# IT308: Operating Systems

## Lockless List

# Lock-free algorithms

- protecting DS (e.g. BST, linked list) with a single lock is pessimistic as it assumes conflicts will occur resulting in no concurrency
- a lockless algorithm is optimistic as it assumes conflicts unlikely to occur and, when they are detected, they are resolved
  - allows concurrency while there are no conflicts which hopefully is most of the time

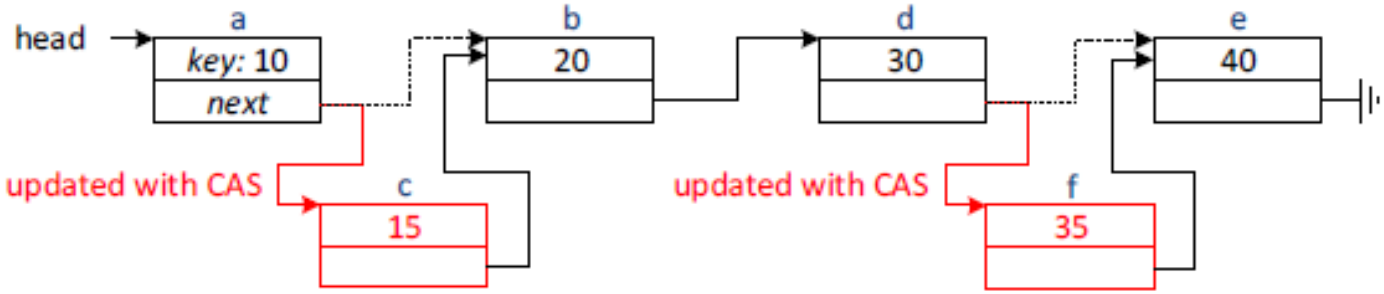
# Atomic Compare-and-Swap (CAS)

```
bool CAS(
     memory location L,
     expected value V at L,
     desired new value V1 at L
);
```

If (the expected value V at memory location L == the current value at L), CAS succeeds by storing the the desired value V1 at L and returns TRUE.

# Using CAS to add nodes

- use CAS to add nodes 15 and 35



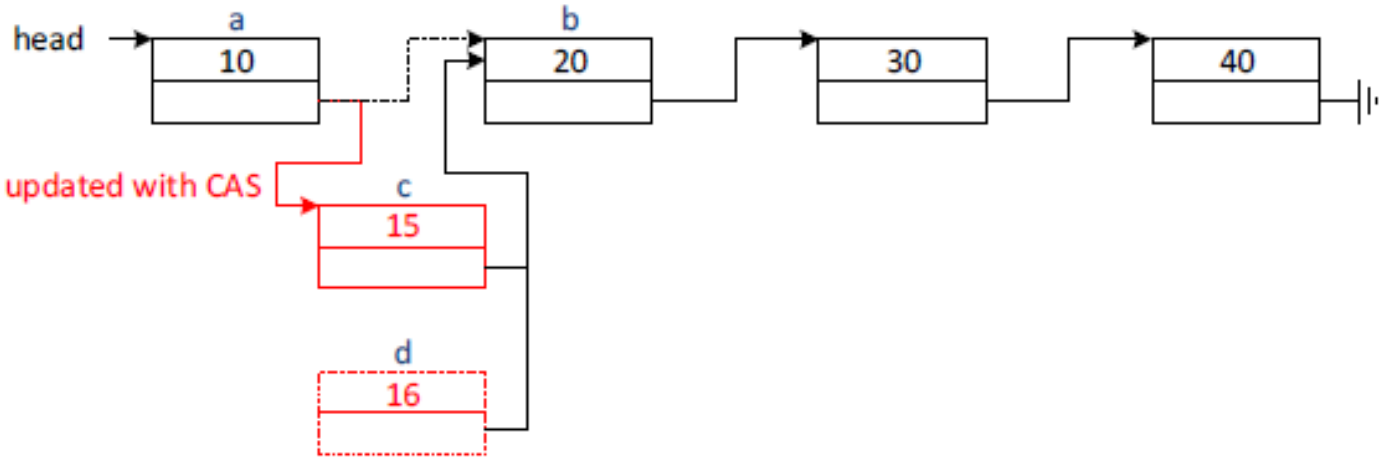

- search for insertion point, initialise next pointer and then execute with correct parameters to insert node into list

```
CAS(&a->next, b, c);     // add node c between a and b
CAS(&d->next, e, f);     // add node f between d and e
```

