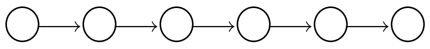
6.897: Advanced Data Structures Spring 2003

Lecture 19 — May 5, 2003

Prof. Erik Demaine *Scribe: Ilya Baran, Deniss Cebikins, and Lev Teytelman*

1 Overview

In this lecture, we examine fault-tolerant data structures. To motivate their study, consider the standard implementation of a linked list:



If the first node in the list is corrupted, the entire list is lost. We would like to construct data structures with good bounds on the damage that a small number of faults may cause.

We will construct fault-tolerant versions of the stack and of the linked list and analyze their performance in terms of the damage that faults can cause and the time it takes to recover from faults. It is also possible to construct fault-tolerant trees, but we will not cover it. All of these results are based on a paper by Aumann and Bender [AB96].

1.1 Model

We will work in the pointer-machine model of computation. Our model of faults is as follows:

1. A node may become faulty at any time. If a node becomes faulty, all of its data and pointers are lost forever.
2. An algorithm can detect whether a node is faulty, but only upon attempt to access the node. In practice, this may be accomplished with error-correcting codes.
3. The number of nodes is unlimited. When replacing faulty nodes, we never run out of nodes.
4. There are s "secure" words of memory which cannot become faulty. This is necessary to have an "entry-point" into the data structure. This is justified in real life by the existence of the memory hierarchy: for example, disks fail more often than RAM and RAM fails more often than registers.

1.2 Goals

We wish to build fault-tolerant versions of familiar pointer-machine data structures. The behavior of fault-tolerant data structures should be the same as that of their non-fault-tolerant counterparts and their asymptotic performance should not be worse. Furthermore, when faults occur, we would like the following characteristics:

1. There should be a constant $c > 0$ such that, as long as there are fewer than cs faults, we don't lose the entire data structure. Notice that c cannot be greater than or equal to 1, because if we lose all of the nodes to which the s secure words point, the entire data structure is necessarily lost.

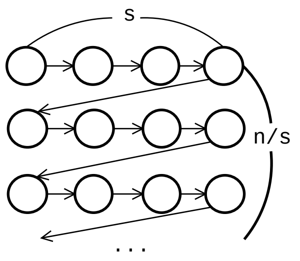
2. Suppose $f \leq cs$ faults occur before a fault is detected. The number of nodes we lose should depend only on f, not on the size of the entire data structure, n. We should lose no more than $O(\ell(f))$ nodes for some reasonable (polynomial) function ℓ.

3. Similarly, if f faults occur before a fault is detected, the recovery should take time that is independent of n. The data structure should be able to recover to a consistent state using $O(r(f, s))$ time for some polynomial function r.

4. A recovery should preserve some basic properties of the data structure. For example, the elements of a linked list or a stack that are not lost should remain in the same order they were before the fault. The recovery procedure must guarantee such a specified property of the structure.

2 Stack

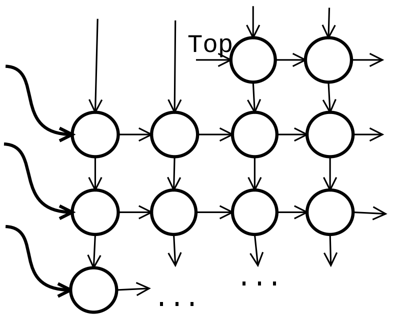
In this section we consider the stack data structure. We divide the n elements into $\lceil n/s \rceil$ layers of consecutive elements as shown. The top layer is the only layer with possibly fewer than s elements.



Horizontal links.

Horizontal links indicate the list order and are not actually stored, because the Stack data structure does not require functionality for finding the sucessor of a given node.

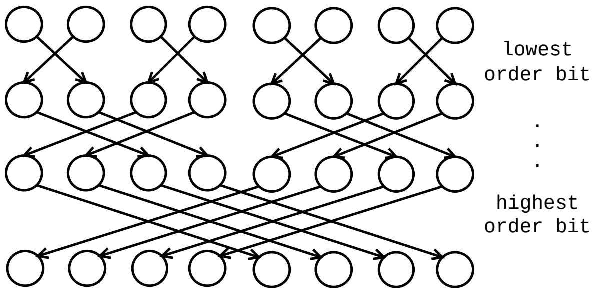
We now connect the layers vertically, and use the secure storage to point to the topmost element in each column:



Vertical links.

We are not yet done: even using both horizontal and vertical links, we can easily lose $\Omega(s^2)$ nodes with $O(s)$ pointer faults, and we would like to do better.

We now connect every $\lceil \lg s \rceil + 1$ layers with a butterfly exchange network. This means connecting x in level $i+1$ to $x \oplus 2^{i \bmod \lceil \lg s \rceil + 1}$ at level i. In other words, we flip the lowest-order bit in the binary representation of the column number in the first layer, then the next bit in the next layer, and so on to the highest-order bit. These links are called diagonal links.



Diagonal Links.

In the structure with vertical and diagonal links, every node has exactly 2 outgoing pointers, and either 2 incoming pointers or 1 secure incoming pointer (the top-layer nodes).

