

## AN INTRODUCTION TO HEAPSORT

First appearing in the June 1964 issue of *Communications of the ACM*, the concept of Heapsort was introduced by J. W. J. Williams as a space-conscious, pointer-free alternative for sorting arrays of elements (347). As its name suggests, this algorithm makes use of a data structure known as a heap to repeatedly arrange, compare, and manipulate values in a given data set. Heapsort is a type of selection sort that performs consistently at  $O(n \log n)$  complexity and proves useful in modern applications by outperforming others like Quicksort under worst-case scenarios. This paper outlines the details of the algorithm, as well as its historical context, comparative analysis, and modern application within the computer science realm and beyond.

### The Heap Data Structure

At the very heart of this algorithm is the data structure known as a heap, which provides the basic foundation for its structure. In his initial publication, Williams distinguishes a heap as any array  $A$  of elements  $1$  to  $n$ , such that  $A[i] \leq A[j]$  for  $2 \leq j \leq n$ ,  $i = j \div 2$  (348). While his primary example is based on a sequence of words, this array could also be extended to include a collection of numerical values or other ordered data types in theory. In this sense, elements are not stored and processed in linear sequence but, instead, can be randomly accessed for swapping.

In Chapter 12 of *Programming Pearls*, Jon Bentley illustrates heaps as modified tree structures that conform to specific requirements. Because binary trees can be conveniently stored in arrays, the values of array elements can also be thought of as node labels in a tree (Lang 1). To fit the criteria for a heap, all nodes must be less than or equal to the values of their immediate children, and the distance from each node to the root must not exceed  $\log_2 n$  in a tree of  $n$  nodes (Bentley 126). As illustrated in Figure 1.1, a heap can be visually represented as either an array or a tree for the same collection of values.

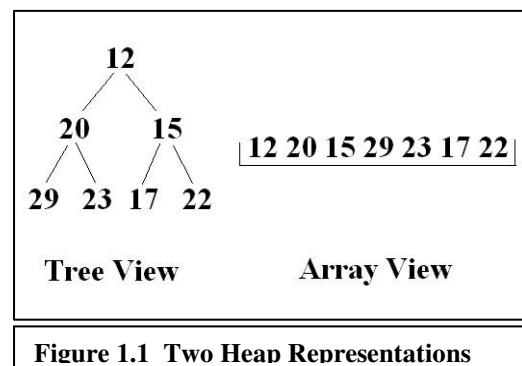


Figure 1.1 Two Heap Representations

### The Sorting Algorithm

A more formal implementation of Heapsort was presented by Robert W. Floyd in the December 1964 issue of *Communications of the ACM* (701). In essence, it is a two-fold algorithm in that it first “builds a heap, then repeatedly extracts the maximum item” to

produce a final array of decreasing values from right to left (Black 1). It is important to note that many implementations of this algorithm handle the first heap property (as mentioned above) in reverse, so that nodes of greater value find their way to the top of the binary tree first. In this sense, adjustments are made during execution to move the greatest value to the root of the heap before extraction and placement at the end of the final sorted list.

In *Programming Pearls*, Bentley presents a simplified version of the Heapsort algorithm, which is condensed into the following five executable lines (133):

```

for I := 2 to N do           /* Invariant: Heap(1,I-1) */
  SiftUp(I)                 /* Heap(1,I) */
for I := N downto 2 do     /* Heap(1,I), Sorted(I+1,N) and X[1..I]<=X[I+1..N] */
  Swap(X[1], X[I])        /* Heap(2,I-1), Sorted(I,N) and X[1..I-1]<=X[I..N] */
  SiftDown(I-1)          /* Heap(1,I-1), Sorted(I,N) and X[1..I-1]<=X[I..N] */

```

The first two lines serve to establish a single integer heap that stores elements  $I$  through  $N$ ; in this case, the largest element always appears at the top of the heap (133). Unlike others that use twice the storage space by saving the heap and final sequence in separate arrays, an in-place implementation of Heapsort saves space by requiring only one array  $X$  for its operation. This also saves time and boosts efficiency by cutting down on the total number of moves to be made.

Once a heap is created from the data values, the last three lines of code are reserved for extracting the maximum element from the heap and tacking it on to the end of the list to produce a sorted sequence from right to left (Wang 1). “The loop body maintains the invariant in two operations. Because  $X[I]$  is the largest among the first  $I$  elements, swapping it with  $X[I]$  extends the sorted sequence by one element. That swap compromises the heap property, which we regain by sifting down the new top element” (Bentley 133). The `SiftUp()` and `SiftDown()` functions are structured as follows (129-130):

```

procedure SiftUp(N)
  pre   Heap(1, N-1) and N < 0
  post  Heap(1, N)
  I := N
  loop
    if I = 1 then break
    P := I div 2
    if X[P] >= X[I] then break
    Swap(X[P], X[I])
    I := P

```

```

procedure SiftDown(N)
  pre   Heap(2, N) and N >= 0
  post  Heap(1, N)
  I := 1
  loop
    C := 2*I
    if C > N then break
    if C+1 <= N then
      if X[C+1] < X[C] then
        C := C+1
    if X[I] <= X[C] then break
    Swap(X[C], X[I])
    I := C

```

These procedures are repeated until the heap is empty and all that is left is a sorted sequence of values.

