Modularity - Processing on Linked List Ordered

Our purpose is to analyze the functions needed to maintain and use a linked list whose nodes are ordered by some key, and, which is held on a random access file (i.e., a relative file, in COBOL parlance).

We will, for the beginning, ignore the way the list is stored. In a way it does not matter. See Fig. 1 below.

![Diagram of linked list](image)

Figure I. A linked list on a random access file

Note that the list is to be ordered,

i.e. for all \( i, \ 1 \leq i < n \) where \( n \) is the number nodes in the list, it must always be true that

\[
K_i \rho K_{i+1} \quad (1)
\]

where \( \rho \) is any binary relation which is a total order.

We begin by examining the operations to be performed on the list. They are tabulated below: (Table 1)
Table 1 - Operations On A Linked List

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT</td>
<td>puts a record in its correct place</td>
</tr>
<tr>
<td>DELETE</td>
<td>removes a record</td>
</tr>
<tr>
<td>FIND</td>
<td>prints a record</td>
</tr>
<tr>
<td>CHANGE</td>
<td>alters either the key or data or both of a record</td>
</tr>
<tr>
<td>PRINTALL</td>
<td>prints the file in key order</td>
</tr>
</tbody>
</table>

Other functions might also be useful, e.g.:

CREATE     creates the file if it does not exist
DUMP       dumps the file in record order so that it can be examined visually.

However, we will look at those in Table I first.

TECHNIQUE
Examine each function "graphically" if appropriate, to see what it must do. **DO NOT** implement one function first.

YOU MAY NOT PICK THE RIGHT ONE!
(You should look at the list in Table 1 and see if you can specify the right order of implementation).

In any case, we will "build" our solution from the bottom up so even picking the "right" function may not help!

The figures below show what will be done to change the list for the INSERT, DELETE and CHANGE OF KEY commands.

We assume that the node is held in the node pointed to by NEW
3

START

record1

record2

k1

record3

k2

record3

k3

NEW

INSERT a node with key $k^*$ ($k_2, p, k^*, p, k_3$)

FIG 2. (Deleted links shown ---)

START

k1

k2

k3

k4

FIG 3. DELETE the node with $k^* = k_3$

FIG 4. CHANGE the key of the node with $k^* = k_3$ to $k^{**}$ such that $k_1, p, k^{**}, p, k_2$
We now have a clear picture of the way links will be changed, and can state them verbally.

However, let us refer to Table 1 and ask if we can extend it to include some simple statement of the error conditions:

In particular, we ought to note that:

Table 2

CONDITIONS FOR EACH FUNCTION

(a) We cannot insert \( K^* \) if there is no place \(^\S\) for it in list
   (i.e. there must exist some \( K_i \) in the set of keys such that
   \( K_i \preceq K \preceq K_{i+1} \))

(b) We cannot delete \( K^* \) if there is no key \( K_j \) in the list such that
    \( K_i = K^* \).

(c) We cannot find \( K^* \) if there is no key \( K_i \) in the list such that
    \( K_i = K^* \).

(d) We cannot change the key of a record from \( K^* \) to \( K^{**} \) unless:
    (i) there exists \( K_j = K^* \) in the list, and
    (ii) there exists \( K_i \) such that \( K_i \preceq K^{**} \preceq K_{i+1} \)
    (i.e. there is a place for \( K_i \)).

\(^\S\) Note that the concept of place (i.e.
\[ \exists K_i : K_i \preceq K \preceq K_{i+1} \]) is more general than that originally used, and allows for relations which are satisfied by equal keys.
2. "DISCOVERING" PRIMITIVE FUNCTIONS

2.1 LINKING

We can begin by examining the Figures 2 through 4 and note that:

(a) the insertion process involves:
   two "linking" operations

(b) the deletion process involves:
   one "linking" operation

(c) the change of key operation involves a delete followed by
   an insert; i.e. three linking operations!

SO, WE CONCLUDE THAT LINKING OPERATIONS ARE PRIMITIVES!

However, we do not know at this stage exactly what they look like.

