Non-Blocking Linked List

Marek Cuchý

marek.cuchy@agents.fel.cvut.cz

B4M36ESW

March 25, 2019

Non-Blocking Stack (LIFO)

```
static class Node<E> {
    final E item;
    Node<E> next;

    public Node(E item) { this.item = item; }
}

AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();

public void push(E item) {
    Node<E> newHead = new Node<E>(item);
    Node<E> oldHead;
    do {
        oldHead = head.get();
        newHead.next = oldHead;
    } while (!head.compareAndSet(oldHead, newHead));
}

public E pop() {
    Node<E> oldHead;
    Node<E> newHead;
    do {
        oldHead = head.get();
        if (oldHead == null)
            return null;
        newHead = oldHead.next;
    } while (!head.compareAndSet(oldHead,newHead));
    return oldHead.item;
```

Linked-List: AtomicReference - Node

```
private static class Node {

    private final int value;
    private final AtomicReference<Node> next;

    public Node(int value, Node next) {
        this.value = value;
        this.next = new AtomicReference<>(next);
    }
}
```

Linked-List: AtomicReference - Add

```
@Override
public boolean add(int value) {
    Node previous, current;
    while (true) {
        previous = head;
        current = previous.next.get();
        while (value > current.value) {
            previous = current;
            current = current.next.get();
        }
        Node newNode = new Node(value, current);
        if (previous.next.compareAndSet(current, newNode)) {
            return true;
        }
    }
}
```

Linked-List: AtomicReference - Delete

```java
@Override
public boolean delete(int value) {
    Node previous, current;
    while (true) {
        previous = head;
        current = previous.next.get();
        while (value > current.value) {
            previous = current;
            current = current.next.get();
        }
        if (current.value == value) {
            if (previous.next.compareAndSet(current, current.next.get())) {
                return true;
            }
        } else {
            return false;
        }
    }
}
```

Linked-List: AtomicReference - Delete

```java
@Override
public boolean delete(int value) {
    Node previous, current;
    while (true) {
        previous = head;
        current = previous.next.get();
        while (value > current.value) {
            previous = current;
            current = current.next.get();
        }
        if (current.value == value) {
            if (previous.next.compareAndSet(current, current.next.get())) {
                return true;
            }
        } else {
            return false;
        }
    }
}
```

- NOT SAFE

Linked-List: AtomicReference - Example

Thread A – delete(a)

```
@Override
public boolean delete(int value) {
    Node previous, current;
    while (true) {
        previous = head;
        current = previous.next.get();
        while (value > current.value) {
            previous = current;
            current = current.next.get();
        }
        if (current.value == value) {
            if (previous.next.compareAndSet(current, current.next.get())) {
                return true;
            }
        } else {
            return false;
        }
    }
}
```

previous: head

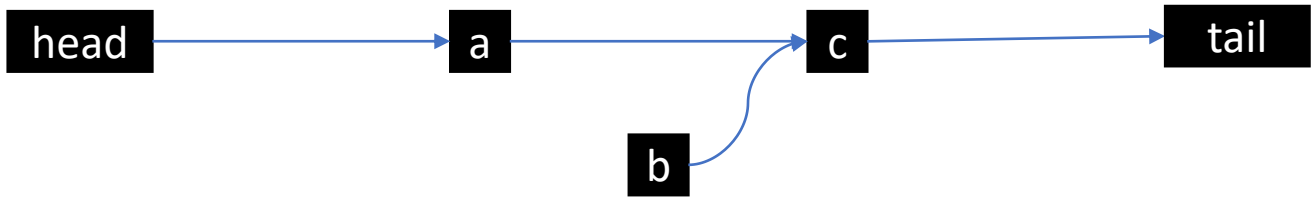
current: a

Thread B – add (b)

```
@Override
public boolean add(int value) {
    Node previous, current;
    while (true) {
        previous = head;
        current = previous.next.get();
        while (value > current.value) {
            previous = current;
            current = current.next.get();
        }
        Node newNode = new Node(value, current);
        if (previous.next.compareAndSet(current, newNode)) {
            return true;
        }
    }
}
```

