Heapsort

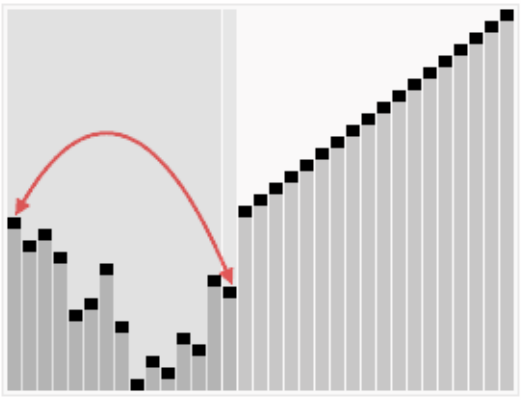
From Wikipedia, the free encyclopedia

In computer programming, **heapsort** is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort: like that algorithm, it divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the smallest element and moving that to the sorted region. The improvement consists of the use of a heap data structure rather than a linear-time search to find the minimum.[2]

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.[3] This was also the birth of the heap, presented already by Williams as a useful data structure in its own right.[4] 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.[4]

Heapsort



A run of the heapsort algorithm 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	$\Omega(n), O(n \log n)$ [1]
Average case performance	$O(n \log n)$
Worst case space complexity	$O(1)$ auxiliary

Contents

Overview

The heapsort algorithm can be divided into two parts.

In the first step, a heap is built out of the data. 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     = floor((i-1) / 2)
iLeftChild  = 2*i + 1
iRightChild = 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. 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.

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]`.

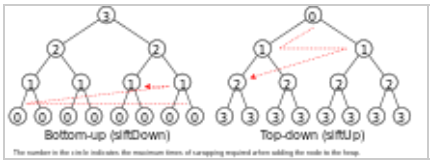
```
procedure heapsort(a, count) is
    input: an unordered array a of length count

    (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 ← floor ((count - 2) / 2)

    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 root * 2 + 1 ≤ end do    (While the root has at least one child)
        child ← root * 2 + 1       (Left child)
        swap ← root                (Keeps track of child to swap with)

        if a[swap] < a[child]
            swap ← child
        (If there is a right child and that child is greater)
        if child+1 ≤ end and a[swap] < a[child+1]
            swap ← child + 1
        if swap = root
            (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.[5] 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 has an impact on asymptotic analysis.



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 := floor((child-1) / 2)
        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
```

Variations

- The most important variation to the simple variant is an improvement by Floyd that uses only one comparison in each siftup run, which must be followed by a siftdown for the original child. 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 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.[6]
- Ternary heapsort[7] 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 step in the shift operation of a ternary heap requires three comparisons and one swap, whereas in a binary heap two comparisons and one swap are required. The ternary heap does two steps in less time than the binary heap requires for three steps, which multiplies the index by a factor of 9 instead of the factor 8 of three binary steps.

- The **smoothsort** algorithm[8][9] 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[10] describe a variation of heapsort based on a Cartesian tree that does not add an element to the heap until smaller values on both sides of it have already been included in the sorted output. As they show, this modification can allow the algorithm to sort more quickly than O(n log n) for inputs that are already nearly sorted.

Bottom-up heapsort

Bottom-up heapsort was announced as beating quicksort (with median-of-three pivot selection) on arrays of size ≥16000.[11] This version of heapsort keeps the linear-time heap-building phase, but changes the second phase, as follows. Ordinary heapsort 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. Bottom-up heapsort instead finds the element to fill the gap, by tracing a path of maximum children down the heap as before, but then sifts that element *up* the heap, which is likely to take fewer steps.[12]

```
function leafSearch(a, end, i) is
    j ← i
    while 2×j ≤ end do
        (Determine which of j's children is the greater)
        if 2×j+1 < end and a[2×j+1] > a[2×j] then
            j ← 2×j+1
        else
            j ← 2×j
    return j
```

The return value of the `leafSearch` is used in a replacement for the `siftDown` routine:[12]

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

Bottom-up heapsort requires only $1.5\ n \log n + O(n)$ comparisons in the worst case and $n \log n + O(1)$ on average. A 2008 re-evaluation of this algorithm showed it to be no faster than ordinary heapsort, though, presumably because modern branch prediction nullifies the cost of the comparisons that bottom-up heapsort manages to avoid.[13]

Comparison with other sorts

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

Quicksort is typically somewhat faster due to some factors, but the worst-case running time for quicksort is $O(n^2)$, which is unacceptable for large data sets and can be deliberately triggered given enough knowledge of the implementation, creating a security risk. See quicksort for a detailed discussion of this problem and possible solutions.

Thus, because of the $O(n \log n)$ upper bound on heapsort's running time and constant upper bound on its auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use heapsort.