2.2 FINDPLACE

Examining our list of CONDITIONS FOR EACH FUNCTION, we note that each
is basically interested in the same question, but with a different
answer, i.e.

either $K_i \neq K^*$ $\rho K_i \rho K_{i+1}$ is to be true

or $K_i = K^*$ is to be true.

BOTH OF THESE INVOLVE A SEARCH OF THE LIST! (OBVIOUSLY)

Hence we need to examine this search function (which is obviously
a primitive) and see what it looks like.

Before we do, let us state each operation verbally:

**INSERT** $K^*$

Search for $K^*$.

If place found then
   begin link new to $K_{i+1}$
   end link $K_i$ to new

**DELETE** $K^*$

Search for $K^*$ which will be at $K_i$ if it exists.
if found then link $K_{i-1}$ to $K_{i+1}$

**FIND** $K^*$

Search for $K^*$
if found then print $K^*$
CHANGE

K* key to K** (tricky)
Search for K*, search for K**
if K* found and place for K**
then begin

link K_{j-1} to K_{j+1} (remove K* from chain)
link K_j to K_{i+1}
link K_i to K_j
change K_j key to K**

end

(Note that CHANGE could be written:
DELETE K*; copy data in K* to a NEW
If successful then INSERT K** (from NEW)).

However, we have missed an opportunity for optimization, since we are
forced to begin our search for the keys from the beginning.

NOTE if K* pk** holds, then the record with key K* occurs before
the K**.

This suggests that :

THERE IS NO NEED TO RETURN TO THE START OF THE LIST,
WE CAN SEARCH FORWARD FROM THE POINT WHERE THE SEARCH
FOR K* TERMINATED!

We can only do this if the search function commences at a nominated
starting point, not the beginning!

THIS SUGGESTS THAT THE SEARCH FUNCTION MUST BE TOLD WHERE TO START,
i.e., that the starting point is a parameter.

We can now attempt to define the SEARCH function in more detail.

§ Unless p includes equality - in which case it does not matter.
SEARCH FUNCTION

SEARCH {start:in; search key:in; pointers to found node:out}

"pointers to found node" ...... we should consider exactly what we
mean by this, and how the search is
to be carried out ......

Let us re-examine the list, and the search procedure.

CURRENT+START; if $\text{RT}(K^* \rho \text{CURRENT}^{+}, \text{KEY})$ then finish.

\begin{center}
\begin{tikzpicture}
  \node (start) at (0,0) {START};
  \node (r1) at (1,0) {$K_1$};
  \node (r2) at (2,0) {$K_2$};
  \node (r3) at (3,0) {$K_3$};
  \node (r4) at (4,0) {$K_4$};
  \draw (start) -- (r1);
  \draw (r1) -- (r2);
  \draw (r2) -- (r3);
  \draw (r3) -- (r4);
  \draw (r4) -- (start);
\end{tikzpicture}
\end{center}

FIG. 5 - First step of Search

Notice only one step is considered!

Also the form of the relationship.

$\text{RT}(K^* \rho \text{CURRENT}^{+}, \text{KEY})$ the choice here depends upon the properties
of the relation $\rho$, and the "ordering" if $\rho$ includes equality.

(Consider the effect of CURRENT$^{+} \text{KEY} \rho K^*$. This must be revealed)

However, referring to Figs. 2 through 4, we note that we require two
pointers from SEARCH, formally, $\text{nodes yo-ya}$ nodes which satisfy
$K_i \rho K_{i+1}$ in all cases.

Hence, the search process should, on successive steps, look
like Figs. 6 and 7.
CONSIDER ASCENDING KEYS,

AND \( p \) IS <

i.e. \( k_1 \ k_2 \ k_3 \ k_4 \)

3 5 7 9

If NEW.KEY = 8, it belongs between \( k_3 \) \& \( k_4 \)

If we start with CURRENT \( \rightarrow \) START

IF \( (k*p \ CURRENT*KEY) \) then stop

else step on one.

