Partition Algorithms for the Doubly Linked List

John M. Boyer
Department of Computer Science and Statistics
University of Southern Mississippi
Hattiesburg, MS 39406

Abstract

The purpose of this paper is to present partition sort and search algorithms developed by the author for the doubly linked list. First, the binary search concept will be modified to enable its use on a linked list, and the resulting algorithm, the *Blink Search*, will be shown to be more than twice as time efficient as the sequential search. The Blink Search will then be used to enhance a standard insertion sort. Upon finding that the improvement still leaves the insertion sort with the unacceptable $O(n^2)$ rating, several linked list partition sorts will be developed. We will see later that the best of these, the *Blink Sort*, shows very comparable execution time to the Quicksort on an equivalent array; in fact, the Blink Sort is rated O(nlogn) for random data and O(n) for reversed order and sorted order lists.

Introduction

Synthesis of doubly linked list partition sort and search algorithms necessarily began with an analysis of the array partition algorithms. A substantial theoretical discussion of these algorithms appears in Knuth [2], and a more intuitive discussion appears in Koffman [3]. Like many other authors, Knuth and Koffman expressed concern for Quicksort performance on pre-sorted data. A corollary concern questions the time efficiency of the Quicksort on reversed order data. Tenenbaum and Augenstein [4] discuss an ample solution to both the reversed and sorted order problems: modify the partition operation to use the median of the first, last and middle sublist elements as the pivot value. The only drawback to this *median-of-three* method is that it increases execution time on random data by as much as 50% in spite of the fact that it promotes more balanced partitions; this is due to the special Quicksort implementation required for it to work on reversed data and, of course, the cost of calculating the median. A more elegant solution to the reversed and sorted order problems is simply to swap the first and middle elements before performing the normal partition. Intuitively, the median-of-three method seems to do this, but in practice it does not work with all Quicksort implementations.

Neither the simple Mid-swap method nor the median-of-three method provide adequate solutions to the reversed and sorted order problems on a linked list because O(n) link assignments are required to access the middle node in a sublist. Nevertheless, the doubly linked list cannot be a competitive data handling construct without solutions to the reversed and sorted order problems. In addition, this difficulty in accessing the middle node casts a shadow of doubt on the efficacy of a linked list partition search, which requires access to the middle sublist element-- the concept seems preposterous.

Despite these concerns, development of linked list partition algorithms would yield the speed and simplicity offered by a list structure without the size bounding characteristic of the array data structure. This paper is devoted to presenting algorithms developed by the author to meet this need.

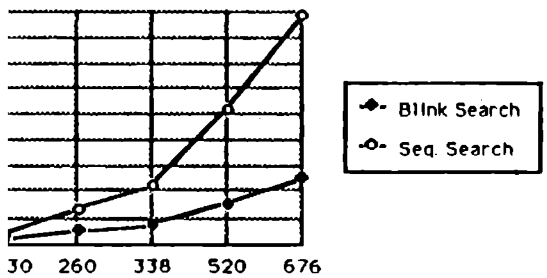
The Partition Search

The *Blink Search* (Appendix 1) and the binary search are very similar, taken to a significant degree of abstraction. The array binary search is a more time efficient algorithm than the array sequential search method because the binary search makes far less comparisons. The Blink Search also makes many less comparisons than a linked list sequential search, however the Blink Search must also access each node between a certain node and the next middle node. Thus, it will average $\log_2 n$ key comparisons but n link assignments; on the other hand, the sequential search averages only $n/2$ link assignments but also $n/2$ key comparisons. In deciding the more efficient search, we must note that the problem reduces to deciding the cost of a key comparison in terms of a certain number of link assignments.

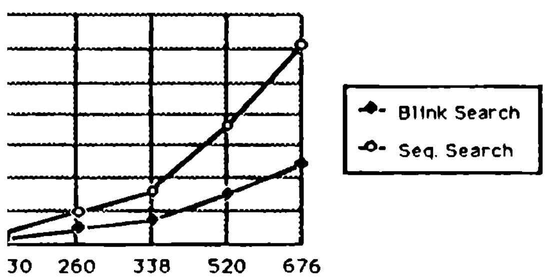
Given the mathematical relations above, inductive reasoning dictates that if the Blink Search can be shown to be more time efficient than a sequential search for some number of nodes n, then it will be more efficient for all lists with more than n nodes. The test below will provide the 'anchor' for this inductive hypothesis and sample test the hypothesis by examining average case performance of the Blink Search for various values of n.

