WIKIPEDIA

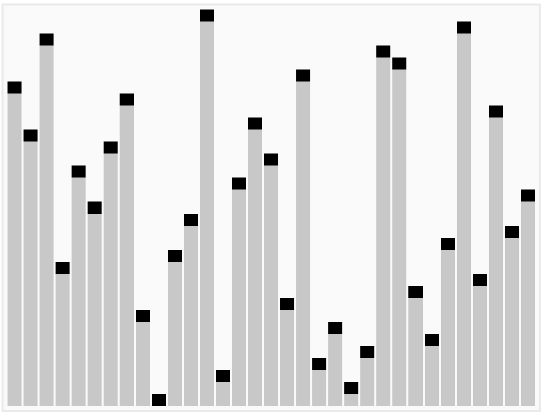
Heapsort

In computer science, **heapsort** is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort: like selection sort, heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. Unlike selection sort, heapsort does not waste time with a linear-time scan of the unsorted region; rather, heap sort maintains the unsorted region in a heap data structure to more quickly find the largest element in each step.[1]

Although somewhat slower in practice on most machines than a well-implemented quicksort, it has the advantage of a more favorable worst-case O(*n* log *n*) runtime. Heapsort is an in-place algorithm, but it is not a stable sort.

Heapsort was invented by J. W. J. Williams in 1964.[2] This was also the birth of the heap, presented already by Williams as a useful data structure in its own right.[3] In the same year, R. W. Floyd published an improved version that could sort an array in-place, continuing his earlier research into the treesort algorithm.[3]

Heapsort



A run of heapsort sorting an array of randomly permuted values. In the first stage of the algorithm the array elements are reordered to satisfy the heap property. Before the actual sorting takes place, the heap tree structure is shown briefly for illustration.

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n \log n)$
Best-case performance	$O(n \log n)$ (distinct keys) or $O(n)$ (equal keys)
Average performance	$O(n \log n)$
Worst-case space complexity	$O(n)$ total $O(1)$ auxiliary

Contents

Overview

The heapsort algorithm can be divided into two parts.

In the first step, a heap is built out of the data (see Binary heap § Building a heap). The heap is often placed in an array with the layout of a complete binary tree. The complete binary tree maps the binary tree structure into the array indices; each array index represents a node; the index of the node's parent, left child branch, or right child branch are simple expressions. For a zero-based array, the root node is stored at index 0; if `i` is the index of the current node, then

```
iParent(i)     = floor((i-1) / 2) where floor functions map a real number to the largest leading integer.
iLeftChild(i)  = 2*i + 1
iRightChild(i) = 2*i + 2
```

In the second step, a sorted array is created by repeatedly removing the largest element from the heap (the root of the heap), and inserting it into the array. The heap is updated after each removal to maintain the heap property. Once all objects have been removed from the heap, the result is a sorted array.

Heapsort can be performed in place. The array can be split into two parts, the sorted array and the heap. The storage of heaps as arrays is diagrammed here. The heap's invariant is preserved after each extraction, so the only cost is that of extraction.

Algorithm

The Heapsort algorithm involves preparing the list by first turning it into a max heap. The algorithm then repeatedly swaps the first value of the list with the last value, decreasing the range of values considered in the heap operation by one, and sifting the new first value into its position in the heap. This repeats until the range of considered values is one value in length.

The steps are:

1. Call the `buildMaxHeap()` function on the list. Also referred to as `heapify()`, this builds a heap from a list in $O(n)$ operations.
2. Swap the first element of the list with the final element. Decrease the considered range of the list by one.
3. Call the `siftDown()` function on the list to sift the new first element to its appropriate index in the heap.
4. Go to step (2) unless the considered range of the list is one element.

The `buildMaxHeap()` operation is run once, and is $O(n)$ in performance. The `siftDown()` function is $O(\log n)$, and is called n times. Therefore, the performance of this algorithm is $O(n + n \log n) = O(n \log n)$.

Pseudocode

The following is a simple way to implement the algorithm in pseudocode. Arrays are zero-based and `swap` is used to exchange two elements of the array. Movement 'down' means from the root towards the leaves, or from lower indices to higher. Note that during the sort, the largest element is at the root of the heap at `a[0]`, while at the end of the sort, the largest element is in `a[end]`.