previous: a

current: c



Linked-List: AtomicReference - Example

Thread A – delete(a)

```
@Override
public boolean delete(int value) {
    Node previous, current;
    while (true) {
        previous = head;
        current = previous.next.get();
        while (value > current.value) {
            previous = current;
            current = current.next.get();
        }
        if (current.value == value) {
            if (previous.next.compareAndSet(current, current.next.get())) {
                return true;
            }
        } else {
            return false;
        }
    }
}
```

previous: head

current: a

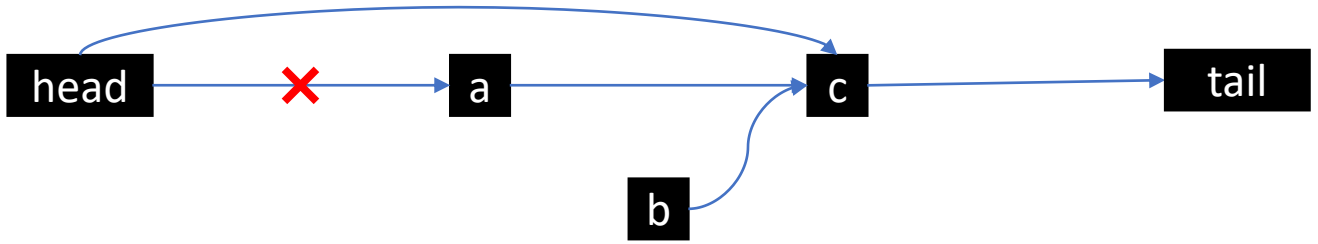
Thread B – add (b)

```
@Override
public boolean add(int value) {
    Node previous, current;
    while (true) {
        previous = head;
        current = previous.next.get();
        while (value > current.value) {
            previous = current;
            current = current.next.get();
        }
        Node newNode = new Node(value, current);
        if (previous.next.compareAndSet(current, newNode)) {
            return true;
        }
    }
}
```

previous: a

current: c

Variable previous still refers to the deleted node



Linked-List: AtomicReference - Example

Thread A – delete(a)

```
@Override
public boolean delete(int value) {
    Node previous, current;
    while (true) {
        previous = head;
        current = previous.next.get();
        while (value > current.value) {
            previous = current;
            current = current.next.get();
        }
        if (current.value == value) {
            if (previous.next.compareAndSet(current, current.next.get())) {
                return true;
            }
        } else {
            return false;
        }
    }
}
```

previous: head

current: a

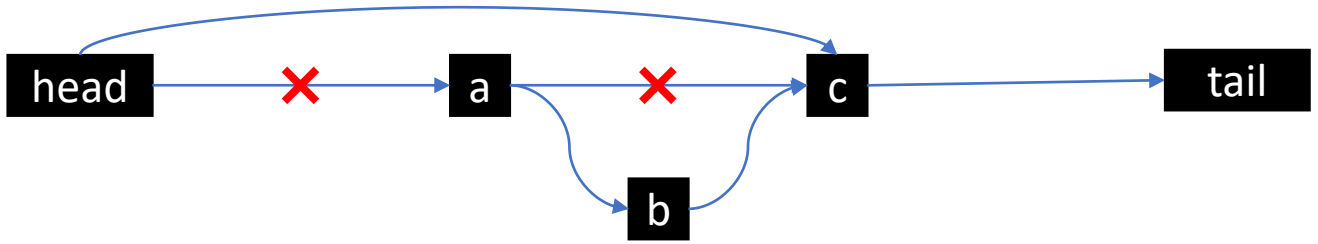
Thread B – add (b)

```
@Override
public boolean add(int value) {
    Node previous, current;
    while (true) {
        previous = head;
        current = previous.next.get();
        while (value > current.value) {
            previous = current;
            current = current.next.get();
        }
        Node newNode = new Node(value, current);
        if (previous.next.compareAndSet(current, newNode)) {
            return true;
        }
    }
}
```

