Decoupling Lock-Free Data Structures from Memory Reclamation for Static Analysis

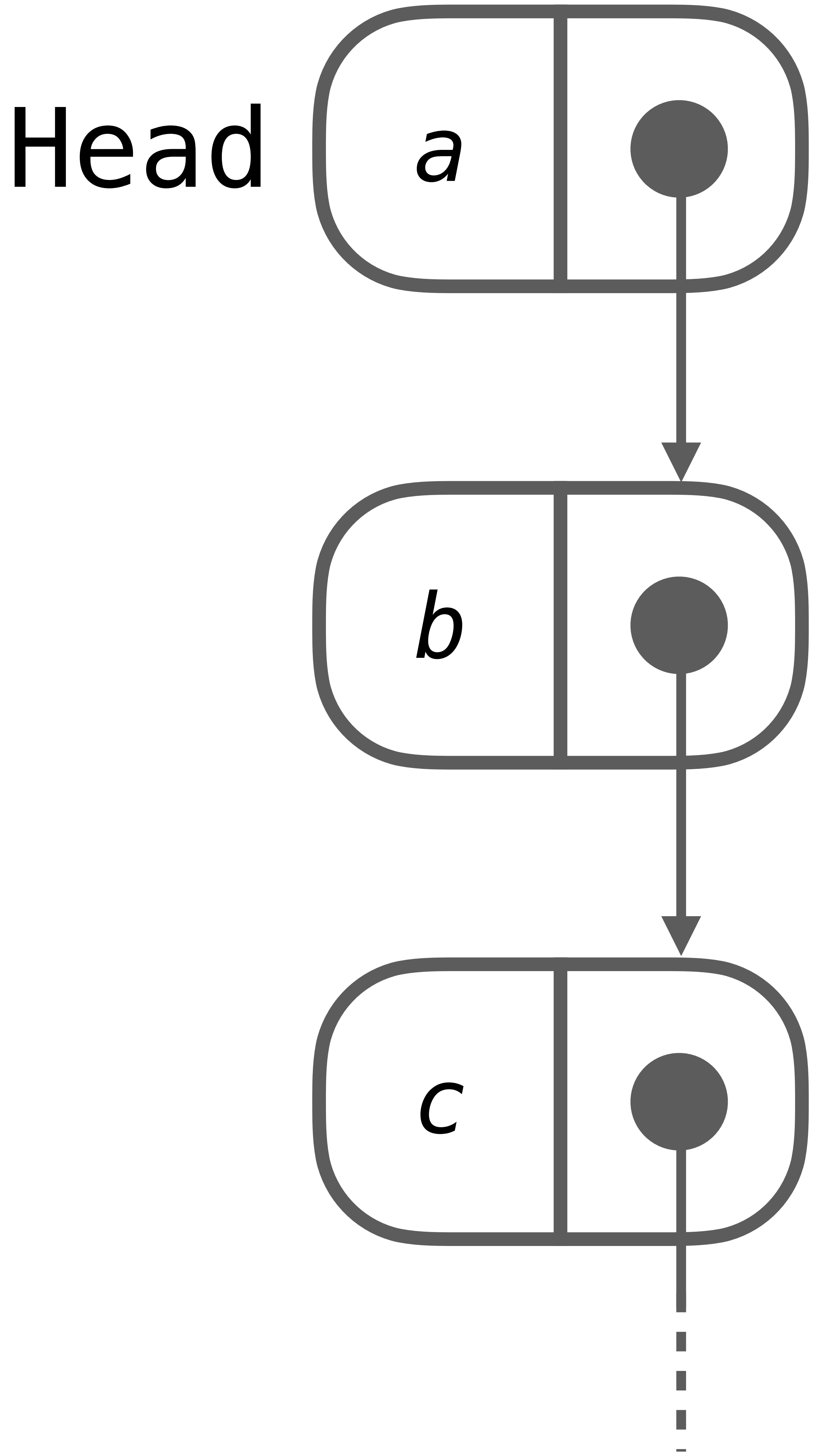
[POPL'19]

Roland Meyer and Sebastian Wolff

TU Braunschweig, Germany

