International Journal of Computing Academic Research (IJCAR)
ISSN 2305-9184 Volume 3, Number 2(April 2014), pp. 39-57

http://www.meacse.org/ijcar

Comparative Performance Evaluation of Heap-Sort and Quick-Sort Algorithms

Ebtisam S. Al-Jaloud, Haifaa A. Al-Aqel and Ghada H. Badr
King Saud University. Riyadh, Saudi Arabia

Abstract

Sorting is a fundamental operation in computer science (many programs use it as an intermediate step). A large number of sorting algorithms have been made in order to have a best performance in terms of computational complexity (best, average and worst), memory usage, stability and method. One of sorting algorithms can perform much better than another. For that, we intend to make a comparison research paper and two representative algorithms were chosen to show these differences. In the paper, a comparative performance evaluation of two sorting algorithms Heap-Sort and Quick-Sort will be presented depending on the performance factors shown above. Furthermore, the most significant enhancements of these algorithms will be discussed.

Keywords: HeapSort, QuickSort, pivot

1. Introduction

A number of sorting algorithms have been developed like include heap sort , merge sort, quick sort, selection sort all of which are comparison based sort . This paper gives the brief history about two sorting algorithms QuickSort and HeapSort and show the biggest differences between them.

Quick Sort is a sorting algorithm which is easier, used to code and implement. It is a recursive algorithm that uses the divide and conquer method. It is also the fastest generic sorting algorithm in practice.

Heap Sort combines the best of both merge sort and insertion sort. Like merge sort, the worst case time of heap sort is O(n log n) and like insertion sort, heap sort sorts in-place.

In section 2, Quicksort and Heapsort algorithms described and compared. In section 3, the updates on the both algorithms discussed and compared. In section 4, we finish the paper with efficient mix of the both algorithms into one.

2. Comparative Analysis

There are differences between Quicksort and heap sort. How much they are different? Does one of them can beat the other in some factors? In this section a comparative analysis is described according to the used method, stability, memory usage and complexity.

A. *QuickSort*

Quicksort is a sorting algorithm developed by Tony Hoare that, It is also known as partition exchange sort.

1) *How It Works*

The idea is to choose a pivot element, and to compare each element to this pivot element. If an element is inferior or equal to the pivot element, this element is put on the left of the pivot element. If the element is strictly superior to the pivot element, it is put on the right of the pivot element. This operation is called partitioning, and is recursively called until all the elements are sorted.

2) *Method*

Quicksort is a divide and conquer algorithm which relies on a partition operation.

3) Stability

Quicksort is not stable, that means Quicksort may change the relative order of records with equal keys.

4) Memory usage

Quicksort is in-place algorithm (don't need an auxiliary array).

5) Coding Algorithm

QuickSort

```
1. If p < r then
2.    q Partition (A, p, r)
3.    Recursive call to Quick Sort (A, p, q)
4.    Recursive call to Quick Sort (A, q + r, r)
```

PARTITION (A, p, r)

```
1.  x ← A[p]
2.  i ← p-1
3.  j ← r+1
4.  while TRUE do
5.     Repeat j ← j-1
6.     until A[j] ≤ x
7.     Repeat i ← i+1
8.     until A[i] ≥ x
9.     if i < j
10.       then exchange A[i] ↔ A[j]
11.       else return j
```

6) Complexity

The complexity of the quick-sort algorithm, for sorting a N elements array, is not always $O(n \log_2 n)$.

In fact, it can vary between ($n \log_2 n$ and n^2 $O(n \log_2 n)$).The complexity of the quick-sort algorithm is essentially $O(n \log_2 n)$. when the array's elements are not almost all sorted (assuming the pivot element is the first one of the array). The complexity is the same (close to a factor) for the heapsort algorithm. Quicksort takes $O(n \log_2 n)$ time on average, when the input is a random permutation.

In the most unbalanced case, each time we perform a partition we divide the list into two sublists of size 0 and n-1 (for example, if all elements of the array are equal). This means each recursive call processes a list of size one less than the previous list. Consequently, we can make n-1 nested calls before we reach a list of size 1. This means that the call tree is a linear chain of n-1 nested calls. The ith call does O(n-i) work to do the partition, and $\sum_{i=0}^{n}(n-i) = O(n^2)$, so in that case Quicksort take $O(n^2)$ time. That is the worst case: given knowledge of which comparisons are performed by the sort, there are adaptive algorithms that are effective at generating worst-case input for quicksort on-the-fly, regardless of the pivot selection strategy.

In the most balanced case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size. Consequently, we can make only $\frac{\log n}{\log 2}$ nested calls before we reach a list of size 1. This means that the depth of the call tree is $\frac{\log n}{\log 2}$. But no two calls at the same level of the call tree process the same part of the original list; thus, each level of calls

needs only O(n) time all together (each call has some constant overhead, but since there are only O(n) calls at each level, this is subsumed in the O(n) factor). The result is that the algorithm uses only O(n log n) time.

In fact, it's not necessary to be perfectly balanced; even if each pivot splits the elements with 75% on one side and 25% on the other side (or any other fixed fraction), the call depth is still limited to $\frac{\log n}{\log \frac{4}{3}}$, so the total running time is still O(n log n).

