UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT

Robust Storage Structures for Crash Recovery

D.J. Taylor
C.-J. Seger

Data Structuring Group
CS-85-18

July, 1985

Robust Storage Structures for Crash Recovery

David J. Taylor

Carl-Johan Seger

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

A robust storage structure is intended to provide the ability to detect and possibly correct damage to the structure. One possible source of damage is the partial completion of an update operation, due to a "crash" of the program or system performing the update. Since adding redundancy to a structure increases the number of fields which must be changed, it is not clear whether adding redundancy will help or hinder crash recovery. This paper examines some of the general principles of using robust storage structures for crash recovery. It also describes a particular class of linked list structures which can be made arbitrarily robust, and which are all suitable for crash recovery.

1. Introduction

A robust storage structure is an implementation of a data structure which is intended to provide the ability to detect and possibly correct errors in structural data. The two principal measures of robustness are detectability and correctability [11]. The detectability is the maximum number of changes to a correct instance of the structure which is guaranteed to leave it incorrect. This depends on some notion of correctness of instances, which may be defined in terms of an axiomatic description or implicitly by a detection procedure. The definition of correctabilty depends explicitly on the existence of a correction procedure. The correctability is the maximum number of changes whose correction by such a procedure can be guaranteed.

The damage (erroneous changes) to be detected or corrected can come from a variety of sources: incorrect update procedures, "wild stores" by unrelated update procedures, failure of the underlying storage mechanism, incomplete updates due to crashes, etc. The damage caused by incorrect update procedures is very hard to predict. The damage caused by "wild stores" or failure of the underlying storage mechanism will presumably affect only one field (or possibly, one node) per occurrence. Thus, all analysis has previously been performed using the total number of fields (or nodes) modified as the measure of damage. In the case of a crash affecting a correct update operation, the possible damage can be exactly determined. The purpose of this paper is to analyse the effects of

such damage on robust storage structures, in particular those for linked lists.

Of course, it is possible to design storage structures which are robust against crashes, but not against other types of damage. One early example, described by Lockemann and Knutsen [6] used redundancy to recover disk allocation data after a crash. In his Ph.D. thesis [13], J. E. Vandendorpe described a B-tree storage structure which allows partially-completed updates to be completed after a crash.

The usual procedure to recover from crashes is to perform backward recovery [1, Chapter 7] to recreate a previous system state. In this paper, we consider an alternative: performing forward recovery, by the use of storage structure correction routines. Such routines, if successful, will either recreate a previous state of the storage structure, as backward recovery would, or will complete a partially completed update operation.

In the remainder of the paper, we first define some terminology and notation. Then, we consider crash recovery using global correction routines: we consider two models for the updating of storage structures, one suited to main storage and one suited to external storage. Then, we examine the somewhat surprising results which can be obtained using a local correction routine for crash recovery. Finally, we present a summary and some possible directions for future work.

2. Terminology and notation

When discussing recovery of a storage structure after a crash, the situation of interest occurs when the structure was being updated at the time of the crash. We refer to the structure instance which existed prior to the update operation as the *"before" instance*. We refer to the structure instance which would have existed if the update operation had completed as the *"after" instance*. The instance which actually exists when the crash occurs is referred to as the *partially updated instance*. We say that a correction procedure *recovers to* either the "before" or the "after" instance if the correction procedure modifies the partially updated instance so that it becomes one or the other of these correct instances. Finally, crash recovery is *successful* if a correction (or other) procedure recovers to either the "before" or the "after" instance.

We will use the term *r-correction routine* to mean a procedure which performs correction of up to r errors in a particular storage structure, and whose behaviour is unspecified if there are more than r errors. Such a correction routine will also be referred to as a *global correction routine* to distinguish it from a local correction routine, as discussed in Section 5.

In this paper, linked lists are used as examples. All of the linked lists considered have a "uniform" pointer arrangement, with each pointer field pointing to a node at some fixed number of nodes following or preceding the node which contains the pointer. Such structures can be conveniently described by giving a vector of pointer distances, with positive values for forward pointers and negative values for back pointers. For example, a standard double-linked list is $(1, -1)$ and a modified(2) double-linked list is $(1, -2)$. Examples are drawn from three parameterised families of lists: modified(k) double-linked (pointers $1, -k$) [11], k-spiral (pointers $1, 2, \cdots, k-1, -k$) [3], and k-linked

(pointers $1, 2, \cdots, k, -1, -2, \cdots, -k$) [2]. Four specific lists are used as examples, standard double-linked (which is also modified(1) double-linked and 1-linked), 2-spiral (which is also modified(2) double-linked), 3-spiral, and 2-linked.

Each list has as many header nodes as the maximum pointer distance (k in each of the above cases). These headers are all assumed to be directly accessible, without following pointers from other headers. For all lists, we assume each node has an *identifier field,* containing a value uniquely identifying the instance, and that the (first) header node contains a count of the number of non-header nodes in the instance. For these lists, and all other structures, we assume the Valid State Hypothesis [11], which asserts that the identifier field values for a structure instance occur only in identifier fields of that instance, and nodes outside the instance do not contain pointers into the instance.

