BIT 33 (1993), 372–381.

WEAK-HEAP SORT

RONALD D. DUTTON

Dept. of Computer Science, University of Central Florida, Orlando, FL 32816, USA

Abstract.

A data structure called a *weak-heap* is defined by relaxing the requirements for a heap. The structure can be implemented on a 1-dimensional array with one extra bit per data item and can be initialized with n items using exactly $n - 1$ data element compares. Theoretical analysis and empirical results indicate that it is a competitive structure for sorting. The worst case number of data element comparisons is strictly less than $(n - 1)\log n + 0.086013n$ and the expected number is conjectured to be approximately $(n - 0.5)\log n - 0.413n$.

CR Categories: E.1, F2.2.

Keywords: Algorithms, data structures, heap, priority queue.

1. Introduction.

This paper introduces a sorting algorithm based upon the *weak-heap* data structure defined initially in [3]. A weak-heap uses a relaxed, or weakened, version of the standard *heap property* first described by Williams in [11] for the sorting algorithm *heap sort*. A tree in which the value in each node, other than the root, is less than or equal to that in its parent node has the heap property. We assume the maximum heap property throughout. A binary tree with the heap property (and with supporting algorithms) is often simply called a *heap*. In an implicit implementation, nodes with less than two children appear on the bottom two levels of the tree with those on the bottom level being as far to the left as possible.

In terms of the number of comparisons, the information theoretic lower bound on the maximum number of data compares for comparison based sequential sorting algorithms is $\log(n!)$, approximately $n\log n - 1.442695n$. Logarithms are assumed to be taken base two. The literature abounds with algorithms approaching this limit in one sense or the other. Most variants of *Quicksort* [5] have a worst case bound of order n^2, but have a much more efficient expected case. The *best-of-three* version called *Clever Quicksort* has been analyzed (see [9]) to have an expected number of compares of approximately $1.188n\log(n - 1) - 2.255n + 1.188\log(n - 1) + 2.507$.

Received July 1992. Revised December 1992.

The original versions of heapsort [4, 11] have a worst case complexity of $2n \log n$. A variant introduced by McDiarmid and Reed, called *Bottom-Up Heapsort* [6, 9], was shown to be bounded by $1.5n \log n - 0.4n$. Recently, Wegner [10] introduced a clever implementation of Bottom-Up Heapsort, using n additional bits, which he called *MDR-Heapsort* and showed it had a worst case bound of $(n+1)\log n + 1.086072n$.

No exact average case results exist for any of the Heapsort variants because the intermediate heaps formed during the execution of the algorithms quickly become nonrandom and it is not clear how to analyze this phenomenon. Simplifying assumptions, supported by simulations, though have given results indicating that the average is $n \log n + f(n)n$ where $f(n) \in [0.355, 0.39]$ for Bottom-Up Heapsort and $f(n) \in [-0.05, 0.10]$ for MDR-Heapsort (see [6, 9] and [10], respectively).

The Weak-Heap sorting algorithm described here, also using n additional bits, is shown to have a worst case number of compares that is less than $(n-1)\log n + 0.086013n$ and we conjecture the average number required is approximately $(n - 0.5)\log n - 0.413n$.

A weak-heap is formally defined as a binary tree where:

(1) the value in any node is at least the value of any node in its right subtree,
(2) the root has no left subtree, and
(3) nodes having fewer than two children, except the root, appear on the bottom two levels of the tree.

Weak-heaps do not require leaf nodes on the bottom level to be as far to the left as possible, nor that a relationship exists between a node's value and the values in the left subtree of that node. Condition (2) guarantees the root contains the largest value. Because weak-heaps are a variant of heaps, they bear a striking similarity to other variants of heaps, e.g., *general heaps* and *binomial heaps* (see [2, 7, and 8]). One can often incorporate the features of one variant within those of another by suitable restrictions and/or relaxations. Nevertheless, different variants can provide subtle insights into algorithm design, implementation and analysis.

Figure 1 gives two views of a weak-heap: the second being a more detailed version of the first. The root contains the largest value and, for $1 \le i \le k$, all values in subtree T_i are less than or equal to the value in the corresponding node i, that is, the subtree induced by node i and T_i is a sub-weak-heap. In fact, any node and its right subtree is a weak-heap. Notice, because of (3) in the definition, the subtree T_i has between $2^{k-i} - 1$ and $2^{k-i+1} - 1$ nodes.