So what happens on average? If the pivot has rank somewhere in the middle 50 percent, that is, between the 25th percentile and the 75th percentile, then it splits the elements with at least 25% and at most 75% on each side. If we could consistently choose a pivot from the two middle 50 percent, we would only have to split the list at most $\frac{\log n}{\log \frac{4}{3}}$ times before reaching lists of size 1, yielding an O(nlog n) algorithm.

When the input is a random permutation, the pivot has a random rank, and so it is not guaranteed to be in the middle 50 percent. However, when we start from a random permutation, in each recursive call the pivot has a random rank in its list, and so it is in the middle 50 percent about half the time. That is good enough. Imagine that you flip a coin: heads means that the rank of the pivot is in the middle 50 percent, tail means that it isn't. Imagine that you are flipping a coin over and over until you get k heads. Although this could take a long time, on average only 2k flips are required, and the chance that you won't get $k$heads after 100k flips is highly improbable (this can be made rigorous using Chernoff bounds). By the same argument, Quicksort's recursion will terminate on average at a call depth of only $2(\frac{\log n}{\log \frac{4}{3}})$. But if its average call depth is O(log n), and each level of the call tree processes at most n elements, the total amount of work done on average is the product, O(n log n). Note that the algorithm does not have to verify that the pivot is in the middle half—if we hit it any constant fraction of the times, that is enough for the desired complexity.

An alternative approach is to set up a recurrence relation for the T(n) factor, the time needed to sort a list of size n. In the most unbalanced case, a single Quicksort call involves O(n) work plus two recursive calls on lists of size 0 and n-1, so the recurrence relation is:

$$T(n) = O(n) + T(0) + T(n-1) = O(n) + T(n-1).$$

This is the same relation as for insertion sort and selection sort, and it solves to worst case $T(n) = O(n^2)$. .

In the most balanced case, a single quicksort call involves O(n) work plus two recursive calls on lists of size $\frac{n}{2}$, so the recurrence relation is

$$T(n) = O(n) + 2T(\frac{n}{2})$$

The master theorem tells us that T(n) = O(n log n).

B. *Heapsort*

Heapsort is a comparison-based sorting algorithm to create a sorted array (or list),

1) *How It Works*

The idea is to look at the array to sort as it was a binary tree. The first element is the root, the second is descending from the first element.

The aim is to obtain a heap, thus a binary tree verifying the following properties:

a) The maximal difference between the two leaves is 1 (all the leaves are on the last, or on the penultimate line).

b) The leaves having the maximum depth are "heaped" on the left.

c) Each node has a bigger value than its of its two children, for an ascending sort.

2) Method

Heapsort is a much more efficient version of selection sort.

3) Stability

HeapSort is not stable, that means heapsort may change the relative order of records with equal keys.

4) Memory usage

HeapSort is in-place algorithm (don't need an auxiliary array).

5) Coding Algorithm

HEAPSORT (A)

1. BUILD_HEAP (*A*)
2. for i ← length (A) down to 2 do
   exchange A[1] ↔ A[i]
   heap-size [A] ← heap-size [A] - 1

Heapify (A, 1)

BUILD_HEAP (A)

1. heap-size (A) ← length [A]
2. For i ← floor(length[A]/2) down to 1 do
3. Heapify (A, i)

Heapify (A, i)

1. l ← left [i]
2. r ← right [i]
3. if l ≤ heap-size [A] and A[l] > A[i]
4. then largest ← l
5. else largest ← i
6. if r ≤ heap-size [A] and A[i] > A[largest]
7. then largest ← r
8. if largest ≠ i
9. then exchange A[i] ↔ A[largest]
10. Heapify (A, largest)

6) Complexity

The complexity of the heap-sort algorithm, for sorting a n elements array, is $O(n \log_2 n)$.

Let T(n) be the time to run Heapsort on an array of size n. Examination of the algorithms leads to the following formulation for runtime:

$$T(n) = TB(n) + \sum_{k=1}^{n-1} TH(k) + \theta(n-1)$$

TB is the time complexity of BuildHeap .

TH is the time complexity of Heapify.

HEAPIFY.

Since Heapify is also used in Buildheap, we will attack it first:

$$TH(n) = \theta(1) + TH(\text{size of subtree})$$

So we need to know how big the subtrees of a heap with n elements can be. A first guess might be that a subtree can only be half as big as the tree, since a heap is a binary tree. However, a little reflection shows that a better guess is 2/3:

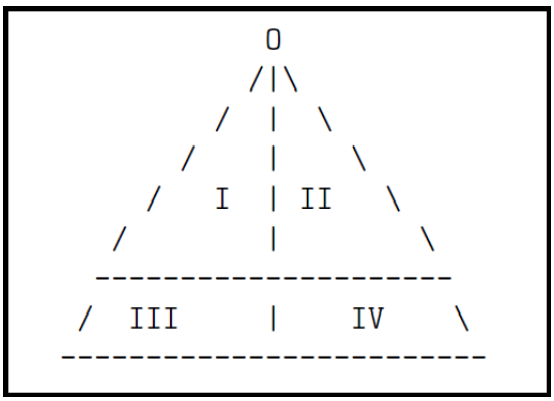

Figure 1. Binary tree

In a complete binary tree, each of the regions shown has approximately the same number of nodes. In a heap, region IV may be empty while region III is full. Since the left subtree consists of regions I and III, it has approximately 2/3 of the nodes in the heap.