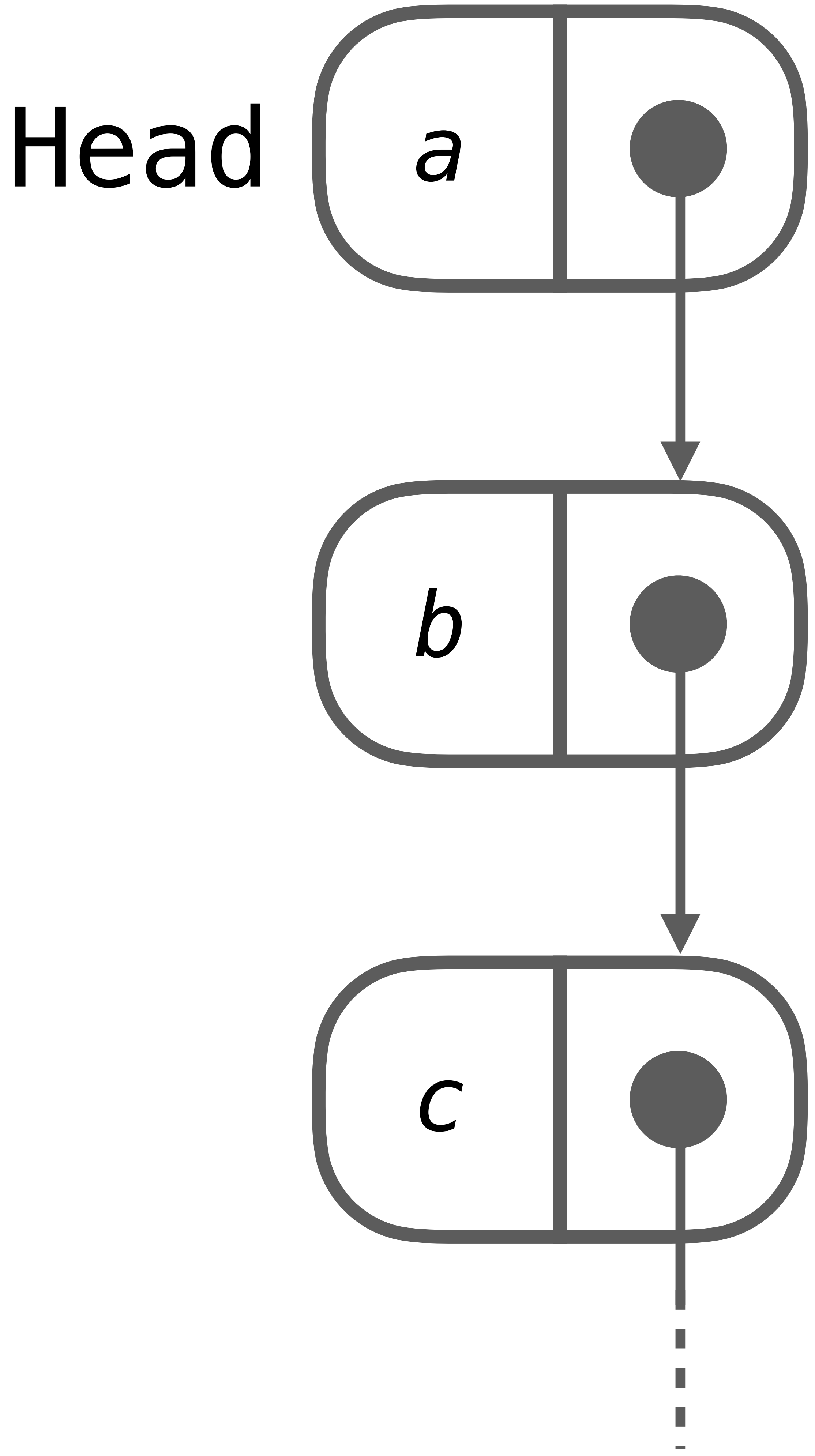
Lock-free Queue (Michael&Scott)

```
void dequeue() {
  while (true) {
    head = Head;
    next = head->next;
    // ...
    if(CAS(Head,head,next)){
      // leak head?
