is 1101 in binary, and thus a binomial heap with 13 nodes will consist of three binomial trees of orders 3, 2, and 0.



The roots of the binomial trees can be stored in a linked list, ordered by increasing order of the tree.

Merge two binomial heaps: merge the binomial trees respectively (if both are present). As their root node is the smallest element within the tree, by comparing the two keys of the roots, the smaller of them is the minimum key, and becomes the new root node. Then the other tree becomes a subtree of the combined tree.



If only one of the heaps contains a tree of order *j*, this tree is moved to the merged heap. If both heaps contain a tree of order *j*, the two trees are merged to one tree of order *j*+1 so that the minimum-heap property is satisfied. Note that it may later be necessary to merge this tree with some other tree of order j+1 present in one of the heaps. In the course of the algorithm, we need to examine at most three trees of any order (two from the two heaps we merge and one composed of two smaller trees).

Because each binomial tree in a binomial heap corresponds to one bit in the binary representation of its size, there is an analogy between the merging of two heaps and the binary addition of the *sizes* of the two heaps, from right-to-left (smallest digit to greatest digit). Whenever a carry occurs during addition, this corresponds to a merging of two binomial trees during the merge.

Each tree has order at most log n and therefore the running time is $O(\log n)$.

Insert an element: Creating a new heap containing only this element and then merging it with the original heap. Due to the merge, insert takes O(log n) time. However, across a series of n consecutive insertions, **insert** has an *amortized* time of O(1) (similar as the incrementing a binary counter).

Find minimum in the heap: Find the minimum among the roots of the binomial trees. This can again be done easily in $O(\log n)$ time, as there are just $O(\log n)$ trees. If we add a link to the root with the minimum key among all O(log n) binomial trees, the minimum can be retrieved in O(1) time. The link can be updated in O(1) amortized cost when an element is inserted, and in O(log n) time when two binomial heaps are merged.

Delete the minimum element from the heap (*Extract-min or delete-minimum*), First find this element, remove it from its binomial tree, and transform its subtrees into a separate binomial heap. Then merge this heap with the original heap. Since each tree has at most log n children, creating this new heap is $O(\log n)$. Merging heaps is $O(\log n)$, so the entire delete minimum operation is $O(\log n)$.

Decrease the key of an element. After decreasing the key, min-heapify the corresponding binary tree (heap), i.e., exchange the element with its parent, and possibly also with its grandparent, and so on, until the heap becomes a min-heap. Each binomial tree has height at most log n, so this takes $O(\log n)$ time.

Increase the key of an element. Similarly.

Delete an element from the heap. Increase its key to negative infinity (that is, some value smaller than any element in the heap) and then delete the minimum in the heap. O(log n) time.

Fibonacci heap

Fibonacci heaps support the merging operation but have the advantage that operations except deleting the minimum element run in **O**(1) amortized time (i.e., to reduce the amortized cost of decreasing the key of an element from O(log n) to O(1)).

To start off building the Fibonacci heap, we begin with a binomial heap and modify it try to make insertions take time O(1). The worst-case runtime of inserting a into a binomial heap is Θ(log n), because we might have Θ(log n) trees that need to get merged together. Those trees need to be merged together only because we need to keep the number of trees low when attempting to find (and delete) the minimum key in the heap (priority queue).

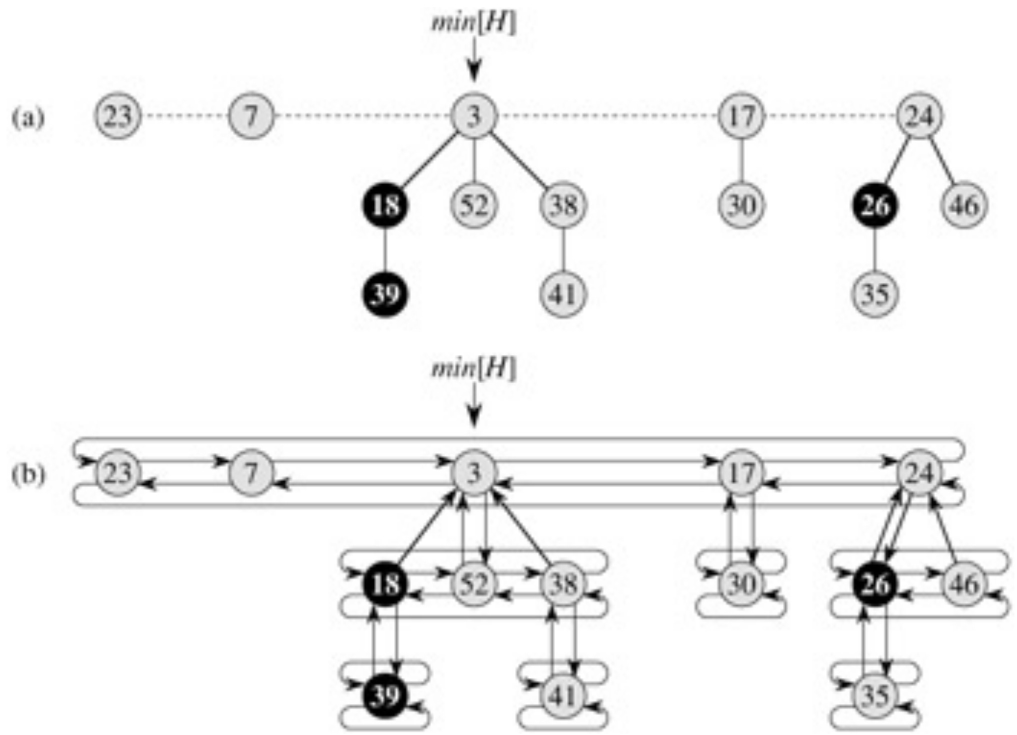
Modification 1: When inserting a node into the heap, just create a tree of order 0 and add it to the existing collection of trees.