This argument can be formalized. Let _ be the depth of heap A, and let n be the number of nodes in A.

$$2^l \leq n < 2^{l+1}$$

The right subtree of *A* cannot be larger than the left subtree. A complete tree of height *h* has $2^{h+1} - 1$ nodes. Regions I and II in Figure.1 each have $2^{l-1} - 1$ nodes. Region III has 2^{l-1} nodes, and the size of the left subtree is $2^l - 1$.

Let $n = 2^l + k$, where $0 \leq k < 2^l$. The left subtree contains $2^{l-1} + j$ nodes and $0 \leq j < 2^{l-1}$.

If $i = k - j, 0 \leq i \leq 2^{l-1}$,then $n = 2^l + j + i$ and

$$\frac{size\ of\ left\ subtree}{n} = \frac{2^{i-1}+j}{2^i+j+i} \leq \frac{2^{i-1}+j}{2 \cdot 2^{i-1}+j} = f(j)$$

Let $\propto = 2^{l-1}$. Then

$$f(j) = \frac{\propto + j}{2\propto + j}.$$

Extend *f* to [0, α] and analyze:

$$f(0) = \frac{\propto +0}{2 \propto +0} = \frac{1}{2}$$

$$f(\propto) = \frac{\propto + \propto}{2 \propto + \propto} = \frac{2}{3}$$

$$f`(j) = \frac{\propto}{(2 \propto +j)^2} \geq 0$$

So *f* is an increasing function that takes on a maximum at the left endpoint, and the max is 2/3. We state our result as a theorem: If a heap *A* has size *n*, its subtrees have size less than or equal to 2*n*/3.

Applying this theorem to Equation (2), we have:

$$TH(n) = TH\left(\frac{2n}{3}\right) + \theta(1).$$

Using Case 2 of the Master Theorem, we have

$$TH(n) = \theta(\log n)$$

- *Buildheap*

The complexity of *Buildheap* appears to be $\theta(n \log n) - n$ calls to *Heapify* at a cost of $\theta(\log n)$ per call, but this result can be improved to Θ(n). The intuition is that most of the calls to heapify are on very short heaps.

Putting our results together with Equation (1) gives us the following run-time complexity for Heapsort:

$$T(n) = TB(n) + \sum_{k=1}^{n-1} TH(k) + \theta(n-1)$$

$$= \theta(n) + \sum_{k=1}^{n-1} \log k + \theta(n-1)$$

$$= \theta(n \log n)$$

The main characteristics of Quick sort and Heap sort are listed in the table bellow

		Quick Sort	Heap Sort
Method		Divide & Conquer (partition)	Selection
Stability		NOT stable	NOT stable
Memory usage		In place	In place
Complexity	Best	(n log n)	(n log n)
	Average	(n log n)	(n log n)
	Worst	(n^2)	(n log n)

C. ***which one is the best :***

Heapsort is typically somewhat slower than quicksort, but the worst-case running time is always Θ(nlogn). Quicksort is usually faster, though there remains the chance of worst case performance except in

the introsort variant, which switches to heapsort when a bad case is detected. If it is known in advance that heapsort is going to be necessary, using it directly will be faster than waiting for introsort to switch to it.

If you want to predict precisely when your algorithm will have finished, it is better to use the heap-sort algorithm, because the complexity of this one is always the same. It is also the case if you know in advance that your array is almost sorted.

3. Quicksort & Heapsort Enhancement steps

Many updates have been made on the both algorithms to improve their performance. In this section the most significant changes on each one of them are reviewed.

1. Quick sort

Many studies and researches have been made on Quicksort algorithms. These studies led to improve the algorithm performance, as well as, it introduces new enhanced versions of it.

Since its development in 1961 by Hoare, the Quicksort algorithm has experienced a series of modifications aimed at improving the worst case behavior. The improvements can be divided into four categories: improvements on the choice of pivot, algorithms that use another sorting algorithm for sorting sublists of certain smaller sizes , different ways of partitioning lists and sublists, and adaptive sorting that tries to improve on the behavior of the Quicksort algorithm when used for sorting lists that are sorted or nearly sorted.

In 1965 Scowen [5] developed a Quicksort algorithm called Quickersort in which the pivot is chosen as the middle key in the array of keys to be sorted. If the array of keys is already sorted or nearly sorted, then the middle key will be an excellent choice since it will split the array into two subarrays of equal size. Choosing the pivot as the middle key, the running time on sorted arrays becomes

```
void partition(int A[ ], int i, int j)
1.  int pivot = A[(i+j)/2];
2.  int p = i – 1;
3.  int q = j + 1;
4.  for(;;)
5.          do q = q – 1; while A[q] > pivot;
6.          do p = p – 1; while A[p] < pivot;
7.          if (p < q)
8.                  exchange (A[p], A[q]);
9.          Else
10.                 return q;
```