To gain information regarding average case performance, an ordered linked list is created; the search being tested is then called n times to seek each element in the list. This test method incorporates every case, from worst to best, into one performance evaluation. Since we seem to be comparing link assignments to key risons, both the size of the list and the type of key aried. Graphs 1 below contain the results of this the Blink Search and the sequential search as run MHz, 1 wait state IBM PC clone.

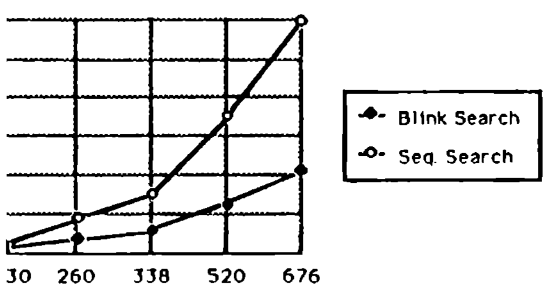
30BS



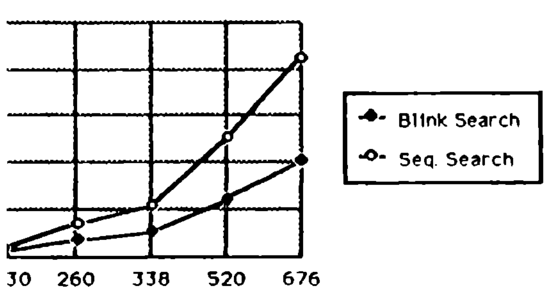
2BS



Real



Integer



Graphs 1: Searches

The types of keys were integer, real, 2 byte string (2BS) containing any two letters of the alphabet (such as 'AA', 'MN' and 'ZY'), and 30BS (consisting of 28 A's followed by the same 2BS). Thus, the data points in each graph indicate the amount of time in seconds required by the test algorithm to operate on an *n*-element list using the key type indicated above the graph. Since the Blink Search's superiority proved to be monotonic increasing with *n*, as predicted above, the restriction of list size to 676 (26x26) was not constraining.

The first fact observed in Graphs 1 is that the Blink Search is 2 to 3.5 times more time efficient than the sequential search in the average case, regardless of key type (including hardware integer comparisons) and linked list size. A corollary fact verified in Graphs 1 is that the Blink Search seems relatively unencumbered by the data type of the key because it does so little comparing. Note that both searches were also tested using $n < 130$, but the results were severely distorted by the IBM PC timer's inaccuracy. However, the average of several test iterations using very small *n* (down to n=13) yielded exactly the same trends as those in Graphs 1.

Graphs 1 illustrates one last important fact, though it is not quite as obvious. Since the test algorithm performs *n* search operations, its big-O rating will be *n* multiplied by the big-O rating of the search. In the test of the sequential search, doubling the list size quadrupled the execution time. This is the hallmark of an $O(n^2)$ algorithm, which implies that a single element sequential search is O(n), the expected result. In the case of the Blink Search test, $O(n^2)$ performance was only observed on numeric key types. This alone is enough to assert that the Blink Search is an O(n) algorithm. Even using the more complex types, the Blink Search is still not an O(logn) algorithm, though. Recall that for the search test to be rated O(nlogn) means that Time = knlogn for some constant *k*; this means that the ratio Time/(nlogn) should hold constant (k) for any list size *n*. Neither the 30BS nor the 2BS data displayed this characteristic for any reasonable level of error tolerance. In the end, Graphs 1 show that the Blink Search is simply a much faster O(n) search than the sequential search-- regardless of (non-trivial) list size and key data type.

The Insertion Sorts

One very important characteristic of the Blink Search in Appendix 1 is that it will always return a valid pointer value. If the search succeeds, then the desired node address is returned; if the search fails, then the Blink Search *always* returns the address of the node prior to where the search value would be if it existed in the list (a *nil* value is returned if the search value belongs before the first node). This is crucial to a linked list insert operation and, hence, to an insertion sort. With this in mind, it seems natural to begin the investigation of efficient linked list sorting by exploring the impact of the Blink Search on the standard insertion sort algorithm.

