WIKIPEDIA

Binary heap

A **binary heap** is a heap data structure that takes the form of a binary tree. Binary heaps are a common way of implementing priority queues.[1]:162–163 The binary heap was introduced by J. W. J. Williams in 1964, as a data structure for heapsort.[2]

Binary Heap		
Type	tree	
Time complexity in big O notation		
Algorithm	**Average**	**Worst case**
Space	O(n)	O(n)
		)
		og n)
		og n)
		)

A binary heap is defined as a binary tree with two additional constraints:[3]

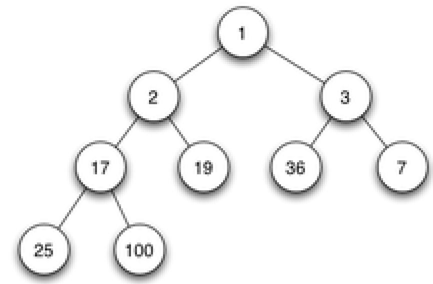
- Shape property: a binary heap is a *complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is no nodes of that level are filled from left to right.
- Heap property: the key stored in each node is either greater than or equal to (≥) o equal to (≤) the keys in the node's children, according to some total order.

John William Joseph Williams was a Welsh-Canadian computer scientist best known for inventing in 1964 heapsort and the binary heap data structure. He was born in Chippenham, Wiltshire and spent the latter part of his career in Canada, moving to Ka

Heaps where the parent key is greater than or equal to (≥) the child keys are called m where it is less than or equal to (≤) are called *min-heaps*. Efficient (logarithmic time) known for the two operations needed to implement a priority queue on a binary he element, and removing the smallest or largest element from a min-heap or max-hea Binary heaps are also commonly employed in the heapsort sorting algorithm, whic algorithm because binary heaps can be implemented as an implicit data structure, st array and using their relative positions within that array to represent child-parent relationships.

Example of a complete binary max heap

Contents



Example of a complete binary min heap

Heap operations

Both the insert and remove operations modify the heap to conform to the shape property first, by adding or removing from the end of the heap. Then the heap property is restored by traversing up or down the heap. Both operations take O(log n) time.

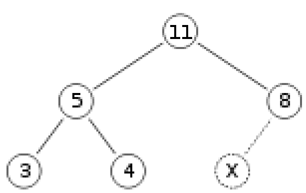
Insert

To add an element to a heap we must perform an *up-heap* operation (also known as *bubble-up*, *percolate-up*, *sift-up*, *trickle-up*, *swim-up*, *heapify-up*, or *cascade-up*), by following this algorithm:

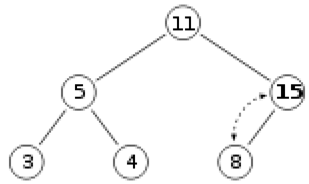
1. Add the element to the bottom level of the heap at the most left.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

The number of operations required depends only on the number of levels the new element must rise to satisfy the heap property, thus the insertion operation has a worst-case time complexity of O(log n) but an average-case complexity of O(1).

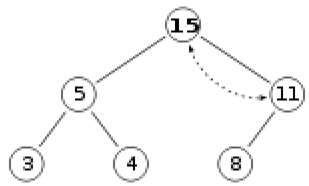
As an example of binary heap insertion, say we have a max-heap



and we want to add the number 15 to the heap. We first place the 15 in the position marked by the X. However, the heap property is violated since 15 > 8, so we need to swap the 15 and the 8. So, we have the heap looking as follows after the first swap:



However the heap property is still violated since 15 > 11, so we need to swap again:



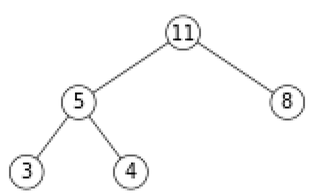
which is a valid max-heap. There is no need to check the left child after this final step: at the start, the max-heap was valid, meaning 11 > 5; if 15 > 11, and 11 > 5, then 15 > 5, because of the transitive relation.

Extract

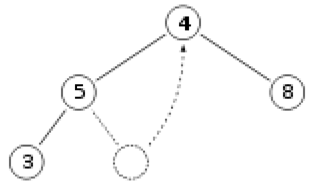
The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called *down-heap* (also known as *bubble-down*, *percolate-down*, *sift-down*, *sink-down*, *trickle down*, *heapify-down*, *cascade-down*, and *extract-min/max*).

