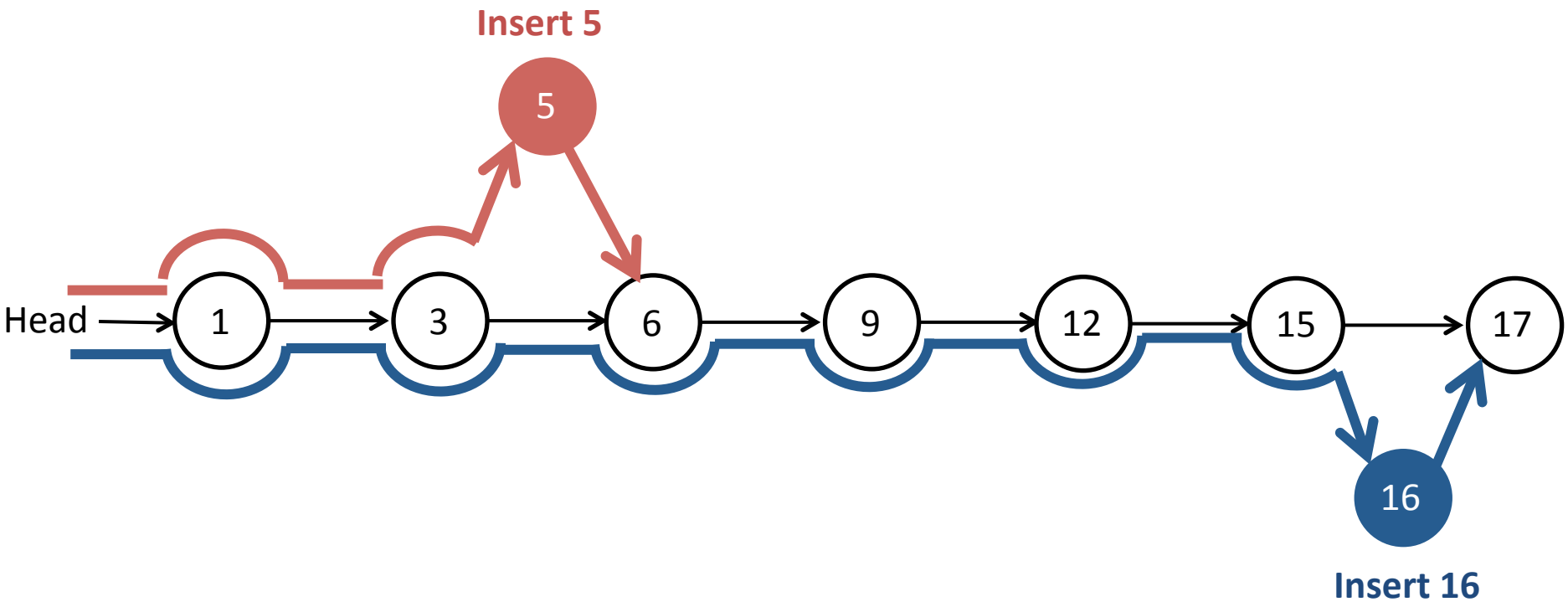
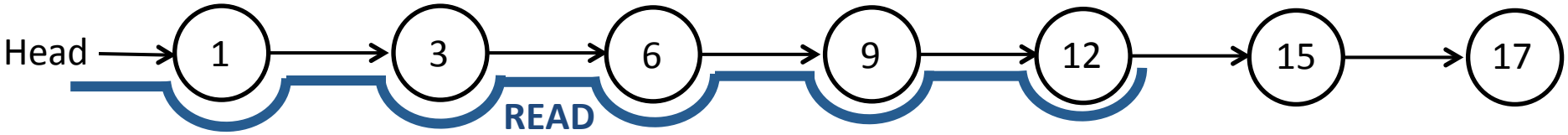
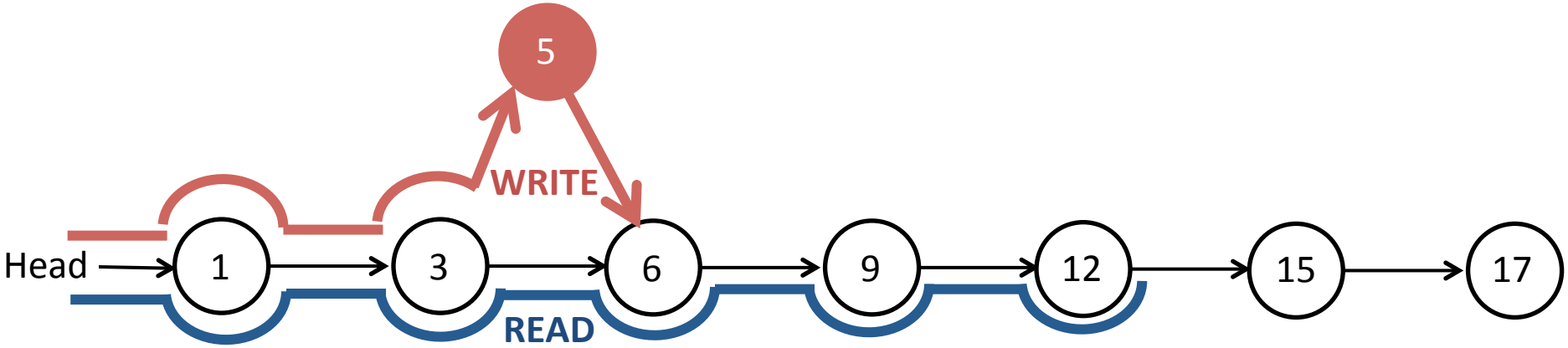
Verifying Transactional Programs with Programmer-Defined Conflict Detection