The insertion sort sequences a linked list by inserting each node into its proper position in a new list; it uses a search to find that proper position. The insertion sorts referred to here sort the linked list in memory, using the old nodes to create the new list. The S-insertion sort is implemented with a sequential search; the B-insertion sort is implemented with the Blink Search (Appendix 2). Chart 1 shows relative performance of the S-insertion sort and the B-insertion sort on random data as list size is varied up to 2000.

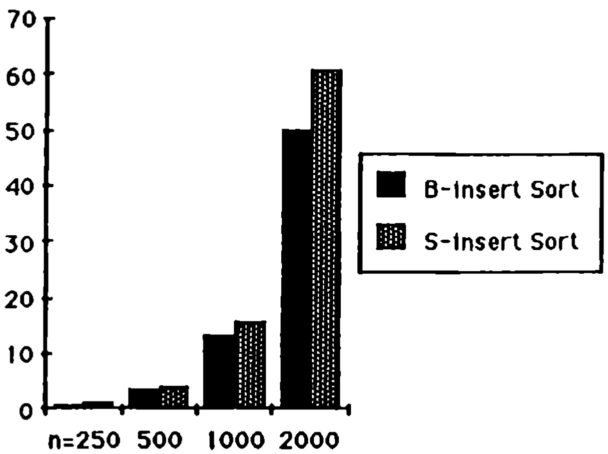


Chart 1: Insertion Sorts

Chart 1 shows some improvement in time efficiency of the insertion sort using the Blink Search. Despite this, execution time is still quite slow because it is not logarithmically related to list size. The linked list cannot be a truly competitive data handling construct without a much faster sort-- namely a partition algorithm with O(nlogn) time efficiency.

The Partition Sorts

The partition sorts below differ more noticeably from the array Quicksort paradigm than does the Blink Search from the array binary search. In fact, even the idea of two partitions is no longer sacrosanct. All of the sort algorithms discussed below are conceptually similar to the array sort in general operation. Each sort algorithm partitions the list then calls itself recursively to sort the resulting sublists. However, the sorts below differ considerably from the array Quicksort in how the partition operation is performed.

The *Separation Sort* (Appendix 3) is a partition-exchange sort without the exchange. The partition operation starts by choosing the first sublist node as the *pivot*. A single *down* pointer is then initialized to the address of the last sublist node. The *down* node's previous node is recorded for later use. The *down* node is moved behind the *pivot* node if its key value is less than that of the *pivot* (i.e. the *down* node becomes the *pivot* node's new previous element). The *down* pointer is then reassigned the value of the previous node saved above. This process is repeated until the *down* pointer has moved down the list from the last node to the *pivot* node, at which time the partition is complete.

The first noticeable change is the use of only one pointer to move 'down' the list toward the pivot element. The Quicksort requires finding two elements to swap because insertion is not a trivial operation on an array. This constraint is not shared by the linked list. Another interesting feature of this particular method is the fact that a reversed order list will be converted to a sorted order list in the first partition. Consider a list that is in descending order which we would like to put in ascending order. The first element has the greatest key, and it becomes the *pivot*. As the *down* pointer moves from the last element toward the *pivot*, it encounters a sequence of keys of increasing value, each of which it inserts just before the pivot element. Inserting successively greater nodes just before the pivot implies inserting just after all of the nodes containing keys which have a lesser value-- the first sublist will contain n-1 nodes in sorted order.

While this does account for the reversed order problem by reducing it to a sorted order problem, the sorted order problem remains. The solution is the reliability adjustment (Appendix 3) for the Separation Sort. The concept is simple: move the *pivot* pointer forward in the list for as long as the nodes contain a monotonic increasing sequence of key values (monotonic decreasing if sorting into descending order); if the *pivot* pointer reaches the last element, then the sublist is already sorted-- neither the remaining partition nor the two recursive sort calls need be performed. If the *pivot* pointer does not reach the end of the sublist, then the normal Separation Sort partition can occur by starting a *down* pointer at the last sublist node.

The *Reliable Separation Sort* is rated O(nlogn) for random data. For reversed order and pre-sorted data, the Reliable Separation Sort requires only O(n) operations, far surpassing the O(nlogn) rating of the reliable Quicksort for these cases. However, the Separation Sort's reliability adjustment causes the pivot pointer to gravitate to a local maximum in the sequence of elements, greatly increasing the likelihood of suboptimal partitioning-- the left partition will very often contain more elements than the right partition.