- disjoint-access parallelism

# Using CAS to add nodes

- if 2 threads try to add nodes at the same position

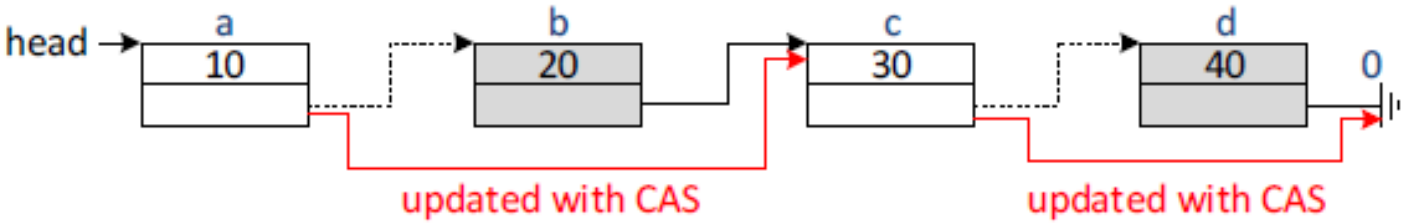



```
CAS(&a->next, b, c); // first CAS executed will succeed..
CAS(&a->next, b, d); // and thus second CAS executed will FAIL
```

- first CAS executed succeeds, second will fail as `a->next != b`
- RETRY on failure, which means searching for insertion point AGAIN and, if key not found, set up and re-execute CAS

# Using CAS to remove nodes

- search for node and then execute CAS with correct parameters to remove node from list
- consider 2 threads removing non-adjacent nodes

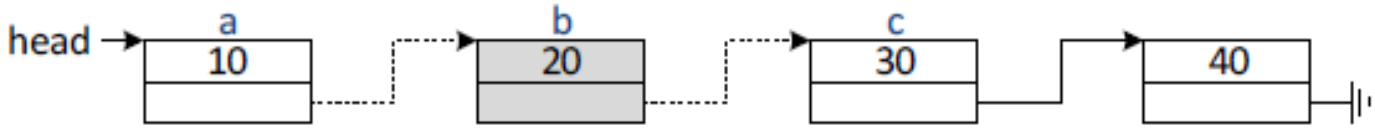



```
CAS(&a->next, b, c); // remove node b (20)
CAS(&c->next, d, 0); // remove node d (40)
```

- disjoint access parallelism

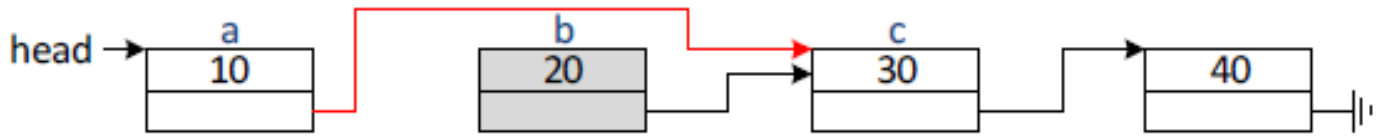
# Using CAS to remove nodes

- if two threads try to remove the same node
- consider 2 threads removing non-adjacent nodes




```
CAS(&a->next, b, c);
CAS(&c->next, b, c);
```