3. Analysis for structures in main storage

In this section, we examine crash recovery for structures held in main storage. For our purposes, the distinguishing feature is that each modification to a field is reflected immediately in the structure. In the next section, we consider structures on external storage. There, several modifications may be made to a node of the structure, and then all appear simultaneously as the node is written back to external storage. In most systems, structures in main storage do not survive system crashes, but it is still important to consider the effects of partially completed updates in this context. One reason is that an update routine might fail during its operation, leaving a partially updated instance to be accessed later by that program (if it continues execution) or some other program sharing the structure. In such cases, a backward recovery technique will often be appropriate, but it is also interesting to consider recovery using a correction routine. Another reason is simply to provide background for the next section.

Following a crash, we must detect errors and, if possible, correct them. For error detection, the situation is, in practice, straightforward. It would be possible for an update routine to modify a structure in such a way that one or more intermediate correct instances were created, between the initial and final correct instances. In practice, update routines do not have such behaviour: all intermediate states are incorrect instances and therefore any partially completed update can be detected.

The harder problem is correction. To deal with this problem, we must consider in some detail the operation of update and correction routines. We normally restrict correction routines (and other routines) to accessing nodes by following pointers from already accessible nodes. Then, any nodes which are not connected to the header(s) of the instance are irrelevant to the correction routine. We divide the changes made by an insertion routine into two categories: (1) changes to fields in the node being inserted and (2) other changes. (Denote the number of changes in these categories by c_1 and c_2.) Then, all the changes in category 1 can be made first, without having any effect on the correction routine. All of these changes will appear simultaneously when the first change creating a pointer to the new node is made. Denote the number of pointers to the new node by p.

The description in the preceding paragraph is for the insertion of a single node, but can readily be applied to deletion as well. For example, the node being deleted effectively disappears when the last pointer to it is changed. As well, it can be adapted to the situation in which a single update operation inserts or deletes several nodes, such as a key insertion into a B-tree which splits both a leaf and its parent branch node. For simplicity, we assume throughout that an insertion operation only adds a single node.

We can prove the following result, by using the ability to make c_1+1 changes effectively simultaneously.

Theorem: Recovery by an r-correction routine from a crash during an insertion operation can be guaranteed iff (a) $c_1+c_2 \leq 2r+1$ or (b) $c_2 \leq 2r+1$ and $p \leq r+1$.

Proof: (If part). If case (a) holds, the result is trivial, since any sequencing of the desired changes will always leave the partially updated instance within r changes of either the "before" or "after" instance. So, consider case (b).

We will perform the insertion by making changes in this order: all the changes in the first category, exactly $min(r, c_2-p)$ of the changes in the second category excluding changes which create pointers to the new node, one change creating a pointer to the new node, the remaining changes. For convenience, we will call the change creating the first pointer to the new node the *central* change. As we make changes from the first category, they are invisible to the correction routine (or any other routine accessing the instance), so crash recovery requires no changes to the instance. Since we make at most r changes from the second category before the central change, the correction routine will successfully recover to the "before" instance if a crash occurs here. For crashes after the central change, we must consider two cases. If $r \leq c_2-p$, then we make r changes from category 2 before the central change and since there are at most $2r+1$ changes in category 2, there can be at most r changes remaining after the central change. If $r > c_2-p$, then we make all category 2 changes before the central change except for pointers to the new node and hence exactly $p-1$ changes remain after the central change, which is less than or equal to r by assumption. Thus, immediately after the central change, the partially updated instance is within at most r changes of the "after" instance, and the r-correction routine is guaranteed to succeed.

(Only if part). We begin by observing that the correction routine will "see" changes either when they are made, or when the node containing them becomes accessible due to a pointer change. We also observe that if any node which is part of the "before" instance becomes inaccessible during the insertion, the correction routine must fail, since that node is part of both the "before" and "after" instances, but cannot be located by the correction routine. Thus, the only changes which can appear simultaneous to the correction routine are changes in the node being inserted, combined with a pointer change creating a pointer to the node.

Since we now must show that successful crash recovery implies either (a) or (b), we assume that (a) is false and show that (b) then follows. If there are more than $r+1$ pointers to the new node, then after the first pointer to the new node is created, the partially updated instance will not be within r changes of the

"after" instance. If $r+1$ or more changes have been made to that point, it will also not be within r changes of the "before" instance, so recovery will be impossible. If not, then following the $r+1$'th change, recovery will be impossible, since all changes will be visible and by assumption the total number of changes required exceeds $2r+1$.

If there are more than $2r+1$ changes in category 2, then consider the situation after $r+1$ of these changes have been made. Regardless of the visibility of the changes in category 1, at least $r+1$ changes are required to reach either the "before" or the "after" instance, so the r-correction routine cannot succeed. Thus, successful crash recovery implies either (a) or (b) holds. □

Although this result is stated for insertion, by appropriately restating the two categories and the definition of p so that they apply to deletion, it can be trivially extended to apply to deletion as well. For structures such as linked lists, for which insertion and deletion are exactly symmetric, no separate consideration of deletion is required.

To assess the practical implications of the above result, we can consider some linked list examples, as described in the preceding section. Table I includes four specific lists and three parameterised families, and simply gives the correctability (r), the number of changes in each category, and the value of p, for each list.