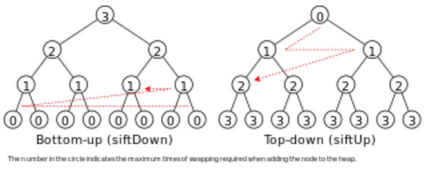
```
procedure heapsort(a, count) is
    input: an unordered array a of length count

    (Build the heap in array a so that largest value is at the root)
    heapify(a, count)

    (The following loop maintains the invariants that a[0:end] is a heap and every element
     beyond end is greater than everything before it (so a[end:count] is in sorted order))
    end ← count - 1
    while end > 0 do
        (a[0] is the root and largest value. The swap moves it in front of the sorted elements.)
        swap(a[end], a[0])
        (the heap size is reduced by one)
        end ← end - 1
        (the swap ruined the heap property, so restore it)
        siftDown(a, 0, end)
```

The sorting routine uses two subroutines, `heapify` and `siftDown`. The former is the common in-place heap construction routine, while the latter is a common subroutine for implementing `heapify`.

```
(Put elements of 'a' in heap order, in-place)
procedure heapify(a, count) is
    (start is assigned the index in 'a' of the last parent node)
    (the last element in a 0-based array is at index count-1; find the parent of that element)
    start ← iParent(count-1)

    while start ≥ 0 do
        (sift down the node at index 'start' to the proper place such that all nodes below
         the start index are in heap order)
        siftDown(a, start, count - 1)
        (go to the next parent node)
        start ← start - 1
    (after sifting down the root all nodes/elements are in heap order)

(Repair the heap whose root element is at index 'start', assuming the heaps rooted at its children are valid)
procedure siftDown(a, start, end) is
    root ← start

    while iLeftChild(root) ≤ end do    (While the root has at least one child)
        child ← iLeftChild(root)   (Left child of root)
        swap ← root                (Keeps track of child to swap with)

        if a[swap] < a[child] then
            swap ← child
        (If there is a right child and that child is greater)
        if child+1 ≤ end and a[swap] < a[child+1] then
            swap ← child + 1
        if swap = root then
            (The root holds the largest element. Since we assume the heaps rooted at the
             children are valid, this means that we are done.)
            return
        else
            swap(a[root], a[swap])
            root ← swap          (repeat to continue sifting down the child now)
```

The `heapify` procedure can be thought of as building a heap from the bottom up by successively sifting downward to establish the heap property. An alternative version (shown below) that builds the heap top-down and sifts upward may be simpler to understand. This `siftUp` version can be visualized as starting with an empty heap and successively inserting elements, whereas the `siftDown` version given above treats the entire input array as a full but "broken" heap and "repairs" it starting from the last non-trivial sub-heap (that is, the last parent node).

Also, the `siftDown` version of heapify has $O(n)$ time complexity, while the `siftUp` version given below has $O(n \log n)$ time complexity due to its equivalence with inserting each element, one at a time, into an empty heap.[4] This may seem counter-intuitive since, at a glance, it is apparent that the former only makes half as many calls to its logarithmic-time sifting function as the latter; i.e., they seem to differ only by a constant factor, which never affects asymptotic analysis.


Bottom-up (siftDown)
Top-down (siftUp)
The number in the circle indicates the maximum times of swapping required when adding the node to the heap.
Difference in time complexity between the "siftDown" version and the "siftUp" version.

To grasp the intuition behind this difference in complexity, note that the number of swaps that may occur during any one `siftUp` call *increases* with the depth of the node on which the call is made. The crux is that there are many (exponentially many) more "deep" nodes than there are "shallow" nodes in a heap, so that siftUp may have its full logarithmic running-time on the approximately linear number of calls made on the nodes at or near the "bottom" of the heap. On the other hand, the number of swaps that may occur during any one siftDown call *decreases* as the depth of the node on which the call is made increases. Thus, when the `siftDown` `heapify` begins and is calling `siftDown` on the bottom and most numerous node-layers, each sifting call will incur, at most, a number of swaps equal to the "height" (from the bottom of the heap) of the node on which the sifting call is made. In other words, about half the calls to `siftDown` will have at most only one swap, then about a quarter of the calls will have at most two swaps, etc.

The heapsort algorithm itself has $O(n \log n)$ time complexity using either version of heapify.

```
procedure heapify(a,count) is
    (end is assigned the index of the first (left) child of the root)
    end := 1

    while end < count
        (sift up the node at index end to the proper place such that all nodes above
         the end index are in heap order)
        siftUp(a, 0, end)
        end := end + 1
    (after sifting up the last node all nodes are in heap order)

procedure siftUp(a, start, end) is
    input:  start represents the limit of how far up the heap to sift.
                  end is the node to sift up.
    child := end
    while child > start
        parent := iParent(child)
        if a[parent] < a[child] then (out of max-heap order)
            swap(a[parent], a[child])
            child := parent (repeat to continue sifting up the parent now)
        else
            return
```

