<u>Analysis-Synthesis Approach to Design</u>

Project for CSMSE, Software Engineering

by Michael Cooper, 82505793

V03. 26/5/2013

Introduction

In Software Engineering, the benefits of modularity and re-use have been well documented in the past. Advantages of these techniques include shortened development time, additional flexibility, and the ability to build and enhance systems from libraries of conceptually simple, re-useable modules. Building and enhancing systems becomes faster and more cost effective.

Techniques for effective module decomposition have been practiced since the late 1950's. Parnas [3] outlined methods for the decomposition of systems into modular form, contrasting both the stepwise refinement and information hiding techniques. The aim of this paper is to present and analyse another technique for the decomposition of a system into modular form. This technique is the *analysis-synthesis* approach to system or program design. We will also attempt to show that a good modular design will allow for more flexible software construction.

The basic principal of the analysis-synthesis approach as stated by Reed [1] in his software engineering lecture series is quoted below.

"Given a collection of (related) functions on a collection of (related) data, there should exist a set of primitive functions from which complete systems may be built."

Using the strategy we will outline in the following paragraphs, we will develop parts of a small test system to demonstrate the analysis-synthesis principle, and contrast it some of the other techniques of system decomposition. What follows is in fact a re-write of the case-study by Reed in 1980 [5], with additional discussion.

The Analysis-Synthesis Approach

Our first task is to perform the initial system design. This involves identifying the functions and subsystems that need to be developed to achieve the specified system functionality. Once this is complete, we should then attempt to find what appear to be *sensible* components in each identified module. By *sensible* components we mean look for a plausible breakdown of tasks that each module is likely to perform. This only needs to be done at a general level initially, since we will examine each component in more detail at a later stage.

The second part of our strategy is to utilise our initial breakdown, our sensible components, to identify any components which are common across each module. These components could possibly be routines that may be used throughout the system in some other modules, but not necessarily all. It is these routines that will form the basis from which we will build the system, and minimise the number of components required for construction.

Thirdly we analyse each component design, and make the identified common useable components identical. This means that we need only support a single version of the identical code across the system. Each identical component identified should then encompass the functionality required by all programs or modules, and ensure their correct operation.

Example of Analysis-Synthesis

In the following example we are going to analyse the functions necessary to use and maintain a linked list of nodes. All nodes in the list are ordered by some key, and also contain some form of other data. The list is to be stored in some form of random access file whose physical structure may be quite different to that of it's logical organisation. The list is to be ordered such that for each node *i*, *1 <= i <= n* where *n* represents the number of nodes currently maintained in the list. It must also be true that k_i *relop ki+1* where *"relop"* represents any binary relation which is in a total order.

We begin the design process by examining the operations which may be performed on the list and giving a brief description of their actions. This list of operations has been tabulated below.

Operation	Description
INSERT	Will place a record in it's correct place
DELETE	Will remove a record from the list
FIND	Will locate and print a records details
CHANGE	Will alter a records data, key or both
PRINTALL	Prints the file in key order

Other useful operations may be defined such as functions to create the file if it does not exist, or a function to dump the file in record order.

The first step toward our design should begin with a graphical examination of each function if possible. This will give us more of an insight into what we are trying to achieve, and help us better understand the manipulations which will be applied to the list. It is important not begin by implementing a single function as we choose incorrectly, and as a consequence could incorrectly identify or specify some of the systems *primitive* components. We may also miss our chance to minimise the number of functions or modules which need to be developed. The graphical representations on the following page give us a clearer picture of what is required to change the list for the *insert, delete,* and *change of key* operations.

Following our graphical analysis, we should now re-examine each of the listed functions and include some statement eluding to error conditions which may arise during processing. This just needs to be some form of written statement expressed in english, and establishing the criterion under which our functions will operate. We will scrutinise our operations in greater detail at a later stage. A list of function conditions follows.

Function Conditions

INSERT :-

We cannot perform an insertion if there is not a place to insert into the list. That is, there must be place which satisfies ki relop "new" relop ki+1.

