A COMPARATIVE STUDY OF LINKED LIST SORTING ALGORITHMS

by Ching-Kuang Shene[1]
Michigan Technological University
Department of Computer Science
Houghton, MI 49931-1295
shene@mtu.edu

1 Introduction

Carraway recently published an article [2] describing a sorting algorithm (the sediment sort) for doubly linked lists. He claimed that the sediment sort is one of the fastest and most efficient sorts for linked list, and planned to determine its complexity. In this article, a comparative study will be presented to show that the sediment sort is only a minor variation of the bubble sort which has been known to the computer science community for more than three decades and that the sediment sort is perhaps the slowest algorithm for sorting linked lists.

In my data structures class I taught two years ago, students were required to compare various sorting algorithms for arrays of different size. It was followed by a study of fine tuning the quick sort by removing recursion, using median-of-three, and sorting small files with other algorithms. Students were asked to run their programs with different small file sizes and to choose an optimal one. They also ran the same program under different hardware (PCs and SPARCstations) with different compilers (Borland C++, Turbo C++, Microsoft C, and GCC). Different configuration yields different optimal size. Students were excited about this approach because they believed they learn something "practical" rather than a theoretical treatment of different algorithms.

Students were also encouraged to compare linked list sorting algorithms with tree-based ones (binary search trees, AVL trees, and B-trees). Usually, bucket sort was chosen as a benchmark since it is a linear algorithm. Although I have not been teaching a data structures course for two years, it would be worthwhile to share some sample solutions and comparison results in my file with other educators. As a result, in addition to the sediment sort, five other linked list sorting algorithms are selected for a comparative study. They are bubble sort, selection sort, merge sort, quick sort, and tree sort. All of them take a linked list as input.

In the following, Section 2 reviews the similarity between the sediment sort and the traditional bubble sort. Section 3 gives a description of tree sort, which uses a doubly linked list implementation, while Section 4 presents the other four singly linked list sorting algorithms. Section 5 provides a comparison and finally Section 6 has our conclusion.

2 Sediment Sort

The sediment sort uses a bounding variable `new_tail`, which is set when a pair of nodes are swapped, to limit the range for next scan. This algorithm was known to the computer community much earlier and was discussed in Knuth's monumental work (Knuth [4]). Figure 1 is a direct translation from Knuth's description, where `SWAP()` is a macro that swaps two integers `a[i]` and `a[i+1]`. Notice the striking similarity between this one and the sediment sort.

```
void  BUBBLEsort(int a[], int n)
{
   int  bound = n-1, done = 0;
   int  swapped, i;

   do {
      swapped = -1;
      for (i = 0; i < bound; i++)
         if (a[i] > a[i+1]) {
            SWAP(a[i], a[i+1]);
            swapped = i;
         }
      if (swapped < 0)
         done = 1;
      else
         bound = swapped;
    } while (!done);
}
```

Figure 1: Bubble Sort

The complexity of this algorithm is $O(n^2)$. The worst case happens when the given array is reversely sorted and in this case exactly $n(n-1)/2$ comparisons and swaps are required. The "best" case, however, only

[1] This work was supported in part by a NSF grant CCR-9410707.

requires n-1 comparisons and no swap when the array is already sorted. Note that theoretically bubble sort is one of the several $O(n^2)$ algorithms, the others being the insertion sort and selection sort. However, since worst-case study usually does not always provide the average behavior of an algorithm, the remaining for me to do is a comparative study.

3 Doubly Linked List Sorting Algorithms

Two doubly linked list sorting algorithms are included in this study, the sediment sort and the tree sort. There is no need to repeat the sediment sort here and the interested reader should refer to [2] for the details.