Check this for \( p \) is \( > \) and

NEW.KEY = 10

\[ \begin{array}{cccc}
\text{START} & k_1 & k_2 & k_3 & k_4 \\
\text{9} & 7 & 5 & 3 
\end{array} \]
If \( p \) is \( \leq \) and \( k^* = 7 \)

i.e. \( k_1 \quad k_2 \quad k_3 \quad k_4 \)

\[
\begin{array}{cccc}
3 & 5 & 7 & 9 \\
\hline
4
\end{array}
\]

Place for 7

A formal proof could be developed!
while not (l and p current != key) begin
  if current.p. point != null then
    draw step
  end
end
NOTE that the action between steps was

**PRED+CURRENT**

**CURRENT+CURRENT+ PT**

AND BEWARNED one should be aware that the operation CURRENT+ may require a procedure call !! We will look at this last !

These two steps, then, combined with Fig. 5, read as :
if NOT(k^p CURRENT+KEY) then PLACE FOUND
else
begin
   PRED=CURRENT
   CURRENT=CURRENT.PT
end

The compound statement (between the begin and end) actually
could be described as a "primitive" function STEP, e.g.

STEP {PRED, CURRENT:IN; PRED, CURRENT:OUT}

Out temptation, at this point, for the INSERT, to just write

    while NOT k^p CURRENT+.KEY do
    begin
       PRED=CURRENT
       CURRENT=CURRENT+.PT
    end
    (* place found *)

Indeed, this would not be a bad choice.

We see that the search actually locates the place for an insertion.

We do not know the exact reason for the search termination

QUESTION
Do we have a useful primitive function? Consider the requirements
for an INSERT

ASSUME primitive SEARCH(k^STARTPT:IN; PRED; CURRENT:OUT)

(* obtain first element, if necessary *)
(* initialize PRED, CURRENT *)
(* but first, check that list is not empty *)

Note: SPECIFICATION FOR SEARCH
CURRENT POINTS TO THE KEY FOR WHICH k^p CURRENT+.KEY IS TRUE.
if START ≠ "null" then
begin
STARTPT+START
PRED+START
CURRENT+START
SEARCH(K*, STARTPT;PRED,CURRENT)
end

/* Assume that a place has been found - we have no warning
at this point */

/* Assume that the new node is pointed to by NEW */

/* We now have the situation in Fig. 2 */

/* Hence: */

NEW+.PT+PRED+.PT  (* or NEW+.PT+CURRENT*)
PRED+.PT+NEW

This is not really satisfactory. (Why?)

It would be possible to see from Table 2 that we might have chosen
a better primitive by examining the conditions which are involved.

Table 3 shows the "results" which are needed from the searches for
each function.

**TABLE 3q - SEARCH RESULTS**

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT</td>
<td>PLACE FOUND</td>
</tr>
<tr>
<td>DELETE</td>
<td>KEY FOUND</td>
</tr>
<tr>
<td>FIND</td>
<td>KEY FOUND</td>
</tr>
<tr>
<td></td>
<td>KEY NOT FOUND</td>
</tr>
<tr>
<td>CHANGE KEY</td>
<td>KEY FOUND</td>
</tr>
<tr>
<td></td>
<td>PLACE FOUND</td>
</tr>
</tbody>
</table>
It is clear then, that we must return a RESULT. However, we should ask ourselves how we handle the situation for insertions where key is to be inserted at the end of the list. This does of course, qualify as a "place found", but, how do we set the pointers and actually terminate the search?

![Diagram of START, K1, K2, K3, and NEW nodes with arrows indicating PRED and CURRENT]

**Fig. 8** K* to go after K3 - pointers at search termination

![Diagram of START, K1, K2, K3, and NEW nodes with arrows indicating PRED and CURRENT]

**Fig. 9** K* inserted after search

After the search, and the standard linking step

- \text{NEW$.PT$} \rightarrow \text{PRED$.PT$}
- \text{PRED$.PT$} \rightarrow \text{NEW}

SO, WE DO NOT NEED A SPECIAL RESULT FOR INSERT IF THERE IS NO "TRUE" PLACE IN THE LIST!
However, we note that we do need a result "key not found" for FIND (see Table 3a).