This problem can be alleviated by using the pivot from the Reliable Separation Sort and the pivot from the plain Separation Sort, causing the creation of three partitions. The resulting algorithm, the *Blink Sort* (appendix 4), maintains O(n) efficiency on reversed and sorted order data. It also averages $n\log_3 n$ operations on random data, but it is still rated O(nlogn) because irrelevant constants are ignored in O-notation. Nevertheless, that constant does provide 25% more time efficiency than the Reliable Separation Sort .

The Blink Sort's partition operation is somewhat more complex than the others. It starts by moving the second node behind the first if it contains a lesser key value. With this change, the present first node becomes the first pivot and the second node becomes the second pivot. The second pivot is then moved up to successive sublist elements until it reaches the sublist's end or until the next node has a lesser key value. If the second pivot reaches the last node, then neither the rest of the partition nor the recursive calls are performed (just as in the Reliable Separation Sort). The second half of the partition starts, as expected, by assigning the address of the last node to the *down* pointer. The *down* pointer is moved toward the second pivot by the same mechanism as in the Separation Sort; each successive down node is compared to both pivot keys, and moved before the appropriate one (or left alone if its key is greater than that of the second pivot).

Both the Blink Sort and the Reliable Separation Sort showed O(n) time efficiency on reversed and sorted data; both reversed the order of a 2000 element list in under a second and handled the sorted order problem in under an eighth of a second. The array Quicksort required over two seconds to handle the reversed and sorted order cases on a 2000 element array. Chart 2 compares the performance on equivalent lists of the Reliable Separation Sort, the Blink Sort and the array Quicksort using the Mid-swap method presented in the introduction. The test lists contained 250 to 2000 randomly generated key values.

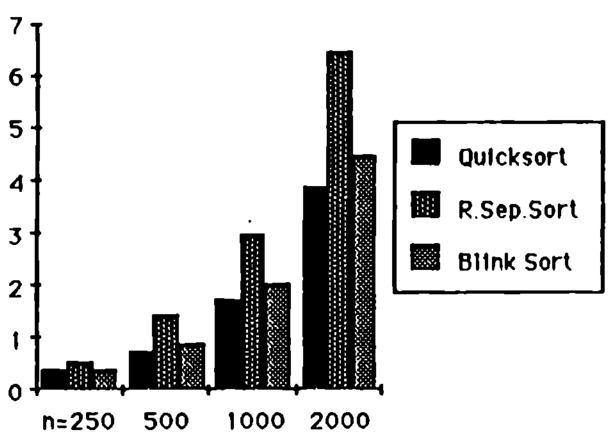


Chart 2: Partition Sorts

Every sort in Chart 2 clearly shows O(nlogn) efficiency. Perhaps the most dramatic illustration of just how much improvement in linked list performance has been made is the fact that time is measured in seconds in Chart 2 and in tens of seconds in Chart 1.

Conclusion

The field of software engineering is one of the fastest growing concerns in computer science because we are finally realizing that programs cannot be efficiently written by depending on software wizardry alone [1]. Many espouse the concept of information hiding as a method for forcing programmers to stay at the appropriate level of abstraction required to solve a computing problem [1]. We must be sure, however, that abstraction does not lead to lethargy. For example, one should *never* say, "A sort is a sort; I should not have to care how it works." The package author is responsible for insuring *that* the package algorithms work; the package user is still responsible for knowing *how* they work. The programmer cannot effectively decide whether to use a package without being cognizant of the exact strengths and weaknesses of the package algorithms. For example, if a linked list package author employs an insertion sort using a sequential search, then the prospective package user may well choose not to use the package, but instead, write another package using the Blink algorithms.

In conclusion, the Blink Search and Blink Sort developed by the author in this paper add to the efficiency of the doubly linked list, and improve its status as a competitive data handling construct.

[1] Booch, Grady. Software Engineering with Ada. 2nd ed. Menlo Park: Benjamin/Cummings Publishing, 1987.

[2] Knuth, Donald E. The Art of Computer Programming: Sorting and Searching. 3rd vol. Reading: Addison-Wesley, 1973.

[3] Koffman, Elliot B. Problem Solving and Structured Programming in Pascal. 2nd ed. Addison-Wesley, 1985.