For any node x in a weak-heap, let

$$S_x = \{Rson(x)\} \cup \{y \mid y \text{ is reachable from } Rson(x) \text{ by left branches only}\}$$

with x being the unique "Grandparent" of the nodes in S_x; i.e.,

$$x = Gparent(y) \text{ if and only if } y \in S_x.$$

In Figure 1, $S_h = \{1, 2, \ldots, k\}$ and h is the grandparent of all nodes i, $1 \le i \le k$.

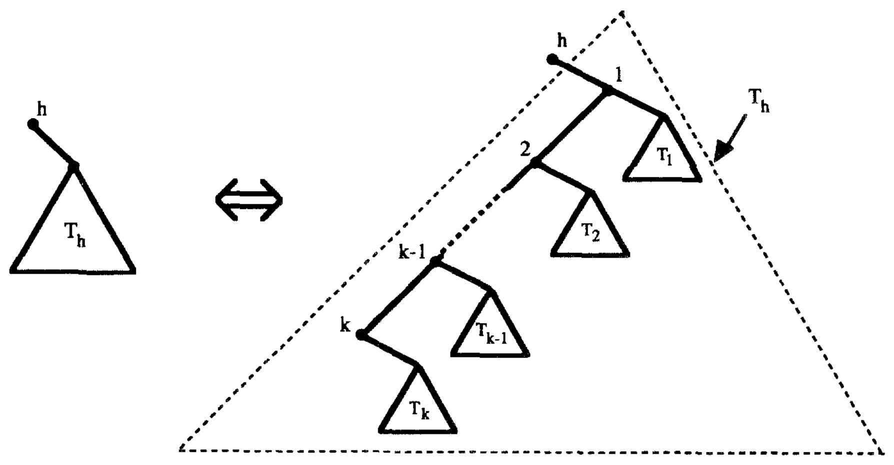

Figure 1. Two views of a weak-heap.

Similarly, for each i, node i is the grandparent of nodes in S_i, a subset of T_i. Node k is called the "leftmost" node in T_h, the right subtree of h.

The significance of the set S_h is that, when the root node h is removed, its elements become the roots of a list or "forest" of weak-heaps that must be restructured, i.e., merged, into a single weak-heap. Two weak-heaps can be merged as follows:

(1) the root node containing the largest value becomes the root of the merged tree, and

(2) the smaller root becomes the right son of the larger and obtains, as its left subtree, the original right subtree of the larger root.

These actions are illustrated in Figure 2 when the value in h is greater than or equal to the value in x. Otherwise, the tree on the right would have the left and right subtrees of node x interchanged. Also, the values in nodes h and x would have been exchanged.

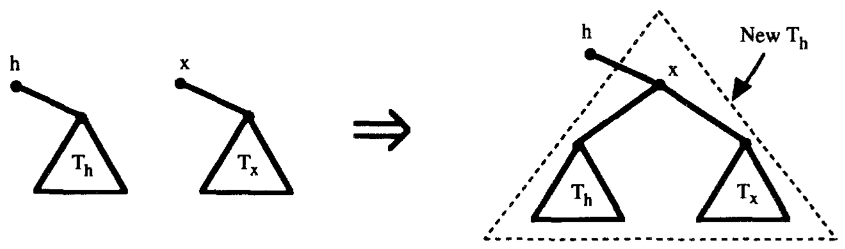

Figure 2. Merging of weak-heaps.

Merging produces a binary tree satisfying conditions (1) and (2) of the weak-heap definition, but not necessarily condition (3). To result in a weak-heap, all distances from nodes with less than two children to their respective root nodes must be one of two consecutive values. Such pairs of weak-heaps will be called *compatible*.

In Section 2, a 1-dimensional array implementation will be presented. The sorting algorithm, *WeakHeapSort*, is given in Section 3, followed in Section 4 by an analysis of the number of data compares and empirical results comparing WeakHeapSort with three of the sorting algorithms mentioned above.

2. An implicit data structure.

To implement the merge procedure described above using a 1-dimensional array requires that values in the respective subtrees be interchanged. It is sufficient to maintain an extra bit, or a boolean variable, for each node to indicate which of its children is the root of its weak-heap right subtree. Then, when interchanging left and right subtrees, we simply complement the extra bit or boolean variable of the (common) parent node. Notice that interchanging subtrees does not alter the distance of any node to the root. Thus, if the initial structures were weak-heaps, the resulting structure will be a weak-heap.