Before considering this, let us consider the "standard linking step", and see what happens if we need to INSERT before $k_1$.

![Diagram showing standard linking operation with $k^*$ to go before $k_1$](image)

Fig.10 $k^*$ to go before $k_1$ - pointers at search termination

The standard linking operation will not work in this case, since it assumes that PRED and CURRENT are distinct, which they are not.

Note that it is START which is to be altered, see Fig. 11.

![Diagram showing correct linking operation](image)

Fig.11 - After Correct Linking

The standard linking operation, as performed will not work because it assumes that PRED points to a node.

A possible solution

Let start point to start node. (See Fig. 12)
Fig. 12 Use of a Header Node - end of search

Inserting via the standard linking sequence puts NEW after the first node, and works well.

**YOU MAY OF COURSE DO THIS IF PRACTICAL.**

However, one may not be able to do this because

- (a) the first node may not be identical to other nodes.
- (b) the nodes may be physically large, and therefore it may be impractical to hold more than a few in memory.

(Objection a) can be overcome by the use of undiscriminated unions in PASCAL and REDEFINES in COBOL - but care is needed!)

so let us revert to Fig. 10, i.e. START points to the first real link.

**A Linguistic Interlude**

It is clear that a procedure

```pascal
LINKIN(NEWPT,PREDPT,NEW)
begin
  NEWPT=PREDPT
  PREDPT=NEW
end
```

will work if called by

```pascal
LINKIN(NEW+.PT,PRED+.PT,NEW)
```
CONSIDER the language statement

"PRED+.PT IS START"

Semantics

An assignment to PRED+.PT alters START unless PRED has been altered
since the execution of the IS statement.

We could then write

\[
\begin{align*}
\text{PRE} & \text{D+.PT IS START;}
\text{CUR} & \text{RRENT+.START}
\text{SE} & \text{ARCH(K*, START;PRED,CURRENT)}
\text{LIN} & \text{KIN(NEW .PT,PRED+.PT,NEW)}
\end{align*}
\]

end of interlude

However, we cannot.

Hence we must write, for our insert:

\[
\begin{align*}
\text{PRE} & \text{D+.START}
\text{CUR} & \text{RRENT+.START}
\text{SE} & \text{ARCH(K*,STARTPT;PRED,CURRENT)}
\text{IF} \; \text{PRED} = \text{START THEN}
& \text{LIN} \text{KIN(NEW+.PT;START;NEW)}
\text{ELSE}
& \text{LIN} \text{KIN(NEW+.PT,PRED+.PT,NEW)}
\end{align*}
\]

which is not as bad as all that!

Notice that we have not worried about the problem of equality
in the search.

LET US NOW EXAMINE THE OTHER FUNCTIONS
DELETE could be described as:
search for key
link it out.

This translates to:
PRED+START
CURRENT+START
SEARCH\{K^*,\text{STARTPT};\text{PRED},\text{CURRENT}\}

IF key is found THEN (* PRED points to KEY *)

BEGIN
IF PRED=START THEN
START=PRED+.PT, return(PRED)
ELSE

FIND
Find is basically a delete with a different action.
PRED+START
CURRENT+START
SEARCH\{K^*,\text{STARTPT};\text{PRED},\text{CURRENT}\}

IF key is found THEN (* PRED points to Key *)
DISPLAY (KEY)

CHANGE Key value $K^*$ to $K^{**}$
here we would code:
IF $K^*$ pK** in then
case 1,
else (* K^{**} pK*) case 2.

where case 1 is a "procedure" which does a change in the first case,
and $K^{**} pK^*$ does it in the second case.
SO suppose we invent a procedure "CHANGE-IN-ORDER" with two parameters, \( k_1 \) and \( k_2 \) but, this becomes a real mess! (try it and see).

(Back out a little!)

We need to ask ourselves -
"What do we need to perform this function?"

From Fig. 4, we see we need four pointers.

![Diagram of pointers for key change](image)

**Fig. 13 Pointers for Key Change**

These four pointers can be picked up by two calls to search.