1. Replace the root of the heap with the last element on the last level.
2. Compare the new root with its children; if they are in the correct order, stop.
3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

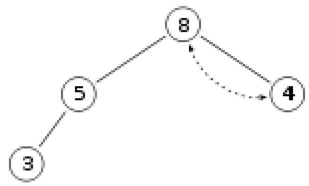
So, if we have the same max-heap as before



We remove the 11 and replace it with the 4.



Now the heap property is violated since 8 is greater than 4. In this case, swapping the two elements, 4 and 8, is enough to restore the heap property and we need not swap elements further:



The downward-moving node is swapped with the *larger* of its children in a max-heap (in a min-heap it would be swapped with its smaller child), until it satisfies the heap property in its new position. This functionality is achieved by the **Max-Heapify** function as defined below in pseudocode for an array-backed heap *A* of length *heap_length*[*A*]. Note that "A" is indexed starting at 1.

```
Max-Heapify (A, i):
    left ← 2×i     // ← means "assignment"
    right ← 2×i + 1
    largest ← i

    if left ≤ heap_length[A] and A[left] > A[largest] then:
        largest ← left

    if right ≤ heap_length[A] and A[right] > A[largest] then:
        largest ← right

    if largest ≠ i then:
        swap A[i] and A[largest]
        Max-Heapify(A, largest)
```

For the above algorithm to correctly re-heapify the array, the node at index *i* and its two direct children must violate the heap property. If they do not, the algorithm will fall through with no change to the array. The down-heap operation (without the preceding swap) can also be used to modify the value of the root, even when an element is not being deleted. In the pseudocode above, what starts with // is a comment. Note that *A* is an array (or list) that starts being indexed from *1* up to *length(A)*, according to the pseudocode.

In the worst case, the new root has to be swapped with its child on each level until it reaches the bottom level of the heap, meaning that the delete operation has a time complexity relative to the height of the tree, or O(log *n*).

Building a heap

Building a heap from an array of n input elements can be done by starting with an empty heap, then successively inserting each element. This approach, called Williams' method after the inventor of binary heaps, is easily seen to run in $O(n \log n)$ time: it performs n insertions at $O(\log n)$ cost each.[a]

However, Williams' method is suboptimal. A faster method (due to Floyd[4]) starts by arbitrarily putting the elements on a binary tree, respecting the shape property (the tree could be represented by an array, see below). Then starting from the lowest level and moving upwards, shift the root of each subtree downward as in the deletion algorithm until the heap property is restored. More specifically if all the subtrees starting at some height h have already been "heapified" (the bottommost level corresponding to $h = 0$), the trees at height $h + 1$ can be heapified by sending their root down along the path of maximum valued children when building a max-heap, or minimum valued children when building a min-heap. This process takes $O(h)$ operations (swaps) per node. In this method most of the heapification takes place in the lower levels. Since the height of the heap is $\lfloor \log n \rfloor$, the number of nodes at height h is $\leq \frac{2^{\lfloor \log n \rfloor}}{2^h} \leq \frac{n}{2^h}$. Therefore, the cost of heapifying all subtrees is:

$$\begin{aligned}\sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^h} O(h) &= O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) \\ &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n)\end{aligned}$$

This uses the fact that the given infinite series $\sum_{i=0}^{\infty} i/2^i$ converges.

The exact value of the above (the worst-case number of comparisons during the heap construction) is known to be equal to:

$$2n - 2s_2(n) - e_2(n),$$ [5][b]

where $s_2(n)$ is the sum of all digits of the binary representation of n and $e_2(n)$ is the exponent of 2 in the prime factorization of n.

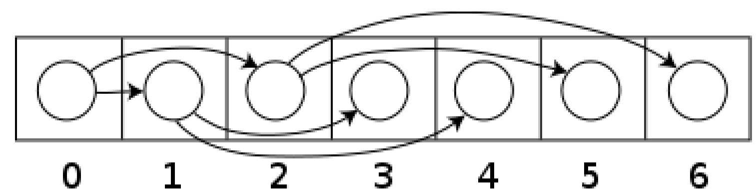
The average case is more complex to analyze, but it can be shown to asymptotically approach $1.8814\ n - 2\ \log_2 n + O(1)$ comparisons.[6][7]

The **Build-Max-Heap** function that follows, converts an array *A* which stores a complete binary tree with *n* nodes to a max-heap by repeatedly using **Max-Heapify** in a bottom up manner. It is based on the observation that the array elements indexed by *floor*(n/2) + 1, *floor*(n/2) + 2, ..., n are all leaves for the tree (assuming that indices start at 1), thus each is a one-element heap. **Build-Max-Heap** runs **Max-Heapify** on each of the remaining tree nodes.