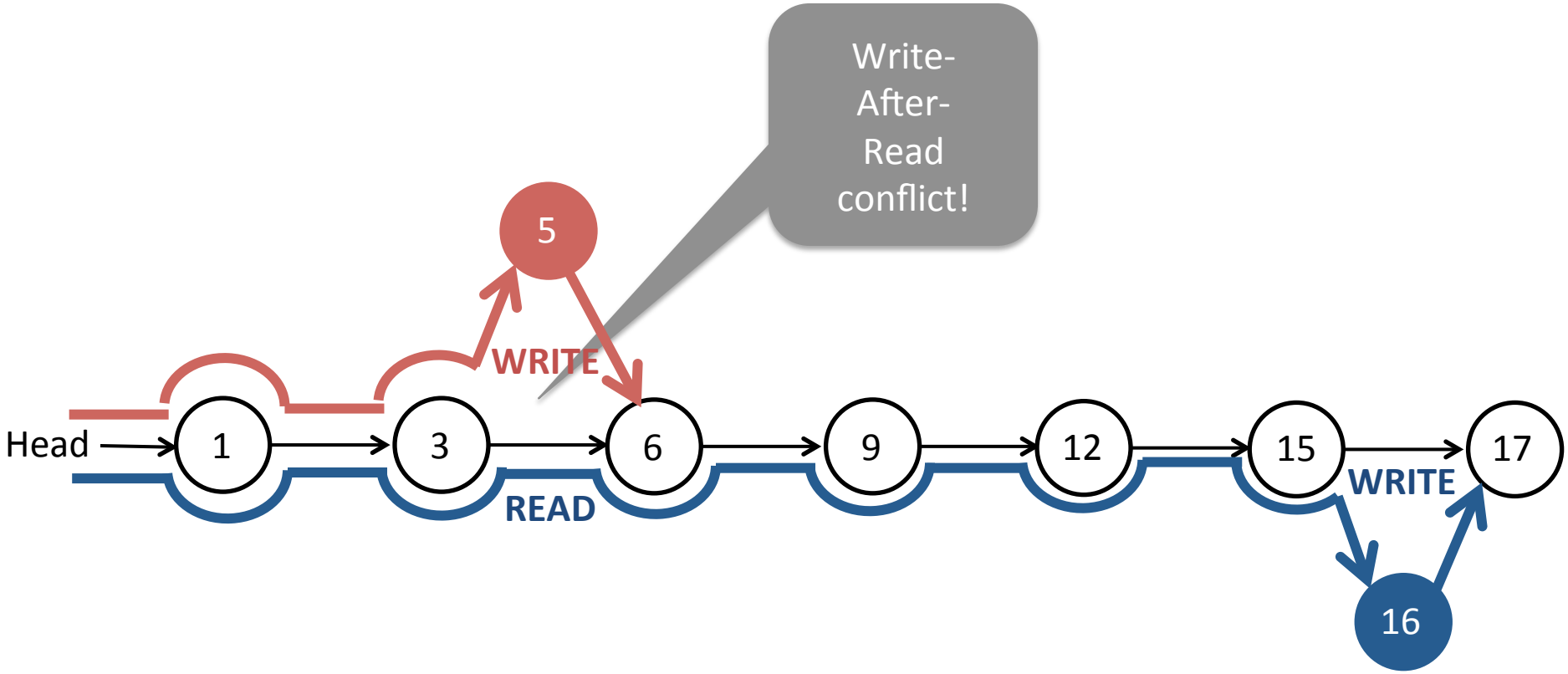
Omer Subasi, **Serdar Tasiran (Koç University)**
Tim Harris (Microsoft Research)
Adrian Cristal, Osman Unsal (Barcelona SC)
Ruben Titos-Gil (University of Murcia)