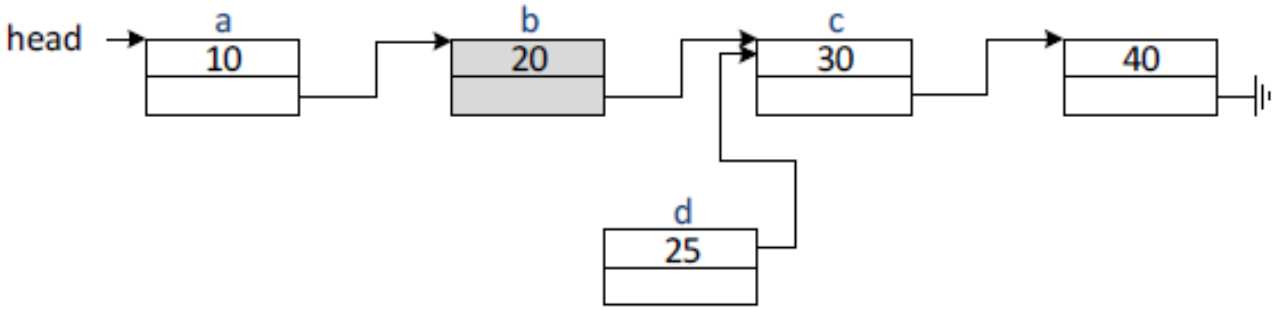
- first CAS executed succeeds



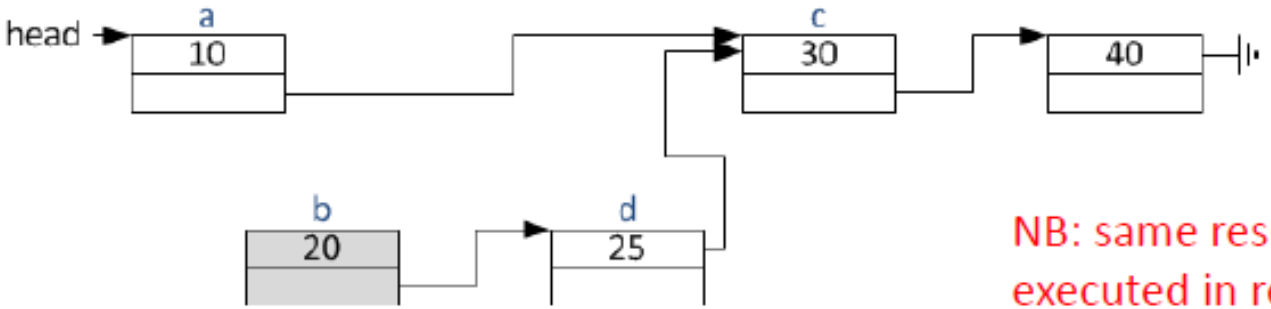

- second CAS executed fails as `a->next != b`
- RETRY on failure, which means searching AGAIN for node (which may not be found)

# What can go wrong with remove?

- imagine removing node 20 and adding node 25 concurrently




```
CAS(&a->next, b, c);  // remove 20
CAS(&b->next, c, d);  // add 25
```

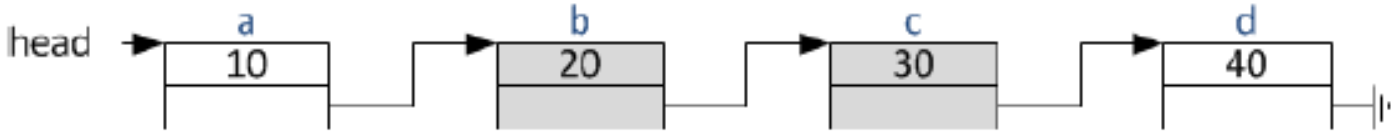



NB: same result if CAS instructions executed in reverse order

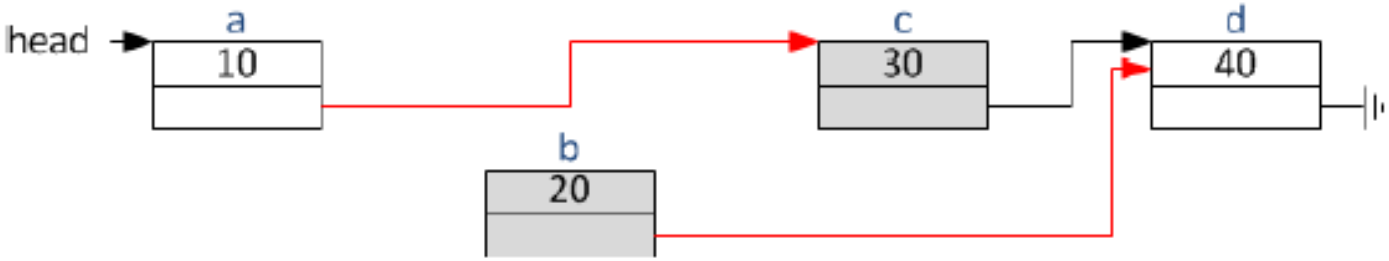
- NOT what was intended!

# What else can go wrong with remove?

- consider deleting adjacent nodes



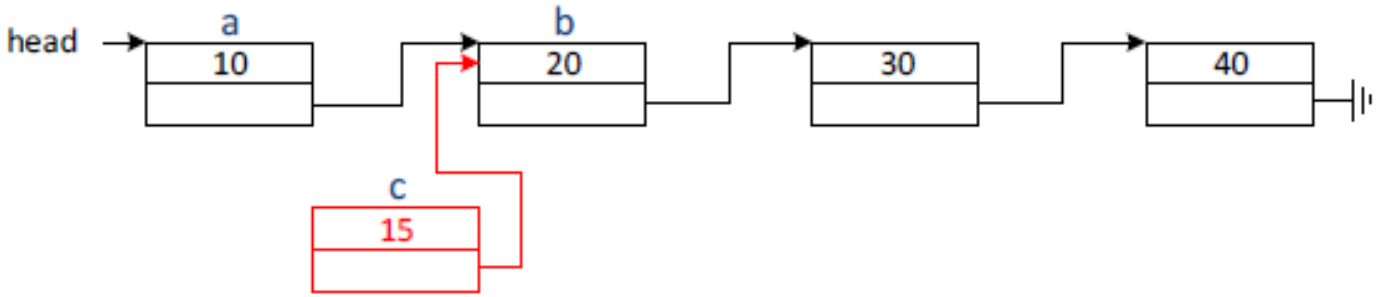

```
CAS(&a->next, b, c);  // remove 20
CAS(&b->next, c, d);  // remove 30
```