The n values in a weak-heap, with nodes labeled $0, 1, \ldots, n-1$ as encountered in a breadth-first traversal from the root, will be maintained in the first n positions of a 1-dimensional array $h[0 \ldots n]$, where the value of node i is $h[i]$. The purpose of $h[n]$ is resolved in the next section. Node 0, or $h[0]$, contains the maximum value and the children of node i are nodes $2i$ (when $i > 0$) and $2i + 1$. Which of these is actually in the weak-heap rooted at node i, i.e., its "right" child, will be indicated by the ith entry of a boolean valued 1-dimensional array, *Reverse* $[0 \ldots n-1]$, with *Reverse*$[i]$ *true* when it is node $2i$, and *false* when it is node $2i + 1$. As indicated earlier, this information could be recorded by a single extra bit within each node. Here, for clarity of presentation, we choose to use named boolean variables, i.e., entries of the boolean array Reverse. For sorting, it is unnecessary to maintain boolean variables for the leaf nodes of the binary tree, but their inclusion is not incorrect and actually simplifies the statements of the algorithms.

The following routines will be used in the algorithms presented below. The first, $\{$*boolean*$\}$, is a function returning one when the logical expression *boolean* is *true*, and zero otherwise. Then the left and right children of node i are nodes $2i + \{Reverse[i]\}$ and $2i + 1 - \{Reverse[i]\}$, respectively. If a value is zero or greater than $n - 1$, the interpretation is that the subtree is empty. The second routine is *odd*(i), a boolean function (supplied by many programming languages) that returns *true* if i is an odd integer and *false* otherwise. Finally, *Swap*(i,j) interchanges the values in $h[i]$ and $h[j]$.

3. WeakHeapSort.

The basic procedure is straightforward: first WeakHeapify, that is, construct a weak-heap from a given set of arbitrary values; then as long as the weak-heap is not empty, remove the root (and value, the largest remaining in the structure) and merge the resulting forest of weak-heaps.

```
WeakHeapSort(n)
  WeakHeapify(n)
  h[n] ← h[0]                                  {Move the root value to h[n].}
  for i = n − 1 downto 2 do MergeForest(i)     {ReWeakHeapify the forest. }
```

The algorithm assumes that all the values in the boolean array Reverse have been set to *false* and an initial array of arbitrary values $h[0 \ldots n-1]$ is given. It then rearranges them into $h[1 \ldots n]$ in ascending order. Routines *WeakHeapify* and *MergeForest* invoke two other routines, *Gparent* and *Merge* which are described next. A more rigorous justification of the correctness of these algorithms can be found in [3]. The algorithm Gparent acts as a function that returns the Grandparent, defined in Section 1, of a given node $j > 0$.

```
Gparent(j)
  while odd(j) = Reverse[⌊j/2⌋] do j ← ⌊j/2⌋   {While j is the left son of ⌊j/2⌋. }
  return(⌊j/2⌋)                               {Now, j is the right son of ⌊j/2⌋.}
```

In the next algorithm, the weak-heaps rooted at i and j are merged into a single weak-heap rooted at i. The fact that these are actually compatible weak-heaps when Merge is entered will be established later when considering WeakHeapify and MergeForest, the places from which Merge is invoked.

```
Merge(i,j)                               {Merges two weak-heaps rooted at i and j. }
  If h[i] < h[j] then                    {            i's right son must be j's left son. }
    Swap(i,j)                            {Interchanges roots i ad j.                }
    Reverse[j] ← not(Reverse[j])         {Interchanges the left and right subtrees  }
  end if                                 {            of j, i.e., complements Reverse[j]}
```

WeakHeapify, below, is an iterative version of a recursive algorithm presented in [3], that is, "unMerge" a binary tree with the weak-heap shape into two smaller binary trees, both with the weak-heap shape; recursively WeakHeapify each; and, finally, Merge them back together. Observe that the subproblems on each level of the recursion tree are independent and therefore may be solved in any order. The only stipulation is that subproblems at lower levels must be solved prior to those at higher levels. Thus, we may solve them "bottom-up" by iterating from $n - 1$ down to 1. Further, the "right" subtrees of the roots of the weak-heaps being merged are actually the two subtrees of node j. Thus, the two weak-heaps are compatible.