      return;
}}}
```



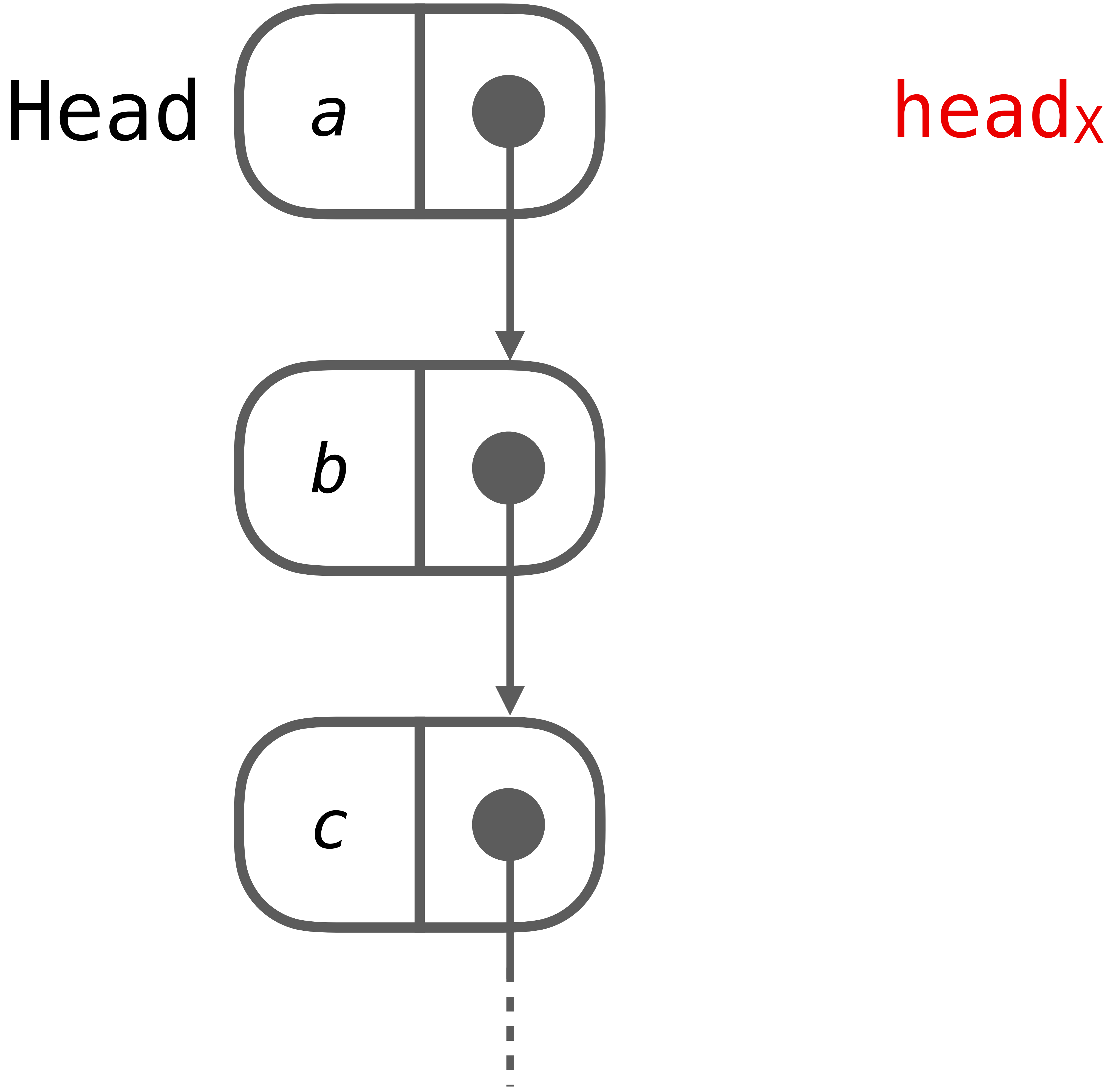
Lock-free Queue (Michael&Scott)

```
   void dequeue() {
     while (true) {
①②Ⓧ   head = Head;
       next = head->next;
       // ...
       if(CAS(Head,head,next)){
         // leak head?