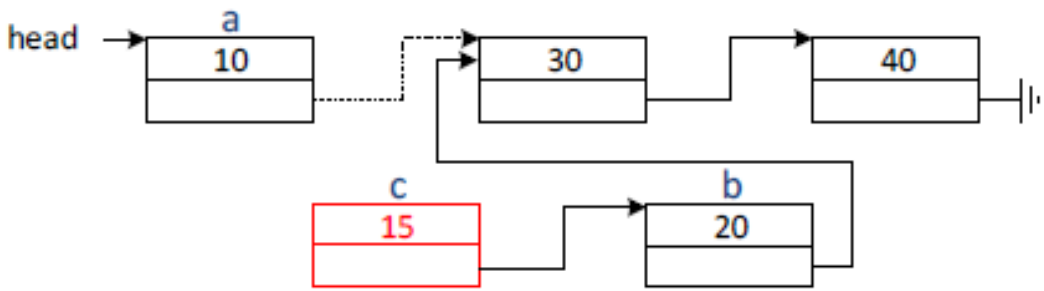

- AGAIN NOT what was intended!

# ABA Problem

- imagine insertion point found, BUT before `CAS(a->next, b, c)` is executed, thread is pre-empted

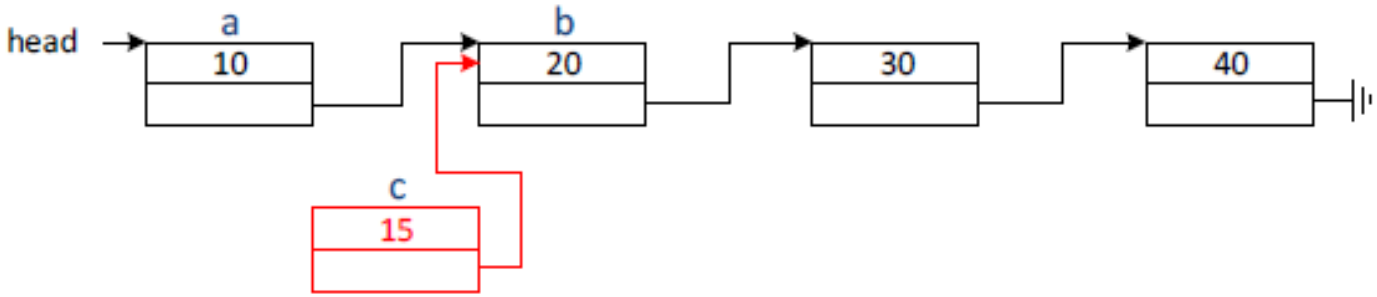



- another thread then removes b from list

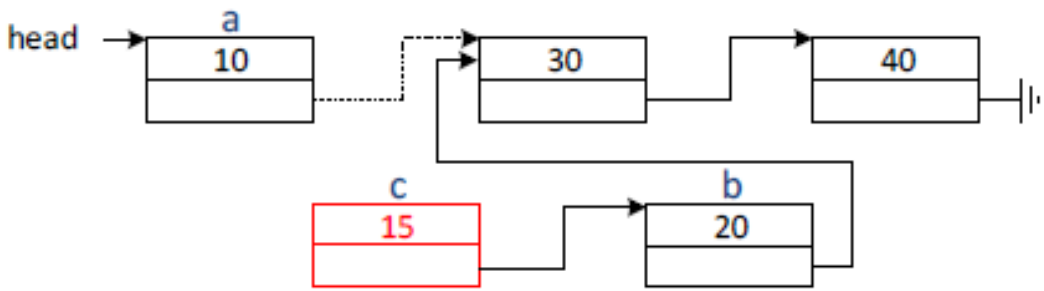


# ABA Problem

- imagine insertion point found, BUT before `CAS(a->next, b, c)` is executed, thread is pre-empted