Notice that lower level subproblems need not be reWeakHeapified after a higher level problem is solved, as opposed to the **heapify** algorithm for heaps.

```
WeakHeapify($n$)                                              {$h[0 \ldots n-1]$ implicity has the weak-heap  }
  for $j = n - 1$ downto 1 do Merge(Gparent($j$),$j$)   {shape, Reverse[$0, \ldots n-1$] is assumed false.}
```

Since each call to Merge executes one data element comparison and the function Gparent uses none, WeakHeapify requires exactly $n - 1$ compares.

MergeForest, given next, is invoked with the values in $h[1 \ldots m]$ representing a forest of weak-heaps. The **do-until** statement "walks" x through the nodes in $S_0 - \{m\}$ to the leftmost node. Each node encountered, except node 1, is the (weak-heap) left son of its parent, i.e. $2x$ if Reverse[x] is **false**, and $2x + 1$, otherwise. Node m is initially merged with node x in order that the underlying binary tree remain balanced. This also guarantees that the pairs of weak-heaps being merged, as we walk back up to the root, are compatible weak-heaps.

As described thus far, MergeForest would leave the resulting weak-heap in $h[0 \ldots m-1]$. Then WeakHeapSort would need to move the value in $h[0]$ to $h[m]$ before the next call to MergeForest. To eliminate this redundant data transfer, MergeForest simply considers node m to be the root of the weak-heap being constructed by the **while** statement.

```
MergeForest($m$)                                         {$h[1 \ldots m]$ is a forest of  }
  $x \leftarrow 1$                                              {              weak-heaps. }
  Reverse[$\lfloor m/2 \rfloor$] ← false                       {$m \geq 2$ is assumed            }
  if $m \geq 3$ then do $x \leftarrow 2*x + \{\text{Reverse}[x]\}$    {Walk down $S_0$ until          }
          until $2*x + \{\text{Reverse}[x]\} \geq i$           {  $x$ is the leftmost node.}
  while $x > 0$ do
      Merge($m,x$)                                       {Walk back up the tree    }
      $x \leftarrow \lfloor x/2 \rfloor$                                    {  merging weak-heaps    }
    end while                                            {  on the way.             }
```

If $h[1 \ldots m]$ represents a forest of weak-heaps, then MergeForest causes either $\lceil \log(m+1) \rceil - 1$ or $\lceil \log(m+1) \rceil$ data comparisons to be executed. When m is a power of two, $\lceil \log(m+1) \rceil - 1$ always holds, otherwise both values are possible. The importance of this is that, during the sorting process, we are guaranteed that at least once on each level a "short" path will be identified by MergeForest. Thus, since there are $n - 1$ calls to MergeForest and $\lceil \log(n) \rceil$ levels, the difference between the minimum and maximum number of compares is exactly $n - 1 - \lceil \log(n) \rceil$.

We now argue the correctness of the main algorithm. After returning from WeakHeapify, the largest value, $h[0]$, is immediately moved to $h[n]$. This effectively "removes" the root of the weak-heap. Then, MergeForest reWeakHeapifies and moves the (new) root to $h[i]$ (the first time, $i = n - 1$). We repeatedly decrease i and invoke MergeForest, until $i < 2$. At that point, one value remains: the smallest, and it is in $h[1]$.

These algorithms have been collected into a single Pascal-like procedure in the appendix. Notice that WeakHeapify does not appear explicitly, but its one line body has replaced the statement invoking it.

4. Analysis and empirical results.

The measure of complexity will be the number of two-way data element compares. We ignore the integer divisions and multiplications by two required to walk up and down a binary tree. Those portions of the program could be coded in a lower level language so that shift register instructions might be used to make these operations less by a factor in the execution time. Also, when the data items are more complex, these operations become less significant than data manipulations.