DELETE :-

We cannot delete a key from the list if the key we wish to delete does not exist.

FIND :-

We cannot find a key and print the node details if that key does not exist in the list.

CHANGE :-

We cannot change a key if the key does not exist in the list, or if there is no place for the new key in the list.

Function Descriptions

The function conditions give us an idea about some of the restrictions which must be considered during the design phase. Now we are in a position to make some form of verbal description of each functions operation.

INSERT :- to insert a node into the list.

```
search for node with the corresponding key
if a place is found then
  link new node to key+1 node
  link key node to new node
```

DELETE :- to delete a node from the list.

```
search for node with the corresponding key
if node with required key is found then
  link key-1 node to key+1 node
```

FIND   :- to find a node with a key value.

```
search for node with the corresponding key
if node with required key is found then
  print key and contents of the node
```

CHANGE :- to change a nodes key value.

```
search for nodes with key* value and key** value
if found node with key* value and
   place for node with key** value then
  link key* -1 node to key*  +1 node
  link key*    node to key** +1 node
  link key**   node to key*     node
  change key* value to key** value
```

Discovering Primitive Functions

By this stage we should have gathered enough information to begin recognising some of the primitive functions used in our list operations. The first observation about the performed actions notes the necessity for a linking operation. Both the insert and delete list operators perform one or more linking operations. The change key action consists of a delete operation followed by an insert operation. From this we conclude that *linking* must be a primitive operation.

Notice that above we had an option to specify the change operation in terms of insert and delete operations. This may have presented problems if a place for the changed key cannot be found. We would then need the ability to be able to recover from the previously preformed deletion.

If we examine the conditions applied to each function we find that our chief concerns are the existence of a *place* in the list, or whether or not a node with a matching key value to that sought can be found. That is, either there is a position in the list such that $k_i \ relop \ k^* \ relop \ k_{i+1}$ can be satisfied, or $k^* = ki$ can be satisfied. In each case we are performing basically the same test, but with the required result being the only difference. Determination of these conditions obviously involves a *search* of the list.

The Search Primitive

If we examine the search primitive in more detail, we find that if the list follows some kind of order, there is no need to return to the beginning of the list to resume searching. In our first optimisation step we can pass the starting point as a parameter to the search function. The format of the call to the search function will then resemble that given below.

```
procedure search {keysought, startpt : in, node pointer : out}
if (search key relop current^.key) then
  finish searching
```

If we consider a single step search process as written above, we find are comparing the current key to the search key passed as a parameter. The result of the comparison, and hence our choice, depend upon the relation *relop* in use. The ordering of the keys is also important if equality is to be included. The graphical representations of our operations listed previously reveals that to satisfy the above condition such that the association will hold in all cases, we require a second pointer indicating the predecessor to the current node. The current node will be the node where the key establishing the relationship as true has been found. This also implies that we need another parameter to be returned representing this predecessor to the current node.

As a side note, the statements which advance the pointers during the search operation could also be described as a primitive function. Writing the code fragment for the *search* routine, including pointer advancement, reveals the following.

```
procedure search {k*, startpt : in, pred, current : out}
while not k* relop current^.key do begin
  pred = current
  current = current^.next
end
```

To investigate the usefulness of our newly defined primitive, we should consider the requirements of the originally specified list operations. If we consider the *insert* operation, and what is required from the search function during processing, our list of actions to be performed would be as follows.

1) Check to see if the list is empty,
2) Obtain the first element if necessary,
3) Initialise both *pred* and *current*,
4) Find the key which satisfies *k* relop current^.key*,
5) Assume a place is found, and *current* points to this node,
6) Assume *new* represents the node to be inserted,
6) Link *pred* to *new* and *new* to *current*.

The assumption we are making with the above list of actions is that a place was successfully found. This is an unsatisfactory supposition caused by the fact that no values indicating the success of the operation are returned by the search function. The table below examines the results required from the search function by each of the specified list operations.