Another improvement to Quicksort was introduced by Singleton in 1969, in which he suggested the median-of-three method for choosing the pivot. One way of choosing the pivot would be to select the three elements from the left, middle and right of the array. Then, the three elements sorted and placed back into the same positions in the array. The pivot is the median of these three elements. The median-of-three method improves Quicksort in three ways [6]: First, the worst case is much more unlikely to occur. Second, it eliminates the need for a sentinel key for partitioning, since this function is served by one of the three elements that are examined before partitioning. Third, it reduces the average running time of the algorithm by about 5%. The following is a C++ implementation for the Quicksort algorithm that uses the median-of-three method for choosing the pivot:

```
void Singleton( long unsigned a[], int left, int right )
1.  int i, j;
2.  long unsigned pivot;
3.  if (left + CUTOFF <= right)
4.          /* CUTOFF = 20 */
5.          pivot = median3( a, left, right );
6.          i = left; j = right-1;
7.          for (;;)
8.              while (a[++i] < pivot);
9.                  while (a[--j] > pivot);
10.                     if (i < j)
11.                         swap( i, j );
12.                     Else
13.                         break;
14.         swap( i, right-1 );
15.         q_sort( a, left, i-1 );
16.         q_sort( a, i+1, right);
```

Quicksort with three way partitioning has been suggested as the method for handling arrays with duplicate keys . Basically, the array of keys is partitioned into three parts: one part contains keys smaller than the pivot, the second part contains keys equal to the pivot, and the third part contains all keys that are larger than the pivot.

In 1978, Sedgewick suggested a version of Quicksort that minimizes the number of swaps by scanning in from the left of the array then scanning in from the right of the array then swapping the two numbers found to be out of position. This algorithm is called SedgewickFast which was declared to be a fast implementation of Quicksort. the algorithm makes comparisons, while for sorted data the number of comparisons is linear. The following is a C++ implementation of the partition algorithm:

```
void Partition(int low, int high, int& pos)
1.  int i, j;
2.  int pivot;
3.  i = low-1;
4.  j = high;
5.  pivot = ar[high];
6.  for(;;)
7.          while (ar[++i] < pivot);
8.          while (pivot < ar[--j])
9.                  if (j == low)
10.                     break;
11.                 if (i >= j)
```

```
12.                     break;
13.                 swap(i, j);
14.  swap(i, high);
15.  pos = i;
```

In 1984, Bentley developed a version of Quicksort based on an algorithm suggested by Nico Lomuto, in which the partition function uses a for loop to roll the largest keys in the array to the bottom of the array. To partition the array around the pivot T (chosen at random) a simple scheme attributed to Nico Lomuto of Alsys, Inc is used. Given the pivot T, an index called LastLow is computed and used to rearrange the array X[A]…X[B] such that all keys less than T are on one side of LastLow, while all other keys are on the other side.

Bsort is a sorting algorithm developed by Wainwright in 1985 which is a variation of Quicksort. It is designed for nearly sorted lists as well as lists that are nearly sorted in reverse order. The algorithm combines the interchange technique used in Bubble sort with the Quicksort algorithm. The algorithm chooses the middle key as the pivot during each pass, then it proceeds to use the traditional Quicksort method.

Qsorte is a quicksort algorithm with an early exit for sorted arrays developed by Wainright in 1987. It is based on the original Quicksort algorithm with a slight modification in the partition phase, where the left and right sublists are checked to see if they are sorted or not. The algorithm chooses the middle key as the pivot in the partitioning phase. But Qsorte has a worst case time complexity of O(n2) . This occurs when the chosen pivot is always the smallest value in the sublist. Qsorte will repeatedly partition the sublist into two sublists where one of the sublists only contains one key[7]. The following is a C++ implementation of the algorithm:

```
void Qsorte (int m, int n)
1.  int k, v;
2.  bool lsorted, rsorted;
3.  if ( m < n )
4.          FindPivot (m, n, v);
5.          Partition (m, n, k, lsorted, rsorted);
6.          if (! lsorted)
7.                  Qsorte(m, k-1);
8.          if (! Rsorted)
9.                  Qsorte(k, n);
```

McDaniel in 1991 provided variations on Hoare's original partition algorithm. In one version called the Rotate version, the pivot is compared with the value in the bottom'th position. If the pivot is less than that value, then the bottom index is decremented. Otherwise a rotate left operation is performed using the call Rotate_Left(bottom,pivot+1,pivot).

```
void Rotate_Left(int a,int b,int c)
1.  int t;
2.  t = ar[a];
3.  ar[a] = ar[b];
4.  ar[b] = ar[c];
```

```
5.  ar[c] = t;
void Partition(int left,int right, int& pos)
1.  int pivot, bottom;
2.  pivot = left;
3.  bottom = right;
4.  while (pivot < bottom)
5.          if (ar[pivot] > ar[bottom])
6.                  Rotate_Left(bottom,pivot+1,pivot);
7.                  pivot = pivot + 1;
8.          Else
9.                  bottom = bottom - 1;
```

Comparison between enhanced algorithms (Random Data)

	Average running time	Average number of comparison
Original	n^2	
Quickersort	(n log n)	(n log n)
Singleton		(n log n)
SedgewickFast	(n log n)	n
Nico		(n log n)
Bsort		
Qsorte	n^2 (worst case)	(n log n)

Comparison between enhanced algorithms (Sorted Data)

	Ordered by average running time	Average number of comparison
Original	4	n log n
Quickersort	3	n log n
Singleton	2	n log n
SedgewickFast		n^2
Nico	5	n log n
Bsort	1	n
Qsorte	1(fastest)	n

Comparison between enhanced algorithms (Reverse Order)

	Ordered by average running time	Average number of comparison(n log n)
Original	5	
Quickersort	3	4
Singleton	4	3
SedgewickFast	6	1
Nico	6	
Bsort	2	2
Qsorte	1(fastest)	1(fewest comparisons)

2. Heapsort

Heapsort is simply an implementation of the general selection sort using the input array L as a heap representing a descending priority queue. This is a two-stage algorithm that works as follows [11]:

1) (Preprocessing Phase) Heap construction – Construct the initial heap out of the list L given as input.

