Concurrent Singly-Linked Lists

Amgad Rady

DisCoVeri Group
Department of Electrical Engineering and Computer Science
York University

November 10, 2015

Outline

Difficulties with Fomitchev and Ruppert's algorithm

- No good solution for the ABA problem arising from decoupling COMPARE&SET and READ operations.
- AtomicStampedReference<V> is slow due to being implemented as a boxed [reference, integer] pair.
- The problem of adversarial scheduling Fomitchev and Ruppert's algorithm is unlikely to be encountered in practice.
- There would be a significant performance penalty in having two operations on shared memory.
- Perhaps the most important reason...

Difficulties with Fomitchev and Ruppert's algorithm

- No good solution for the ABA problem arising from decoupling COMPARE&SET and READ operations.
- AtomicStampedReference<V> is slow due to being implemented as a boxed [reference, integer] pair.
- The problem of adversarial scheduling Fomitchev and Ruppert's algorithm is unlikely to be encountered in practice.
- There would be a significant performance penalty in having two operations on shared memory.
- Perhaps the most important reason...

Difficulties with Fomitchev and Ruppert's algorithm

- No good solution for the ABA problem arising from decoupling COMPARE&SET and READ operations.
- AtomicStampedReference<V> is slow due to being implemented as a boxed [reference, integer] pair.
- The problem of adversarial scheduling Fomitchev and Ruppert's algorithm is unlikely to be encountered in practice.
- There would be a significant performance penalty in having two operations on shared memory.
- Perhaps the most important reason...

Difficulties with Fomitchev and Ruppert's algorithm

- No good solution for the ABA problem arising from decoupling COMPARE&SET and READ operations.
- AtomicStampedReference<V> is slow due to being implemented as a boxed [reference, integer] pair.
- The problem of adversarial scheduling Fomitchev and Ruppert's algorithm is unlikely to be encountered in practice.
- There would be a significant performance penalty in having two operations on shared memory.
- Perhaps the most important reason...

Difficulties with Fomitchev and Ruppert's algorithm

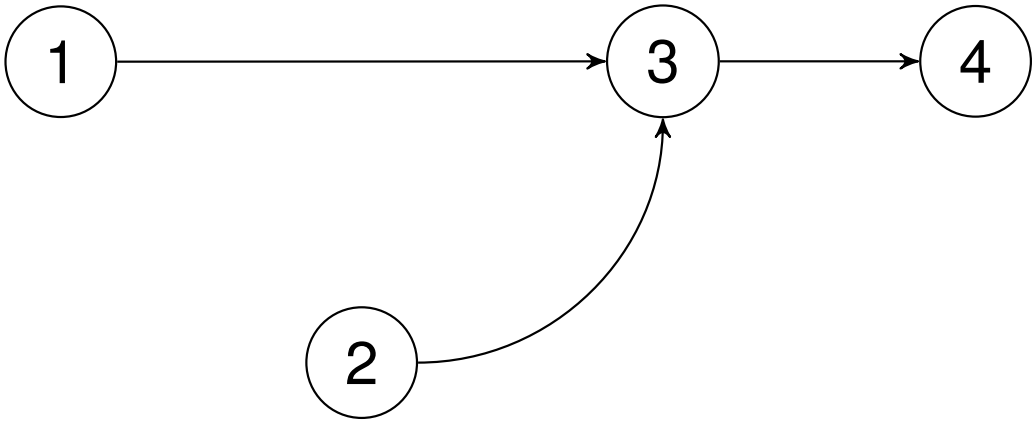
- No good solution for the ABA problem arising from decoupling COMPARE&SET and READ operations.
- AtomicStampedReference<V> is slow due to being implemented as a boxed [reference, integer] pair.
- The problem of adversarial scheduling Fomitchev and Ruppert's algorithm is unlikely to be encountered in practice.
- There would be a significant performance penalty in having two operations on shared memory.
- Perhaps the most important reason...

The Real Reason to Implement Harris's Algorithm

I found it too difficult then to implement Fomitchev and Ruppert's algorithm.