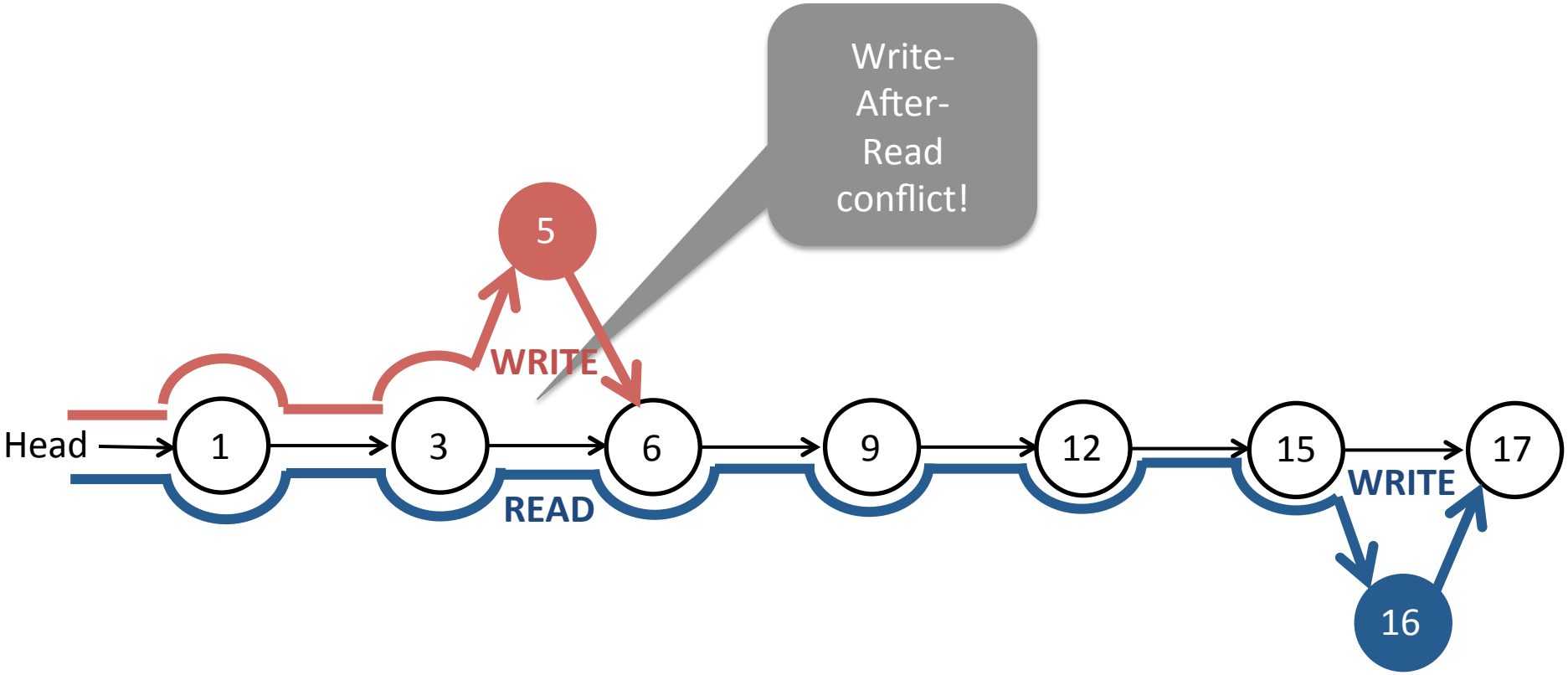
Motivation: Linked List from Genome



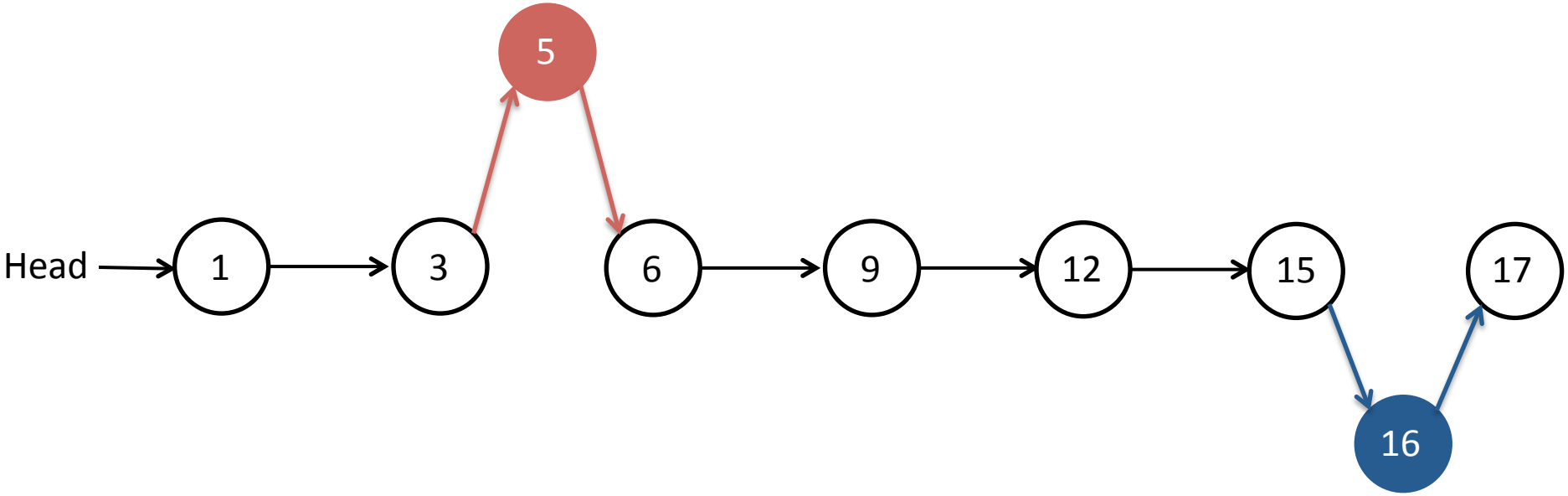

Head
1
3
6
9
12
15
17
READ


5
WRITE
Head
1
3
6
9
12
15
17
READ


Write-
After-
Read
conflict!
5
WRITE
Head
1
3
6
9
12
15
17
READ
WRITE
16



- Conventional TM conflict detection
  - Insertions conflict often



- But, both insertions OK even if we ignore WAR conflict

⇒ Relaxed conflict detection

User-Defined Conflict Detection

- Ignore write-after-read conflicts (Titos et al):

```
atomic[!WAR]{
        insert method body
}
```