## Comparative Analysis

When comparing sorting algorithms, it is important to remember that there is no best one for all cases. Several factors should always be taken into account when selecting an optimal sort, including: code simplicity, the size of the data set, the type of values to be sorted, the amount of required storage space, the amount of available memory, and the number of required moves or overhead involved. In nearly all instances, there exist best- and worst-case scenarios, which have the ability to significantly help or hinder the efficiency of a sorted outcome.

In general, selection and insertion sorts are relatively equivalent in efficiency. Each have an outer loop that iterates through every value in the unsorted array and an inner loop that compares this value with what elements remain. Both algorithms perform  $O(n^2)$  comparisons, where  $n$  dictates the number of elements in the array. Selection sort, however, is typically more straightforward to implement and performs fewer swaps if used efficiently.

A member of the selection sort family, Heapsort has a constant complexity of  $O(n \log n)$ , making it a more favorable and consistent comparison sort (Heapsort 1). This is a considerable improvement over other selection and insertion sorts, which normally become far more inefficient as the number of elements increases. “A not in-place Heapsort algorithm needs  $n$  times  $O(\log n)$  time to build the heap and another  $n$  times  $O(\log n)$  time to extract the minima. This results in a total runtime of  $O(n \log n)$ . A in-place version needs only  $O(n)$  time to build the heap but still  $n$  times  $O(\log n)$  time to extract the maxima. Thus the runtime remains  $O(n \log n)$ ” (Wang 1).

Heapsort rivals Quicksort and Mergesort in best-case efficiency but only outperforms the former in worst-case  $O(n^2)$  scenarios however. Attempting to process nearly sorted data is unnecessarily expensive. Because the algorithm strongly relies on random access, Heapsort also tends to run slower in practice on large data caches, linked lists, and media with long access times; for faster machines, Quicksort is usually considered to be a better option on average (Heapsort1).

## Further Variations

Over the years, the Heapsort algorithm has witnessed its share of improvements and modifications—many of which have evolved into cross-bred variations of several different sorts. While Floyd’s implementation is still used and regarded as standard, there exist many offshoots that attempt to improve upon the original by fusing characteristics existing algorithms. One example of this is Introsort, which combines Heapsort and Quicksort to retain the advantages of the former’s worst case speed and the latter’s average speed (Heapsort 1).

Another recognized derivation is Smooth Sort, a 1981 algorithm that was developed by Edsger Dijkstra in response to Heapsort’s inefficiency with handling nearly sorted data. “While sharing in general its  $O(n \log n)$  characteristic, smooth sort does not share in this disadvantage: for an initially (nearly) sorted sequence, smooth sort is of order  $n$  with a smooth transition between the two” (Dijkstra 2). Likewise, it “improves the best case to  $O(n)$  on an already sorted list. But in order to achieve this, the heap needs to be implemented

backwards which causes great overhead. Also because average and worst case are not improved, Smoothsort is seldom used in practice” (Wang 1).

## Conclusion

Since its inception by Williams and Floyd, Heapsort has gained popularity as a traditional sorting algorithm used to store and sequence values via unique data structures known as heaps. Because runtime remains at a  $O(n \log n)$  complexity, it provides reliable consistency in manipulating array elements and excels in situations where faster algorithms like Quicksort are reduced to their  $O(n^2)$  worst-cases. Heapsort presents a secure alternative to other sorts in time-critical applications where storage space is limited and reliable performance is required, securing its status as a fundamental component of computer science curricula.

## Works Cited

- Bentley, Jon. *Programming Pearls*. Murray Hill, New Jersey: Addison-Wesley, 1986.
- Black, Paul E. “Heapsort” in *Dictionary of Algorithms and Data Structures* [online]. 14 May 2007. U.S. National Institute of Standards and Technology. 17 Apr 2008. <<http://www.nist.gov/dads/HTML/heapsort.html>>.
- Dijkstra, Edsger. 16 Aug 1981. “Smoothsort: An Alternative for Sorting in Situ.” 17 Apr 2008. <<http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD796a.PDF>>.
- Floyd, Robert W. “Algorithm 245 Treesort 3” in *Communications of the ACM*, vol. 7, no.12 (December 1964): p. 701.
- “Heapsort.” Wikipedia.org. 17 Apr 2008. <<http://en.wikipedia.org/wiki/Heapsort>>.
- Lang, Hans Werner. “Heapsort.” 25 Jan 2008. Flensburg University of Applied Sciences. 17 Apr 2008. <<http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/heap/-heapen.htm>>.
- Wang, Miao. “Heap Sort.” 25 July 2001. 17 Apr 2008. <[http://www.wanginator.de/studium/applets/heapsort\\_en.html](http://www.wanginator.de/studium/applets/heapsort_en.html)>.
- Williams, J. W. J. “Algorithm 232 Heapsort” in *Communications of the ACM*, vol. 7, no. 6 (June 1964): pp. 347-348.