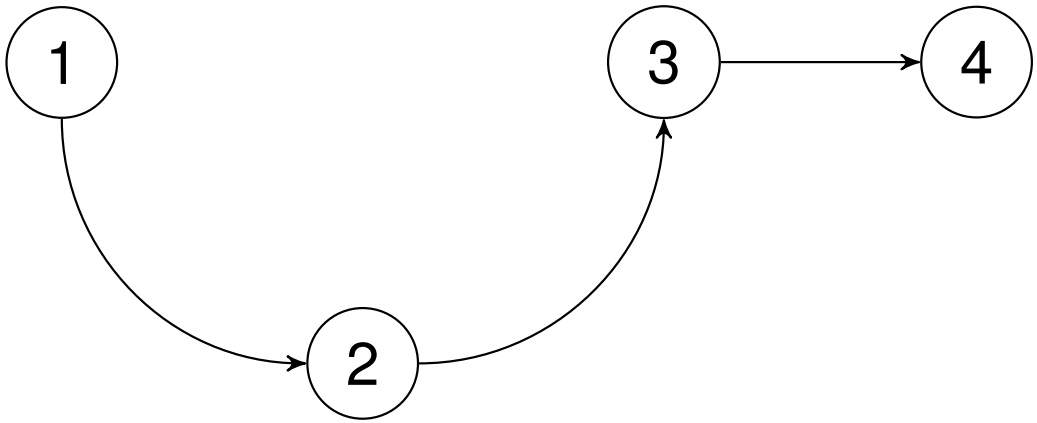
SLL Operations: INSERT

Inserting the node containing 2 into the list $\{1, 3, 4\}$. First, find the appropriate successor for 2 by searching the list from the head.



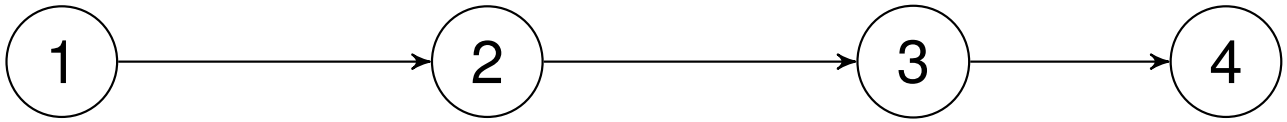
SLL Operations: INSERT

Inserting the node containing 2 into the list $\{1, 3, 4\}$. Next, swing the pointer from the predecessor (1) to the node (2).



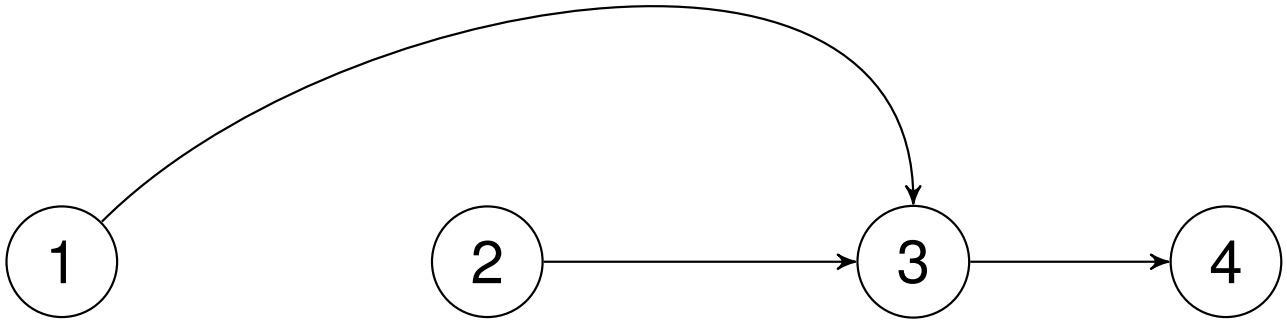
SLL Operations: DELETE

Deleting the node containing 2 from the list $\{1, 2, 3, 4\}$. First, find the node's predecessor by searching the list from the head.