Since a node in a doubly linked list has two fields, `prev` and `next`, pointing to the previous node and the next node, it is good enough for representing a binary tree. Therefore, we can use these fields to build a binary search tree and reconstruct a sorted doubly linked list as the binary search tree is traversed with inorder. Since building a binary search tree is quite popular, the following only describes the reconstruction phase.

```
static  NodePTR     head, tail;

void  Traverse(NodePTR  root)
{
     NodePTR  work;

     if (root != NULL) {
          Traverse(root->LEFT);
          work = root->RIGHT;
          APPEND_NODE(root);
          Traverse(work);
     }
}
```

Figure 2: Reconstruct a List from a Tree

Figure 2 shows a modified recursive inorder traversal of a binary search tree. Two static variables, `head` and `tail`, are set to `NULL` before calling `Traverse()`. Function `Traverse()` receives the current root pointer. If it is not `NULL`, the left subtree is traversed. Then, the pointer to `root`'s right subtree is saved to `work`, the root is appended to the end of the doubly linked list with head and tail pointers `head` and `tail`, and finally the right subtree pointed to by `work` is traversed. Note that the pointer to the right subtree must be saved before the root is appended to the doubly linked list since macro `APPEND_NODE` destroys `prev` and `next`.

As is well-known, the complexity of binary search tree insertion is $O(n^2)$, since in a binary search tree, except for one leaf, all nodes could have exactly one child and in this case the tree reduces to a linked list. However, if the input data are random, the resulting binary search tree could be reasonably balanced and the complexity would be approximately $O(n\log_2 n)$.

4 Singly Linked List Sorting Algorithms

Since a singly linked list has only one link field, any sorting algorithm for a singly linked list can only scan the list along one direction. Thus, the selection sort, insertion sort and bubble sort can easily be tuned into a list sorting algorithm. Although Shell sort can also be made into a list sorting algorithm, it could be inefficient since we have to step through nodes in order to find a neighboring node if the gap is greater than one.

An efficient implementation of heap sort requires an array that is accessed almost randomly (*i.e.*, accessing the index sequence i, $i/2$, $i/2^2$, and so on). Although it could be done with other heap data structures (see, for example, Weiss [8]), the material might be inappropriate for a CS2 type course.

For quick sort, Hoare's original algorithm [3] cannot be used since this algorithm “burns a candle from both ends”. Nico Lomuto's algorithm as described in Bentley [1] could be a better candidate for our study since it keeps two forward scanning pointers. However, since quick sort is not stable (Sedgewick [6]), it is not included. Instead, an algorithm which was originally designed to make quick sort stable and to handle equal keys is selected for this study. This algorithm was first proposed by Motzkin [5] and then analyzed by Wegner [7]. In fact, Wegner showed that on average this algorithm is of order $O((m+n)\log_2(n/m))$, where n is the number of keys in an input linked list in which each key occurs m times.

The idea of Wegner's algorithm is simple. Three linked lists are used, `less`, `equal` and `larger`.

The first node of the input list is chosen to be a pivot and is moved to `equal`. The value of each node is compared with the pivot and moved to `less` (*resp*., `equal` or `larger`) if the node's value is smaller than (*resp*., equal to or larger than) the pivot. Then, `less` and `larger` are sorted recursively. Finally, joining `less`, `equal` and `larger` into a single list yields a sorted one. Figure 3 shows the basic concept, where macro `APPEND()` appends the first argument to the tail of a singly linked list whose head and tail are defined by the second and third arguments. On return, the first argument will be modified so that it points to the next node of the list. Macro `JOIN()` appends the list whose head and tail are defined by the third and fourth arguments to the list whose head and tail are defined by the first and second arguments. For simplicity, the first and fourth arguments become the head and tail of the resulting list.

```
void Qsort(NodePTR *first, NodePTR *last)
{
   NodePTR lesHEAD=NULL, lesTAIL=NULL;
   NodePTR equHEAD=NULL, equTAIL=NULL;
   NodePTR larHEAD=NULL, larTAIL=NULL;
   NodePTR current = *first;
   int     pivot, info;

   if (current == NULL)
      return;
   pivot = current->data;
   APPEND(current, equHEAD, equTAIL);
   while (current != NULL) {
      info = current->data;
      if (info < pivot)
         APPEND(current,lesHEAD,lesTAIL)
      else if (info > pivot)
         APPEND(current,larHEAD,larTAIL)
      else
         APPEND(current,equHEAD,equTAIL);
     }
   Qsort(&lesHEAD, &lesTAIL);
   Qsort(&larHEAD, &larTAIL);
   JOIN(lesHEAD,lesTAIL,equHEAD,equTAIL);
   JOIN(lesHEAD,equTAIL,larHEAD,larTAIL);
   *first = lesHEAD;
   *last  = larTAIL;
}
```