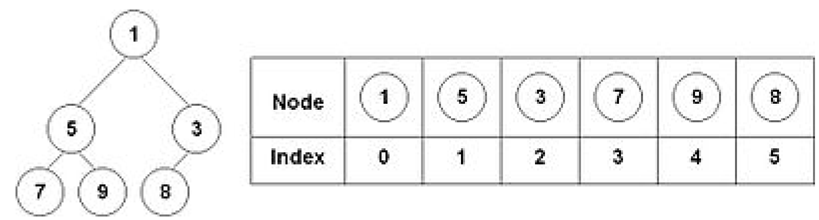
```
Build-Max-Heap (A):
    heap_length[A] ← length[A]

    for each index i from floor(length[A]/2) downto 1 do:
        Max-Heapify(A, i)
```

Heap implementation

Heaps are commonly implemented with an array. Any binary tree can be stored in an array, but because a binary heap is always a complete binary tree, it can be stored compactly. No space is required for pointers; instead, the parent and children of each node can be found by arithmetic on array indices. These properties make this heap implementation a simple example of an implicit data structure or Ahnentafel list. Details depend on the root position, which in turn may depend on constraints of a programming language used for implementation, or programmer preference. Specifically, sometimes the root is placed at index 1, in order to simplify arithmetic.



A small complete binary tree stored in an array



Comparison between a binary heap and an array implementation.

Let n be the number of elements in the heap and i be an arbitrary valid index of the array storing the heap. If the tree root is at index 0, with valid indices 0 through $n - 1$, then each element a at index i has

- children at indices $2i + 1$ and $2i + 2$
- its parent at index *floor*($(i - 1)/2$).

Alternatively, if the tree root is at index 1, with valid indices 1 through n, then each element a at index i has

- children at indices $2i$ and $2i + 1$
- its parent at index *floor*($i/2$).

This implementation is used in the heapsort algorithm, where it allows the space in the input array to be reused to store the heap (i.e. the algorithm is done in-place). The implementation is also useful for use as a Priority queue where use of a dynamic array allows insertion of an unbounded number of items.

The upheap/downheap operations can then be stated in terms of an array as follows: suppose that the heap property holds for the indices b, $b+1$, ..., e. The sift-down function extends the heap property to $b-1$, b, $b+1$, ..., e. Only index $i = b-1$ can violate the heap property. Let j be the index of the largest child of $a[i]$ (for a max-heap, or the smallest child for a min-heap) within the range b, ..., e. (If no such index exists because $2i > e$ then the heap property holds for the newly extended range and nothing needs to be done.) By swapping the values $a[i]$ and $a[j]$ the heap property for position i is established. At this point, the only problem is that the heap property might not hold for index j. The sift-down function is applied tail-recursively to index j until the heap property is established for all elements.

The sift-down function is fast. In each step it only needs two comparisons and one swap. The index value where it is working doubles in each iteration, so that at most $\log_2 e$ steps are required.

For big heaps and using virtual memory, storing elements in an array according to the above scheme is inefficient: (almost) every level is in a different page. B-heaps are binary heaps that keep subtrees in a single page, reducing the number of pages accessed by up to a factor of ten.[8]

The operation of merging two binary heaps takes $\Theta(n)$ for equal-sized heaps. The best you can do is (in case of array implementation) simply concatenating the two heap arrays and build a heap of the result.[9] A heap on n elements can be merged with a heap on k elements using $O(\log n \log k)$ key comparisons, or, in case of a pointer-based implementation, in $O(\log n \log k)$ time.[10] An algorithm for splitting a heap on n elements into two heaps on k and n-k elements, respectively, based on a new view of heaps as an ordered collections of subheaps was presented in.[11] The algorithm requires $O(\log n * \log n)$ comparisons. The view also presents a new and conceptually simple algorithm for merging heaps. When merging is a common task, a different heap implementation is recommended, such as binomial heaps, which can be merged in $O(\log n)$.