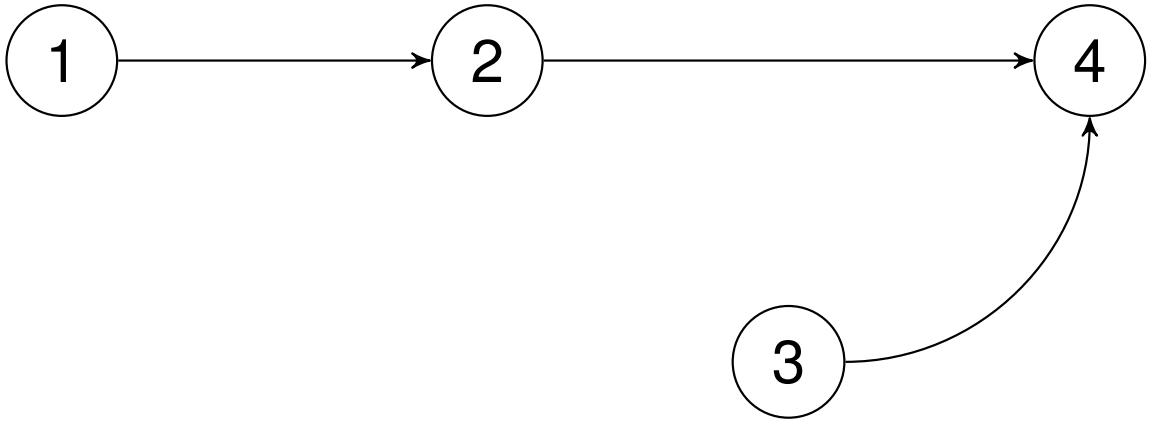
SLL Operations: DELETE

Deleting the node containing 2 from the list $\{1, 2, 3, 4\}$.
Next, swing (2)'s predecessor's pointer to (2)'s successor.



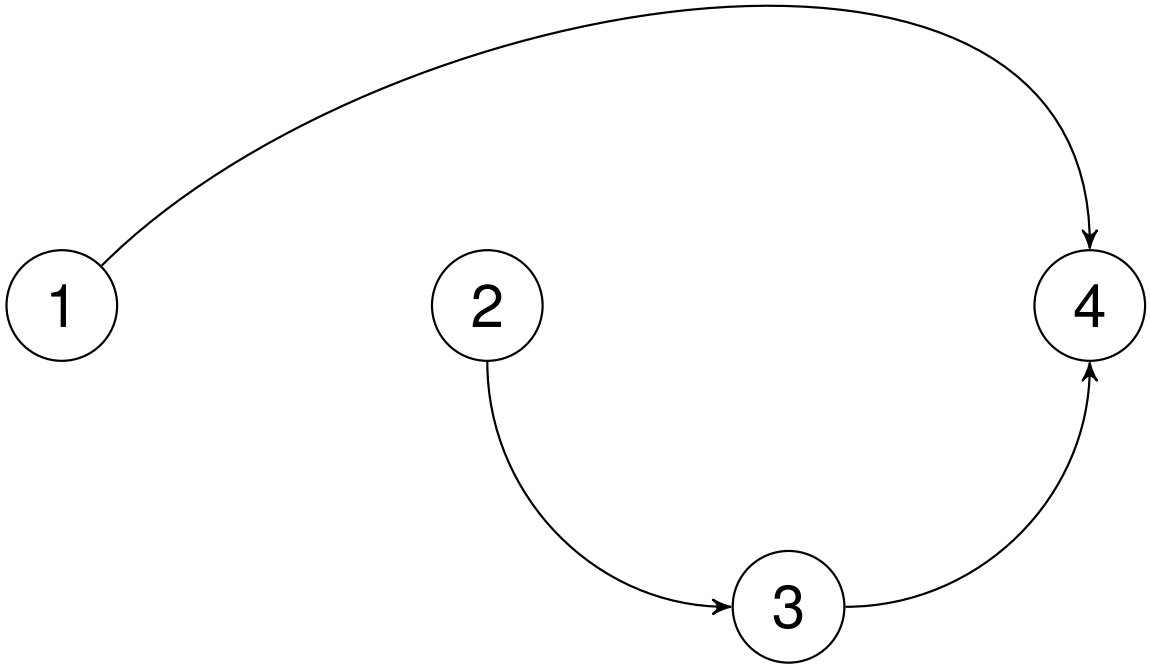
Concurrent INSERT and DELETE

We delete the node (2) and insert the node (3) concurrently into the list $\{1, 2, 4\}$.