It is not possible to provide a brief justification of all the values in the table, but consider the fourth line as an illustration. The correctability of a 2-linked list has been published previously [2]. There are four pointers and an identifier field to be changed in the new node, so c_1 is 5. Outside the new node, we must change one $+1$ pointer, two $+2$ pointers, one -1 pointer, two -2 pointers, and the count, so c_2 is 7. There are four pointers to the new node, one for each pointer distance, so p is 4.

We observe that none of the structures satisfy $c_1+c_2 \leq 2r+1$, all of the structures satisfy $p \leq r+1$, but only the standard double-linked list and the 2-linked list satisfy $c_2 \leq 2r+1$. (Some elementary algebra shows that these observations hold for the parameterised families, as well as the specific lists shown.) A complete analysis of all uniform linked list structures, in the Appendix, shows that there are many others which satisfy the conditions, but almost all of them are structures of very dubious utility. We thus conclude that very few robust lists are suitable for crash recovery. However, this conclusion is based on two assumptions used throughout this section of the paper: that the structure is in main storage and that a "standard" r-correction routine is used. The following sections examine the alternatives to these assumptions.

List	r	c_1	c_2	p
(1,–1)	1	3	3	2
(1,–2)	1	3	4	2
(1,2,–3)	2	4	7	3
(1,2,–1,–2)	3	5	7	4
modified(k)	1	3	$k+2$	2
k-spiral	$k-1$	$k+1$	$\frac{k^2+k+2}{2}$	k
k-linked	$2k-1$	$2k+1$	k^2+k+1	$2k$

Table I: Crash recovery in main storage

4. Analysis for structures on external storage

In the preceding section, guaranteeing successful crash recovery depended on making several changes appear to be simultaneous, by modifying the node being inserted before creating the first pointer to it. For structures on external storage, it is possible to do this for all nodes: the node may be read into main storage, an arbitrary number of changes applied, and then written back to external storage. We assume that writes take place atomically, that is, that a crash cannot cause a write to be partially completed. Some crashes, originating with hardware problems, may violate this constraint, but techniques are known for making writes appear atomic even in such cases [5].

We now need to consider the effect of allowing all the changes to a node being made simultaneously. We observe that by writing the new node before writing the first node containing a pointer to it, it is possible to make all the changes to the new node and one other node appear simultaneous. As in the preceding section, we restrict the explicit discussion to insertion operations which add a single node, we divide changes into two categories, and we denote the number of changes in category i by c_i. For a node in the "before" instance, M,

we denote the number of changes in M by m_M, and the number of pointers to the new node, other than in M, by p_M.

Theorem: Recovery by an r-correction routine from a crash during an insertion operation can be guaranteed iff there exists a node M such that (a) $c_1+c_2-m_M \leq 2r$ or (b) M contains a pointer to the new node, $c_2-m_M \leq 2r$, and $p_M \leq r$.

Proof: (If part). Although for efficiency, changes to most nodes would probably be accomplished with single write operations, in the proof we assume all changes are done singly except for those to node M. In case (a), we simply need divide the changes other than to node M such that no more than r of them precede the writing of node M, and no more than r follow.

In case (b), we perform the changes in the following order: changes to the new node, $min(r, c_2-m_M-p_M)$ changes from category 2 excluding node M and all pointers pointing to the new node, changes to node M (simultaneously), all remaining changes.

The changes to the new node do not become "visible" until node M is written, so the number of visible changes prior to writing node M is clearly less than r, and thus an r-correction routine will recover to the "before" instance. To examine the number of changes immediately after writing node M, we must consider two cases. First, if $r \leq c_2-m_M-p_M$, then the number of changes remaining after writing node M is c_2-m_M-r, which is less than or equal to r, so recovery to the "after" instance will occur. Second, if $r > c_2-m_M-p_M$, then only p_M changes remain after writing node M, which is less than or equal to r by assumption, so again, recovery is guaranteed.

(Only if part). We begin by noting that the only case in which changes to two nodes can appear simultaneous occurs when the first node is written containing a pointer to the new node. (As in the preceding section, a node in the "before" instance which becomes inaccessible will prevent crash recovery. Thus, no other node which is part of the "after" instance can change from inaccessible to accessible.) We must show that successful crash recovery implies either (a) or (b), so we assume that (a) is false and show that (b) then follows.

Let M be the node which is written when the number of changes first exceeds r. Unless this write creates the first pointer to the new node, it can cause a maximum of m_M changes. Since (a) is false, the partially updated instance will then be more than r changes distant from both the "before" and "after" instances. So, we conclude that this write must also create the first pointer to the new node.

Immediately after writing M, at least p_M changes will remain to be made. To allow recovery, this must be less than or equal to r, so we conclude $p_M \leq r$. The maximum number of changes which can become "visible" when M is written is m_M+c_1. The number of changes excluding these is c_2-m_M. Since the number of changes preceding the writing of M is not greater than r, and the number of changes remaining after the writing of M also must not be greater than r, we conclude that $c_2-m_M \leq 2r$, as required. □

The theorem requires that we select a node, M, to satisfy two inequalities, but in practice selecting M is quite simple. Usually, a node will contain at most one pointer to any other specified node, and therefore p_M will be the same for all nodes containing a pointer to the new node. Thus, M must simply be selected as the node containing a maximal number of changes among those nodes containing a pointer to the new node.