Additionally, a binary heap can be implemented with a traditional binary tree data structure, but there is an issue with finding the adjacent element on the last level on the binary heap when adding an element. This element can be determined algorithmically or by adding extra data to the nodes, called "threading" the tree—instead of merely storing references to the children, we store the inorder successor of the node as well.

It is possible to modify the heap structure to allow extraction of both the smallest and largest element in $O(\log n)$ time.[12] To do this, the rows alternate between min heap and max heap. The algorithms are roughly the same, but, in each step, one must consider the alternating rows with alternating comparisons. The performance is roughly the same as a normal single direction heap. This idea can be generalised to a min-max-median heap.

Derivation of index equations

In an array-based heap, the children and parent of a node can be located via simple arithmetic on the node's index. This section derives the relevant equations for heaps with their root at index 0, with additional notes on heaps with their root at index 1.

To avoid confusion, we'll define the **level** of a node as its distance from the root, such that the root itself occupies level 0.

Child nodes

For a general node located at index i (beginning from 0), we will first derive the index of its right child, $\text{right} = 2i + 2$.

Let node i be located in level L, and note that any level l contains exactly 2^l nodes. Furthermore, there are exactly $2^{l+1} - 1$ nodes contained in the layers up to and including layer l (think of binary arithmetic; 0111...111 = 1000...000 - 1). Because the root is stored at 0, the kth node will be stored at index $(k - 1)$. Putting these observations together yields the following expression for the **index of the last node in layer l**.

$$\text{last}(l) = (2^{l+1} - 1) - 1 = 2^{l+1} - 2$$

Let there be j nodes after node i in layer L, such that

$$\begin{aligned} i = & \quad \text{last}(L) - j \\ = & \quad (2^{L+1} - 2) - j \end{aligned}$$

Each of these j nodes must have exactly 2 children, so there must be $2j$ nodes separating i's right child from the end of its layer $(L + 1)$.

$$\begin{aligned} \text{right} = & \quad \text{last(L + 1)} - 2j \\ = & \quad (2^{L+2} - 2) - 2j \\ = & \quad 2(2^{L+1} - 2 - j) + 2 \\ = & \quad 2i + 2 \end{aligned}$$

As required.

Noting that the left child of any node is always 1 place before its right child, we get $\text{left} = 2i + 1$.

If the root is located at index 1 instead of 0, the last node in each level is instead at index $2^{l+1} - 1$. Using this throughout yields $\text{left} = 2i$ and $\text{right} = 2i + 1$ for heaps with their root at 1.

Parent node

Every node is either the left or right child of its parent, so we know that either of the following is true.

1. $i = 2 \times (\text{parent}) + 1$
2. $i = 2 \times (\text{parent}) + 2$

Hence,

$$\text{parent} = \frac{i-1}{2} \text{ or } \frac{i-2}{2}$$

Now consider the expression $\left\lfloor \dfrac{i-1}{2} \right\rfloor$.

If node i is a left child, this gives the result immediately, however, it also gives the correct result if node i is a right child. In this case, $(i-2)$ must be even, and hence $(i-1)$ must be odd.

$$\begin{aligned}\left\lfloor \frac{i-1}{2} \right\rfloor &= \left\lfloor \frac{i-2}{2} + \frac{1}{2} \right\rfloor \\ &= \frac{i-2}{2} \\ &= \text{parent}\end{aligned}$$

Therefore, irrespective of whether a node is a left or right child, its parent can be found by the expression:

$$\text{parent} = \left\lfloor \frac{i-1}{2} \right\rfloor$$

Related structures

Since the ordering of siblings in a heap is not specified by the heap property, a single node's two children can be freely interchanged unless doing so violates the shape property (compare with treap). Note, however, that in the common array-based heap, simply swapping the children might also necessitate moving the children's sub-tree nodes to retain the heap property.

The binary heap is a special case of the d-ary heap in which d = 2.