2) (Selection Phase) Maximum deletion – Apply the root deletion operation n - 1 times to the remaining heap. This procedure involves the reconstruction of heap after construction of heap and the root node and the largest numbered node have been swapped.

The variants of heapsort algorithms were based on changes in one of those stages.

D. Basic Heapsort

Heap sort was originally proposed by William in 1964 [10]. A heap of size n is an array a[1..n] containing n elements satisfying the following conditions:

(1) Each component of the array stores exactly one element (2) The array represents a binary tree completely filled on all levels except possibly at the lowest, which is filled from the left up to appoint

(3) The root of the tree is a[1]

(4) for a node I in the binary tree, a[i] is its key parent(i) = |_ i/2 _| is its parent and 2i and 2i+1 are its children, if they exist; (5) The heap property is for all $2 \leq i \leq n$, a{parent(i)}$\geq$ a[i].Thus the largest element in a heap is always at the root of the heap.

There are two principle alternatives for constructing a heap,

referred as heapifying, for a given list of keys [10]:

1) Top-down heap construction,

2) Bottom-up heap construction.

In the top-down heapifying, used in the original articles by Williams and Johnson, the shift down path is traversed down as long as the node under consideration is not a leaf and the element stored at that node is larger than the element put aside. Simultaneously, the elements met are moved one level up to get the hole down. When the traversal stops, the element put aside is stored at the hole. If the depth of the tree is *d*, at

most *d* element comparisons are done at each level: at most *d*-1 to find the maximum element stored at the children and one to determine whether we should stop or not.

```
SIFT-UP(int i)

1.  if i ≥ 2 then
2.          k = |_ i/2 _|
3.          if A[i] < A[k] then
4.                  SWAP(i; k)
5.                  SIFT-UP(k)
```

```
TOPDOWN-HEAP

1.  for i = 1 to n do
2.      SIFT-UP(i)
```

In the worst case, the i-th item moves up all the way to the root. This approach TOPDOWN-HEAP requires O(n log n) time because each SIFT-UP takes O(log n) time and there are $n$elements. Algorithm of William uses nlogn + O(n) comparisons in the worst case to build a heap on n elements and more than 2nlogn comparisons in the worst case to sort the elements. However this is not the optimal method.

In 1965, Floyd proposed an improved technique for building a heap with better average case performance and a worst case of *O*(*n)* steps to construct the initial heap in time O(*n*) by building it from bottom to top.

```
SIFT-DN(int i)

1.  if 2i _ n then
2.  k = 2i; // left child
3.  if k + 1 _ n and A[k + 1] < A[k] then
k = k + 1  // right child

4.  if A[k] < A[i] then
SWAP(i; k);

SIFT-DN(k)
```

```
BOTTOMUP-HEAP

1.  for i = n downto 1 do
2.  SIFT-DN(i)
```

His algorithm uses 2n comparisons in the worst case to build a heap. The sorting phase requires at most 2nlogn comparisons. The average case is hardly better than the worst case. Williams' base algorithm with Floyd's improvement is still the most prevalent heapsort variant in use.

E. Bottom up Heapsort

A variant of Heap sort proposed by Carlson in 1987 [12]. In this improved version, no comparisons are made between a node and its children. Instead, a path of maximum sons is found, by letting the larger of the sons of the root be in that list, and connecting it with the path of maximum sons for that node. A binary insertion in such a path can be performed, since any path from the root to a leaf in a heap is an ordered list. The length of that path is at most [log k]. The only part of the HEAPSORT algorithm that has to be changed

is the rearrangement procedure. First, a path of maximum sons is found. Choose the element to be inserted as the root of the heap to be rearranged during the creation of the heap, and the last leaf during the sorting. Then, a binary search for the place of the element, which has to be inserted in the path, is performed. This can easily be done after observing that any node j has the ancestors j div 2 k, where k is the length of the path between the node and its ancestor (j div 2^k can be computed by shifting j k steps to the right).