The order of these searches depends upon the order of the two keys.
IF \( K^* \neq K^{**} \) THEN

BEGIN
\( K^* \text{PRED} \rightarrow \text{START}; \)
\( K^* \text{CURR} \rightarrow \text{START}; \)
\( \text{SEARCH}(K^* \text{STARTPT} ; K^* \text{PRED} , K^* \text{CURR}) \)
\( K^* \text{RES} = \text{RES} \)
(* NOW FIND THE OTHER PAIR *)
\( K^{**} \text{PRED} \rightarrow K^{**} \text{CURR} \)
\( K^{**} \text{CURR} \rightarrow K^{**} \text{CURR} \)
(* start search from the point just reached *)
\( \text{SEARCH}(K^{**} \text{STARTPT} ; K^{**} \text{PRED} , K^{**} \text{CURR}) \)
\( \text{END} \)
\( \text{RES} = \text{RES} \)
ELSE (* \( K^{**} \neq K^* \) *)
BEGIN
(* repeat above using \( K^{**} \) in place of \( K^* \), and vice versa *)
\( \text{END} \)
IF \( K^* \text{RES} = \text{FOUND} \) and \( K^{**} \text{RES} = \text{PLACE FOUND} \) THEN
BEGIN
\( \text{OKDELETE}(K^* \text{PRED} ; K^* \text{CURR} , \text{START}) \)
alter Key of \( K^* \text{CURR} \) to \( K^{**} \)
\( \text{OKLINKIN}(K^{**} \text{CURR} + . \text{PT} , K^{**} \text{PRED} + . \text{PT} , K^* \text{CURR} , \text{START}) \)
\( \text{END} \)
(* process errors *)

Note that we have used the section of code from Fragment 8 of beginning "IF PRED = " as the procedure OKLINKIN.

THERE ARE A NUMBER OF THINGS WHICH STILL NEED TO BE CLEANED UP.

THESE INCLUDE:

(A) clumsy use of procedure SEARCH and its parameters

SEARCH ought to begin from STARTPT - this would save some initialization.
(B) The result of SEARCH is not a parameter, nor has it been defined.
That is OK, at least we know what results we require - or do we?

(C) There is not test for end of list.
otherwise we are in good shape.

LET US EXAMINE THE RESULTS
Consider $p$ is $\leq$.
Then
SEARCH(KEYSOUGHT,IN;STARTPT,IN:PRED,CURR:OUT*)
CURR=STARTPT

WHILE NOT CURR=null DO
IF KEYSOUGHT $\leq$ CURR .KEY THEN
GO TO FOUNDPLACE
ELSE BEGIN
PRED=CURR
CURR=CURR+.PT
END
FOUNDRES=NOTFOUND;
RETURN (* exits procedure *)

FOUNDPLACE: IF KEYSOUGHT=CURR+.KEY THEN
FOUNDRES=FOUND
ELSE FOUNDRES=NOTFOUND

we note that there is always a place for the key, in this case found or not.

Exactly what we need depends upon the relationship $p$.

$p$ may or may not include equality (e.g. $p = \leq$) or it may not,
(e.g. $p = >$). Clearly, if we stop our search when we have found
the first item for which $p$ is true, then the key of the "sought" item
may or not be equal to that of the stopping point.
It is interesting to note that,

while our external action does not depend on the relation (we are interested in three results, key found, key not found, place found). The action inside SEARCH does indeed.
Consider $\rho$ is $\leq$

then

```
SEARCH{KEYSOUGHT, IN: STARTPT, IN: PRED, CURR: OUT;}
CURR=STARTPT
WHILE NOT CURR=null DO
IF KEYSOUGHT $\leq$ CURR+.KEY THEN
  GO TO FOUNDPLACE
ELSE BEGIN
  PRED=Curr
  CURR=Curr+.PT
END
FOUNDRES=NOTFOUND;
RETURN (* exits procedure *)
FOUNDPLACE: IF KEYSOUGHT=CURR+.KEY THEN
  FOUNDRES=FOUND
ELSE
  FOUNDRES=NOTFOUND
```

We note that there is always a place for the key, in this case - found or not.