Note that unlike `siftDown` approach where, after each swap, you need to call only the `siftDown` subroutine to fix the broken heap; the `siftUp` subroutine alone cannot fix the broken heap. The heap needs to be built every time after a swap by calling the `heapify` procedure since "siftUp" assumes that the element getting swapped ends up in its final place, as opposed to "siftDown" allows for continuous adjustments of items lower in the heap until the invariant is satisfied. The adjusted pseudocode for using `siftUp` approach is given below.

```
procedure heapsort(a, count) is
   input: an unordered array a of length count

   (Build the heap in array a so that largest value is at the root)
   heapify(a, count)

   (The following loop maintains the invariants that a[0:end] is a heap and every element
    beyond end is greater than everything before it (so a[end:count] is in sorted order))
   end ← count - 1
   while end > 0 do
       (a[0] is the root and largest value. The swap moves it in front of the sorted elements.)
       swap(a[end], a[0])
       (rebuild the heap using siftUp after the swap ruins the heap property)
```

```
        heapify(a, end)
        (reduce the heap size by one)
        end ← end - 1
```

Variations

Floyd's heap construction

The most important variation to the basic algorithm, which is included in all practical implementations, is a heap-construction algorithm by Floyd which runs in $O(n)$ time and uses siftdown rather than siftup, avoiding the need to implement siftup at all.

Rather than starting with a trivial heap and repeatedly adding leaves, Floyd's algorithm starts with the leaves, observing that they are trivial but valid heaps by themselves, and then adds parents. Starting with element $n/2$ and working backwards, each internal node is made the root of a valid heap by sifting down. The last step is sifting down the first element, after which the entire array obeys the heap property.

The worst-case number of comparisons during the Floyd's heap-construction phase of Heapsort is known to be equal to $2n - 2s_2(n) - e_2(n)$, where $s_2(n)$ is the number of 1 bits in the binary representation of n and $e_2(n)$ is number of trailing 0 bits.[5]

The standard implementation of Floyd's heap-construction algorithm causes a large number of cache misses once the size of the data exceeds that of the CPU cache.[6]:87 Much better performance on large data sets can be obtained by merging in depth-first order, combining subheaps as soon as possible, rather than combining all subheaps on one level before proceeding to the one above.[7][8]

Bottom-up heapsort

Bottom-up heapsort is a variant which reduces the number of comparisons required by a significant factor. While ordinary heapsort requires $2n \log_2 n + O(n)$ comparisons worst-case and on average,[9] the bottom-up variant requires $n \log_2 n + O(1)$ comparisons on average,[9] and $1.5n \log_2 n + O(n)$ in the worst case.[10]

If comparisons are cheap (e.g. integer keys) then the difference is unimportant,[11] as top-down heapsort compares values that have already been loaded from memory. If, however, comparisons require a function call or other complex logic, then bottom-up heapsort is advantageous.

This is accomplished by improving the `siftDown` procedure. The change improves the linear-time heap-building phase somewhat,[12] but is more significant in the second phase. Like ordinary heapsort, each iteration of the second phase extracts the top of the heap, $a[0]$, and fills the gap it leaves with $a[end]$, then sifts this latter element down the heap. But this element comes from the lowest level of the heap, meaning it is one of the smallest elements in the heap, so the sift-down will likely take many steps to move it back down.[13] In ordinary heapsort, each step of the sift-down requires two comparisons, to find the minimum of three elements: the new node and its two children.

Bottom-up heapsort instead finds the path of largest children to the leaf level of the tree (as if it were inserting −∞) using only one comparison per level. Put another way, it finds a leaf which has the property that it and all of its ancestors are greater than or equal to their siblings. (In the absence of equal keys, this leaf is unique.) Then, from this leaf, it searches *upward* (using one comparison per level) for the correct position in that path to insert $a[end]$. This is the same location as ordinary heapsort finds, and requires the same number of exchanges to perform the insert, but fewer comparisons are required to find that location.[10]

Because it goes all the way to the bottom and then comes back up, it is called **heapsort with bounce** by some authors.[14]

```
function leafSearch(a, i, end) is
    j ← i
    while iRightChild(j) ≤ end do
        (Determine which of j's two children is the greater)
        if a[iRightChild(j)] > a[iLeftChild(j)] then
            j ← iRightChild(j)
        else
            j ← iLeftChild(j)
    (At the last level, there might be only one child)
    if iLeftChild(j) ≤ end then
        j ← iLeftChild(j)
    return j
```

The return value of the `leafSearch` is used in the modified `siftDown` routine:[10]

```
procedure siftDown(a, i, end) is
    j ← leafSearch(a, i, end)
    while a[i] > a[j] do
        j ← iParent(j)
    x ← a[j]
    a[j] ← a[i]
    while j > i do
        swap x, a[iParent(j)]
        j ← iParent(j)
```

Bottom-up heapsort was announced as beating quicksort (with median-of-three pivot selection) on arrays of size ≥16000.[9]

A 2008 re-evaluation of this algorithm showed it to be no faster than ordinary heapsort for integer keys, presumably because modern branch prediction nullifies the cost of the predictable comparisons which bottom-up heapsort manages to avoid.[11]

A further refinement does a binary search in the path to the selected leaf, and sorts in a worst case of $(n+1)(\log_2(n+1) + \log_2 \log_2(n+1) + 1.82) + O(\log_2 n)$ comparisons, approaching the information-theoretic lower bound of $n \log_2 n - 1.4427n$ comparisons.[15]