Heapsort also competes with merge sort, which has the same time bounds. Merge sort requires $\Omega(n)$ auxiliary space, but heapsort requires only a constant amount. Heapsort typically runs faster in practice on machines with small or slow data caches. On the other hand, merge sort has several advantages over heapsort:

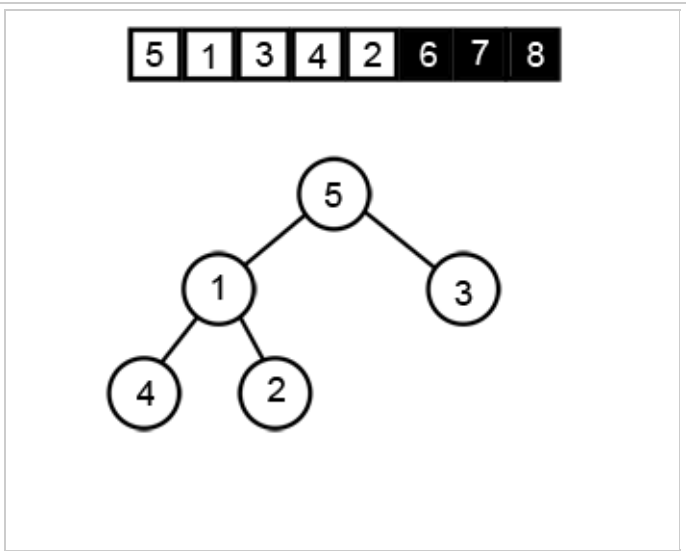
- Merge sort on arrays has considerably better data cache performance, often outperforming heapsort on modern desktop computers because merge sort frequently accesses contiguous memory locations (good locality of reference); heapsort references are spread throughout the heap.
- Heapsort is not a stable sort; merge sort is stable.
- Merge sort parallelizes well and can achieve close to linear speedup with a trivial implementation; heapsort is not an obvious candidate for a parallel algorithm.
- Merge sort can be adapted to operate on **singly** linked lists with O(1) extra space. Heapsort can be adapted to operate on **doubly** linked lists with only O(1) extra space overhead.
- Merge sort is used in external sorting; heapsort is not. Locality of reference is the issue.

Introsort is an alternative to heapsort that combines quicksort and heapsort to retain advantages of both: worst case speed of heapsort and average speed of quicksort.

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.)

1. Build the heap



An example on heapsort.

Heap	newly added element	swap elements
nil	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		

2. 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. **^** http://dx.doi.org/10.1006/jagm.1993.1031
2. **^** Skiena, Steven (2008). *The Algorithm Design Manual*. Springer. p. 109. doi:10.1007/978-1-84800-070-4_4 (https://dx.doi.org/10.1007%2F978-1-84800-070-4_4). "[H]eapsort is nothing but an implementation of selection sort using the right data structure."
3. **^** Williams 1964
4. **^** ***a*** ***b*** Brass, Peter (2008). *Advanced Data Structures*. Cambridge University Press. p. 209. ISBN 9780521880374.
5. **^** "Priority Queues" (http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L10-PQueues.htm). Retrieved 24 May 2011.
6. **^** 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* (IOS Press) **120** (1): 75–92, doi:10.3233/FI-2012-751 (https://dx.doi.org/10.3233%2FFI-2012-751)
7. **^** "Data Structures Using Pascal", 1991, page 405, gives a ternary heapsort as a student exercise. "Write a sorting routine similar to the heapsort except that it uses a ternary heap."
8. **^** http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD796a.PDF
9. **^** http://www.cs.utexas.edu/~EWD/transcriptions/EWD07xx/EWD796a.html
10. **^** 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 **382**, London, UK: Springer-Verlag, pp. 499–509, doi:10.1007/3-540-51542-9_41 (https://dx.doi.org/10.1007%2F3-540-51542-9_41).
11. **^** Wegener, Ingo (1993). "Bottom-up heapsort, a new variant of heapsort beating, on an average, quicksort (if n is not very small)". *Theoretical Computer Science* **118** (1): 81–98. doi:10.1016/0304-3975(93)90364-y (https://dx.doi.org/10.1016%2F0304-3975%2893%2990364-y).
12. **^** ***a*** ***b*** Fleischer, Rudolf (1994). "A tight lower bound for the worst case of Bottom-Up-Heapsort". *Algorithmica* **11** (2): 104–115. doi:10.1007/bf01182770 (https://dx.doi.org/10.1007%2Fbf01182770).
13. **^** Mehlhorn, Kurt; Sanders, Peter (2008). *Algorithms and data structures: The basic toolbox*. Springer. p. 142.

References

- Williams, J. W. J. (1964), "Algorithm 232 - Heapsort", *Communications of the ACM* **7** (6): 347–348
- Floyd, Robert W. (1964), "Algorithm 245 - Treesort 3", *Communications of the ACM* **7** (12): 701, doi:10.1145/355588.365103 (https://dx.doi.org/10.1145%2F355588.365103)
- Carlsson, Svante (1987), "Average-case results on heapsort", *BIT* **27** (1): 2–17, doi:10.1007/bf01937350 (https://dx.doi.org/10.1007%2Fbf01937350)
- Knuth, Donald (1997), "§5.2.3, Sorting by Selection", *Sorting and Searching*, The Art of Computer Programming **3** (third ed.), Addison-Wesley, pp. 144–155, ISBN 0-201-89685-0
- 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

The Wikibook *Algorithm implementation* has a page on the topic of: ***Heapsort***

- Animated Sorting Algorithms: Heap Sort (http://www.sorting-algorithms.com/heap-sort) – graphical demonstration and discussion of heap sort
- Courseware on Heapsort from Univ. Oldenburg (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 (http://www.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 color graphical Java applet (http://coderaptors.com/?HeapSort) that allows experimentation with initial state and shows statistics
- 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#SECTION00141300000000000000)

Retrieved from "http://en.wikipedia.org/w/index.php?title=Heapsort&oldid=646698919"

Categories: Sorting algorithms | Comparison sorts | Heaps (data structures)