Modification 2: When merging two heaps together, just combine all their trees together without doing any merging.

Modification 3: Keep a pointer to the minimum element among all roots in a heap. It can be updated in O(1) time when insertion or merging is done.

Modification 3: When the extract-*min* (find and delete the minimum element) is operated, consolidate (merge) all trees (including the subtrees of the deleted node) to ensure there is at most one tree of each order.

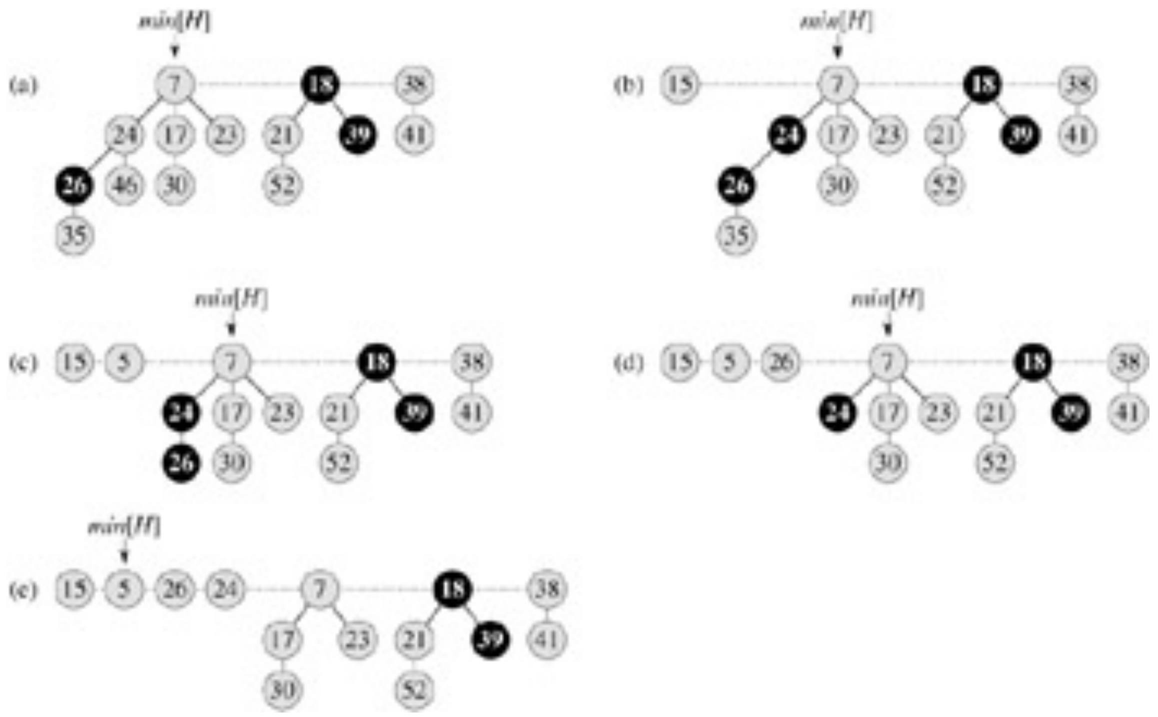
Note that unlike the binomial heap, the Finonacci heap, there's no guarantee that the trees will be in any order. The following algorithm can be used to merge trees so that there is at most one tree per order. Suppose we maintain a hash table that maps from tree orders to trees, starting from an empty set of trees and going through one tree after another. We could then do the following operation for each tree in the data structure: 1) see if there's already a tree of the order of the given tree; 2) if not, insert the current tree into the hash table; 3) otherwise, merge the given tree with the tree of that order, removing the old tree from the hash table. This process can be recursively carried out for each tree. Obviously, when we're done, there's at most one tree of each order. Also, it is quite efficient, the merge is conducted for each tree that is NOT in the set of (Θ(n)) trees at the end of the process.