A variant which uses two extra bits per internal node (n−1 bits total for an n-element heap) to cache information about which child is greater (two bits are required to store three cases: left, right, and unknown)[12] uses less than $n \log_2 n + 1.1n$ compares.[16]

Other variations

- Ternary heapsort uses a ternary heap instead of a binary heap; that is, each element in the heap has three children. It is more complicated to program, but does a constant number of times fewer swap and comparison operations. This is because each sift-down step in a ternary heap requires three comparisons and one swap, whereas in a binary heap two comparisons and one swap are required. Two levels in a ternary heap cover 3^2 = 9 elements, doing more work with the same number of comparisons as three levels in the binary heap, which only cover 2^3 = 8. This is primarily of academic interest, or as a student exercise,[17] as the additional complexity is not worth the minor savings, and bottom-up heapsort beats both.

- Memory-optimized heapsort[6]: 87 improves heapsort's locality of reference by increasing the number of children even more. This increases the number of comparisons, but because all children are stored consecutively in memory, reduces the number of cache lines accessed during heap traversal, a net performance improvement.
- Out-of-place heapsort[18][19][13] improves on bottom-up heapsort by eliminating the worst case, guaranteeing $n \log_2 n + O(n)$ comparisons. When the maximum is taken, rather than fill the vacated space with an unsorted data value, fill it with a $-\infty$ sentinel value, which never "bounces" back up. It turns out that this can be used as a primitive in an in-place (and non-recursive) "QuickHeapsort" algorithm.[20] First, you perform a quicksort-like partitioning pass, but reversing the order of the partitioned data in the array. Suppose (without loss of generality) that the smaller partition is the one greater than the pivot, which should go at the end of the array, but our reversed partitioning step places it at the beginning. Form a heap out of the smaller partition and do out-of-place heapsort on it, exchanging the extracted maxima with values from the end of the array. These are less than the pivot, meaning less than any value in the heap, so serve as $-\infty$ sentinel values. Once the heapsort is complete (and the pivot moved to just before the now-sorted end of the array), the order of the partitions has been reversed, and the larger partition at the beginning of the array may be sorted in the same way. (Because there is no non-tail recursion, this also eliminates quicksort's $O(\log n)$ stack usage.)
- The smoothsort algorithm[21] is a variation of heapsort developed by Edsger Dijkstra in 1981. Like heapsort, smoothsort's upper bound is $O(n \log n)$. The advantage of smoothsort is that it comes closer to $O(n)$ time if the input is already sorted to some degree, whereas heapsort averages $O(n \log n)$ regardless of the initial sorted state. Due to its complexity, smoothsort is rarely used.
- Levcopoulos and Petersson[22] describe a variation of heapsort based on a heap of Cartesian trees. First, a Cartesian tree is built from the input in $O(n)$ time, and its root is placed in a 1-element binary heap. Then we repeatedly extract the minimum from the binary heap, output the tree's root element, and add its left and right children (if any) which are themselves Cartesian trees, to the binary heap.[23] As they show, if the input is already nearly sorted, the Cartesian trees will be very unbalanced, with few nodes having left and right children, resulting in the binary heap remaining small, and allowing the algorithm to sort more quickly than $O(n \log n)$ for inputs that are already nearly sorted.
- Several variants such as weak heapsort require $n \log_2 n + O(1)$ comparisons in the worst case, close to the theoretical minimum, using one extra bit of state per node. While this extra bit makes the algorithms not truly in-place, if space for it can be found inside the element, these algorithms are simple and efficient,[7]: 40 but still slower than binary heaps if key comparisons are cheap enough (e.g. integer keys) that a constant factor does not matter.[24]
- Katajainen's "ultimate heapsort" requires no extra storage, performs $n \log_2 n + O(1)$ comparisons, and a similar number of element moves.[25] It is, however, even more complex and not justified unless comparisons are very expensive.

Comparison with other sorts

Heapsort primarily competes with quicksort, another very efficient general purpose in-place comparison-based sort algorithm.

Heapsort's primary advantages are its simple, non-recursive code, minimal auxiliary storage requirement, and reliably good performance: its best and worst cases are within a small constant factor of each other, and of the theoretical lower bound on comparison sorts. While it cannot do better than $O(n \log n)$ for pre-sorted inputs, it does not suffer from quicksort's $O(n^2)$ worst case, either. (The latter can be avoided with careful implementation, but that makes quicksort far more complex, and one of the most popular solutions, introsort, uses heapsort for the purpose.)

Its primary disadvantages are its poor locality of reference and its inherently serial nature; the accesses to the implicit tree are widely scattered and mostly random, and there is no straightforward way to convert it to a parallel algorithm.