Figure 3: Quick Sort

At a first glance, merge sort may not be a good candidate since the middle node is required to subdivide the given list into two sublists of equal length. Fortunately, moving the nodes alternatively to two lists would also solve this problem (Sedgewick [6]). Then, sorting these two lists recursively and merging the results into a single list will sort the given one. Figure 4 depicts the basic idea of this merge sort.

```
NodePTR  Msort(NodePTR  first)
{
   NodePTR  list1HEAD = NULL;
   NodePTR  list1TAIL = NULL;
   NodePTR  list2HEAD = NULL;
   NodePTR  list2TAIL = NULL;

   if (first==NULL || first->next==NULL)
      return  first;
   while (first != NULL) {
      APPEND(first,list1HEAD,list1TAIL);
      if (first != NULL)
        APPEND(first,list2HEAD,list2TAIL);
   }
   list1HEAD = Msort(list1HEAD);
   list2HEAD = Msort(list2HEAD);
   return  merge(list1HEAD, list2HEAD);
}
```

Figure 4: Merge Sort

Moreover, almost all external sorting algorithms can be used for sorting linked lists since each involved file can be considered as a linked list that can only be accessed sequentially. Note that one can sort a doubly linked list using its `next` fields as if it is a singly linked one and reconstruct the `prev` fields after sorting with an additional scan.

5 Comparisons

Of these six algorithms, two (sediment sort and tree sort) use a doubly linked list while the other four (bubble sort, selection sort, quick sort and merge sort) use a singly linked list. Due to the similarity between sediment sort and bubble sort, one can immediately conclude that the later is faster since fewer pointer manipulations are involved. Furthermore, the selection sort should be faster than the bubble sort since the former requires only n-1 swaps while the later may require as many as $n(n-1)/2$. Thus, for these three algorithms, the matter is not which one is faster than the other, but determining the relative efficiency.

All of these six algorithms were coded in ANSI C and `SWAP()`, `APPEND()` and `JOIN()` are C macros rather than functions except for the sediment

sort whose swap function is taken directly from Carraway's paper.[2] For those who love C++, these macros and variable parameters can easily be changed to `inline` functions and aliases, respectively. Each sorting algorithm is repeated several times sorting the same set of input to minimize timing error and the average elapsed time is recorded. The `clock()` function is used to retrieve the elapsed time between the start and the end of a sorting algorithm, excluding data generation and all other operations. Note that `clock()` returns the number of clock ticks rather than the number of seconds. Moreover, since `clock()` returns elapsed time rather than user time (*i.e.*, the CPU time used by a user program), this test is performed under MS-DOS rather than Windows and Unix to minimize the multitasking effect. The machine used for this test is an Intel 66mhz 486DX2 IBM PC compatible and the compiler is Watcom C/C++ Version 10.0 with compiler options set to `/oneatx/zp4/4/fp3` as suggested by Watcom for maximum efficiency.