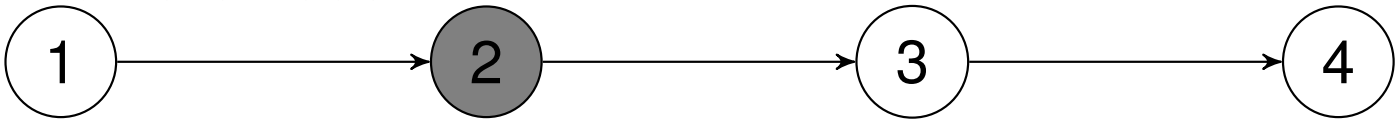
Concurrent INSERT and DELETE

The resulting list is $\{1, 4\}$, rather than the correct $\{1, 3, 4\}$.



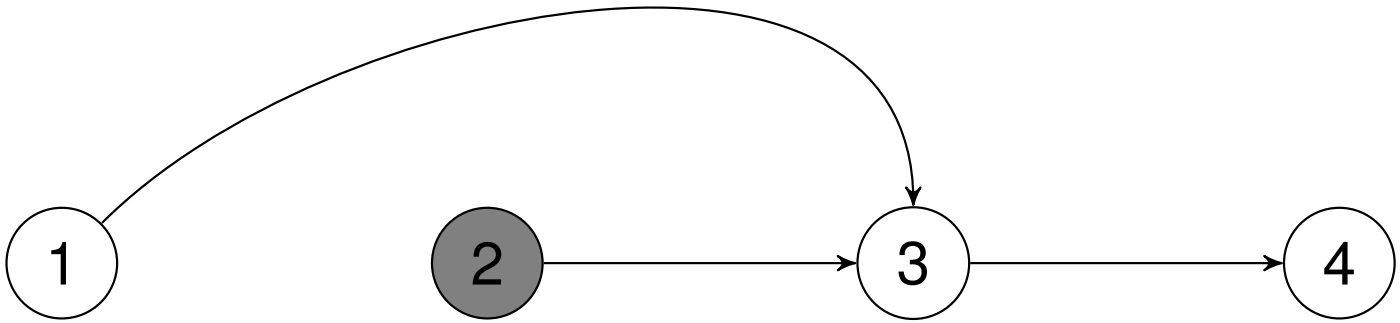
DELETE Procedure

Deleting the node containing 2 from the list $\{1, 2, 3, 4\}$.
Mark the node.



DELETE Procedure

Deleting the node containing 2 from the list $\{1, 2, 3, 4\}$.
Mark the node.



Node and SinglyLinkedList

- The overarching class is SinglyLinkedList and it contains two classes:
- A Node class representing a node in the linked list.
- A Pair class representing a pair of nodes (this is needed by the SEARCH method.)

Node and SinglyLinkedList

- The overarching class is SinglyLinkedList and it contains two classes:
- A Node class representing a node in the linked list.
- A Pair class representing a pair of nodes (this is needed by the SEARCH method.)

Node and SinglyLinkedList

- The overarching class is SinglyLinkedList and it contains two classes:
- A Node class representing a node in the linked list.
- A Pair class representing a pair of nodes (this is needed by the SEARCH method.)

The Node class

- The Node class contains two fields: *key* and *next*.
- The *key* field is of type **int** and represents the key stored in the node. It is a **final** variable.
- The *next* field is of type **AtomicMarkableReference<Node>** and is the pointer to the next node in the list. It is **volatile**.
- In addition, we have the usual constructor, getters and setters.

The Node class

- The Node class contains two fields: *key* and *next*.
- The *key* field is of type **int** and represents the key stored in the node. It is a **final** variable.
- The *next* field is of type **AtomicMarkableReference<Node>** and is the pointer to the next node in the list. It is **volatile**.
- In addition, we have the usual constructor, getters and setters.