This makes it popular in embedded systems, real-time computing, and systems concerned with maliciously chosen inputs,[26] such as the Linux kernel.[27] It is also a good choice for any application which does not expect to be bottlenecked on sorting.

A well-implemented quicksort is usually 2–3 times faster than heapsort.[6][28] Although quicksort requires fewer comparisons, this is a minor factor. (Results claiming twice as many comparisons are measuring the top-down version; see § Bottom-up heapsort.) The main advantage of quicksort is its much better locality of reference: partitioning is a linear scan with good spatial locality, and the recursive subdivision has good temporal locality. With additional effort, quicksort can also be implemented in mostly branch-free code, and multiple CPUs can be used to sort subpartitions in parallel. Thus, quicksort is preferred when the additional performance justifies the implementation effort.

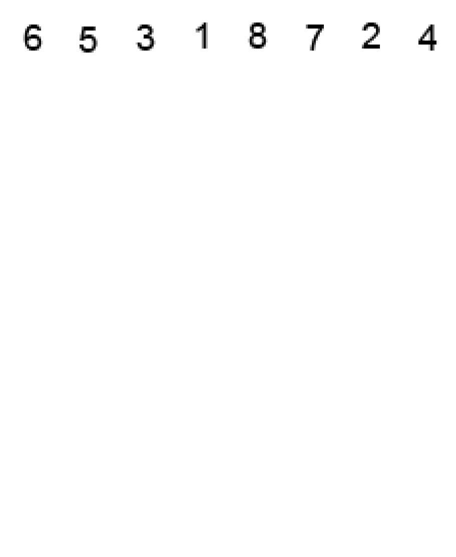
The other major $O(n \log n)$ sorting algorithm is merge sort, but that rarely competes directly with heapsort because it is not in-place. Merge sort's requirement for $\Omega(n)$ extra space (roughly half the size of the input) is usually prohibitive except in the situations where merge sort has a clear advantage:

- When a stable sort is required
- When taking advantage of (partially) pre-sorted input
- Sorting linked lists (in which case merge sort requires minimal extra space)
- Parallel sorting; merge sort parallelizes even better than quicksort and can easily achieve close to linear speedup
- External sorting; merge sort has excellent locality of reference

Example

Let { 6, 5, 3, 1, 8, 7, 2, 4 } be the list that we want to sort from the smallest to the largest. (NOTE, for 'Building the Heap' step: Larger nodes don't stay below smaller node parents. They are swapped with parents, and then recursively checked if another swap is needed, to keep larger numbers above smaller numbers on the heap binary tree.)

Build the heap



An example on heapsort.

Heap	newly added element	swap elements
NULL	6	
6	5	
6, 5	3	
6, 5, 3	1	
6, 5, 3, 1	8	
6, **5**, 3, 1, **8**		5, 8
6, **8**, 3, 1, 5		6, 8
8, 6, 3, 1, 5	7	
8, 6, **3**, 1, 5, **7**		3, 7
8, 6, 7, 1, 5, 3	2	
8, 6, 7, 1, 5, 3, 2	4	
8, 6, 7, **1**, 5, 3, 2, **4**		1, 4
8, 6, 7, 4, 5, 3, 2, 1		

Sorting

Heap	swap elements	delete element	sorted array	details
8, 6, 7, 4, 5, 3, 2, **1**	8, 1			swap 8 and 1 in order to delete 8 from heap
1, 6, 7, 4, 5, 3, 2, **8**		8		delete 8 from heap and add to sorted array
1, 6, **7**, 4, 5, 3, 2	1, 7		8	swap 1 and 7 as they are not in order in the heap
7, 6, **1**, 4, 5, **3**, 2	1, 3		8	swap 1 and 3 as they are not in order in the heap
7, 6, 3, 4, 5, 1, **2**	7, 2		8	swap 7 and 2 in order to delete 7 from heap
2, 6, 3, 4, 5, 1, **7**		7	8	delete 7 from heap and add to sorted array
2, **6**, 3, 4, 5, 1	2, 6		7, 8	swap 2 and 6 as they are not in order in the heap
6, **2**, 3, 4, **5**, 1	2, 5		7, 8	swap 2 and 5 as they are not in order in the heap
6, 5, 3, 4, 2, **1**	6, 1		7, 8	swap 6 and 1 in order to delete 6 from heap
1, 5, 3, 4, 2, **6**		6	7, 8	delete 6 from heap and add to sorted array
1, **5**, 3, 4, 2	1, 5		6, 7, 8	swap 1 and 5 as they are not in order in the heap
5, **1**, 3, **4**, 2	1, 4		6, 7, 8	swap 1 and 4 as they are not in order in the heap
5, 4, 3, 1, **2**	5, 2		6, 7, 8	swap 5 and 2 in order to delete 5 from heap
2, 4, 3, 1, **5**		5	6, 7, 8	delete 5 from heap and add to sorted array
2, **4**, 3, 1	2, 4		5, 6, 7, 8	swap 2 and 4 as they are not in order in the heap
4, 2, 3, **1**	4, 1		5, 6, 7, 8	swap 4 and 1 in order to delete 4 from heap
1, 2, 3, **4**		4	5, 6, 7, 8	delete 4 from heap and add to sorted array
1, 2, **3**	1, 3		4, 5, 6, 7, 8	swap 1 and 3 as they are not in order in the heap
3, 2, **1**	3, 1		4, 5, 6, 7, 8	swap 3 and 1 in order to delete 3 from heap
1, 2, **3**		3	4, 5, 6, 7, 8	delete 3 from heap and add to sorted array
1, **2**	1, 2		3, 4, 5, 6, 7, 8	swap 1 and 2 as they are not in order in the heap
2, **1**	2, 1		3, 4, 5, 6, 7, 8	swap 2 and 1 in order to delete 2 from heap
1, **2**		2	3, 4, 5, 6, 7, 8	delete 2 from heap and add to sorted array
1		1	2, 3, 4, 5, 6, 7, 8	delete 1 from heap and add to sorted array
			1, 2, 3, 4, 5, 6, 7, 8	completed