Finally, all elements that are larger than the last leaf are moved one step up, and the element is inserted. Note that the path of maximum sons depends only on the current configuration of the heap and not on the value of the element, which is to be inserted. In the procedure HEAPSORT, the auxiliary procedure REARRANGE is at first used for the creation of the heap and then for the necessary rearrangement after removing the current maximum element from the root of the heap.

```
REARRANGE( A , I,  N )

1.  define J,K, NrOfLevels, Bot, Top, Mid : integer;
2.   J=2*I
3.  //  * * * LINEAR SEARCH * * *
4.  while J < N do
5.   if A[J] < A[J+I] then
6.          J=2*(J+I)
7.  else J == 2 * J
8.  if J>N then
9.          J=J/2                  // J is now a leaf
10. // * * * BINARY SEARCH * * *
11. NrOfLevels = log (J / I)  // Number of levels between  I and J
12. ToP = NrOfLevels
13. BOT = 0
14. while Top > Bot do
15.         Mid = (Top + Bot) / 2
16.         if X < A[J/2^Mid] then
17.                Top = Mid
18. else Bot = Mid + 1
19. // * * *  INSERTION * * *
20. for K = NrOfLevels-1 down to Top do
21. A[J / 2^(K+1)] = A[J / 2^K]
22. A[J / 2^Top] = X
```

```
HEAPSORT( A , N )

1.  // This procedure is similar to the usual HEAPSORT
2.  define I : integer
3.  define Temp: elementtype
4.  // * * * HEAP CREATION * * *
5.  for I = N/2 down to 1 do
6.          REARRANGE(A, A(I), I, N);
7.  // * * * SORTING * * *
8.  for I = N-1 down to 1 do
9.         Temp = A[I + 1]
10.        A[I + 1] := A[1]
11.        REARRANGE(A, Temp, 1, I)
```

The maximum number of comparisons during one call of the REARRANGE procedure is h + |_ log h _| + 1, where h is the height of the heap to be rearranged. If the number of elements to be sorted is 2^k- 1, the number of comparisons between elements can be show as ½(n + 1) heaps of height log(n + 1) - 1, ¼(n + 1) heaps of height log(n + 1)- 2, and so on, are rearranged to give total ≤ (n + 1)(log(n + 1) + log log(n + 1)).

In 1990 [13], Wegner proved that upper bound for comparisons in Bottom-up-heap sort of 1.5nlogn.

F. MDR-Heapsort

A variant of bottom-up heapsort has been presented by McDiarmid and Reed in 1989 which we call MDR-HEAPSORT.

In many cases procedure reheap needs almost 2d comparisons, if d is the depth of the heap. We like to get by with approximately d comparisons. Therefore we use only 1 comparison at each vertex. With 1 comparison we can decide which son of the vertex just considered contains the smaller object and we go to this son. By this procedure it is not possible to stop at the correct position, since we do not consider the object at the root. Hence, we walk down a path in the heap until we reach a leaf. This path will be called special path, and the last object on this path will be called special leaf: This part of the algorithm is done by the procedure leaf-search, which returns the special leaf. The procedure Reheap of heapsort places the root object at some position p of the special path and all objects on the special path from the root to position p are shifted ‘into their father vertices. Bottom-up heapsort and MDR-HEAPSORT look for the same position p. But these algorithms start their search from the special leaf and search bottom-up by the procedure bottom-up search. Finally, the procedure interchange performs the data transport. Hence, all heapsort variants construct the same heaps.

Let d be the length of the special path and let j be the length of the path from the root of the special path to the vertex at position p. The procedure reheap of HEAPSQRT needs 2 min { d, j + 1 > comparisons while the procedure Bottom-up reheap needs d+ min(d, d-j+ 1 } comparisons (see below). For large j we save almost half of the comparisons.

With MDR-HEAPSORT we save even more comparisons by using old information. MDR-HEAPSORT works with an extra array called info. The possible values of info(j) are:

- *Unknown, abbreviated by u,*
- *Left, abbreviated by l,*
- *Right, abbreviated by r*

and can be coded by 2 bits. The interpretation is the following. If info(j) = i (or r), then the left son contains a smaller object than the right one (or vice versa). If info(j) = u, nothing is known about the smaller son. We need info(j) only for j= 1,.., |_(n - 1)/2_|, since the other nodes do not have two sons. Hence, 2|_(n - 1)/2_| < n extra bits are sufficient. The parameters are initialized as u. Now we are prepared to describe the procedure MDR-REHEAP:

MDR-REHEAP (m , i)

1. LEAF-SEARCH (m, i).
2. BOTTOM-UP search (i, j).
3. INTERCHANGE (i, j).

LEAF-SEARCH(m , i)

```
1.  j=i.
2.  while 2j < m do begin
3.  if info(j) = l then
4.          j = 2j
5.          else if info(j) = r then
6.                  j = 2j+ 1,
7.                  else if a(2j) < a(2j+ 1) then
8.                          info(j) = l
9.                          j = 2j
10.                         else info(j) = r
11.                                 j = 2j+ 1.
12. if 2j=m then
13.         j=m.
14. return j
```

BOTTOM-UP SEARCH(i, j) // j is the output of leaf-search

```
1.  while (i<j and a(i)<a(j)) do
2.   j= |_ j/2 _|
3.  return
```

The procedure bottom-up search walks the special path bottom-up until the new position for the root object is found. The data transport is done by a cyclic shift on the special path from the root i to the position j computed by bottom-up search. By bin(j) we denote the length of the binary representation of j.