Consider $\rho$ is $<$

then

```
SEARCH{KEYSOUGHT, STARTPT: IN; PRED, CURR: OUT}
CURR=STARTPT
WHILE NOT CURR=null DO
IF KEYSOUGHT$<$CURR+.KEY THEN GO TO FOUNDPLACE
IF KEYSOUGHT=CURR+.KEY THEN GO TO FOUND
ELSE BEGIN
  PRED=Curr
  CURR=Curr+.PT
END
FOUNDPLACE: FOUNDRES=FOUNDPLACE RETURN;
FOUND: FOUNDRES=FOUND RETURN;
```
Comparing FRAGMENTS 12 and 13 we see that they are equivalent.
(Why? make sure you see why!)

Except that we are calling the result of FRAGMENT 12 "NOTFOUND"
instead of "PLACE FOUND"

THIS WILL NOT ALWAYS BE TRUE, SO, THE DETAIL OF SEARCH WILL
NEED TO BE RE-WRITTEN FOR EACH CASE.
NOTE this sort of problem can be easily handled when a procedure
can be passed as a parameter (HOW?)

What is important, however, is that we conclude that if the
list is ordered by "p" then, when SEARCH STOPS

(a) the target may be found
(b) if it is not found we have the place for an insertion.

SO, WE ARE ONLY INTERESTED IN TWO RESULTS, NOT THREE.
EXCEPT THAT FOUND MAY OR MAY NOT MEAN PLACEFOUND!

NOTE ALSO FROM FRAGMENT 12, we have cleared up the problem of the
start and initialization of PRED,CURR.

We assume the SEARCH commences from STARTPT ....... lets clean
it up finally!

SEARCH
Definition
Searches for the list item with key KEYSOUGHT commencing from
the node pointed to by STARTPT.

It stops when either:
(a) the KEYSOUGHT is found
or (b) its place is found

and returns separate indications for these two.

Note that when SEARCH stops with CURR = STARTPT, PRED is
meaningless,
otherwise PRED points to the successor to CURR+, and
CURR points to the first item for which p is true.

CODE FOR SEARCH
procedure SEARCH(KEYSOUGHT,STARTPT, IN; PRED,CURR,RESULT:OUT)
CU educación STARTPT

WHILE NOT CURR=NULL DO {search while}

IF KEYSOUGHT=p CURR+.KEY THEN {have we a termination}

BEGIN {check for equality}

IF KEYSOUGHT=CURR+.KEY

THEN RESULT+FOUND

RETURN

ELSE GO TO FOUNDPLACE

END {check for equality}

ELSE {have we a termination? no, not here}

BEGIN {step forwards one link}

PRED+CURR

CURR+CURR+.PT

END {step forwards one link}

{ENDIF have we a termination, no, we will go on}

{ENDWHILE: search while}

FOUNDPLACE: {we have found a place, either by termination or by finding a place}

RESULT+PLACEFOUND

RETURN

END {SEARCH}.

NOW WE CAN CODE OUR PROCEDURES.

START WITH FIND

PROCEDURE FIND(START, KEYSOUGHT:IN)

SEARCH(KEYSOUGHT, START, PRED, CURR, RESULT)

IF RESULT=FOUND THEN PRINT(CURR)

ELSE LOGERR("KEYSOUGHT")

END
NEXT DELETE

PROCEDURE DELETE(START, KEYSOUGHT; IN)

SEARCH(START, KEYSOUGHT, PRED, CURR, RESULT)
IF RESULT=FOUND THEN
BEGIN {PROCESS THE found record}
IF CURR=START {Bypass first item}
THEN LINK(START, CURR+.PT)
ELSE {all other cases}
    LINK(PRED+.PT, CURR+.PT)
{end of nested if}
RECLAIM(CURR) {put object pointed to by
    CURR on delete chain}
END
ELSE LOGERR("RECORD NOT FOUND")