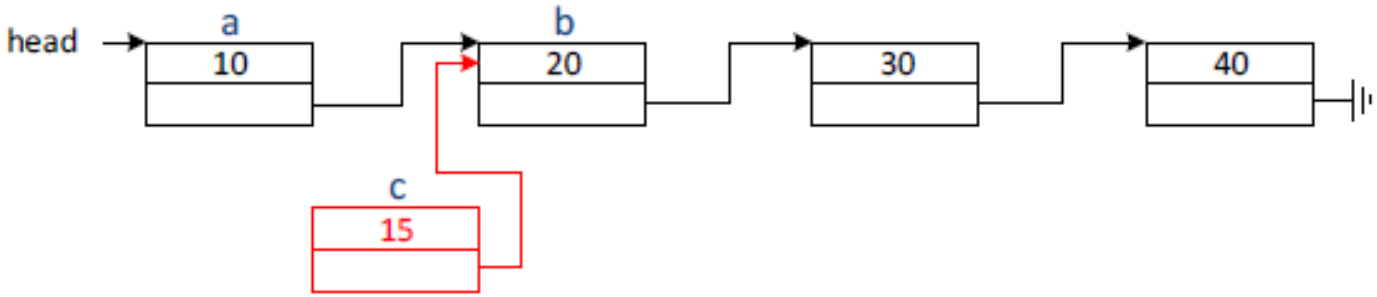

- another thread then removes b from list



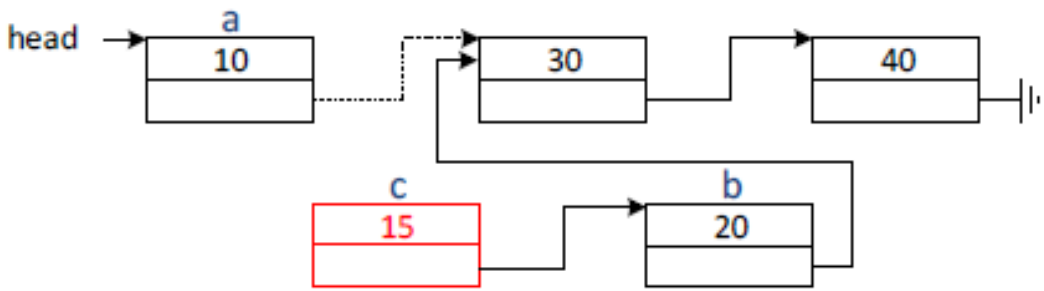

- if thread adding 15 resumes execution, the CAS fails which is OK in this case

# ABA Problem

- imagine insertion point found, BUT before `CAS(a->next, b, c)` is executed, thread is pre-empted



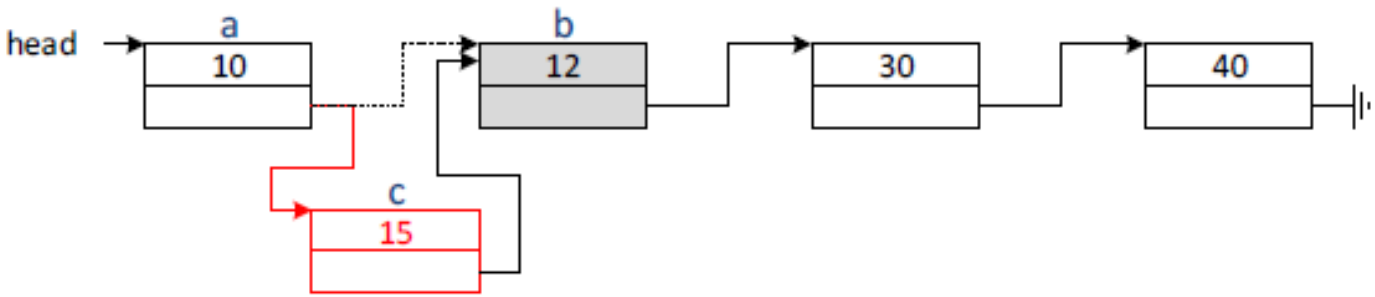

- another thread then removes b from list




- if thread adding 15 resumes execution, the CAS fails which is OK in this case
- BUT what bad thing can happen?

# ABA Problem

- if the memory used by b is reused, for example by a thread adding key 12 to the list before thread adding 15 resumes . . .



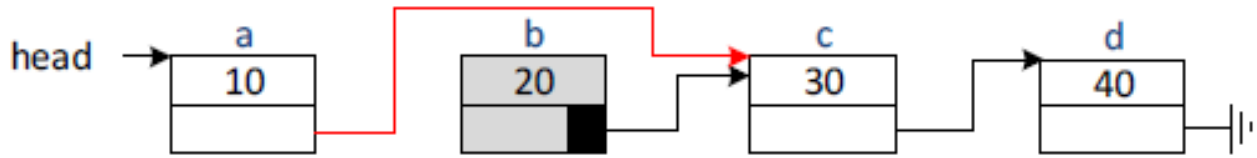

- when the thread adding 15 to list resumes, its CAS will succeed and 15 will be added into the list at the wrong position

# ABA Problem

- ignore the ABA problem by not reusing nodes (will quickly run out of memory)
  - nodes cannot be reused if any thread has or can get a pointer to the node

# Remove a node

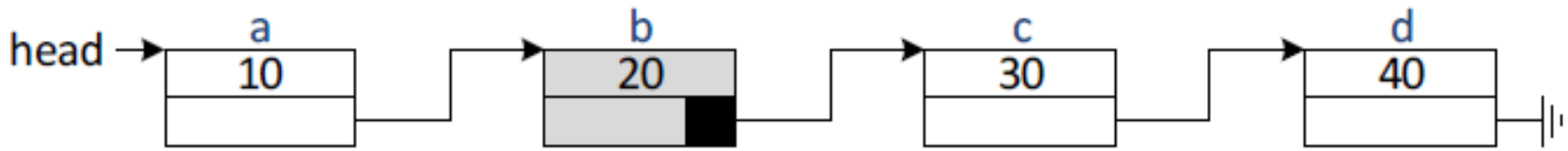
- Use two step removal e.g. remove(20)