We can consider again the linear list examples used in the previous section. Table II gives the values of r, c_1, c_2, m_M, p_M, and c_2-m_M. M is chosen to maximise m_M, among nodes which contain pointers to the new node. The value of p_M does not depend on the choice of M for these lists.

Again, we justify the fourth line of the table, for illustrative purposes. The values of correctability, c_1, and c_2 were discussed in the previous section. The maximum number of changes to any node other than the new node occurs for the nodes immediately preceding and following the new node. In each of these, two pointers must be changed, so m_M is 2. Since p is 4 and each of these nodes contains one pointer to the new node, p_M is 3.

List	r	c_1	c_2	m_M	p_M	c_2-m_M
(1,–1)	1	3	3	1	1	2
(1,–2)	1	3	4	1	1	3
(1,2,–3)	2	4	7	2	2	5
(1,2,–1,–2)	3	5	7	2	3	5
modified(k)	1	3	$k+2$	1	1	$k+1$
k-spiral	$k-1$	$k+1$	$\frac{k^2+k+2}{2}$	$k-1$	$k-1$	$\frac{k^2-k+4}{2}$
k-linked	$2k-1$	$2k+1$	k^2+k+1	k	$2k-1$	k^2+1

Table II: Crash recovery on external storage

The k-spiral data applies only for $k>1$ (for a 1-spiral, m_M is 1 not 0, so m_M and c_2-m_M are incorrect for $k=1$).

For very short lists, the values may be different from those indicated, since nodes "following" the new node and nodes "preceding" the new node may overlap. Also, insertions near the beginning or end of a list may produce different values, by causing other changes in the header node containing the count. In all cases, the differences will make m_M larger or p_M smaller, so that recovery will not be precluded in cases where the table indicates recovery is possible. In some cases, this might allow recovery where the table indicates recovery is not possible, but this is of little interest, since we would like to guarantee recovery for all insertions regardless of list length or insertion position.

The only specific lists in the table for which recovery can be guaranteed are the standard double-linked list and the 2-linked list. Here, some algebra reveals one additional list in the parameterised families for which recovery can be guaranteed, the 3-linked list. However, it is disappointing that the only new list structure for which crash recovery can be guaranteed is a 6-pointer list. A complete analysis of all uniform linked list structures, in the Appendix, shows that there are a few others which could be of some practical use.

The conclusion which must be drawn from this and the preceding section is that crash recovery can be guaranteed only for a very limited selection of "useful" list structures, if recovery is performed using a standard r-correction routine.

5. Local error correction

The intuitive idea of local error correction is that an arbitrary number of errors can be corrected, provided not too many are encountered at once. A precise characterisation of this property is complex [3], and is not attempted here. It seems unlikely that a local correction routine should be suitable for crash recovery, since the "erroneous" changes to be fixed by the correction routine will be all or almost all near the new node being inserted. Any correction routine which uses a conventional traversal should encounter at least most of them "all at once" and thus offer no improvement over a global correction routine. In this case, intuition is wrong.

The results in this section concern a particular local correction routine, used for k-spiral lists, thus we begin with a brief description of its operation. A more complete description and a proof of its behaviour is available elsewhere [3].

In a k-spiral list, there is one back pointer (which spans k nodes) and $k-1$ forward pointers $(1, 2, \cdots, k-1)$. The local correction routine traverses the list backwards. Its principal task is the verification and correction of the back pointers. Once a back pointer has been verified or corrected, then the forward pointers in the node it points to can easily be corrected, since they must point to the last $k-1$ nodes traversed. The routine maintains a vector of the last k nodes traversed. It obtains k *constructive votes* for the location of the next node by following the $-k$ pointer in the "oldest" node in the vector, $-k$ then $+1$ from the next node in the vector, ..., $-k$ then $k-1$ from the node most recently placed in the vector. For each distinct node located in this way, the routine evaluates $k-1$ *diagnostic votes,* by checking whether the forward pointers have the expected

values. The correct next node in the traversal is that one which receives at least k votes in total. Once the identity of this node has been established, the forward pointers in the node and the $-k$ pointer in the oldest node in the vector may be corrected if necessary. This procedure is guaranteed to correct an arbitrary number of errors in a list, provided that the total number of errors in all the pointers just described is less than k each time the voting is carried out. (Details of the handling of identifier fields and the count have been omitted.)

Figure 1 shows that part of a 4-spiral list relevant to one application of the voting procedure just described. In the figure, the correction procedure is attempting to reach node 4, having already traversed nodes 8, 7, 6, and 5. Only the pointers used by the voting procedure are shown: one forward pointer in each of nodes 1, 2, and 3; all forward pointers in node 4; and the back pointer in each of nodes 5, 6, 7, and 8.