INTERCHANGE(i, j) // j is the output of bottom-up-search

```
1.  l = bin(j) - bin(i)
2.   x = a(i).
3.  for k= l – 1 down to 0 do
4.      a|_ j/2^(k+1) _| = a|_ j/2^k _|
5.      info(a|_ j/2^(k+1) _|) = u
6.  a(j): = x.
```

Now we analyze the call of mdr-reheap with this actual special path. The number of comparisons during the procedure leaf-search is k + 1, and the number of comparisons during the procedure bottom-up-search is min{d, d-j+ l), since the root object is not compared with itself. We estimate the number of comparisons by: $k+l+d-j+l=d+l-(d-l-k)+(d-j)$.

We interpret this number in the following way.

- d is the length of the special path; i.e., d is either the depth of the heap or one less than the depth.
- 1 stands for the comparison between b(0) and b(d).
- d-1-k is the number of "old" pebbles on this path before the call of procedure leaf-search (this number is d - 2 -k, if b(d - 1) has only one son).
- d-j is the number of "new" pebbles on this path after the call of the procedure interchange (this number is d- 1 -j, if b(d- 1) has only one son).

We sum up these terms in order to estimate the worst case number of comparisons.

The algorithm starts without any pebble and stops without any pebble, since the last heap consists of two nodes and, therefore, it does not contain any node with two sons. Let OP be the sum of the numbers of "old" pebbles and NP be the sum of the numbers of "new" pebbles. Then NP - OP is the number of vanishing pebbles. This number is bounded by |_(n - 1)/2_|, since pebbles lie only on the nodes 1, .. , |_(n - 1)/2_| and since for each node only one pebble may vanish.

Now we sum the terms "1"; i.e., we count the number of calls of procedure bottom-up search. This number equals n+|_n/2_| - 2. At last we sum the depths of the considered heaps.

LEMMA 1. The sum of the depths of the heaps considered during the heap creation phase is in the interval [n - |_logn_| - 1, n – 1].

LEMMA 2. The sum of the depths of the heaps considered during the selection phase equals n|_logn_| - $2^{\lfloor \log n \rfloor + 1}$ + |_logn_| + 2.

We like to express n|_logn_| - $2^{\lfloor \log n \rfloor + 1}$ as n log n - A(n)n. Hence, A(n): =log n- |_logn_| - $2^{\lfloor \log n \rfloor + 1}$ /n.

We consider the interval l= [2^k, 2^{k+1}) such that |_logx_| = k for x ∈ l. It is easy to see that A(x) takes its maximum for x=2^k, A(2^k)= 2, and A(x) takes its minimum for x = (2 In 2)2^k, where A((2 In 2)2^k) = log(2 In 2) + (In 2)$^{-1}$ ≥ 1.913928.

The worst case number of comparisons of MDR-HEAPSORT is bounded by

n log n + (3 - A(n))n + |_logn_| - 1≤(n + 1) log n + 1.086072n

For n = 2^k, the number of comparisons can be bounded by (n + 1) log n + n. However, Wegner showed that McDiarmid and Reed's variant of Bottom-up-heap sort needs nlogn+1.1 n comparisons in 1991 [14].

G. WEAK HEAPSORT

WEAK HEAPSORT proposed by Dutton in 1993 [15]. 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.

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. Notice, because of (3) in the definition, the subtree T_i has between 2^{k-i}-1 and 2^{k-i+1}-1 nodes.

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. Merging produces a binary tree satisfying conditions (1) and (2) of the weak-heap definition, but not necessarily condition (3).

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.

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.., n - 1] is given. It then rearranges them into h[1.., n] in ascending order. Routines WeakHeapify and MergeForest invoke two other routines, Gparent and Merge. The algorithm Gparent acts as a function that returns the Grandparent of a given node j > 0. Next, 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. Since each call to Merge executes one data element comparison and the function Gparent uses none, WeakHeapify requires exactly n - 1 compares.

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]. The maximum number of data compares required by WeakHeapSort is less than (n - 1)log(n) + 0.086013n.

In 1996, Katajainen uses a median-finding procedure to reduce the number of comparisons required by Bottom-Up-Heapsort in the worst-case, completely eliminating the sift-up phase. This idea has been further refined by Rosaz in 1997. It is to be noted, though, that the algorithms work “in-place” and perform no more than n log n + O(n) key comparisons and n log n + O(n) element moves in the worst case, but they are mostly of theoretical interest only, due to the overhead introduced by the median-finding procedure.

Comparison between Heapsort enhanced algorithms

	At most number of comparison
Basic	2 nlogn
BottomUp	1.5 nlogn
MDR	nlogn+1.1 n
Weak	(n - 1)log(n) + 0.086013n
Median-finding	n log n + O(n)

4. QuickHeapsort

Cantone and Cincotti[16] combine Quicksort and Heapsort in one efficient algorithm. We mainly focus our attention on the number of comparisons, since this often represents the dominant cost in any reasonable implementation. Accordingly, to sort n elements the classical Quicksort algorithm performs on the average 1.386n log n – 2.846n + 1.386 log n key comparisons and $O(n^2)$ key comparisons in the worst-case, whereas Floyd’s version of the Heapsort algorithm performs 2n log n + ɵ (n) key comparisons in the worst-case. QuickHeapsort combines the Quicksort partition step with two adapted min-heap and max-heap variants of the Heapsort algorithm. Analogously to Quicksort, the first step of QuickHeapsort consists in choosing a pivot, which is used to partition the array. We refer to the sub-array with smaller size as heap area and to the one with larger size as work area. Depending on which of the two sub-arrays is taken as heap-area, the adapted max-heap or min-heap variant of Heapsort is applied and the work area is used as an external array. At the end of each stage, the elements moved in the work area are in correct sorted order and the remaining unsorted part of the array can be processed iteratively in the same way.