Operation	Search Result
INSERT	Place Found
DELETE	Key Found
FIND	Key Found, Key not Found
CHANGE KEY	Key Found, Place Found

The Linking Operation

Since it is now necessary for the search function to return a result, we may become aware of situations where our search primitive may produce some erroneous results. This necessitates the testing of some boundary conditions to verify that our search function behaves in the correct manner. Analysing the insert operation once again, we need to consider cases such as insertion at both the head, and tail of the list.

Examining the case of insertion at the tail of the list we find that the list end does actually qualify as a "place found". We must however still analyse the impact on our pointer system and the method of terminating the search. On reaching the end of the list, *current* points to *null*, and *pred* points to the *last node* in the list. So no special case is required here since our linking operation will append the new node to the end of the list regardless of whether a true place was found. Having established that a special result is not necessary for the insert operation, analysis of a similar type performed for the *find* operation reveals that a returned result of "key not found" is necessary (the above table).

If we examine our linking operation and it's behaviour when inserting nodes at the head of the list we find it will fail. This is due to the fact that *current* and *pred* are always assumed to be distinct nodes which they are not. In this case it is really *start* which needs to be modified. For the linking operation to succeed *pred* must point to a node. To overcome this we could either introduce a header node, or initialise *current* and *pred* to be identical in the initial instance. Header nodes may not always be practical due to both physical or relational constraints, so we will use the latter, more elegant method. For the case of our insert operation, the code will start to take the form of that given below.

```
search {k*, startpt; pred, current}
if pred <> start then
  linkin {new^.next, pred^.next, new}
```

```
else linkin {new^.next, start, new}
```

The *linkin* primitive will simply perform a pointer reshuffle to include the new node between the current and previous nodes.

```
function linkin {newpt, predpt, new}
  newpt = predpt
  predpt = new
```

Here we do not need to worry about the case of equality in the search. If we now focus our attention on the other functions to be implemented we find that *delete* can be described as searching for a key, then linking it out of the current list of nodes.

```
search {k*, startpt; pred, current}
if key is found then
  if pred <> start then
    pred^.next = current^.next
  else start = pred^.next
return pred
```

The *find* operation is identical to delete but performs a different action.

```
search {k*, startpt; pred, current}
if key is found then
  display (key)
```

For the change operation we have two cases to consider. The case where the first key precedes the second in the current relationship, and vice versa. Analysing what is needed to perform this operation, we find that we require four pointers. These four pointers can be picked up from two calls to the search routine, one for each key. The order in which the searches are performed depends upon the order of the two keys with respect to the relationship *k* relop k*** or *k** relop K**.

The Search Primitive Refinement

Having defined our search routine to this level, we are still to make the final refinements necessary to render it fully functional across the range of our specified list operations. These refinements involve things like the actual use of the search procedure, and the search parameters. The searching operation would be more efficient if it began from *startpt*, thus saving some initialisation. The result parameter is currently undefined, and there is no test for the end of the list. In the following code fragments we are going to look at the cases where the relationships vary, and the differences for each case. The purpose of this is to attempt to make them identical so they may be applied across the whole system or module. In the first instance we will examine the case where *relop* involves equality.

```
procedure search {keysought, startpt :in, pred, curr : out}
foundres = notfound, curr = startpt
while not curr = null and not foundres = found do
  if keysought <= curr^.key then
```

```
    if keysought = curr^.key then
      foundres = found
  else begin
    pred = curr
    curr = curr^.next
  end
return
```

Note that in the above fragment there is always a place for the key regardless of whether it was found or not. As previously stated, the exact requirement of the search function depends on the relationship *relop* being used in the comparison. When the search terminates, we have located the first instance where the relationship *relop* is satisfied. In this case the sought key may or may not be equal to the required stopping point. It is interesting to note that outside of the search routine we are not concerned with the nature of the relation. Instead we are interested in three results, key found, key not found, and place found. The actions inside the search however are dependant on the nature of the relation. We now analyse the code fragment where the relationship *relop* does not involve equality.

```
procedure search {keysought, startpt : in, pred, curr : out}
curr = startpt
while not (curr = null or foundres = place, found) do begin
  if keysought < curr^.key then
    foundres = foundplace
  if keysought = curr^.key then
    foundres = found
  else begin
    pred = curr
    curr = curr^.next
  end
end
return
```