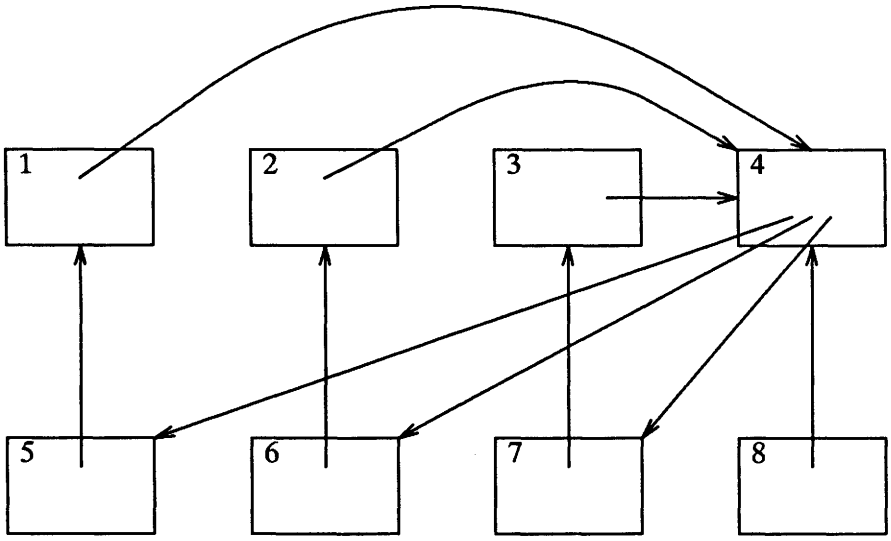

Figure 1. Voting structure in a 4-spiral list

We begin by restricting our attention to main storage. We will make the changes required for insertion in a particular order. The sequence of changes should be intuitively appealing, but the only real justification is the proof of crash recovery which follows. For convenience, let us denote the nodes in the "after" list by their position relative to the new node: N will be the new node, $N+1$ will be the node following the new node, etc. We make changes in the following sequence: all fields in N, back pointer in $N+1$, ..., back pointer in $N+k-1$, $+1$ pointer in $N-1$ (the central change), remaining changes in any order. Note that all changes to nodes following the new node are made before the central change, except the back pointer in $N+k$. After correcting the pointer structure, the local correction routine simply sets the count to the number of nodes it has observed, so we omit any further discussion of the count field.

The new node does not become visible to the correction routine until the central change is made, so after any preceding change, the only changes separating the "before" instance and the partially updated instance are the back

pointers in $k-1$ or fewer nodes following the new node. Since this means there are at most $k-1$ changes in total, local correction will clearly succeed.

Immediately after the central change, the number of pointer changes separating the partially updated instance and the "after" instance is $k-2$ in node $N-1$, $k-i$ in node $N-i$ for $i=2, \cdots, k-1$, and 1 in node $N+k$. Thus, the total number of pointer changes is $(k-1)(k-2)/2+k-1$. For $k=2$ this evaluates to 1: the local correction routine will clearly be successful, since the total number of errors does not exceed $k-1$. For larger values of k, this simple justification does not work, so we must examine the operation of the local correction routine in more detail.

As the correction routine tries to reach nodes $N+k-2$ through $N+1$, it will encounter the "erroneous" forward pointers in nodes $N-k+1$ through $N-1$. If $N+j$ should be the next node reached in the traversal, it will receive at least one constructive vote, since the back pointer to it is correct. It will also receive $k-1$ diagnostic votes, since all its forward pointers are correct. The only other node receiving any constructive votes will be $N+j+1$ and it will receive no diagnostic votes. Thus, $N+j$ will be the only node receiving k votes, and the correction routine will perform the proper traversal, but will not correct any of the forward pointer errors encountered during this part of the traversal.

When the correction routine tries to reach node N, we obtain the following constructive votes: starting from $N+k$, a vote for $N-1$; starting from $N+k-1$, a vote for N; the other $k-2$ votes for $N+1$. N gets $k-1$ diagnostic votes (all forward pointers match), $N+1$ gets no diagnostic votes, and $N-1$ gets $k-2$ diagnostic votes (all forward pointers, except $+1$, match). Thus, we correctly select N as the next node, and correct the back pointer in $N+k$. Figure 2 shows the situation when trying to reach node N in a 4-spiral. All pointers used in constructive votes are shown. Pointers used in diagnostic votes are shown only if they apparently point to the right place, thus no forward pointers are shown in node $N+1$. Pointers which need to be corrected are marked with an "x".

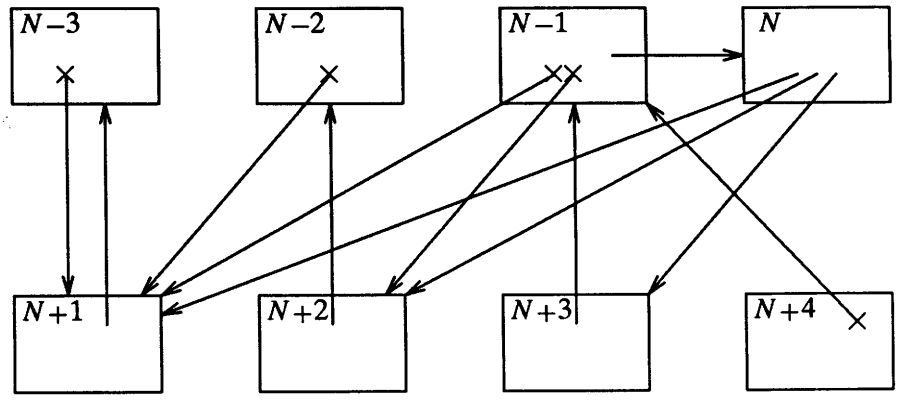

Figure 2. Immediately after central change: Trying to reach node N

The remaining pointers requiring correction are in nodes $N-k+1$ through $N-1$. The errors are all in forward pointers, but none of those pointers are used in constructive votes after node N is reached. (All were used earlier, as discussed above.) Thus, at each step, the proper node will receive k constructive votes, and will be selected regardless of the number of diagnostic votes it receives.