THEOREM 4.1. *The maximum number of data compares required by WeakHeapSort for any number of $n \geq 1$ values is less than $(n-1)\log(n) + 0.086013n$.*

PROOF. We first derive an upper bound on the minimum number of comparisons required, then argue that the maximum is at most $n - \lceil \log(n) \rceil - 1$ more than this result. No comparisons occur when $n = 1$, so we may assume $2^{k-1} < n \leq 2^k$, for positive integer k. Then, $k = \lceil \log(n) \rceil$ is the level number of nodes on the bottom of the tree with the root on level 0, its right son on level 1, etc. As root nodes are removed, MergeForest is invoked with the remaining number of values being, in successive calls, $n-1, n-2, \ldots, 2$. As i decreases from $n-1$ down to 2, the minimum total number of compares, exclusive of WeakHeapify, is

$$\sum_{i=2}^{n-1} (\lceil \log(i+1) \rceil - 1) = nk - 2^k - n + 2.$$

Since WeakHeapify uses exactly $n-1$ data compares, the overall best case number of compares is $nk - 2^k + 1$.

From earlier comments, the worst case total number of comparisons is the best case number plus $n - 1 - \lceil \log n \rceil$(or, $n - k - 1$), that is, a total of $nk - 2^k + n - k$. For any $n \geq 1$, there is some real x, $0 \leq x < 1$, such that $k = \log n + x$. Then, $nk - 2^k + n - k = n \log n + nx - n2^x + n - \log n - x = (n-1)\log n + n(x - 2^x + 1) - x$. The real-valued function $x - 2^x + 1$ is bounded above by 0.086013 and is zero when $x = 0$. Thus, the total number of compares is strictly less than $(n-1)\log n + 0.086013n$. ■

As for heap sorts, and for the same reasons, an average case analysis seems difficult and is unsolved at this time, but simulations with randomly generated data appear to be within 0.1% of the average of the best and worst cases, that is, $nk - 2^k + (n - k + 1)/2 < (n - 0.5)\log n - 0.413n$. The derivation of the real-valued approximation is similar to that for the upper bound given in the proof of Theorem 4.1.

Simulation results are presented in the Table. For each n, identical sets of 20 randomly generated distinct values were given to each sorting routine. The second column, WHS, is the average number of comparisons required by WeakHeapSort. The third column, QSORT, reflects those of an implementation of Quicksort [5] which partitions with the middle of three randomly selected values. The fourth column, BUHS, is the data reported for Bottom-Up heapsort from [1]: the last column, MDRS, is the result of an implementation of Wegener's version of Bottom-Up heapsort [10]. The figures in parenthesis are the expected number of compares for each instance. For example, with WeakHeapSort, this is approximated by an upper bound of the average of the best and worst case number of compares as given above, i.e., $(n - 0.5)\log n - 0.413n$.

Table. *Simulated numbers of compares.*

n	WHS	QSORT	BUHS	MDRS
10	26(27)	28(23)	32(38)	31(33)
50	254(258)	260(231)	293(300)	283(284)
100	614(619)	624(574)	696(701)	669(669)
500	4247(4271)	4543(4211)	4650(4669)	4507(4506)
1000	9497(9547)	10142(9598)	10312(10338)	10032(10013)
5000	59222(59367)	64207(61731)	63309(63303)	61862(61678)
10000	128465(128740)	140570(135326)	136648(136607)	133719(133352)
50000	758835(759824)	850962(814483)	799741(799132)	785982(782857)

In [9], Wegener presents a formula (attributed to Kemp) resulting from an analysis of the expected number of compares for the version of Quicksort used here, that is approximately $1.188n\log(n - 1) - 2.255n + 1.188\log(n - 1) + 2.507$. Our simulation, column QSORT, exceeds the formula values (given in parenthesis) consistently by 4–5%. Even then, when $n > 500$, the formula values are still greater than those observed for WeakHeapSort. The expected values for columns BUHS and MDRS were derived with the formulas $n\log n + 0.373n$ and $n\log n + 0.0475n$, respectively, and arise from using the midpoints of the expected ranges for $f(n)$ given in Section 1.

WeakHeapSort seems always to achieve the best case number of compares, $n\lceil\log n\rceil - 2^{\lceil\log n\rceil} + 1 < \log n - 0.913986n$, when the data are initially in ascending order. The reason for this is not yet understood, but one should not conclude that preprocessing the data will be beneficial since the difference between the best and worst case number of comparisons is at most $n - \lceil\log n\rceil - 1$. It is difficult to imagine preprocessing that would use fewer compares than this difference.