Comparing the two code fragments above we find they are equivalent. The only exception is that we have called the results in one of these fragments *not found* instead of *place found.* This may not always be true so we will need to revise the detail of the search for each relationship *relop* with the aim of making them equivalent. What we have been able to conclude is that if the list is ordered by some relation, then when the search halts, a target may be found. If a target was not found then we still have a valid place for insertion. This indicates we are only interested in two of three results. So *found* may or may not mean place found. Now we are in a position to fully specify the search primitive.

<u>**Search Primitive Final Definition**</u>

The *search* function will scan a list of nodes looking for the node with the key value matching that passed. The search will commence from the node pointed to by *start*. Termination occurs when either the sought key is found, or if not found, a place in the list is found for it. Separate indications of these two events should be returned. If the search terminates with *curr = startpt, pred* is meaningless, otherwise *curr* points to the first item for which the relation *relop* is true, and *pred* points to the predecessor of *curr*. The code fragment for the search will then have the below form with *relop* being substituted for the relation of the designers choice. The result parameter has been included in the following fragment.

SEARCH :-

```
procedure search {key, start :in; pred, curr, result : out}
result = place found, curr = start
while not (curr = null or result = found) do begin
  if key relop curr^.key then
    result = place
  if key = curr^.key then
    result = found
  if result <> found, place begin
    pred = curr
    curr = curr^.next
  end
end
return
```

Review of the Process

A review of the process to this point reveals the following. We began with the specification of a linked list of nodes, and a set of operations which may be applied to the list. We examined these operations graphically to obtain a better understanding of their actions in terms of the list. This examination then yielded the set of conditions which must be satisfied for the successful execution of each operation. From this we were able to form a verbal definition of the functionality we required from each operation. These function descriptions allowed us to begin discovering our primitive components which will constitute the list operations.

The search primitive was chosen for further investigation. We proceeded to define it's operation by specifying code fragments and analysing functionality with respect to the specified list processing operations. We were able to establish requirements of the search in relation to the nodes tracked, the parameters it must be passed, and the results that need to be returned. Once this was completed we would have a useful common useable component. The linking operation was also examined along with it's behaviour under boundary conditions such as insertions at the head or tail of the list.

We were then able to compare the search code fragments, analyse their behaviour under different relational conditions, and highlight the differences. Once these differences were identified we could then attempt to make the code fragments identical for each of the relations. Then we could finally specify the search primitive.

To follow on from this point we must perform the above actions in detail for each primitive component of the system that is common useable. Having completed this, we may then specify or implement our list operations in terms of these primitive components. After initial implementation we may then return and simplify these routines if this is necessary. The code for some of our list operations will resemble that given below.

INSERT :-

```
procedure insert(keysought, start, begin, resultab)
begin { Find a place for the insertion }
  search(start, keysought, pred, curr, result)
  if resultab["search", result] = foundplace then begin
    get_free_rec(new)
    if curr <> start then
      linkin(new^.next,pred^.next, curr^.next)
    else linkin(new^.next, start, curr^.next)
  end else logerr("No place for key")
end {procedure}
```

DELETE :-

```
procedure delete(start, keysought)
begin
  search(start, keysought, pred, curr, result)
  if result = found then begin
    if curr <> start then
      link(pred^.next, curr^.next)
    else link(start, curr^.next)
    reclaim(curr)
  end else logerr("Record not found")
end
```

Note that in the previous examples *resultab* is a table lookup for the type of search we are performing. That is, whether equality is included or not.

Analysis of the Method

In his paper of 1990, Hoffman [4] specified some criteria which could be used for the formalisation of specification of module interfaces. While these criterion were specified for module interfaces using an information hiding decomposition strategy, I believe many are still relevant for the analysis-synthesis approach. Indeed many of these practices should be employed for system design in general. Hoffman stated the following characteristics for module interfaces to which their applicability to analysis-synthesis is contrasted.