Thus, correction will succeed immediately after the central change. Each subsequent change simply removes one of the changes which has to be made by the correction routine, so correction will also succeed for all subsequent partially updated instances. We conclude that, provided changes are made in the specified sequence, the k-spiral correction routine will always recover from a crash during insertion. Unlike the results of the two preceding sections, this provides a class of lists of arbitrarily great robustness all of which are suitable for crash recovery.

If the list is on external storage, then we can make several changes at once, eliminating many of the partially updated instances. However, it is clearly possible to follow the above sequence of changes so that no other partially updated instances are created. In fact, it is possible to follow the above update sequence without writing any node more than once, provided the insertion is not very near the beginning or end of the list. Since we can group the central change with all other changes to node $N-1$, the maximum number of changes ever needing correction is reduced to $(k-1)(k-2)/2+1$. This provides an immediate justification for crash recovery with $k=2$ and $k=3$ on external storage, but the more complex justification is still needed for larger values of k.

6. Conclusions and further work

In this paper, we have shown how to characterise those storage structures which are suitable for crash recovery using r-correction routines. This analysis was performed separately for structures in main storage and on external storage. The implications of these characterisations were examined for uniform linked list structures, yielding the conclusion that few useful robust lists are suitable for this form of crash recovery. Then, we examined the use of a local correction routine for crash recovery. Although this seemed intuitively unsuitable, it proved to be more successful than the global correction routines. In particular, it provides a class of arbitrarily robust, useful, linked lists.

It may be appropriate to mention that the behaviour of various correction routines was first examined by observing the behaviour of actual implementations when subjected to damage caused by crashes. This provided a general understanding of their behaviour which assisted in the development of the theoretical results. In particular, the excellent behaviour of the k-spiral correction routine first came to our attention in this way. However, the results presented here do not depend in any formal sense on the experimentation.

Many aspects of crash recovery remain to be investigated. For example, we have not examined local correction in general, but only for linked lists, and there only for one particular local correction routine. Other local correction routines should also be examined, and, if possible, a general characterisation obtained of those situations in which local correction provides guaranteed crash recovery.

Another aspect which should be examined is case (a) in each of the two theorems. It appears that this can only occur for pathological structures: a characterisation of such structures should be sought.

One problem not addressed in the paper is that recovery to the "before" instance will usually result in the new node becoming lost. That is, it will not be part of the structure into which it was being inserted and it will also not be part of whatever free list structure it was taken from. Ideally, one would like to arrange that crash recovery of both the free list and the other structure would result in the node being placed in exactly one of the two structures. Because it is generally necessary to completely change the new node before creating any pointers to it (and, symmetrically during deletion, to delete all pointers to a node before changing the node itself), this cannot be guaranteed. This leads to two problems: loss of usable storage space and violation of the Valid State Hypothesis. Probably the only general solution is to use some form of garbage collection after crash recovery. If all structures are checked during crash recovery, such garbage collection can probably be achieved at little additional cost.

Another problem which has not been addressed is a crash during execution of a correction routine. Some correction routines, including the k-spiral correction routine, do not modify the structure except to make corrections. Other correction routines use "guessing" [10] which involves making trial corrections which may be wrong. A crash in the middle of such a correction routine could result in additional damage to the structure. In principle, one could modify such routines so that they did not make actual changes to the structure until a guess had proved correct, but this could greatly complicate the checking of a structure to validate a guess.

One advantage which is gained by local correction is that an error in the count field does not affect the operation of the correction routine. For some lists, it is possible to make a minor modification to a global r-correction routine so that it always corrects count errors plus up to r other errors. Such routines should be examined to determine whether they might be significantly better for crash recovery. It is possible that other *ad hoc* modifications might be made which would have no harmful effect on the ordinary behaviour of a correction routine but which would greatly improve crash recovery.

As well, the investigation should be extended to consider other specific examples in addition to linked lists. We do not know of any parameterised families of B-trees or binary trees, but two specific robust B-trees [4, 9], and four specific robust binary trees [7, 8, 11, 14] have been reported. These should be examined for their suitability in crash recovery. It is possible that none of them will prove suitable. In that case, other robust implementations should be sought which will have as good detectability and correctability, and also resistance to crashes.

Appendix: Analysis for all uniform lists

The main body of the paper examines only a few specific list structures and three parameterised families of lists. These probably include all lists which should be used in real systems, but for completeness, this appendix includes an analysis of all uniform list structures, meaning all those with a uniform pointer arrangement as described in Section 2.