previous: a

current: c

Variable previous still refers to the deleted node



Linked-List: AtomicReference - Example

Thread A – delete(a)

```java
@Override
public boolean delete(int value) {
    Node previous, current;
    while (true) {
        previous = head;
        current = previous.next.get();
        while (value > current.value) {
            previous = current;
            current = current.next.get();
        }
        if (current.value == value) {
            if (previous.next.compareAndSet(current, current.next.get())) {
                return true;
            }
        } else {
            return false;
        }
    }
}
```

previous: head

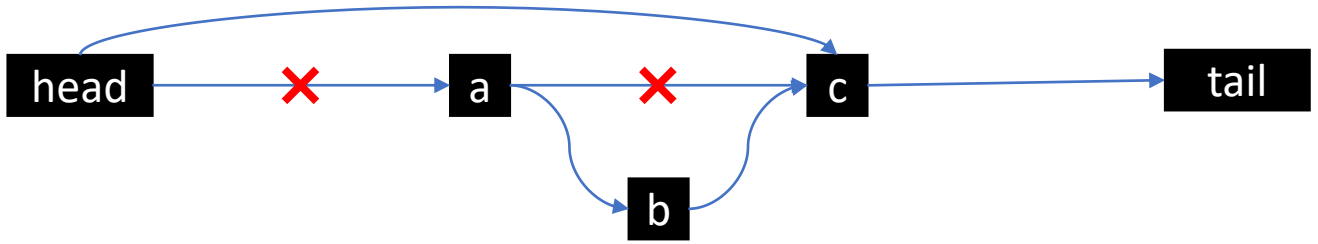
current: a

Thread B – add (b)

```java
@Override
public boolean add(int value) {
    Node previous, current;
    while (true) {
        previous = head;
        current = previous.next.get();
        while (value > current.value) {
            previous = current;
            current = current.next.get();
        }
        Node newNode = new Node(value, current);
        if (previous.next.compareAndSet(current, newNode)) {
            return true;
        }
    }
}
```

previous: a

current: c



SOLUTION: Put a mark on the deleted node first

Linked-List: AtomicMarkableReference - Delete

- Only mark the node – logical delete
- All threads can physically delete the node during traversal

AtomicMarkableReference<T>

- allows atomic operations on <boolean, reference> pair:
  - `boolean compareAndSet(T expectedReference, T newReference, boolean expectedMark, boolean newMark)`
  - `boolean attemptMark(T expectedReference, boolean newMark)`
  - `T get(boolean[] markHolder)` – store current mark to ‚markHolder‘ array and return reference

Linked-List: AtomicMarkableReference - Delete

```java
@Override
public boolean delete(int value) {
    Node[] prevResults = new Node[1], currResults = new Node[1];
    Node prev, curr;
    while(true){
        find(prevResults, currResults, value);
        prev = prevResults[0];
        curr = currResults[0];
        if (curr.value == value){
            Node next = curr.next.getReference();
            boolean marked = curr.next.attemptMark(next, newMark: true);
            if(!marked){
                continue;
            }
            prev.next.compareAndSet(curr, next, expectedMark: false, newMark: false);
            return true;
        }else{
            return false;
        }
    }
}
```

Placeholders for results (workaround for C++ references)

Find the location of the node to be deleted

Try to put mark on the `curr` node

Can be ommitted. Physically deletes the node. No need to check if CAS successfull, because if it returns false another thread already deleted the node

