



# Shareable Data Structures

MALCOLM CROWE

6 NOVEMBER 2018

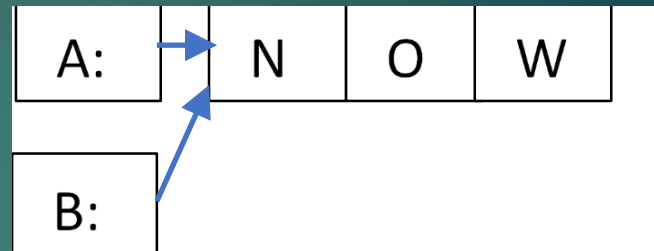
# Shareable data structures

- ▶ Data Structures are in all Computing courses
  - ▶ Revisited when student has reached Threading
- ▶ Threading examples show need for locking
  - ▶ Students learn this is why strings are immutable
    - ▶ At least in C# and Java – “value semantics”
- ▶ But why do we use unsafe data structures?
  - ▶ In this course we focus on SAFE data structures
  - ▶ For sharing and copying between threads
- ▶ The reduces the need for complex locking

# What is unsafe?

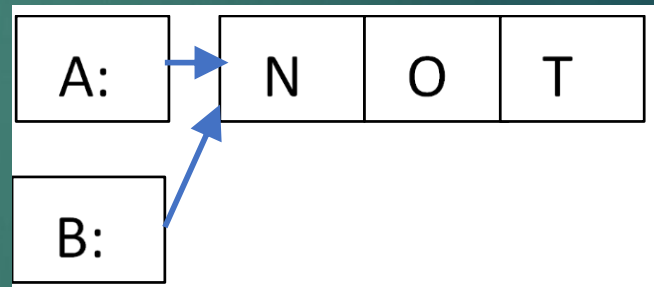
▶ Example: Arrays A and B – in Java (say)

▶ After `B=A` we have



▶ Then `A[2]='T'` gives

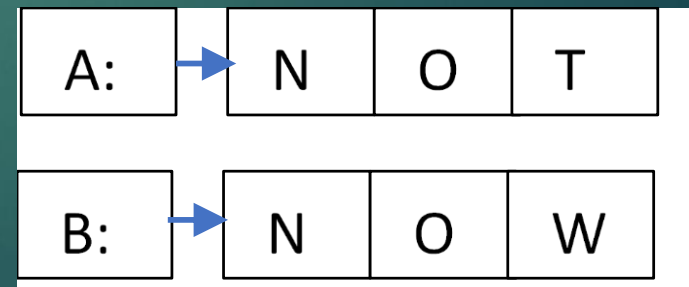
▶ (correct, maybe?)



▶ A safe array would give

▶ `A=A.Set(2,'T')`

▶ Change is just to A



# We learn about cloning?

- ▶ A concept often not grasped by students
- ▶ When a list is passed “by value” to a proc
- ▶ There is nothing to stop the proc changing it
- ▶ With value semantics this shouldn’t occur
- ▶ So maybe we need to stop using lists!
- ▶ Immutable strings are still useful, so
  - ▶ Our data structures have immutable contents
- ▶ We will still need locking for mutable things
  - ▶ We keep it to a minimum to simplify our design

# In database technology

- ▶ Once we have enough structures
  - ▶ We show how a full DBMS can be built
- ▶ Taking a snapshot is as easy as  $B=A$  above
  - ▶ People with copies can consider changes
  - ▶ On ROLLBACK they can simply be forgotten
    - ▶ For  $B=A$  example, simply restore by  $A=B$
  - ▶ On COMMIT we need to check for conflicts
    - ▶ And the DBMS can accept the changes in master copy
- ▶ The list of master copies of databases in use
  - ▶ Will be the DBMS' only unsafe data structure!

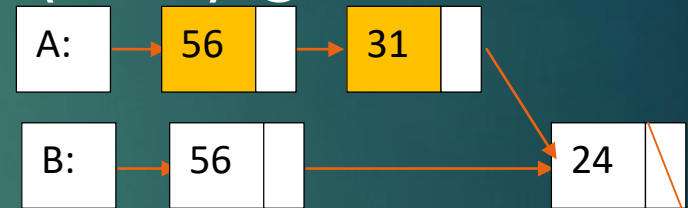
# Example: a safe linked list

- ▶ After  $B=A$  suppose we have linked list (56,24)



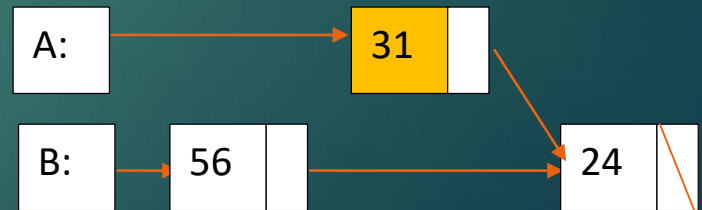
- ▶ For a safe list,  $A=A.InsertAt(1,31)$  gives:

- ▶ Coloured nodes are new



- ▶ Then  $A=A.RemoveAt(0)$ :

- ▶ Note B still has the old list



# Implementation in Java

- ▶ Shareable data structures have all fields `public final`
- ▶ So a safe linked list of integers might be:

```
public class SListOfInt {  
    public final int element;  
    public final SListOfInt next;  
    //.. And we need at least one constructor  
    //    and the methods InsertAt, RemoveAt  
}
```

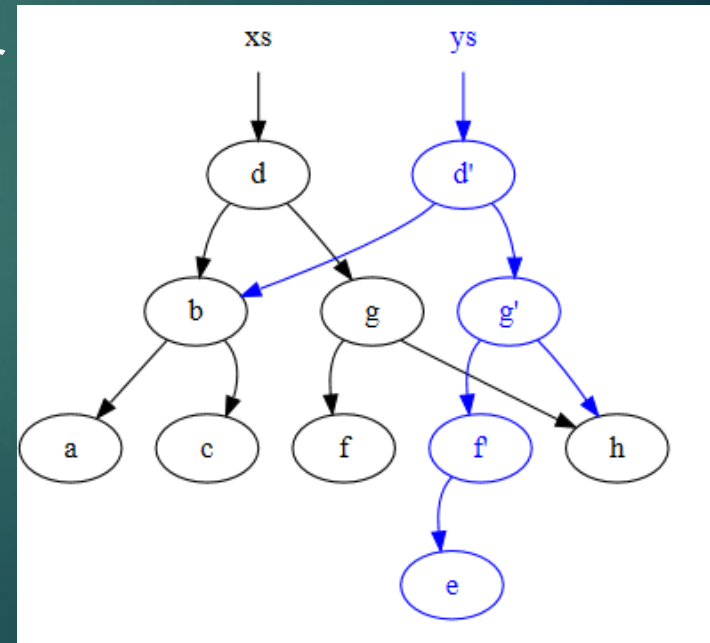
# Memory blocks are shared

- ▶ Thinking about the shareable linked-list
- ▶ The versions share nodes after the change
- ▶ Similarly for a shareable tree structure
  - ▶ For each change the new nodes are a path
  - ▶ From the root to the nodes that were changed
- ▶ More complex data structures are better
  - ▶ Even more efficient since more is shared
- ▶ Contrast with mutable structures
  - ▶ Where the whole thing needs to be cloned



# How new is this?

- ▶ “Persistent Data Structures”
  - ▶ “Multi-version”, “Concurrent” etc
- ▶ These have a similar idea, but a fatal flaw
  - ▶ A mutable root node
- ▶ “Purely functional” is better
  - ▶ See [Okasaki 1996](#)
  - ▶ Wikipedia shows this:
  - ▶ Which is exactly right.



# Is it faster?

- ▶ The memory allocator works harder
- ▶ But we avoid recursive copying
  - ▶ Top parts of structures are cheapest to change
  - ▶ Cheapest of all are stacks
- ▶ Structures such as queues are expensive
  - ▶ Since we always add at the end
  - ▶ This affects speed of breadth-first search
- ▶ This would be something worth analysing

# Bookmark replaces Iterator

- ▶ Iterators are popular for traversing structures
    - ▶ Used internally to implement foreach
  - ▶ But they are not shareable
    - ▶ And prevent change to the structure traversed
  - ▶ A Bookmark starts with snapshot of structure
    - ▶ Bookmark `b=S.First()`
    - ▶ Shareable, traverses using `b=b.Next()`
- `for (var b=S.First(); b!=null; b=b.Next())`
- ▶ Just as efficient (memory allocator works harder)

# Towards Strong DBMS

- ▶ Too much locking is a real pain
- ▶ DBMS (say they) use locking all the time
  - ▶ Deadlocks are a plague and very hard to avoid
  - ▶ Normally detected by inactivity!
- ▶ Pyrrho DBMS uses optimistic execution
  - ▶ Locks and checks everything only during commit
  - ▶ Uses shareable data structures for trees, rows
- ▶ The Strong DBMS will use them for most things
  - ▶ Especially the database and transaction objects

# Strong DBMS so far

- ▶ Every change changes at two levels:
  - ▶ The in-memory versions of the indexes (etc)
  - ▶ The list of transaction steps
- ▶ To have a single pointer to this state
  - ▶ Modified objects must be within the transaction
- ▶ It is a bit confusing to program
  - ▶ A C# constructor for each kind of change
- ▶ Always get from one consistent state to next