5. Summary and acknowledgements.

A new data structure for sorting has been shown to possess several desirable characteristics, not the least of which is simplicity of definition, a feature that appears to be lacking in many data structures introduced since Williams' elegant definition of heaps. The resulting sorting algorithm requires a number of comparisons that is strictly less than $(n-1)\log n + 0.086013n$. The structure, called a weak-heap, arises by relaxing the requirements of a heap and is a special case of a general heap. An implementation in a 1-dimensional array uses one extra bit per data item.

The author would like to express appreciation to the referees for their helpful comments and suggestions, especially for pointing out the existence of the recent paper by Wegener.

REFERENCES

1. S. Carlsson, *Average-case results on heapsort*, BIT, 27, 1987, pp. 2–17.
2. S. Carlsson, J. I. Munro and P. V. Poblete, *An implicit binomial queue with constant insertion time*, Proceedings of 1st Scandinavian Workshop on Algorithm Theory, Halmstad, Sweden, July 5–8, 1988, pp. 1–13.
3. R. D. Dutton, *The weak-heap data structure*, Department of Computer Science Technical Report, CS-TR-92-09, 1992, University of Central Florida, Orlando, FL 32816.
4. R. W. Floyd, *Algorithm* 245, *treesort* 3, Comm. ACM, 1964, p. 701.
5. C. A. R. Hoare, *Algorithm* 63, 64 *and* 65, Comm. ACM, 4 (7), 1961, pp. 321–322.
6. C. J. H. McDiarmid and B. A. Reed, *Building heaps fast*, J. of Algorithms, 10, 1989, pp. 352–365.
7. J. R. Sack and T. Strothotte, *An algorithm for merging heaps*, Acta Informatica 22, 1985, pp. 171–186.
8. J. Vuillemin, *A data structure for manipulating priority queues*, Comm. of the ACM, 21 (4), 1978, pp. 309–315.
9. I. Wegener, *Bottom-up heap sort, a new variant of heap sort beating on average quicksort (if n is not very small)*, Proceedings of Mathematical Foundations of Computer Science 1990, Banska Bystrica, Czechoslovakia, August, 1990, pp. 516–522.
10. I. Wegener, *The worst case complexity of McDiarmid and Reed's variant of Bottom-Up heapsort is less than* $n\log n + 1.1n$, Information and Computation, 97, 1992, pp. 86–96.
11. J. W. J. Williams, *Algorithm* 232, Heapsort, Comm. of the ACM, 7, 1964, pp. 347–348.

Appendix

The WeakHeapSort Algorithm

```
procedure WeakHeapSort(n)                          {Arrays h and Reverse are given.          }
  function Gparent(j)                              {Returns the Grandparent of node j.       }
    while odd(j) = Reverse[⌊j/2⌋] do j ← ⌊j/2⌋     {While j is the left son on ⌊j/2⌋.        }
    return(⌊j/2⌋)                                  {Now, j is the right son of ⌊j/2⌋.        }
  end Gparent

  procedure Merge(i,j)                             {Merges weak-heaps rooted at i and j.     }
    if h[i] < h[j] then                            {i's right son must be j's left son.      }
      Swap(i,j)                                    {Interchange values in roots i and j.     }
      Reverse[j] ← not (Reverse[j])                {Interchanges the subtrees of node j,     }
  end if                                              {  i.e., complements Reverse[$j$].      }
end Merge

procedure MergeForest($m$)                            {$h[1 \dots m]$ is a forest of weak-heaps. }
  $x \leftarrow 1$
  Reverse[$\lfloor m/2 \rfloor$] $\leftarrow$ false
  if $m \geq 3$ then do $x \leftarrow 2 * x + \{$Reverse[$x$]$\}$    {Walk down $S_0$                }
    until $2 * x + \{$Reverse[$x$]$\} \geq m$          {until $x$ is the leftmost node.        }
  while $x > 0$ do
      Merge($m, x$)                                   {Walk back up the tree merging         }
      $x \leftarrow \lfloor x/2 \rfloor$                                   {  weak-heaps on the way.              }
    end while
end MergeForest

for $i = n - 1$ downto 1 do Merge(Gparent($i$), $i$)  {WeakHeapify.                          }
$h[n] \leftarrow h[0]$                                {Move the root value to $h[n]$.         }
for $i = n - 1$ downto 2 do MergeForest($i$)          {ReWeakHeapify and move                }
end WeakHeapSort                                      {  root to $h[i]$.                      }
```