Table 1: Running Time for n = 100 to 1000

	$O(n^2)$ *Group*			$O(n\log_2 n)$ *Group*		
n	*D-Bub*	*S-Bub*	*Select*	*Msort*	*Qsort*	*Tree*
100	0.22	0.12	0.10	0.08	0.07	0.05
200	0.98	0.54	0.44	0.15	0.13	0.10
300	2.20	1.22	0.76	0.23	0.22	0.19
400	4.18	2.42	1.44	0.32	0.30	0.21
500	6.38	3.74	2.18	0.42	0.37	0.29
600	10.22	6.48	4.06	0.53	0.51	0.40
700	15.38	10.10	6.46	0.69	0.57	0.43
800	21.20	14.82	9.68	0.76	0.69	0.51
900	28.34	20.20	13.62	0.88	0.79	0.61
1000	36.58	26.14	17.88	1.01	0.89	0.69

Since some algorithms perform better for small size input but poorly for large ones, timing will be divided into two groups. Table 1 and Table 2 contain the number of clock ticks used for all six algorithms. These two tables show that the fastest algorithm is the tree sort and the slowest is the sediment sort. Merge sort, quick sort and tree sort have very similar timing results. Sediment sort, which is a doubly linked list implementation of bubble sort, is about 1.5 times slower than the bubble sort. Its cause could be some extra time for maintaining two link fields. The function implementation of swapping might consume some processing time as well. Swapping is implemented with C macros in all other algorithms.

Table 2: Running Time for n = 2000 to 10000

	$O(n^2)$ *Group*			$O(n\log_2 n)$ *Group*		
n	*D-Bub*	*S-Bub*	*Select*	*Msort*	*Qsort*	*Tree*
2000	159	127	93	2.75	2.00	1.38
3000	379	302	220	3.38	3.38	2.88
4000	693	549	401	5.50	4.12	4.12
5000	1104	867	643	6.00	6.88	5.50
6000	1763	1395	1082	9.00	8.88	6.38
7000	3037	2604	2169	12.38	11.00	9.62
8000	4449	3850	3252	13.75	11.62	10.25
9000	5515	4630	3917	16.38	14.38	12.25
10000	6591	5509	4619	19.25	16.50	12.25

Note that comparing timing values of two algorithms for a particular n usually does not provide much insight. This is particularly true if two algorithms from different groups are compared. For example, the sediment sort is about four times slower than the tree sort when n = 100 and it becomes 538 times slower when n = 10000. By the definition of $O()$, the number of data items n and the required clock ticks t to sort them satisfy $t=\alpha(n^2)$ and $t=\alpha(n\log_2 n)$ for the $O(n^2)$ group and the $O(n\log_2 n)$ group, respectively. A least square (regression) fit, taking n and t as input, will deliver an estimation of the constant factor α.[3] Table 3 shows this result. The third column is the ratio of the second and the first columns. Note that n is divided by 100 to make the value α larger. So, the equations are $t=\alpha(k^2)$ and $t=\alpha(k\log_2 k)$, where $k=n/100$.

Now we can compare these constant factors to determine their relative efficiency. For the $O(n^2)$ group, if $n \le 1000$, the sediment sort is 1.43 ≈ 0.342743 / 0.239701 (*resp.*, 2.14 ≈ 0.342743 / 0.160253) times slower than the bubble (*resp.*, selection) sort, while the bubble sort is 1.50 ≈ 0.239701 / 0.160253 times slower than the selection sort. If $n > 1000$, the sediment sort is 1.19 ≈ 0.649589 / 0.546267 (*resp.*, 1.42 ≈ 0.649589 / 0.456876) times slower than the bubble (*resp.*, selection) sort, while the bubble sort is 1.19 ≈

[2] All test programs are available on request. Please send an e-mail to the author.

[3] We do not have to write a program to carry out the least square fitting since most commercial spreadsheet packages such as Lotus 1-2-3, Excel and Quatro Pro have this capability built-in.

0.546267 / 0.456876 times slower than the selection sort. Thus, for larger size input, the speed gap is narrower than smaller size input.