Consistency: This is essential in almost all aspects of design regardless of the method employed. eg. naming conventions etc.

Essential: States that interfaces should not have needless features. In fact analysis-synthesis ensures a minimal set of functions offering non-duplicated services to the system.

Minimal: All independent features are also separated which satisfies the minimality criteria. Analysis-Synthesis goes a step further and combines similar features if possible.

Generality: All operations are designed in such a way that they are as general as possible. The design of the search function and the consideration of the relational aspects demonstrated this. When common components are made identical this introduces a greater level of generality.

Opaque: This criteria is more reserved for systems designed with information hiding as the criteria for module breakdown. If we examine our search function more closely however, we find that we have actually hidden a secret from our list manipulation routines. This secret has been addressed continuously throughout the design process, and is in the nature of the relation between the keys.

The analysis-synthesis technique is remarkably similar to the information hiding method outlined by Parnas [3] in the results that are produced. By results it is meant that both techniques produce systems whose design aspects are desirable according to the above criterion. System functionality should be identical regardless of the design methodology employed. Both produce flexible modular code whose modules do not necessarily correspond to sequential steps in processing. Information hiding also seems to be more of a design only technique with the method and design of programs left to the developers discression.

Analysis-synthesis on the other hand is a more flexible methodology which may be applied to both system design and the programming of the system. There is no conscious decision to encapsulate secrets although this may occur in the manner as outlined by our search function. There is also no reason why an information hiding design strategy could not be implemented with an analysis-synthesis approach to the programming of the system. This combination it would appear incorporates all of the desirable features outlined by Hoffman [4] in both the design and programming of any developed systems. It would undoubtedly be one of the more flexible approaches.

Using a bottom up programming technique with a top down design method like stepwise refinement could be usefully employed, although it would be more restrictive than some of the other methods already mentioned. The sequential nature of the processing used by stepwise refinement would seem to indicate that the number of common useable components that we could identify would be less. Therefore a smaller amount of reusable code is available to develop the system.

Another advantage of the analysis-synthesis approach, as with information hiding, is the promotion of a hierarchical model. We find that we have developed a set of primitive functions at the lowest level, and then use these functions to build other more complex functions. This layered approach continues until the most complex function has been implemented, and we have enough functions to implement our target system.

Conclusion

As a concluding note, we have tried to demonstrate the analysis-synthesis approach for a small system where it was possible graphically visualise the operations that were to be performed. This may not always be possible when developing for larger systems. What it does indicate is that a detailed understanding of each modules operation, and the requirements from each common component is needed. If this is achieved then analysis-synthesis offers an efficient means to design, construct, and program systems in a most flexible manner.

References

1. Reed, K. "Lecture notes in Computer Science",
"Complex Example of Analysis-Synthesis".

2. Parnas, D.L. CACM Vol 15 No 5, May 1972 pp 330-336,
"A Technique for Software Module Specification with Examples".

3. Parnas, D.L. CACM Vol 15 No 12, Dec 1972 pp 1053-1058,
"On the Criteria used in Decomposing Systems into Modules".

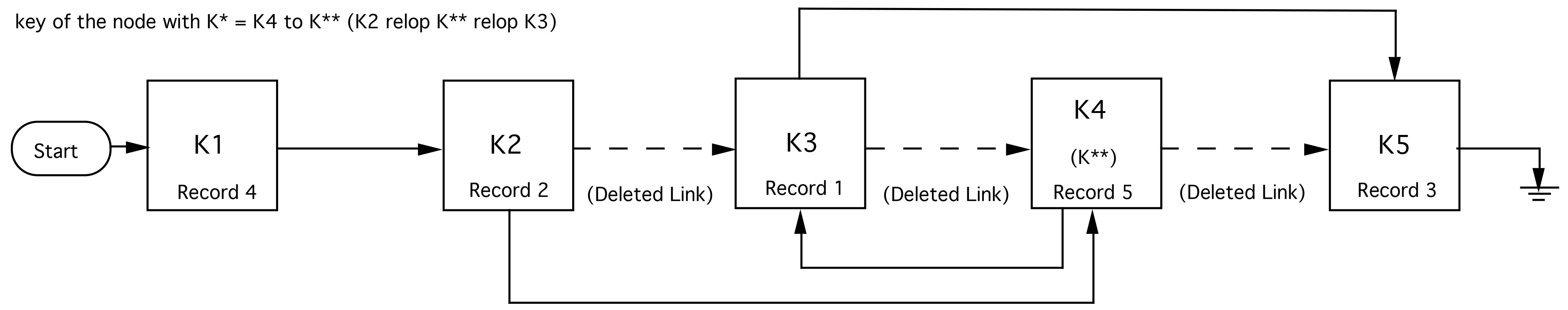
4. Hoffman, D. IEEE-TSE Vol 16 No 5, May 1990 pp 537-542,
"On Criteria for Module Interfaces".