Depending on which sub-array is taken as heap area, the adapted max-heap or minheap variant of Heapsort is called, with the work area used as external array. Moreover occurrences of keys contained in the work area are used in place of the infnity values ±∞.

More precisely, if A[1…Pivot−1] is the heap area, then the max-heap version of Heapsort is applied to it using the right-most region of the work area as external array. In this case, at the end of the stage, the right-most region of the work area will contain the elements formerly in A[1…Pivot−1] in ascending sorted order, and the elements previously contained in the right-most region of the work area have been moved in the heap area. Similarly, if A[Pivot + 1… n] is the heap area, then the min-heap version of Heapsort is applied to it using the left-most region of the work area as external array. In this case, at the end of the stage, the left-most region of the work area will contain the elements formerly in A[Pivot + 1…. n] in ascending sorted order.

The element A[Pivot] is moved in the correct place and the remaining part of A[1…. n], i.e. the heap area together with the unused part of the work area, is sorted iteratively in same fashion.

QuickHeapsort sorts n elements in-place in O(n log n) average-case time. More specifically, it performs on the average n log n+2:996n+O(n) key comparisons and n log n + 2:645n + O(n) element moves.

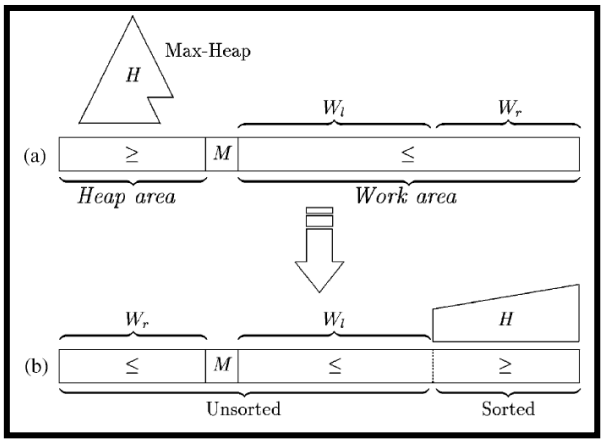

Figure 2: QuickHeapsort

5. Conclusion

Quicksort and heapsort has been described in comparative analysis. The difference between them helps in determining the best usage of them. Furthermore, working on improve such sorting algorithm is needed in order to facilities how its applications work in more efficient way to reduce time & resources. For example, it helps in designing the memories and reduces the power that consumes. In addition, the benefit of comparitve study such the one we discussed help to take the advantages from each algorithm Quicksort and Heapsort and try to combine them efficiently as in we see in Quickheap sort.

References

[1] Thomas H. cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. (2001) Introduction to Algorithms. London: McGraw-Hill Book Company

[2] Sharma, V. and Singh, S. (2008) Performance Study of Improved Heap Sort Algorithm and Other Sorting Algorithms on Different Platforms. IJCSNS International Journal of Computer Science and Network Security, VOL 8, No 4

[3] Muthusundari, S. Baboo, S. (2011) A Divide and Conquer Strategy for Improving The Efficiency and Probability Success In Sorting. International Conference on Advanced Computing, Communication and Networks.

[4] C. A. R. Hoare, "Quicksort," Computer Journal, 5, pp 10 – 15 1962.

[5] R.S. Scowen, "Algorithm 271: Quickersort," Comm. ACM 8, 11, pp 669-670, Nov. 1965..

[6] R. Sedgewick, Algorithms in C++, 3rd edition, Addison Wesley, 1998.

[7] R. L Wainright, "Quicksort algorithms with an early exit for sorted subfiles," Comm. ACM, 1987.

[8] W. Dobosiewicz, "Sorting by distributive partitioning" Information Processing Letters 7, 1 – 5., 1978.

[9] A. LaMarca and E. Ladner, "The Influence of Caches on the Performance of Sorting," Journal of Algorithms 31**,** 66-104 ., 1999.

[10] Williams, J.W.J., 1964. ACM algorithm 232: Heap sort. Commun. ACM., 7: 347-348.

[11] A. Levitin. Introduction to the design & Analysis of Algorithms. Pearson Education, 1st edition, 2008.

[12] S.Carlsson: A variant of HEAPSORT with almost optimal number of comparisons. Information Processing Letters, 24: 247-250, 1987.

[13] I.Wegner: BOTTOM-UP-HEAPSORT beating on average QUICKSORT (if n is not very small). Proceedings of the MFCS90, LNCS 452: 516-522, 1990

[14] I.Wegner: The worst case complexity of Mc diarmid and Reed's variant of BOTTOM-UP-HEAP SORT. Proceedings of the STACS91, LNCS 480: 137-147, 1991.

[15] Dutton R D. Weak Heap Sort. BIT, 1993, 33(3): 372-381.

[16] D. Cantone and G. Cincotti, "QuickHeapsort, an Efficient Mix of Classical Sorting Algorithms," presented at the Proceedings of the 4th Italian Conference on Algorithms and Complexity.