[4] Tenenbaum, Aaron M., and Augenstein, Moshe J. Data Structures Using Pascal. Englewood Cliffs: Prentice Hall, 1986.

NOTE: The code in all appendices assumes SHORT CIRCUIT EVALUATION OF BOOLEAN EXPRESSIONS.

All appendices contain code for the implementation section of a linked list package or unit; FIRST and LAST will point to the beginning and the end of the list, respectively, and COUNTER will keep track of the number of nodes in the list, which is only needed for the Blink Search. In Appendices 1 through 4, the node data structure has the Pascal declaration:

```
type
  Key_Type = (* arbitrary *)
  Node_Ptr = ^Node_Rec;
  Node_Rec = Record
     Key : Key_Type;
     Prev, Next : Node_Ptr
  end;
```

```
function Search (Search_Key : Key_Type) : Node_Ptr;

  function Blink_Search (Search_Key : Key_Type) : Node_Ptr;
    var First_Pos, Mid_Pos, New_Mid, Last_Pos, I, Move : integer;
        Fnd_Ptr : Node_Ptr;
    begin
      First_Pos := 1;
      Last_Pos := COUNTER;
      Fnd_Ptr := FIRST;
      Mid_Pos := (First_Pos + Last_Pos) div 2;
      for I := 2 to Mid_Pos do Fnd_Ptr := Fnd_Ptr^.Next;
      while First_Pos <= Last_Pos do begin
        if Fnd_Ptr^.Key = Search_Key then First_Pos := Last_Pos + 1
        else begin
          if Fnd_Ptr^.Key < Search_Key then First_Pos := Mid_Pos + 1
          else Last_Pos := Mid_Pos - 1;
          New_Mid := (First_Pos + Last_Pos) div 2;
          Move := Abs (New_Mid - Mid_Pos);
          if New_Mid > Mid_Pos then
               for I := 1 to Move do Fnd_Ptr := Fnd_Ptr^.Next
          else for I := 1 to Move do Fnd_Ptr := Fnd_Ptr^.Prev
          Mid_Pos := New_Mid
        end
      end;
      Blink_Search := Fnd_Ptr
    end;

begin
  if COUNTER > 1 then Search := Blink_Search (Search_Key)
end;
```

```
procedure Insertion_Sort;
  var Templ, Temp2 : Node_Ptr;

  procedure Insert (Node : Node_Ptr);    (* see below *)

begin
  Templ := FIRST^.Next;
  FIRST^.Next := Nil;
  LAST := FIRST;
  COUNTER := 1;
  while Templ <> nil do begin
    Temp2 := Templ^.Next;
    Insert (Templ);
    Templ := Temp2
  end;
end;

  procedure Insert (Node : Node_Ptr);    (* Use with B-insertion sort *)
    var Prev_Ptr : Node_Ptr;
    begin
      Prev_Ptr := Blink_Search (Node^.Key);
      if Prev_Ptr = nil then begin
        Node^.Next := FIRST;
        FIRST^.Prev := Node;
        FIRST := Node
      end
      else begin
        Node^.Next := Prev_Ptr^.Next;
        Prev_Ptr^.Next := Node
      end;
      if Prev_Ptr = LAST then LAST := Node
      else Node^.Next^.Prev := Node;
      Node^.Prev := Prev_Ptr;
      COUNTER := COUNTER + 1
    end;

  procedure Insert (Node : Node_Ptr);    (* Use with S-insertion sort *)
    var Next_Ptr : Node_Ptr;
    begin
      Next_Ptr := Sequential_Search (Node^.Key);
      if Next_Ptr = nil then begin
        Node^.Prev := LAST;
        LAST^.Next := Node;
        LAST := Node
      end
      else begin
        Node^.Prev := Next_Ptr^.Prev;
        Next_Ptr^.Prev := Node
      end;
      if Next_Ptr = FIRST then FIRST := Node
      else Node^.Prev^.Next := Node;
      Node^.Next := Next_Ptr;
      COUNTER := COUNTER + 1
    end;
```