5. Reed, K. "Modularity-Processing on a linked list ordered" Lecture Notes, cs280 Software Engineering I RMIT 1980

Appendix A - A Sample Program

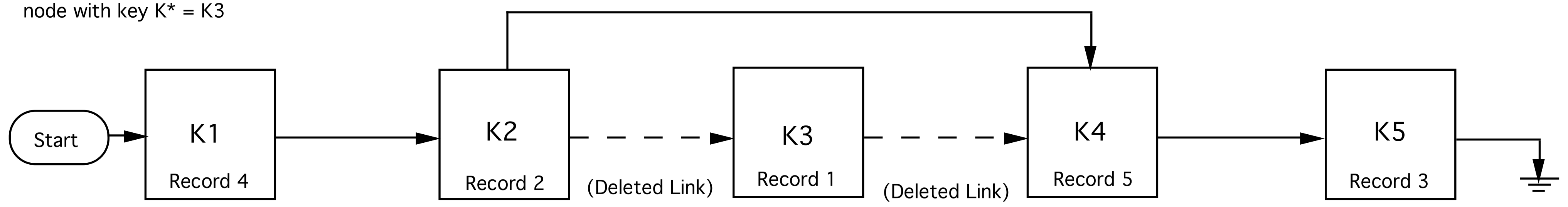
CHANGE

key of the node with K* = K4 to K** (K2 relop K** relop K3)



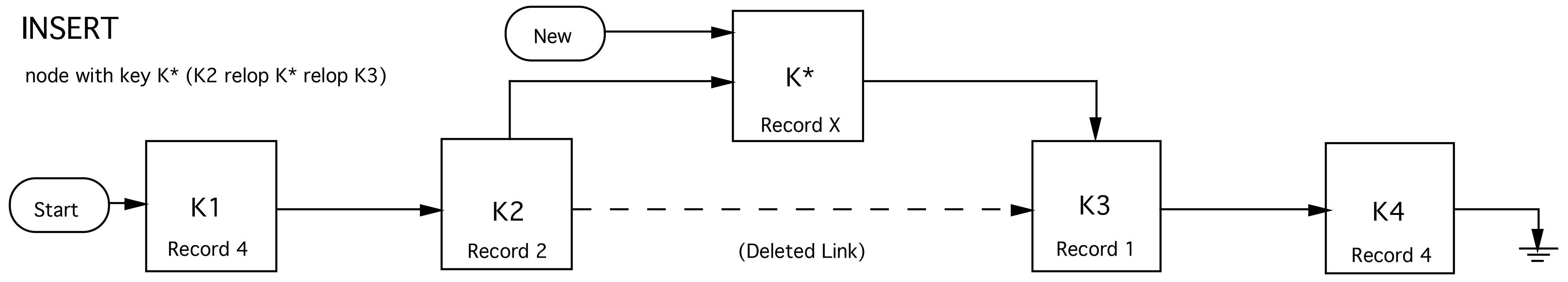
DELETE

node with key K* = K3



INSERT

node with key K* (K2 relop K* relop K3)


New
K*
Record X
Start
K1
Record 4
K2
Record 2
(Deleted Link)
K3
Record 1
K4
Record 4