The costs of the operations can be analyzed by amortized analysis using a potential function Φ that is the number of trees in the data structure.

- **Insert**: increases the potential by 1. Amortized cost O(1).
- **Merge**: no net change in potential. Amortized cost O(1).
- **Extract-Min**: O(#trees + #merges) work and decreases the potential down to Θ(log n), the number of trees in the binomial tree. Let's have the number of trees as Θ(log n) + E, where E is the "excess" number of trees. After the consolidation, the potential function reduced from Θ(log n) + E to Θ(log n). Note that we will do one merge per excess tree, and so the total work done is Θ(log n + E). Therefore, the amortized cost of a extract-min is Θ(log n).

Modification 5: To decrease the key of a node and, if its key is now smaller than its parent's key (thus violate the min-heap property), cut it and add it to the root list (i.e., create a new tree rooted by the specific node).
Modification 6: Assign a mark bit to each node that is initially false. When a child is cut from an unmarked parent, mark the parent. When a child is cut from a marked parent, unmark the parent and cut the parent from its parent.



Two calls of **(a)** The initial Fibonacci heap. **(b)** The node with key 46 has its key decreased to 15. The node becomes a root, and its parent (with key 24), which had previously been unmarked, becomes marked. **(c)-(e)** The node with key 35 has its key decreased to 5. In part (c), the node, now with key 5, becomes a root. Its parent, with key 26, is marked, so a cascading cut occurs. The node with key 26 is cut from its parent and made an unmarked root in (d). Another cascading cut occurs, since the node with key 24 is marked as well. This node is cut from its parent and made an unmarked root in part (e). The cascading cuts stop at this point, since the node with key 7 is a root. (Even if this node were not a root, the cascading cuts would stop, since it is unmarked.)

The simple cut takes O(1) because we just cut the tree rooted at the appropriate node and moved it to the root list. However, now we might have to do a "cascading cut," in which we cut a node from its parent, then cut *that* node from *its* parent, etc. We use the accounting method of the amortized analysis. We can add a charge of 1 to each decrease-key operation that we can then spend to cut the parent node from its parent. Since we only cut a node from its parent if it has already lost two children, we can pre-pay each decrease-key operation for the work necessary to cut its parent node later. When we do cut the parent, we can charge the cost of doing so back to one of the earlier decrease-key operations. Consequently, even though any individual decrease-key operation might take a long time to finish, we can always amortize the work across the earlier calls so that the runtime is amortized O(1).

Decrease the key of an element: O(1) amortized time
Delete an element: first decrease the key of the element to negative infinite, and then delete the minimum element. O(log n).

Name of Fibonacci heap comes from the fact that the number of nodes in an **n**-node

Fibonacci heap with degree k is at least F_{k+2}, where F_k is the kth Finonacci number. Therefore, as a result, the maximum degree D(n) of any node in an *n*-node Fibonacci heap is O(log n).

Operation	**Binary**	**Binomial**	**Fibonacci**
find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
extract-min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
insert	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
merge	$\Theta(m \log n)$	$O(\log n)$	$\Theta(1)$

Application of Finonacci heaps

Prim algorithm for MST

```
1. MST ← an arbitrary v
2. Make a priority queue for all nodes with the priority of node z: p(z) ← w(z, v) (or ∞ if (z,v) ∉ E)
3. while MST does not contain all vertices
4.     Extract the node u with the minimum priority from the heap //O(log n)
5.     add v (and corresponding edge with the minimum priority) to MST
6.     for each vertex x incident to u
7.         if(p(x) > w(x, u))      decrease the priority p(x) to w(x, u)  //O(1)
```

ln 7: amortized cost O(1) if the priority queue is implemented in Fibonacci heap. Overall O(m + n log n); otherwise O(m log n + n log n)

Dijkstra's algorithm

```
Input: G(V, E), s
1.  S ← Ø
2.  Make a priority queue for all nodes with the priority of node z, p(z) ← d(z, v) (or ∞ if (z,v) ∉ E)
3. while Q ≠ Ø
4.     Extract the node u with the minimum priority from the heap to v //O(log n)
5.     add v (and corresponding edge with the minimum priority) into S
6.     for each vertex x incident to u
7.         if(p(x) > d(x, u))      decrease the priority p(x) to d(x, u)  //O(1)
```