For this appendix, we need some additional notation. We denote the vector of pointer distances by $(d_i)_{i=1}^{n}$. (Duplication is allowed, although of dubious value, as discussed below.) Since the quantity is needed frequently, we let $A=\sum_{i=1}^{n}|d_i|$. The correctability of such a list is $\min(\lfloor A/2\rfloor,n-1)$ as shown previously [12]. Throughout, we denote the correctability by r.

A.1 Lists in main storage

We can now calculate the quantities c_1, c_2, and p defined in Section 3. The number of changes in the new node is simply one greater (because of the identifier field) than the number of pointers in a node. The remaining changes are a change to the count, and pointer changes. For a pointer with distance d_i, a total of $|d_i|$ changes to pointers not in the new node must be made. The number of pointers to the new node is simply the number of pointers in a node, since one pointer at each distance points to each node. Thus, we have $c_1=n+1$, $c_2=A+1$, and $p=n$. (We will also use these values of c_1 and c_2 in the next section.)

The lists for which we can guarantee crash recovery have either $c_1+c_2\leq 2r+1$ or $c_2\leq r+1$ and $p\leq r+1$. Since $c_1+c_2=A+n+2$, which is greater than both $2(\lfloor A/2\rfloor)+1\leq A+1$ and $2(n-1)+1$ (clearly, $A\geq n$), no uniform linked list meets the first condition. Thus, crash recovery can be guaranteed precisely for those lists meeting the two inequalities of the second condition.

If $r<n-1$, then $p\leq r+1$ is false, so for successful crash recovery, we must have a list with correctability exactly $n-1$. Since we require $c_2\leq 2r+1$ and we have $c_2=A+1$ we obtain $A\leq 2n-2$. If this inequality is strict, the correctability will not be $n-1$, so we conclude that $A=2n-2$.

Since A does not depend on the signs of the pointer distances, we can give unsigned vectors of pointers, to represent a class of lists obtained by attaching any combination of signs to the distances. The class of lists satisfying $A=2n-2$ is infinite: the first few members are: (1,1), (1,1,2), (1,1,2,2), (1,1,1,3), (1,1,2,2,2).

Clearly, most of these are not very intuitively appealing lists. If we assert that we are only interested in lists without duplicated pointer distances, we will eliminate most of them. (We provide no theoretical justification for eliminating all other lists, but simply state that a list in which each node has two or more copies of some pointer field does not seem useful.) For convenience, call lists without duplicated pointer distances *useful*. Now, the signs of the distances become important. If an unsigned pointer distance occurs at most twice, we can avoid duplication. For the lists being considered, if there are more than four pointers, duplication must occur since $1+1+2+2+3(n-4)>2n-2$ for $n>4$ (if we

have only two each of 1 and 2, all the others must be at least three). Thus, only the first three of the lists above are possibilities. From them we obtain four list structures: (1,–1), (1,–1,2), (1,–1,–2), and (1,–1,2,–2). Since the two three-pointer lists are mirror images of each other, there are really only three distinct list structures. Only the three-pointer lists were not included in Section 3.

If the definition of "useful" is accepted, we must conclude that there are very few useful uniform list structures suitable for crash recovery in main storage.

A.2 Lists on external storage

Here, we must find all lists satisfying either $c_1+c_2-m_M \leq 2r$ or both $c_2-m_M \leq 2r$ and $p_M \leq r$, with the conditions on M as stated in Section 4. If the correctability is $n-1$, the first condition becomes $A+n+2-m_M \leq 2n-2$, which is $A-m_M \leq n-4$. Since m_M is at most n (if all pointer distances have the same sign), this condition can only be satisfied if $A \leq 2n-4$. However, if this is the case, the correctability is not $n-1$ since the correctability is at most $\lfloor A/2 \rfloor$. If the correctability is $\lfloor A/2 \rfloor$, then the first condition becomes $A+n+2-m_M \leq 2(\lfloor A/2 \rfloor) \leq A$. Again, $m_M \leq n$, so we have $A+2 \leq A$, a contradiction. We conclude that there are no uniform linked lists satisfying the first condition.

We consider the second condition, made up of two inequalities, as applied to lists with correctability of $\lfloor A/2 \rfloor < n-1$, and let $j=n-\lfloor A/2 \rfloor$. $\lfloor A/2 \rfloor = n-j$ implies $A-1 \leq 2n-2j$. This implies that $(|d_i|)_{i=1}^n$ must include at least $2j-1$ ones, hence $(d_i)_{i=1}^n$ must include at least j positive ones or j negative ones. If there are j positive ones, select M as the node immediately preceding the new node, otherwise select M as the node immediately following the new node. For this node, $p_M \leq n-j = \lfloor A/2 \rfloor$. Thus, all of the lists under consideration satisfy $p_M \leq r$, for at least one node M.

We also require $c_2-m_M \leq 2r$, which becomes $A+1-m_M \leq 2(\lfloor A/2 \rfloor) \leq A-1$. For the node M selected in the preceding paragraph, $m_M \geq j$ since there are j pointers to the new node. Given the initial assumption about correctability, j is at least 2, so this inequality is also satisfied for all of the lists under consideration.