Notes

1. Skiena, Steven (2008). "Searching and Sorting". *The Algorithm Design Manual*. Springer.

p. 109. doi:10.1007/978-1-84800-070-4_4 (https://doi.org/10.1007%2F978-1-84800-070-4_4). ISBN 978-1-84800-069-8. "[H]eapsort is nothing but an implementation of selection sort using the right data structure."

2. Williams 1964
3. Brass, Peter (2008). *Advanced Data Structures*. Cambridge University Press. p. 209. ISBN 978-0-521-88037-4.
4. "Priority Queues" (https://web.archive.org/web/20200909183303/http://www.equestionanswers.com/c/c-queue.php). Archived from the original (http://www.equestionanswers.com/c/c-queue.php) on 9 September 2020. Retrieved 10 September 2020.
5. Suchenek, Marek A. (2012), "Elementary Yet Precise Worst-Case Analysis of Floyd's Heap-Construction Program", *Fundamenta Informaticae*, **120** (1): 75–92, doi:10.3233/FI-2012-751 (https://doi.org/10.3233%2FFI-2012-751)
6. LaMarca, Anthony; Ladner, Richard E. (April 1999). "The Influence of Caches on the Performance of Sorting" (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.456.3616&rep=rep1&type=pdf). *Journal of Algorithms*. **31** (1): 66–104. CiteSeerX 10.1.1.456.3616 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.456.3616). doi:10.1006/jagm.1998.0985 (https://doi.org/10.1006%2Fjagm.1998.0985). S2CID 206567217 (https://api.semanticscholar.org/CorpusID:206567217). See particularly figure 9c on p. 98.
7. Bojesen, Jesper; Katajainen, Jyrki; Spork, Maz (2000). "Performance Engineering Case Study: Heap Construction" (http://hjemmesider.diku.dk/~jyrki/Paper/katajain.ps) (PostScript). *ACM Journal of Experimental Algorithmics*. **5** (15): 15–es. CiteSeerX 10.1.1.35.3248 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.3248). doi:10.1145/351827.384257 (https://doi.org/10.1145%2F351827.384257). S2CID 30995934 (https://api.semanticscholar.org/CorpusID:30995934). Alternate PDF source (https://www.semanticscholar.org/paper/Performance-Engineering-Case-Study-Heap-Bojesen-Katajainen/6f4ada5912c1da64e16453d67ec99c970173fb5b).
8. Chen, Jingsen; Edelkamp, Stefan; Elmasry, Amr; Katajainen, Jyrki (27–31 August 2012). *In-place Heap Construction with Optimized Comparisons, Moves, and Cache Misses* (https://web.archive.org/web/20161229031307/https://pdfs.semanticscholar.org/9cc6/36d7998d58b3937ba0098e971710ff039612.pdf) (PDF). 37th international conference on Mathematical Foundations of Computer Science. Bratislava, Slovakia. pp. 259–270. doi:10.1007/978-3-642-32589-2_25 (https://doi.org/10.1007%2F978-3-642-32589-2_25). ISBN 978-3-642-32588-5. S2CID 1462216 (https://api.semanticscholar.org/CorpusID:1462216). Archived from the original (https://pdfs.semanticscholar.org/9cc6/36d7998d58b3937ba0098e971710ff039612.pdf) (PDF) on 29 December 2016. See particularly Fig. 3.
9. Wegener, Ingo (13 September 1993). "BOTTOM-UP HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small)" (https://core.ac.uk/download/pdf/82350265.pdf) (PDF). *Theoretical Computer Science*. **118** (1): 81–98. doi:10.1016/0304-3975(93)90364-y (https://doi.org/10.1016%2F0304-3975%2893%2990364-y). Although this is a reprint of work first published in 1990 (at the Mathematical Foundations of Computer Science conference), the technique was published by Carlsson in 1987.[15]
10. Fleischer, Rudolf (February 1994). "A tight lower bound for the worst case of Bottom-Up-Heapsort" (http://staff.gutech.edu.om/~rudolf/Paper/buh_algorithmica94.pdf) (PDF). *Algorithmica*. **11** (2): 104–115. doi:10.1007/bf01182770 (https://doi.org/10.1007%2Fbf01182770). hdl:11858/00-001M-0000-0014-7B02-C (https://hdl.handle.net/11858%2F00-001M-0000-0014-7B02-C). S2CID 21075180 (https://api.semanticscholar.org/CorpusID:21075180). Also available as Fleischer, Rudolf (April 1991). *A tight lower bound for the worst case of Bottom-Up-Heapsort* (http://pubman.mpdl.mpg.de/pubman/item/escidoc:1834997:3/component/escidoc:2463941/MPI-I-94-104.pdf) (PDF) (Technical report). MPI-INF. MPI-I-91-104.
11. Mehlhorn, Kurt; Sanders, Peter (2008). "Priority Queues" (http://people.mpi-inf.mpg.de/~mehlhorn/ftp/Toolbox/PriorityQueues.pdf#page=16) (PDF). *Algorithms and Data Structures: The Basic Toolbox* (http://people.mpi-inf.mpg.de/~mehlhorn/Toolbox.html). Springer. p. 142. ISBN 978-3-540-77977-3.