Appendix 3: The Separation Sort

```
procedure Sort;

  procedure Separation_Sort (SubF, SubL : Node_Ptr);
    var Down_Ptr, Pvt_Ptr, Temp : Node_Ptr;

    procedure Move_Behind;
      begin
        if Down_Ptr <> SubL then
          Down_Ptr^.Next^.Prev := Down_Ptr^.Prev
        else begin
          SubL := Down_Ptr^.Prev;
          if Down_Ptr = LAST then LAST := Down_Ptr^.Prev
          else Down_Ptr^.Next^.Prev := Down_Ptr^.Prev;
        end;
        Down_Ptr^.Prev^.Next := Down_Ptr^.Next;
        Down_Ptr^.Next := Pvt_Ptr;
        Down_Ptr^.Prev := Pvt_Ptr^.Prev;
        Pvt_Ptr^.Prev := Down_Ptr;
        if Pvt_Ptr <> SubF then Down_Ptr^.Prev^.Next := Down_Ptr
        else begin
          SubF := Down_Ptr;
          if Pvt_Ptr = FIRST then FIRST := Down_Ptr
          else Down_Ptr^.Prev^.Next := Down_Ptr;
        end;
      end;

    begin
{1}   Pvt_Ptr := SubF;
{2}   Down_Ptr := SubL;
      while Down_Ptr <> Pvt_Ptr do begin
        Temp := Down_Ptr^.Prev;
        if Down_Ptr^.Key < Pvt_Ptr^.Key then Move_Behind;
        Down_Ptr := Temp
      end;
      if (SubF <> Pvt_Ptr) and (SubF^.Next <> Pvt_Ptr) then
        Separation_Sort (SubF, Pvt_Ptr^.Prev);
      if (Pvt_Ptr <> SubL) and (Pvt_Ptr^.Next <> SubL) then
{9}     Separation_Sort (Pvt_Ptr^.Next, SubL)
    end;

begin
  if FIRST <> LAST then Separation_Sort (FIRST, LAST)
end;
```

The reliability adjustment can be made by adding another 'end' after line 9 and the following code between lines 1 and 2:

```
while (Pvt_Ptr <> SubL) and
      (Pvt_Ptr^.Next^.Key >= Pvt_Ptr^.Key) do
  Pvt_Ptr := Pvt_Ptr^.Next;
 if Pvt_Ptr <> SubL then begin
```

```
procedure Sort;

  procedure Blink_Sort (SubF, SubL : Node_Ptr);
    var Down, Pvt1, Pvt2, Temp : Node_Ptr;

    procedure Move_Behind (var Pvt, Down : Node_Ptr);
      begin
        if Down <> SubL then Down^.Next^.Prev := Down^.Prev
        else begin
          if Down = LAST then LAST := Down^.Prev
          else Down^.Next^.Prev := Down^.Prev;
          SubL := Down^.Prev;
        end;
        Down^.Prev^.Next := Down^.Next;
        Down^.Next := Pvt;
        Down^.Prev := Pvt^.Prev;
        Pvt^.Prev := Down;
        if Pvt <> SubF then Down^.Prev^.Next := Down
        else begin
          if Pvt = FIRST then FIRST := Down
          else Down^.Prev^.Next := Down;
          SubF := Down;
        end;
      end;

    begin
      Pvt2 := SubF^.Next;
      if SubF^.Key > Pvt2^.Key then Move_Behind (SubF, Pvt2);
      Pvt1 := SubF;
      while (Pvt2 <> SubL) and (Pvt2^.Next^.Key >= Pvt2^.Key) do
        Pvt2 := Pvt2^.Next;
      if Pvt2 <> SubL then begin
        Down := SubL;
        while Down <> Pvt2 do begin
          Temp := Down^.Prev;
          if Down^.Key < Pvt1^.Key then Move_Behind (Pvt1, Down)
          else if Down^.Key < Pvt2^.Key then Move_Behind (Pvt2, Down)
          Down := Temp;
        end;
        if (Pvt1 <> SubF) and (Pvt1^.Prev <> SubF) then
          Blink_Sort (SubF, Pvt1^.Prev);
        if (Pvt1^.Next <> Pvt2) and (Pvt1^.Next <> Pvt2^.Prev) then
          Blink_Sort (Pvt1^.Next, Pvt2^.Prev);
        if (Pvt2 <> SubL) and (Pvt2^.Next <> SubL) then
          Blink_Sort (Pvt2^.Next, SubL)
      end
    end;

begin
  if FIRST <> LAST then Blink_Sort (FIRST, LAST)
end;
```