2.1 Implementation

The top-layer element in each column has a secure pointer. There is also a secure counter to store which column starts the topmost layer. We now need to implement *Top*, *Pop*, and *Push* methods of stack:

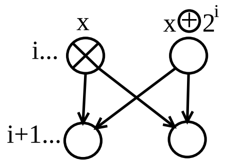
- *Top*:
  - Follow the secure pointer corresponding to the first column of the topmost layer.
- *Pop*:
  - Remove the Top element
  - Follow the Down pointer to update the secure pointer
  - Increment the secure counter
- *Push*:
  - Decrement the secure counter modulo s
  - Add a node to this column
  - Add Down, Right, and Diagonal pointers using secure pointers

Next, we are consider recovery algorithm which is invoked any time a failed node is encountered by any of the above procedures.

2.2 Recovery Algorithm

- Repeatedly pop nodes from the stack.
- For every node, check whether it is accessible by either of the incoming pointers from a previous level. If this fails, mark this node as lost and discard.
- Stop when s accessible (not lost and not faulty) consecutive nodes have been encountered.
- We now have all the secure pointers required for the next level.
- Push all popped nodes back onto the stack.

2.3 Fault Analysis



Suppose we have seen f faults so far. (There might be more down below.) We claim that $O(f \lg f)$ extra nodes are lost in this case:

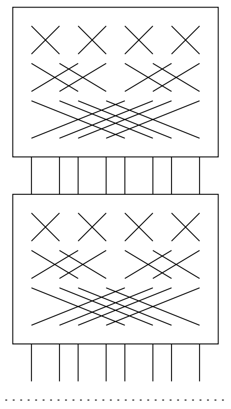
- Consider inaccessible node x in layer i. If $x \oplus 2^{i \bmod \lceil \lg s \rceil + 1}$ (the diagonal pointer from x) is also inaccessible in layer i, then both are inaccessible in layer $i+1$. If $x \oplus 2^{i \bmod \lceil \lg s \rceil + 1}$ is accessible in layer i, then both are accessible in layer $i+1$.
- We can have at most $f \lg f$ pairs $\langle x, i \rangle$ such that x and $x \oplus 2^{i \bmod \lceil \lg s \rceil + 1}$ are both faulty at any layer. Provided that $f \leq s/2$, we stay in one butterfly structure.
- Therefore, the number of lost nodes is $O(f \lg f)$.
- Therefore, the number of inaccessible nodes is $O(f(1 + \lg f)) = O(f \lg f)$.

2.4 Recovery Algorithm Performance

- There are $O(f \lg f)$ inaccessible nodes.
- Therefore, at most $O(f \lg f)$ layers are considered.
- There are s elements per layer, so recovery time is $O(sf \lg f)$.
- We can get rid of the $\lg f$ factor in the recovery time bound by noting that there can be at most f layers with inaccessible nodes before an entirely accessible layer is hit.
- Therefore, recovery time is only $O(sf)$.

3 Linked List

The fault-tolerant version of a linked list has the same underlying structure as the stack except that blocks of $\lceil \lg s \rceil + 1$ layers connected in a butterfly are separated by a layer of vertical pointers. Thus, the nodes are now partitioned into blocks.



The above structure is subject to the following invariant: each block (except maybe the top one) contains $\Theta(s \lg s)$ nodes of the linked list, which are consecutive in the list. Within a block, however, the nodes can be stored in arbitrary order, so that the reading order does not imply the actual order of the nodes in the list. To maintain the actual order of the nodes in a block, we use either a list-labeling data structure or an order-query data structure with indirection. In this way, the recovery algorithm can recover the original order of the nodes despite faults.

In this structure, Insert and Delete operations first act only on a single block where the element in question is located. Balancing out the structure to preserve the invariant on block sizes is done via the B-tree split/merge trick. Recovering data involves the additional structure used to store the order of the nodes within a block.

The performance of the above operations depends on the choice of the order-query data structure within a block. List labeling requires extra space of $O(\lg s)$ bits per node, and results in an additional $O(\lg s)$ factor in the running time of Insert and Delete for relabeling. Using a constant-time order-query data structure using indirection keeps the insertion and deletion bounds as constant but adds an $O(\lg s)$ factor in the number of nodes that can be lost from f faults, because an entire indirection substructure of size $\Theta(\lg s)$ can be lost for each single-node failure.

4 Further Results and Open Questions

A fault-tolerant version of a tree data structure is known. Also, if you use bounded-degree expander graphs instead of a butterfly network allows to get rid of an $O(\lg f)$ factor in the number of lost nodes.

There are fairly general open questions related to fault-tolerant data structures. One problem is to design fault-tolerant RAM data structures, which has the useful consequence that you can never lose any piece of the data structure (though the pieces may be expensive to find). Another problem is to generalize the known constructions to produce a fault-tolerant version of an arbitrary pointer-machine data structure.

References

[AB96] Yonatan Aumann and Michael A. Bender. Fault tolerant data structures. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, 1996, pages 580–589.