Linked List: Insert

```
list_insert(list_t *listPtr, node_t *node) {
    atomic {

        *curr = listPtr->head;
        do {
                prev = curr;
                curr = curr->next;
        }  while (curr != NULL &&
                  curr->key < node->key);

        node->next = curr;
        prev->next = node;
    }
}
```

Linked List: Insert

```
list_insert(list_t *listPtr, node_t *node) {
    atomic {

        *curr = listPtr->head;
        do {
                prev = curr;
                curr = curr->next;
        }  while (curr != NULL &&
                  curr->key < node->key);

        node->next = curr;
        prev->next = node;
        assert(node is in the list && list is sorted);
    }
}
```

Strict conflict detection:

Can reason about transaction code sequentially.

Linked List: Insert

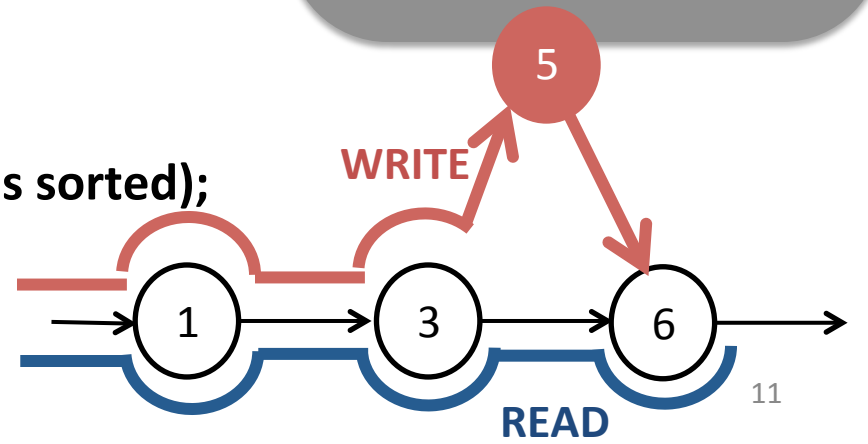
```
list_insert(list_t *listPtr, node_t *node) {
    atomic [!WAR]{

        *curr = listPtr->head;
        do {
            prev = curr;
            curr = curr->next;
        }  while (curr != NULL &&
                  curr->key < node->key);

        node->next = curr;
        prev->next = node;
        assert(node is in the list && list is sorted);
    }
}
```

Ignore Write-After-Read conflicts:

- Writes by others can occur between read phase and write phase
- Lost ability to reason sequentially



Making !WAR block atomic

```
list_insert(list_t *listPtr, node_t *node) {
    atomic [!WAR]{

        *curr = listPtr->head;
        do {
            prev = curr;
            curr = curr->next;
        }  while (curr != NULL &&
             curr->key < node->key);

        node->next = curr;
        prev->next = node;
        assert(node is in the list && list is sorted);
    }
}
```

Would like this action to be "right mover"

- Can commute to the right of any action by another thread

Right Mover

α commutes to the right of β if

α ; β goes to state **S** then

β ; α goes to same state **S**.

If α is right-mover:

$$\alpha ; \gamma \longrightarrow \boxed{\alpha;\gamma}$$

Making !WAR block atomic

```
list_insert(list_t *listPtr, node_t *node) {
    atomic [!WAR]{

        *curr = listPtr->head;
        do {
            prev = curr;
            curr_T1 = curr_T1->next;
        } while (curr != NULL &&
                 curr->key < node->key);

        node->next = curr;
        prev->next = node;
        assert(node is in the list && list is sorted);
    }
}
```

Ignored WAR conflict:

$curr_{T1} = curr_{T1}->next;$

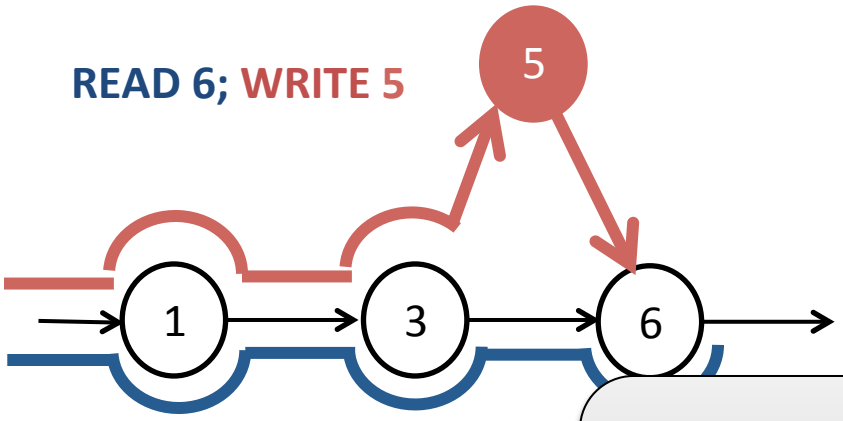
does not move to the right of

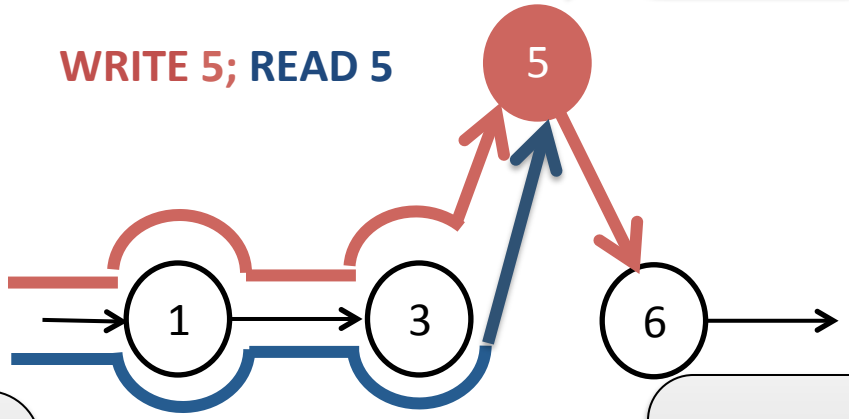
$node_{T2}->next = curr;$

$node_{T2}->next = curr;$

Solution: Abstraction