Summary of running times

Here are time complexities[13] 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[13]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[13][14]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[c]	$\Theta(\log n)$	$O(\log n)$[d]
Fibonacci[13][15]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$
Pairing[16]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$o(\log n)$[c][e]	$\Theta(1)$
Brodal[19][f]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[21]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$
Strict Fibonacci[22]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2-3 heap[23]	$O(\log n)$	$O(\log n)$[c]	$O(\log n)$[c]	$\Theta(1)$	?

a. In fact, this procedure can be shown to take $\Theta(n \log n)$ time in the worst case, meaning that $n \log n$ is also an asymptotic lower bound on the complexity.[1]:167 In the *average case* (averaging over all permutations of n inputs), though, the method takes linear time.[4]
b. This does not mean that *sorting* can be done in linear time since building a heap is only the first step of the heapsort algorithm.
c. Amortized time.
d. n is the size of the larger heap.
e. Lower bound of $\Omega(\log \log n)$,[17] upper bound of $O(2^{2\sqrt{\log \log n}})$.[18]
f. 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 $O(n)$.[20]

See also

- Heap
- Heapsort

References

1. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.
2. Williams, J. W. J. (1964), "Algorithm 232 - Heapsort", *Communications of the ACM*, **7** (6): 347–348, doi:10.1145/512274.512284 (https://doi.org/10.1145%2F512274.512284)

3. eEL,CSA_Dept,IISc,Bangalore, "Binary Heaps" (http://lcm.csa.iisc.ernet.in/dsa/node137.html), *Data Structures and Algorithms* (http://lcm.csa.iisc.ernet.in/dsa/dsa.html)
4. Hayward, Ryan; McDiarmid, Colin (1991). "Average Case Analysis of Heap Building by Repeated Insertion" (https://web.archive.org/web/20160205023201/http://www.stats.ox.ac.uk/__data/assets/pdf_file/0015/4173/heapbuildjalg.pdf) (PDF). *J. Algorithms*. **12**: 126–153. CiteSeerX 10.1.1.353.7888 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.353.7888). doi:10.1016/0196-6774(91)90027-v (https://doi.org/10.1016%2F0196-6774%2891%2990027-v). Archived from the original (http://www.stats.ox.ac.uk/__data/assets/pdf_file/0015/4173/heapbuildjalg.pdf) (PDF) on 2016-02-05. Retrieved 2016-01-28.
5. Suchenek, Marek A. (2012), "Elementary Yet Precise Worst-Case Analysis of Floyd's Heap-Construction Program" (http://www.deepdyve.com/lp/ios-press/elementary-yet-precise-worst-case-analysis-of-floyd-s-heap-50NW30HMxU), *Fundamenta Informaticae*, **120** (1): 75–92, doi:10.3233/FI-2012-751 (https://doi.org/10.3233%2FFI-2012-751).
6. Doberkat, Ernst E. (May 1984). "An Average Case Analysis of Floyd's Algorithm to Construct Heaps" (https://core.ac.uk/download/pdf/82135122.pdf) (PDF). *Information and Control*. **6** (2): 114–131. doi:10.1016/S0019-9958(84)80053-4 (https://doi.org/10.1016%2FS0019-9958%2884%2980053-4).
7. Pasanen, Tomi (November 1996). *Elementary Average Case Analysis of Floyd's Algorithm to Construct Heaps* (Technical report). Turku Centre for Computer Science. CiteSeerX 10.1.1.15.9526 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.9526). ISBN 951-650-888-X. TUCS Technical Report No. 64. Note that this paper uses Floyd's original terminology "siftup" for what is now called sifting *down*.
8. Kamp, Poul-Henning (June 11, 2010). "You're Doing It Wrong" (http://queue.acm.org/detail.cfm?id=1814327). *ACM Queue*. Vol. 8 no. 6.
9. Chris L. Kuszmaul. "binary heap" (http://nist.gov/dads/HTML/binaryheap.html) Archived (https://web.archive.org/web/20080808141408/http://www.nist.gov/dads/HTML/binaryheap.html) 2008-08-08 at the Wayback Machine. Dictionary of Algorithms and Data Structures, Paul E. Black, ed., U.S. National Institute of Standards and Technology. 16 November 2009.
10. J.-R. Sack and T. Strothotte "An Algorithm for Merging Heaps" (https://doi.org/10.1007%2FBF00264229), Acta Informatica 22, 171-186 (1985).
11. Sack, Jörg-Rüdiger; Strothotte, Thomas (1990). "A characterization of heaps and its applications". *Information and Computation*. **86**: 69–86. doi:10.1016/0890-5401(90)90026-E (https://doi.org/10.1016%2F0890-5401%2890%2990026-E).
12. Atkinson, M.D.; J.-R. Sack; N. Santoro & T. Strothotte (1 October 1986). "Min-max heaps and generalized priority queues" (http://cg.scs.carleton.ca/~morin/teaching/5408/refs/minmax.pdf) (PDF). Programming techniques and Data structures. Comm. ACM, 29(10): 996–1000.
13. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. (1990). *Introduction to Algorithms* (1st ed.). MIT Press and McGraw-Hill. ISBN 0-262-03141-8.
14. "Binomial Heap | Brilliant Math & Science Wiki" (https://brilliant.org/wiki/binomial-heap/). *brilliant.org*. Retrieved 2019-09-30.
15. Fredman, Michael Lawrence; Tarjan, Robert E. (July 1987). "Fibonacci heaps and their uses in improved network optimization algorithms" (http://bioinfo.ict.ac.cn/~dbu/AlgorithmCourses/Lectures/Fibonacci-Heap-Tarjan.pdf) (PDF). *Journal of the Association for Computing Machinery*. **34** (3): 596–615. CiteSeerX 10.1.1.309.8927 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.309.8927). doi:10.1145/28869.28874 (https://doi.org/10.1145%2F28869.28874).