         return;
   }}}
```



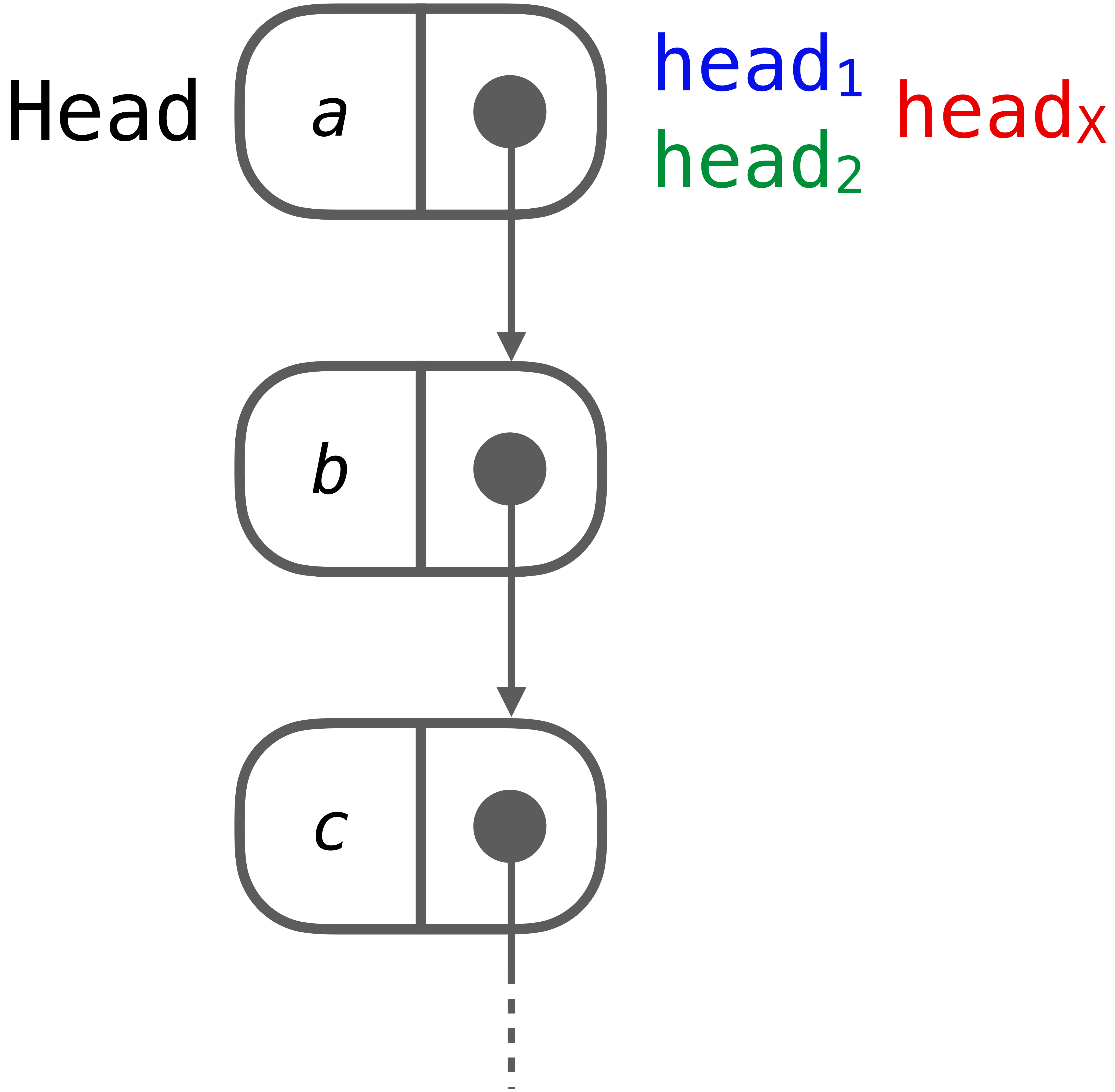
Lock-free Queue (Michael&Scott)

```
     void dequeue() {
       while (true) {
①②       head = Head;
 Ⓧ       next = head->next;
         // ...
         if(CAS(Head,head,next)){
           // leak head?