PROCEDURE INSERT(START, KEYSOUGHT, BEGIN, RESULTTAB)
{First, find a place for insertion}
SEARCH(START, KEYSOUGHT, PRED, CURR, RESULT)
IF RESULTTAB["SEARCH", RESULT] = FOUNDPLACE:
THEN
BEGIN {perform insertion} GET_FREE_REC(NEW)
IF CURR=START {Bypass first item}
THEN LINKIN(NEW+.PT, START, CURR+.PT)
ELSE LINKIN(NEW+.PT, PRED+.PT, CURR+.PT)
END
ELSE LOGERR("NO PLACE FOR KEY")
END {procedure complete}
Finally, the most complicated of all, we re-write Fragment 11.

**PROCEDURE CHANGE_IN_ORDER**(START, OLDKEY, NEWKEY, RESTAB:IN; NEREL; INOUT);

**BEGIN**

IF OLDKEY ≠ NEWKEY THEN

**BEGIN**

SEARCH(START, OLDKEY, OLDPRED, OLDCURR, OLDRS)

IF RESTAB[SEARCH, OLDRES] ≠ FOUND THEN LOGERR("OLD KEY NOT FOUND");

SEARCH(OLDCURR, NEWKEY, NEWPRED, NEWPRED, NEWRES)

BEGIN {Note we continue from the original found point}

IF RESTAB[SEARCH, NEWRES] ≠ PLACEFOUND THEN LOGERR("NEWKEY HAS NO PLACE")

**END (reverse case)**

SEARCH(START, NEWKEY, NEWPRED, NEWCURR, NEWRES)

IF RESTAB[SEARCH, NEWRES] ≠ FOUNDPPLACE THEN LOGERR("NO PLACE FOR NEWKEY")

SEARCH(NEWPRED, OLDKEY, OLDPRED, OLDCURR, OLDRS)

IF RESTAB[SEARCH, OLDRES] ≠ FOUND THEN

**END (reverse case)**

**FRAGMENT 14**

I must then return to the previous definitions and simplify them (DO this, re-write LINKIN as well).

**SECONDLY** I ask myself a question -

What am I trying to do ?

**ACTUALLY** I want to search for the "least" key, then the other one. THEN I want to make the necessary changes !

**SO** If I can somehow "tag" the keys so that I

(a) search for the least key first, the other key next and

(b) remember which key was which, I will succeed.

**HOWEVER.** I do have a technique for doing the reverse.

I can set up a key to be the lowest key, and remember whether it is the NEW key or the OLD key, and vice versa.

The routine "CHANGE_IN_ORDER" follows :
PROCEDURE CHANGE_IN_ORDER(START, OLDDKEY, NEWKEY, RESTAB: IN; NEWREL: INOUT)

BEGIN  {set up keys for correct order of search}
   {note we simulate associative memory}

IF OLDDKEY \neq NEWKEY
   THEN OLDP+FIRST\neq NEWPT+SECOND
   ELSE OLDP+SECOND\neq NEWPT+FIRST

   {"remember" keys}

   KEY[OLDP] = OLDDKEY;
   KEY[NEWPT] = NEWKEY;

   SEARCH(START, KEY[FIRST], PRED[FIRST], CURR[FIRST], RES[FIRST])
   SEARCH(PRED[FIRST], KEY[SECOND], PRED[SECOND], CURR[SECOND],
          RES[SECOND])

NOTICE THIS - WHAT HAVE I DONE

LOGCHANGE I R(RESTAB, RES, OLDP, ERRME'S, ERRFLAC)
LOGCHANGE I R(RESTAB, RES, OLDP, ERRME'S, ERRFLAC)

IF NOT ERRFLAC THEN BEGIN
   {FIRST, delete the OLD KEY's record}
   LINK(START, PRED[OLD], CURR[OLD])
   {alter the keys}
   CURR[OLD].KEY = KEY[NEWPT]
   LINKIN(START, CURR[OLD], PRED[NEW], CURR[NEW])
END
END {of change of key}.

Note 14 lines of executable code.

KR/HG