16. Iacono, John (2000), "Improved upper bounds for pairing heaps", *Proc. 7th Scandinavian Workshop on Algorithm Theory* (http://john2.poly.edu/papers/swat00/paper.pdf) (PDF), Lecture Notes in Computer Science, **1851**, Springer-Verlag, pp. 63–77, arXiv:1110.4428 (https://arxiv.org/abs/1110.4428), CiteSeerX 10.1.1.748.7812 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.748.7812), doi:10.1007/3-540-44985-X_5 (https://doi.org/10.1007%2F3-540-44985-X_5), ISBN 3-540-67690-2
17. Fredman, Michael Lawrence (July 1999). "On the Efficiency of Pairing Heaps and Related Data Structures" (http://www.uqac.ca/azinflou/Fichiers840/EfficiencyPairingHeap.pdf) (PDF). *Journal of the Association for Computing Machinery*. **46** (4): 473–501. doi:10.1145/320211.320214 (https://doi.org/10.1145%2F320211.320214).
18. Pettie, Seth (2005). *Towards a Final Analysis of Pairing Heaps* (http://web.eecs.umich.edu/~pettie/papers/focs05.pdf) (PDF). FOCS '05 Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science. pp. 174–183. CiteSeerX 10.1.1.549.471 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.549.471). doi:10.1109/SFCS.2005.75 (https://doi.org/10.1109%2FSFCS.2005.75). ISBN 0-7695-2468-0.
19. Brodal, Gerth S. (1996), "Worst-Case Efficient Priority Queues" (http://www.cs.au.dk/~gerth/papers/soda96.pdf) (PDF), *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 52–58
20. Goodrich, Michael T.; Tamassia, Roberto (2004). "7.3.6. Bottom-Up Heap Construction". *Data Structures and Algorithms in Java* (3rd ed.). pp. 338–341. ISBN 0-471-46983-1.
21. Haeupler, Bernhard; Sen, Siddhartha; Tarjan, Robert E. (November 2011). "Rank-pairing heaps" (http://sidsen.org/papers/rp-heaps-journal.pdf) (PDF). *SIAM J. Computing*. **40** (6): 1463–1485. doi:10.1137/100785351 (https://doi.org/10.1137%2F100785351).
22. Brodal, Gerth Stølting; Lagogiannis, George; Tarjan, Robert E. (2012). *Strict Fibonacci heaps* (http://www.cs.au.dk/~gerth/papers/stoc12.pdf) (PDF). Proceedings of the 44th symposium on Theory of Computing - STOC '12. pp. 1177–1184. CiteSeerX 10.1.1.233.1740 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.233.1740). doi:10.1145/2213977.2214082 (https://doi.org/10.1145%2F2213977.2214082). ISBN 978-1-4503-1245-5.
23. Takaoka, Tadao (1999), *Theory of 2–3 Heaps* (https://ir.canterbury.ac.nz/bitstream/handle/10092/14769/2-3heaps.pdf) (PDF), p. 12

External links

- Binary Heap Applet (http://people.ksp.sk/~kuko/bak/index.html) by Kubo Kovac
- Open Data Structures - Section 10.1 - BinaryHeap: An Implicit Binary Tree (http://opendatastructures.org/versions/edition-0.1e/ods-java/10_1_BinaryHeap_Implicit_Bi.html), Pat Morin
- Implementation of binary max heap in C (https://robin-thomas.github.io/max-heap/) by Robin Thomas
- Implementation of binary min heap in C (https://robin-thomas.github.io/min-heap/) by Robin Thomas