  - atomically mark node by setting LSB of next pointer (logically removes node)
  - remove node by updating next pointer using CAS

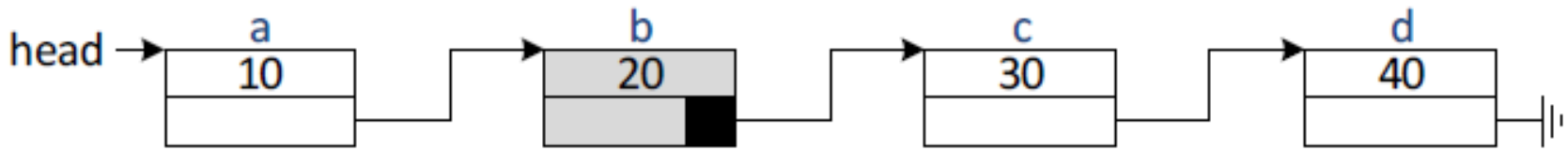
# Marked nodes

- Marked node indicated by an ODD address in its next field
  - OK as addresses normally aligned on at least 4 byte boundary [2 or 3 LSBs normally 0]
- e.g., to atomically mark node b [logically remove]



# Marked nodes

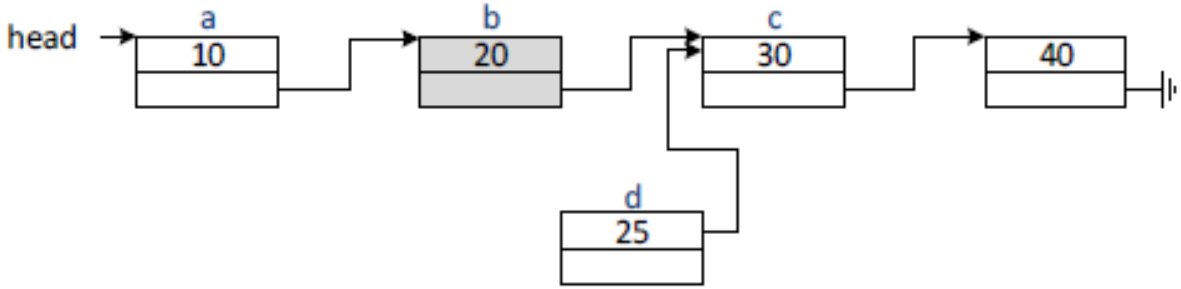
- Marked node indicated by an ODD address in its next field
  - OK as addresses normally aligned on at least 4 byte boundary [2 or 3 LSBs normally 0]
- e.g., to atomically mark node b [logically remove]




```
CAS(&b->next, c, c+1) //assumes node UNMARKED
```

# Revisit adding node and removing node

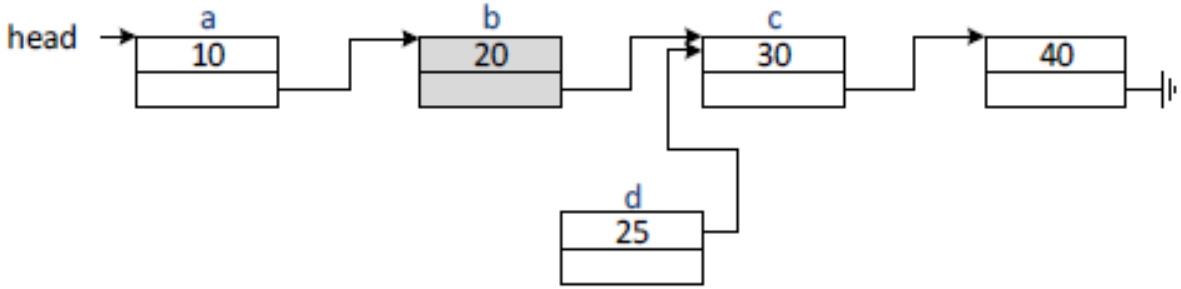
- imagine adding node [25] and removing node [20] concurrently




```
(1) CAS(&b->next, c, d);    // add 25
    and
(2) if (CAS(&b->next, c, c+1) == 1) // MARK node b and then
(3)     CAS(&a->next, b, c);        // remove b [20]
```