Linked-List: AtomicMarkableReference - Find

```
private void find(Node[] prevResult, Node[] currResult, int value) {
    Node prev, curr, next;
    boolean[] marked = {false};
    retry:
    while (true) {
        prev = head;
        curr = prev.next.getReference();
        while (true) {
            next = curr.next.get(marked);
            // while curr is marked as logically deleted try to delete it physically
            while (marked[0]) {
                boolean deleted = prev.next.compareAndSet(curr, next, expectedMark: false, newMark: false);
                if (!deleted) {
                    //if any other thread marks prev or changes prev.next start over
                    continue retry;
                }
                curr = next;
                next = curr.next.get(marked);
            }
            if (curr.value >= value) {
                prevResult[0] = prev;
                currResult[0] = curr;
                return;
            }
            prev = curr;
            curr = next;
        }
    }

}
```

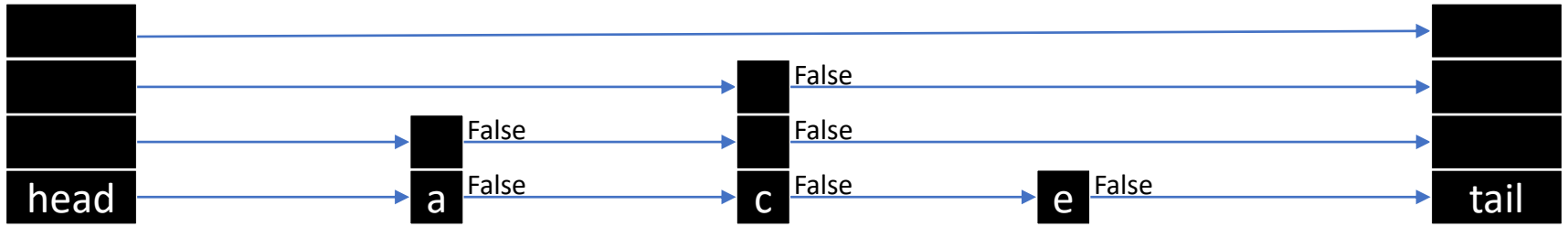
Get next reference and store the mark to ‚marked‘ array

- ‚marked[0]‘ contains the mark of ‚curr‘ node
- Iff ‚curr‘ logically deleted (marked)→marked[0]==true

Try to physically delete marked node

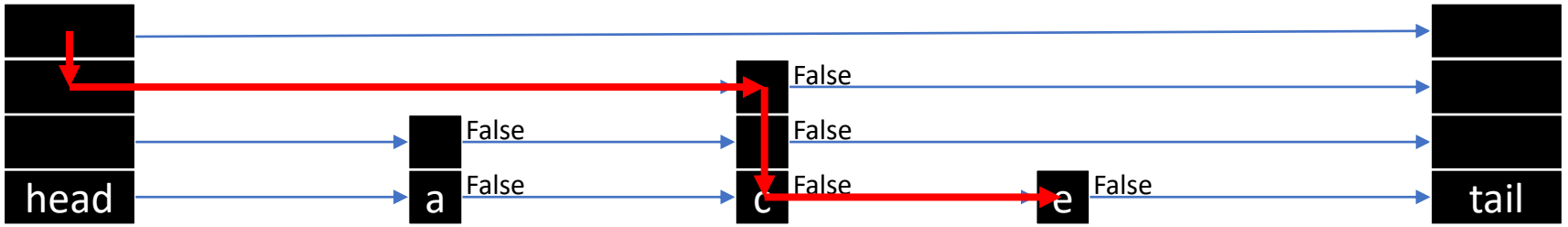
Non-Blocking Skip-List

- Generalization of linked-list
- During insertion/deletion the node is added/removed to/from mutliple-levels sequentially
  - Can take some time to add/remove to/from all levels (thread can be interrupted)
  - →The lowest level is the most important and decisive if a value is contained



Non-Blocking Skip-List – Find(d)

boolean find(Node[] *prevResult*, Node[] *nextResult*, **int** *value*)



Non-Blocking Skip-List – Find(d)

boolean find(Node[] *prevResult*, Node[] *nextResult*, **int** *value*)