12. McDiarmid, C.J.H.; Reed, B.A. (September 1989). "Building heaps fast" (http://cgm.cs.mcgill.ca/~breed/2016COMP610/BUILDINGHEAPSFAST.pdf) (PDF). *Journal of Algorithms*. **10** (3): 352–365. doi:10.1016/0196-6774(89)90033-3 (https://doi.org/10.1016%2F0196-6774%2889%2990033-3).
13. MacKay, David J. C. (December 2005). "Heapsort, Quicksort, and Entropy" (https://inference.org.uk/mackay/sorting/sorting.html). Retrieved 12 February 2021.
14. Moret, Bernard; Shapiro, Henry D. (1991). "8.6 Heapsort". *Algorithms from P to NP Volume 1: Design and Efficiency*. Benjamin/Cummings. p. 528. ISBN 0-8053-8008-6. "For lack of a better name we call this enhanced program 'heapsort with bounce.'"
15. Carlsson, Scante (March 1987). "A variant of heapsort with almost optimal number of comparisons" (https://web.archive.org/web/20161227055904/https://pdfs.semanticscholar.org/caec/6682ffd13c6367a8c51b566e2420246faca2.pdf) (PDF). *Information Processing Letters*. **24** (4): 247–250. doi:10.1016/0020-0190(87)90142-6 (https://doi.org/10.1016%2F0020-0190%2887%2990142-6). S2CID 28135103 (https://api.semanticscholar.org/CorpusID:28135103). Archived from the original (https://pdfs.semanticscholar.org/caec/6682ffd13c6367a8c51b566e2420246faca2.pdf) (PDF) on 27 December 2016.
16. Wegener, Ingo (March 1992). "The worst case complexity of McDiarmid and Reed's variant of BOTTOM-UP HEAPSORT is less than *n* log *n* + 1.1*n*" (https://doi.org/10.1016%2F0890-5401%2892%2990005-Z). *Information and Computation*. **97** (1): 86–96. doi:10.1016/0890-5401(92)90005-Z (https://doi.org/10.1016%2F0890-5401%2892%2990005-Z).
17. Tenenbaum, Aaron M.; Augenstein, Moshe J. (1981). "Chapter 8: Sorting". *Data Structures Using Pascal*. p. 405. ISBN 0-13-196501-8. "Write a sorting routine similar to the heapsort except that it uses a ternary heap."
18. Cantone, Domenico; Concotti, Gianluca (1–3 March 2000). *QuickHeapsort, an efficient mix of classical sorting algorithms* (https://archive.org/details/springer_10.1007-3-540-46521-9). 4th Italian Conference on Algorithms and Complexity. Lecture Notes in Computer Science. Vol. 1767. Rome. pp. 150–162. ISBN 3-540-67159-5.
19. Cantone, Domenico; Concotti, Gianluca (August 2002). "QuickHeapsort, an efficient mix of classical sorting algorithms" (https://core.ac.uk/download/pdf/81957449.pdf) (PDF). *Theoretical Computer Science*. **285** (1): 25–42. doi:10.1016/S0304-3975(01)00288-2 (https://doi.org/10.1016%2FS0304-3975%2801%2900288-2). Zbl 1016.68042 (https://zbmath.org/?format=complete&q=an:1016.68042).
20. Diekert, Volker; Weiß, Armin (August 2016). "QuickHeapsort: Modifications and improved analysis". *Theory of Computing Systems*. **59** (2): 209–230. arXiv:1209.4214 (https://arxiv.org/abs/1209.4214). doi:10.1007/s00224-015-9656-y (https://doi.org/10.1007%2Fs00224-015-9656-y).
21. Dijkstra, Edsger W. *Smoothsort – an alternative to sorting in situ (EWD-796a)* (http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD796a.PDF) (PDF). E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (transcription (http://www.cs.utexas.edu/users/EWD/transcriptions/EWD07xx/EWD796a.html))
22. Levcopoulos, Christos; Petersson, Ola (1989), "Heapsort—Adapted for Presorted Files", *WADS '89: Proceedings of the Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science, vol. 382, London, UK: Springer-Verlag, pp. 499–509, doi:10.1007/3-540-51542-9_41 (https://doi.org/10.1007%2F3-540-51542-9_41), ISBN 978-3-540-51542-5 Heapsort—Adapted for presorted files (Q56049336).
23. Schwartz, Keith (27 December 2010). "CartesianTreeSort.hh" (http://www.keithschwarz.com/interesting/code/?dir=cartesian-tree-sort). *Archive of Interesting Code*. Retrieved 5 March 2019.
24. Katajainen, Jyrki (23 September 2013). *Seeking for the best priority queue: Lessons learnt* (http://hjemmesider.diku.dk/~jyrki/Myris/Kat2013-09-23P.html). Algorithm Engineering (Seminar 13391). Dagstuhl. pp. 19–20, 24.
25. Katajainen, Jyrki (2–3 February 1998). *The Ultimate Heapsort* (http://hjemmesider.diku.dk/~jyrki/Myris/Kat1998C.html). Computing: the 4th Australasian Theory Symposium. *Australian Computer Science Communications*. Vol. 20, no. 3. Perth. pp. 87–96.