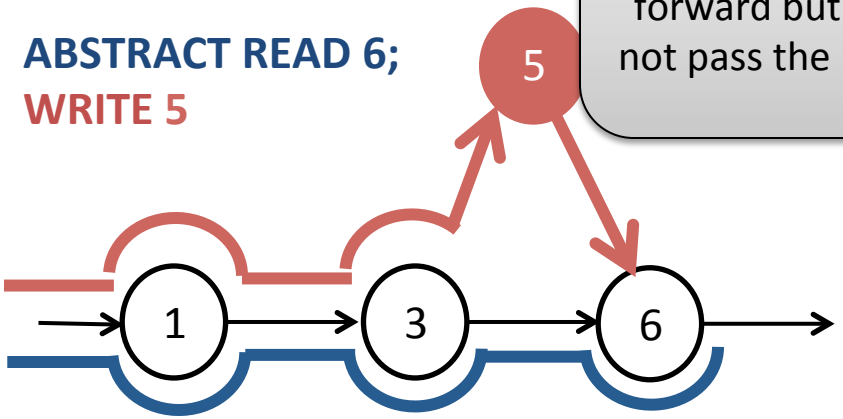
Abstraction intuition

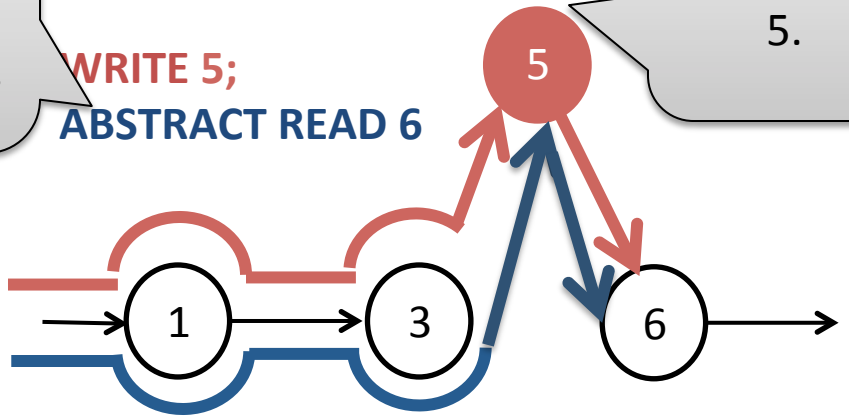




Want to read 6 again!

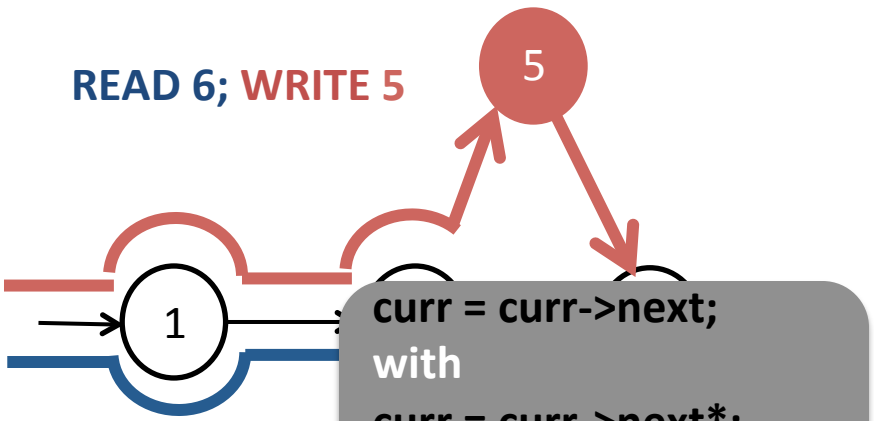
ABSTRACT READ: Read anything forward but do not pass the key.