The Node class

- The Node class contains two fields: *key* and *next*.
- The *key* field is of type **int** and represents the key stored in the node. It is a **final** variable.
- The *next* field is of type **AtomicMarkableReference<Node>** and is the pointer to the next node in the list. It is **volatile**.
- In addition, we have the usual constructor, getters and setters.

The Node class

- The Node class contains two fields: *key* and *next*.
- The *key* field is of type **int** and represents the key stored in the node. It is a **final** variable.
- The *next* field is of type **AtomicMarkableReference<Node>** and is the pointer to the next node in the list. It is **volatile**.
- In addition, we have the usual constructor, getters and setters.

The FIND method

- The find method is of type **int** $\longrightarrow$ **boolean**
- The method returns true if at some specific point between the invocation of the method and its response, the key *k* it was given as input is in the list.

The FIND method

- The find method is of type **int** $\longrightarrow$ **boolean**
- The method returns true if at some specific point between the invocation of the method and its response, the key k it was given as input is in the list.

The FIND method

- The find method is of type **int** $\longrightarrow$ **boolean**
- The method returns true if at some specific point between the invocation of the method and its response, the key k it was given as input is in the list.

The FIND method


```
public boolean find(int key) {
 Node right_node;

 right_node = search(key).getRight();
 if ((right_node == tail) ||
     (right_node.getKey() != key)) {
  return false;}
 else {return true;}
}
```

The INSERT method

- The insert method is of type **int** $\longrightarrow$ **boolean**
- The method returns true if it succeeds in inserting its key into the list and maintaining the ordering at some specific point between its invocation and response.

The INSERT method

- The insert method is of type **int** $\longrightarrow$ **boolean**
- The method returns true if it succeeds in inserting its key into the list and maintaining the ordering at some specific point between its invocation and response.

The INSERT method

- The insert method is of type **int** $\longrightarrow$ **boolean**
- The method returns true if it succeeds in inserting its key into the list and maintaining the ordering at some specific point between its invocation and response.

The INSERT method


```
public boolean insert(int key) {
Node new_node = new Node(key),
left_node, right_node;
Pair pair;
do {
 pair = search(key);
\\If right_node contains the key,
\\return false.
\\Point new_node's next to right_node
\\Attempt to swing left_node's next to
\\new_node. Break if successful.
 } while(true);
}
```

The INSERT method

- The delete method is of type **int** $\longrightarrow$ **boolean**
- The method returns true if it succeeds in deleting its key into the list and maintaining the ordering at some specific point between its invocation and response.

The INSERT method

- The delete method is of type **int** $\longrightarrow$ **boolean**
- The method returns true if it succeeds in deleting its key into the list and maintaining the ordering at some specific point between its invocation and response.

The INSERT method

- The delete method is of type **int** $\longrightarrow$ **boolean**
- The method returns true if it succeeds in deleting its key into the list and maintaining the ordering at some specific point between its invocation and response.

The INSERT method

- The insert method is of type **int** $\longrightarrow$ **boolean**
- The method returns true if it succeeds in inserting its key into the list and maintaining the ordering at some specific point between its invocation and response.

The INSERT method

- The insert method is of type **int** $\longrightarrow$ **boolean**
- The method returns true if it succeeds in inserting its key into the list and maintaining the ordering at some specific point between its invocation and response.

The INSERT method

- The insert method is of type **int** $\longrightarrow$ **boolean**
- The method returns true if it succeeds in inserting its key into the list and maintaining the ordering at some specific point between its invocation and response.

The DELETE method


```
public boolean delete(int key) {
while(true) {
get the left and right nodes using search.
let right_node_next = right_node.next
Attempt to mark right_node_next.}
Attempt to swing left_node.next to
right_node_next.
}
```

The SEARCH method

- The search method is of type **int** $\longrightarrow$ **boolean**
- The method returns a pair of nodes 'left' and 'right' such that *left_node.next* $=$ *right_node* and *right_node.key* $\leq k$ at some specific point between its invocation and response.

The SEARCH method

- The search method is of type **int** $\longrightarrow$ **boolean**
- The method returns a pair of nodes 'left' and 'right' such that *left_node.next* $=$ *right_node* and *right_node.key* $\leq k$ at some specific point between its invocation and response.