26. Morris, John (1998). "Comparing Quick and Heap Sorts" (https://www.cs.auckland.ac.nz/software/AlgAnim/qsort3.html). *Data Structures and Algorithms* (Lecture notes). University of Western Australia. Retrieved 12 February 2021.
27. https://github.com/torvalds/linux/blob/master/lib/sort.c Linux kernel source
28. Maus, Arne (14 May 2014). "Sorting by generating the sorting permutation, and the effect of caching on sorting" (https://www.researchgate.net/publication/228620592). See Fig. 1 on p. 6.

References

- 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)
- Floyd, Robert W. (1964), "Algorithm 245 - Treesort 3", *Communications of the ACM*, **7** (12): 701, doi:10.1145/355588.365103 (https://doi.org/10.1145%2F355588.365103), S2CID 52864987 (https://api.semanticscholar.org/CorpusID:52864987)
- Carlsson, Svante (1987), "Average-case results on heapsort", *BIT*, **27** (1): 2–17, doi:10.1007/bf01937350 (https://doi.org/10.1007%2Fbf01937350), S2CID 31450060 (https://api.semanticscholar.org/CorpusID:31450060)
- Knuth, Donald (1997), "§5.2.3, Sorting by Selection", *Sorting and Searching*, The Art of Computer Programming, vol. 3 (third ed.), Addison-Wesley, pp. 144–155, ISBN 978-0-201-89685-5
- 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. Chapters 6 and 7 Respectively: Heapsort and Priority Queues
- A PDF of Dijkstra's original paper on Smoothsort (http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD796a.PDF)
- Heaps and Heapsort Tutorial (http://cis.stvincent.edu/html/tutorials/swd/heaps/heaps.html) by David Carlson, St. Vincent College

External links

- Animated Sorting Algorithms: Heap Sort (https://web.archive.org/web/20150306071556/http://www.sorting-algorithms.com/heap-sort) at the Wayback Machine (archived 6 March 2015) – graphical demonstration
- Courseware on Heapsort from Univ. Oldenburg (https://web.archive.org/web/20130326084250/http://olli.informatik.uni-oldenburg.de/heapsort_SALA/english/start.html) - With text, animations and interactive exercises
- NIST's Dictionary of Algorithms and Data Structures: Heapsort (https://xlinux.nist.gov/dads/HTML/heapSort.html)
- Heapsort implemented in 12 languages (http://www.codecodex.com/wiki/Heapsort)
- Sorting revisited (http://www.azillionmonkeys.com/qed/sort.html) by Paul Hsieh
- A PowerPoint presentation demonstrating how Heap sort works (http://employees.oneonta.edu/zhangs/powerPointPlatform/index.php) that is for educators.
- Open Data Structures - Section 11.1.3 - Heap-Sort (http://opendatastructures.org/versions/edition-0.1e/ods-java/11_1_Comparison_Based_Sorti.html#SECTION001413000000000000000), Pat Morin

Retrieved from "https://en.wikipedia.org/w/index.php?title=Heapsort&oldid=1075091552"