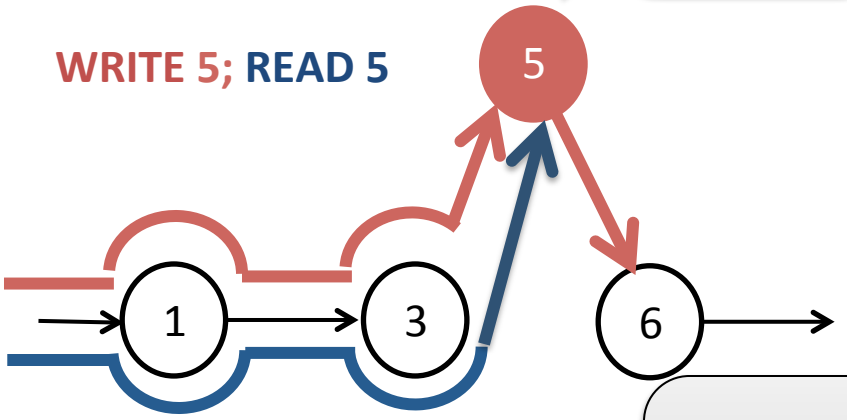


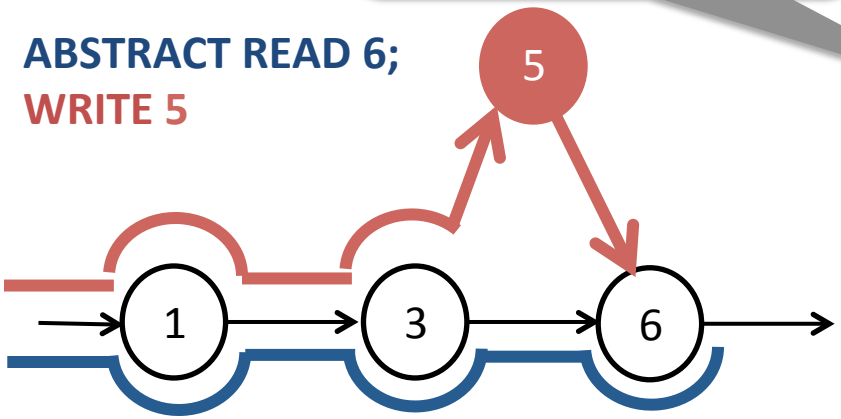


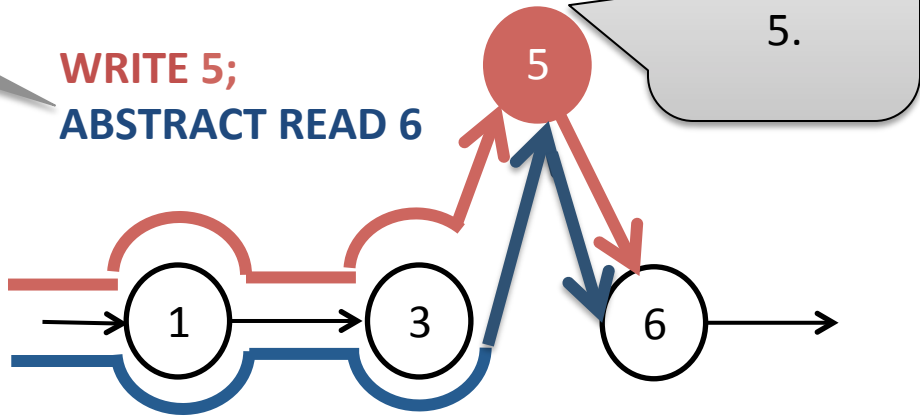
Need to jump over 5.

Abstraction intuition









Abstraction

```
list_insert(list_t *listPtr, node_t *node) {
    atomic [!WAR]{

        *curr = listPtr->head;
        do {
            prev = curr;
            curr_T1 = curr_T1->next*;
        } while (curr != NULL &&
                 curr->key < node->key);

        node->next = curr;
        prev->next = node;
        assert(node is in the list && list is sorted);
    }
}
```

Solution: Abstraction

Replace

$curr_{T1} = curr_{T1}->next;$
with
$curr_{T1} = curr_{T1}->next*;$
but don't go past key.

- Now the block is atomic.
- Can do sequential verification.
- Use a sequential verification tool such as HAVOC, VCC…

1. Sequentially prove the original code

```
list_insert(list_t *listPtr, node_t *node) {

        *curr = listPtr->head;
        do {
                prev = curr;
                curr = curr->next;
        }  while (curr != NULL &&
                  curr->key < node->key);

        node->next = curr;
        prev->next = node;
        assert(node is in the list && list is sorted);
    }
}
```

2. Apply the program transformation

```
list_insert(list_t *listPtr, node_t *node) {

        *curr = listPtr->head;
        do {
                prev = curr;
                curr = curr->next*;
        }  while (curr != NULL &&
                  curr->key < node->key);

        node->next = curr;
        prev->next = node;
        assert(node is in the list && list is sorted);
    }
}
```

Do global read abstractions
➔ Abstract transaction becomes atomic .

3. Prove sequentially properties on abstract code

```
list_insert(list_t *listPtr, node_t *node) {

        *curr = listPtr->head;
        do {
                prev = curr;
                curr = curr->next*;
        }  while (curr != NULL &&
                  curr->key < node->key);

        node->next = curr;
        prev->next = node;
        assert(node is in the list && list is sorted);
    }
}
```

Finally: Soundness theorem says properties hold on original code, with !WAR ignored.

Other proofs

- Labyrinth from STAMP.
- StringBuffer pool.
- Can apply to any program that has the pattern where:
    - a large portion of the shared data is read first
    - local computations is done and then
    - a small portion of shared data is updated