# Revisit adding node and removing node

- imagine adding node [25] and removing node [20] concurrently

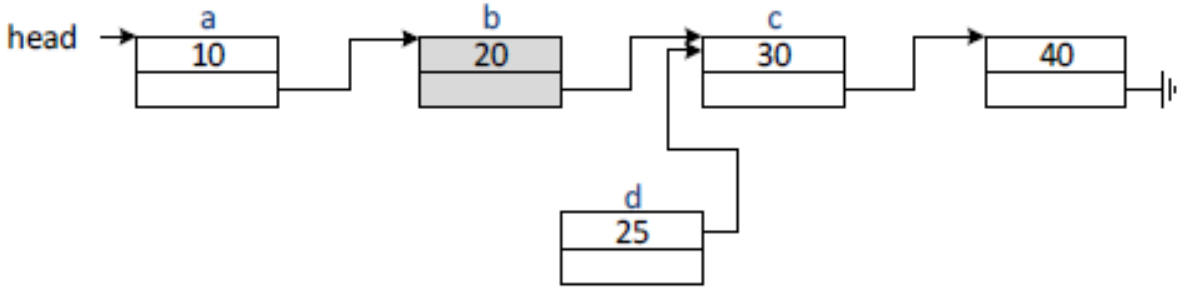



```
(1) CAS(&b->next, c, d);    // add 25
    and
(2) if (CAS(&b->next, c, c+1) == 1) // MARK node b and then
(3)     CAS(&a->next, b, c);        // remove b [20]
```

- if (1) executed first, (2) will fail as `b->next != c`

# Revisit adding node and removing node

- imagine adding node [25] and removing node [20] concurrently

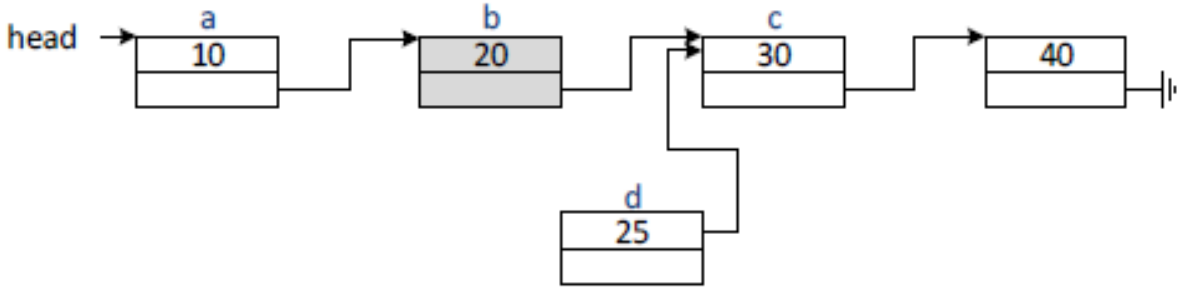



```
(1) CAS(&b->next, c, d);    // add 25
    and
(2) if (CAS(&b->next, c, c+1) == 1) // MARK node b and then
(3)     CAS(&a->next, b, c);        // remove b [20]
```

- if (1) executed first, (2) will fail as `b->next != c`
- if (2) executed first, (1) will fail as `b->next != c`

# Revisit adding node and removing node

- imagine adding node [25] and removing node [20] concurrently




```
(1) CAS(&b->next, c, d);    // add 25
    and
(2) if (CAS(&b->next, c, c+1) == 1) // MARK node b and then
(3)     CAS(&a->next, b, c);        // remove b [20]
```

- if (1) executed first, (2) will fail as `b->next != c`
- if (2) executed first, (1) will fail as `b->next != c`
- if (3) fails, it means that a no longer points to b, BUT b is logically marked and can be removed later
  - OK for list to contain temporary marked nodes

# What still needs to be done?

- Previous solution avoids ABA problem by NOT re-using nodes
- there is no code for freeing or reusing nodes
- Solutions with memory management:
  - A Pragmatic Implementation of Non-Blocking Linked Lists, Tim Harris, 2001
  - Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects, Maged M. Michael, 2004

# Code

- Node class
  - int key
  - Node *next
- List implementated using global variable *head* and functions *add*, *remove* and *find*
  - Node *head
  - int add(Node *head, Node *node)
  - int remove(Mode **head, int key)
  - int find(Node **head, int key)
- per thread local variables
  - Node **prev
  - Node *cur
  - Node *next

# Marked nodes

- Marked node indicated by an ODD address in its next field
- handle marked nodes as follows

```
if (n->next & 1) ... // tests if node n MARKED
CAS(&n->next, v, v+1) // MARK node n (assumes node NOT MARKED)
CAS(&n->next, v, v-1); // UNMARK node n (assumes node MARKED)
```

# find()

```
int find(Node **head, int key) {      // find insertion point or node to remove
retry:                                // NB: thread local variables prev, cur and next
    prev = head;                      // NB: Node **prev, Node **head;
    cur = *prev;                      // *prev and hence cur will be unmarked

    while (cur != NULL) {

        next = cur->next;             // cur unmarked

        if (next & 1) {

            if (CAS(prev, cur, next-1) == 0)
                goto retry;
            cur = next-1;             // cur unmarked

        } else {

            int ckey = cur->key;
            if (*prev != cur)         // optimisation?? will fail sooner?
                goto retry;           // make sure key still in list and no nodes added between prev and cur OTHERWISE retry
            if (ckey >= key)
                return (ckey == key);
            prev = &cur->next;
            cur = next;

        }
    }
    return 0;
}
```