The SEARCH method

- The search method is of type **int** $\longrightarrow$ **boolean**
- The method returns a pair of nodes 'left' and 'right' such that *left_node.next* $=$ *right_node* and *right_node.key* $\leq k$ at some specific point between its invocation and response.

The SEARCH method


```
private Pair search(int key) {
1.Find left_node and right_node.

2.Check that the nodes are physically adjacent.
If so, check that the reference is not marked.
If so, return them; if not, goto 1

3.If the nodes are not physically adjacent
swing left's pointer to right. Check that
the new pointer is not marked, same as 2.
```

Correctness Tests

- I cribbed the RedBlackTreeTest class and modified it to work on SLL's to do preliminary correctness testing.
- All those tests passed on the first try!
- I implemented further ad hoc tests to check correctness by constructing Adder and Deleter threads and observing by their behaviour 'print debugging'.
- Not elegant, but instructive. The expected properties were validated by these tests (e.g. inserting the same element multiple times succeeds only once, inserting an element and deleting it multiple times succeeds only once, etc.)
- However, some oddities emerged...

Correctness Tests

- I cribbed the RedBlackTreeTest class and modified it to work on SLL's to do preliminary correctness testing.
- All those tests passed on the first try!
- I implemented further ad hoc tests to check correctness by constructing Adder and Deleter threads and observing by their behaviour 'print debugging'.
- Not elegant, but instructive. The expected properties were validated by these tests (e.g. inserting the same element multiple times succeeds only once, inserting an element and deleting it multiple times succeeds only once, etc.)
- However, some oddities emerged...

Correctness Tests

- I cribbed the RedBlackTreeTest class and modified it to work on SLL's to do preliminary correctness testing.
- All those tests passed on the first try!
- I implemented further ad hoc tests to check correctness by constructing Adder and Deleter threads and observing by their behaviour 'print debugging'.
- Not elegant, but instructive. The expected properties were validated by these tests (e.g. inserting the same element multiple times succeeds only once, inserting an element and deleting it multiple times succeeds only once, etc.)
- However, some oddities emerged...

Correctness Tests

- I cribbed the RedBlackTreeTest class and modified it to work on SLL's to do preliminary correctness testing.
- All those tests passed on the first try!
- I implemented further ad hoc tests to check correctness by constructing Adder and Deleter threads and observing by their behaviour 'print debugging'.
- Not elegant, but instructive. The expected properties were validated by these tests (e.g. inserting the same element multiple times succeeds only once, inserting an element and deleting it multiple times succeeds only once, etc.)
- However, some oddities emerged...

Correctness Tests

- I cribbed the RedBlackTreeTest class and modified it to work on SLL's to do preliminary correctness testing.
- All those tests passed on the first try!
- I implemented further ad hoc tests to check correctness by constructing Adder and Deleter threads and observing by their behaviour 'print debugging'.
- Not elegant, but instructive. The expected properties were validated by these tests (e.g. inserting the same element multiple times succeeds only once, inserting an element and deleting it multiple times succeeds only once, etc.)
- However, some oddities emerged...

Correctness Tests

- Java's cyclic barrier appears to maintain a large degree of ordering by constructing time of the thread.
- On Add/Delete tests with 2500 of each thread, the same list is nearly always returned.
- This remains the case even when the threads are shuffled.

Correctness Tests

- Java's cyclic barrier appears to maintain a large degree of ordering by constructing time of the thread.
- On Add/Delete tests with 2500 of each thread, the same list is nearly always returned.
- This remains the case even when the threads are shuffled.

Correctness Tests

- Java's cyclic barrier appears to maintain a large degree of ordering by constructing time of the thread.
- On Add/Delete tests with 2500 of each thread, the same list is nearly always returned.
- This remains the case even when the threads are shuffled.

- Try and understand the odd behaviour of these test cases.
- Implement F&R's algorithm (which should now be possible) and compare its performance against Harris's algorithm.

- Try and understand the odd behaviour of these test cases.
- Implement F&R's algorithm (which should now be possible) and compare its performance against Harris's algorithm.

The End

Questions?