We must now consider lists for which the correctability is $n-1$. Here, for any node pointing to the new node, $p_M \leq n-1$, so the only the first inequality need be considered. That inequality becomes $A+1-m_M \leq 2n-2$. There is always one node pointing to the new node in which all the forward pointers must be changed, and another node pointing to the new node in which all the back pointers must be changed. Thus, there is a node for which $m_M \geq \lceil n/2 \rceil$. If we consider starting with a collection of unsigned distances and attaching signs to them, by making all the signs the same we can achieve $m_M = n$. Thus, if $A \leq 2n+\lceil n/2 \rceil - 3$, crash recovery can be guaranteed. If $A \leq 3n-3$, then signs can be attached to the collection of unsigned distances so that crash recovery can be guaranteed.

Each of the inequalities just derived is conditioned on the correctability being $n-1$, but if the correctability is smaller, then crash recovery can always be guaranteed. Since the condition for correctability less than $n-1$ is stronger than both inequalities, we can ignore the distinction and simply use these two

inequalities for all lists. Enumerating lists satisfying these conditions is very tedious, even for small values of n. Omitting one list from each mirror-image pair, the only two-pointer lists are (1,1), (1,–1), and (1,2), but for three pointers there are 14 lists.

Again, we restrict our attention to "useful" lists, with no duplicate pointers. The minimum value of A for such lists is achieved by having the absolute values of the pointer distances as 1, 1, 2, 2, 3, 3, ... Since the sum of these grows quadratically with n and the upper bound on A is $3n-3$, there is clearly an upper bound on the number of pointers per node in such lists. A quick calculation gives $n \leq 8$, although we eventually discover that there are no such lists with seven or eight pointers.

The complete set of lists can be found by enumeration of list structures satisfying the established inequalities. Again omitting mirror-image duplicates, Table III gives all such lists.

n	Pointer vectors
2	(1,–1) (1,2)
3	(1,–1,2) (1,–1,3) (1,2,–2) (1,2,3)
4	(1,–1,2,–2) (1,–1,2,3) (1,–1,2,–3) (1,–1,2,4) (1,2,–2,3)
5	(1,–1,2,–2,3) (1,–1,2,–2,4) (1,–1,2,3,–3) (1,–1,2,3,4)
6	(1,–1,2,–2,3,–3) (1,–1,2,–2,3,4)

Table III: "Useful" lists for crash recovery

The table includes, of course, the k-linked lists for $k \leq 3$. Some of the other lists are probably as useful as the k-linked lists, but for a given number of pointers, likely none of them would be considered particularly good, other than for crash recovery.

References

1. T. Anderson and P. A. Lee, *Fault Tolerance: Principles and Practice,* Prentice-Hall, Englewood Cliffs, N. J. (1981).
2. J. P. Black, D. J. Taylor, and D. E. Morgan, A compendium of robust data structures, *Digest of Papers: Eleventh Annual International Symposium on Fault-Tolerant Computing*, pp. 129-131 (24-26 June 1981).
3. J. P. Black and D. J. Taylor, Local correctability in robust storage structures, CS-84-44, Dept. of Computer Science, University of Waterloo (December 1984). Accepted for publication in *IEEE Transactions on Software Engineering*.
4. J. P. Black, D. J. Taylor, and D. E. Morgan, A robust B-tree implementation, *Proceedings of the Fifth International Conference on Software Engineering*, pp. 63-70 (9-12 March 1981).
5. W. H. Kohler, A survey of techniques for synchronization and recovery in decentralized computer systems, *Computing Surveys* **13**(2) pp. 149-183 (June 1981).
6. P. C. Lockemann and W. D. Knutsen, Recovery of disk contents after system failure, *Communications of the ACM* **11**(8) p. 542 (August 1968).
7. J. I. Munro and P. V. Poblete, Fault tolerance and storage reduction in binary search trees, *Information and Control* **62**(2/3) pp. 210-218 (August/September 1984).
8. S. C. Seth and R. Muralidhar, Analysis and design of robust data structures, *Digest of Papers: Fifteenth Annual International Symposium on Fault Tolerant Computing*, pp. 14-19 (19-21 June 1985).
9. D. J. Taylor and J. P. Black, A locally correctable B-tree implementation, CS-84-51, Dept. of Computer Science, University of Waterloo (December 1984). Accepted for publication in *Computer Journal*.
10. D. J. Taylor and J. P. Black, Principles of data structure error correction, *IEEE Transactions on Computers* **C-31**(7) pp. 602-608 (July 1982).
11. D. J. Taylor, D. E. Morgan, and J. P. Black, Redundancy in data structures: Improving software fault tolerance, *IEEE Transactions on Software Engineering* **SE-6**(6) pp. 585-594 (November 1980).
12. D. J. Taylor, *Robust Data Structure Implementations for Software Reliability,* Ph. D. Thesis, University of Waterloo, Ontario, Canada (August 1977).
13. J. E. Vandendorpe, *A crash tolerant B-tree structure for database retrieval systems,* Ph.D. Thesis, Illinois Institute of Technology (December 1980).
14. K. Yoshihara, Y. Koga, and T. Ishihara, A robust data structure scheme with checking loops, *Digest of Papers: Thirteenth Annual International Symposium on Fault Tolerant Computing*, pp. 241-248 (June 28-30, 1983).