continue until end of list

test if marked node

try to remove marked node

move to next node

make copy of key

check that *prev == cur

return 1 if key found, 0 otherwise

move to next node

# find()

```
int add(Node **head, Node *node)  {
    key = node->key;              // NB: thread local variables prev, cur and next
    while (1) {
        if (find(head, key))
            return 0;
        node->next = cur;
        if (CAS(prev, cur, node) == 1)
            return 1;
    }
}
```

return 0 if key already in list

set up new node's next pointer

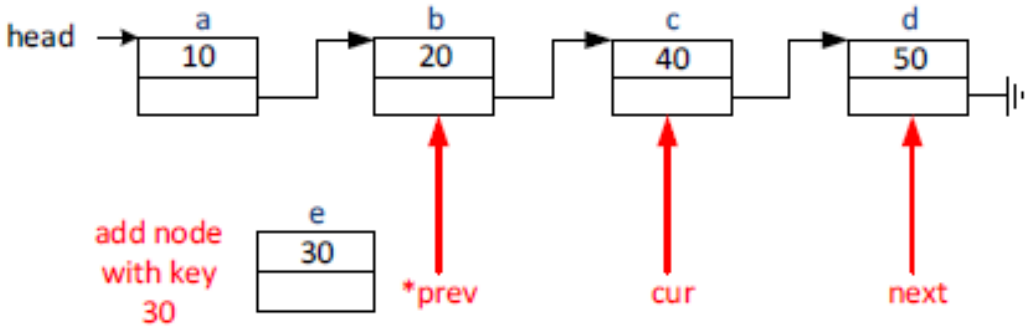
add node between prev and cur

returns 1 on success

keep trying until successful

# add()

- add(key) calls the find() function




NB: Node **prev;
Node *cur, *next;

- find() returns thread local pointers such that the new node should be added between *prev and cur
- if CAS(prev, cur, node) succeeds, it must mean that prev still pointed to cur [nodes have not been added between prev and cur]
- a node CANNOT be added by linking to a MARKED node [logically removed] thus avoid the problem discussed earlier

# remove()

```
int remove(Node ** head, int key) {
    while (1) {                    // NB: thread local variables prev, cur and next

        if (find(head, key) == 0)      // cur and prev will be unmarked
            return 0;

        if (CAS(&cur->next, next, next+1) == 0)
                continue;

        if (CAS(prev, cur, next) == 0)
            find(head, key);

        return 1;

    }
}
```

keep trying until successful

returns 1 if key found
returns prev, cur and next
curr Node removed

return 0 if key not in list

try to MARK UNMARKED node
once marked node is logically removed

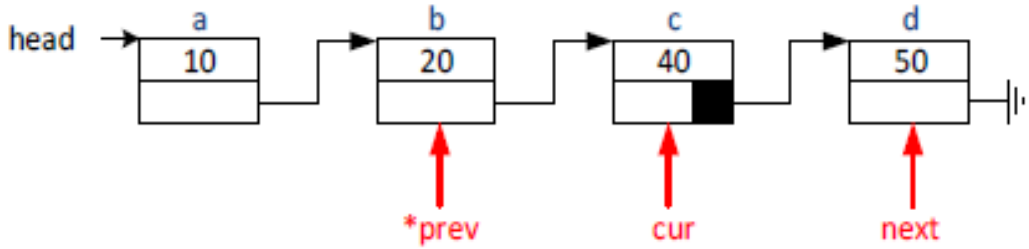
try to remove node from list

if CAS fails, use find() to remove marked node from list

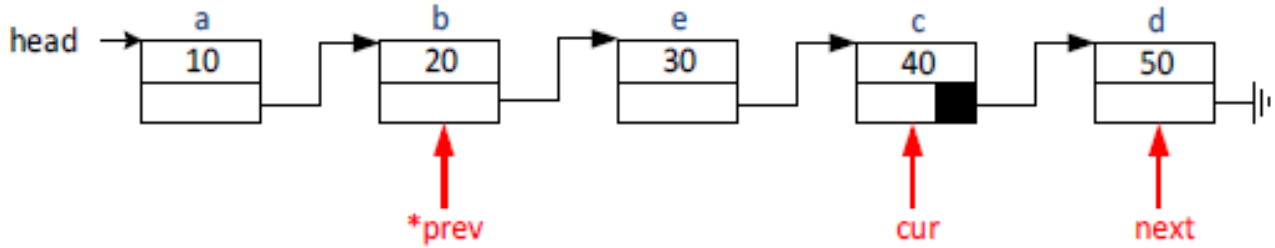
- calls find to remove marked node if CAS fails AND if find fails to remove the marked node, it can be removed by furture calls to find (in add and remove)

# remove()

- assume initial search has returned *prev, cur and next AND cur has been MARKED [logically removed]




- imagine that before CAS(prev, cur, next) is executed to remove node, another thread inserted a node between prev and cur




- CAS(prev, cur, next) will FAIL