Table 3: The Constant Factors

Method	≤ 1000	> 1000	***Ratio***
D-Bub	0.342743	0.649589	1.895
S-Bub	0.239701	0.546267	2.279
Select	0.160253	0.456876	2.851
Msort	0.032090	0.027577	0.859
Qsort	0.028548	0.024358	0.853
Tree	0.021951	0.019862	0.905

For the $O(n\log_2 n)$ group, if $n \le 1000$, the merge sort is 1.12 ≈ 0.032090 / 0.028548 (*resp*., 1.46 ≈ 0.032090 / 0.021951) times slower than the quick sort (*resp*., tree sort), while the quick sort is 1.30 ≈ 0.028548 / 0.021951 times slower than the tree sort. If $n > 1000$, the merge sort is 1.13 ≈ 0.027577 / 0.024358 (*resp*., 1.39 ≈ 0.027577 / 0.019862) times slower than the quick sort (*resp*., tree sort), while the quick sort is 1.22 ≈ 0.024358 / 0.019862 times slower than the tree sort. The speed difference is very similar to that of the $O(n^2)$ group.

Consider the ratios. Since a larger constant means less efficient, a ratio that is larger than (*resp*., smaller than) one means the corresponding algorithm is more efficient (*resp*., less efficient) in handling small input size. Thus, the $O(n^2)$ group algorithms have better performance in handling small data set, and the $O(n\log_2 n)$ group algorithms are more efficient in handling larger data set, although the difference is not as significant as that of the $O(n^2)$ group. Whatever the input size, the $O(n\log_2 n)$ group performs much better than the $O(n^2)$ group. Note that this only shows the test results for $n \ge 100$, it could be different for $n < 100$.

6 Conclusion

The six algorithms included in this test are only a small sample of sorting algorithms. There are other interesting algorithms that are worth to be mentioned. For example, the shaker sort is an extension to the bubble sort in which two bounds are used to limit the range for next scan (Knuth [4] and Wirth [9]). Since the shaker sort scans the list in both directions, it would be very interesting to know the contribution of using two bounds rather than one in the bubble sort and the sediment sort.

Two factors are not addressed in this article. Since the input data for this test are random, some extreme characteristics cannot be tested. For example, the tree sort and the quick sort perform poorly if the input is sorted or reversely sorted, while bubble sort requires only n-1 comparisons and no swap if the input is sorted. Therefore, a comparison could be based on the sortedness of the input data. Second, in practice input data might not be distinct. Yet another comparison could be based on the level of data uniqueness. If there are duplicated items in the input, some algorithms could perform better than the others. For example, the quick sort presented in this paper has the capability of collecting equal items into a list so that they will not involve in subsequent sorting phases, while others (*i.e*., merge sort) are insensitive to the presence of duplicated data.

Please note that performing these comparison tests is not new and has been carried out many times based on different criteria by many researchers ever since people knew sorting is an important and useful technique (see Knuth [4] for historical notes). However, as an educator, I believe that making these theoretical results down to the earth and accessible for students would be an important job.

References

1. Jon Bentley, *Programming Pearls*, Addison-Wesley, 1986.
2. Jim Carraway, Doubly-Linked Opportunities, *ACM 3C ONLINE*, Vol. 3 (1996), No. 1 (January), pp. 9-12.
3. R. Hoare, Quicksort, *The Computer Journal*, Vol. 5 (1962), pp. 10-15.
4. Donald E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching*, second printing, Addison-Wesley, 1975.
5. Dalia Motzkin, A Stable Quicksort, *Software-Practice and Experience*, Vol. 11 (1981), No. 6, pp. 607-611.

6. Robert Sedgewick, *Algorithms in C++*, Addison-Wesley, 1992.
7. Lutz M. Wegner, Sorting a Linked List with Equal Keys, *Information Processing Letters*, Vol. 15 (1982), No. 5 (December), pp. 205-208.
8. Mark Allen Weiss, *Data Structures and Algorithm Analysis in C++*, Benjamin/Cummings, 1994.
9. Niklaus Wirth, *Algorithms & Data Structures*, Prentice-Hall, 1986.