           return;
     }}}
```



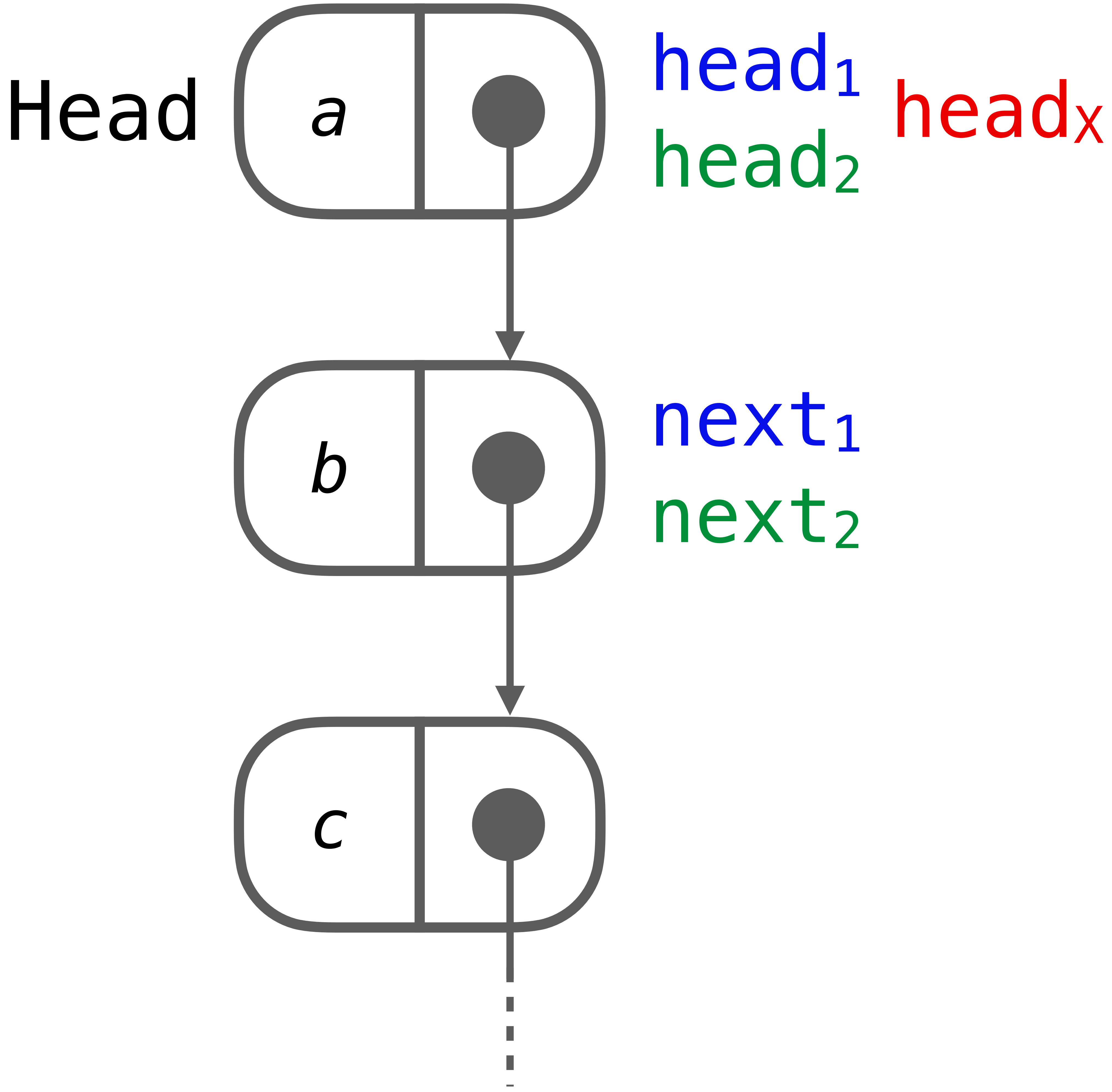
Lock-free Queue (Michael&Scott)

```
void dequeue() {
 while (true) {
  head = Head;
  next = head->next;   1 2 X
  // ...
  if(CAS(Head,head,next)){
   // leak head?
   return;
}}}
```



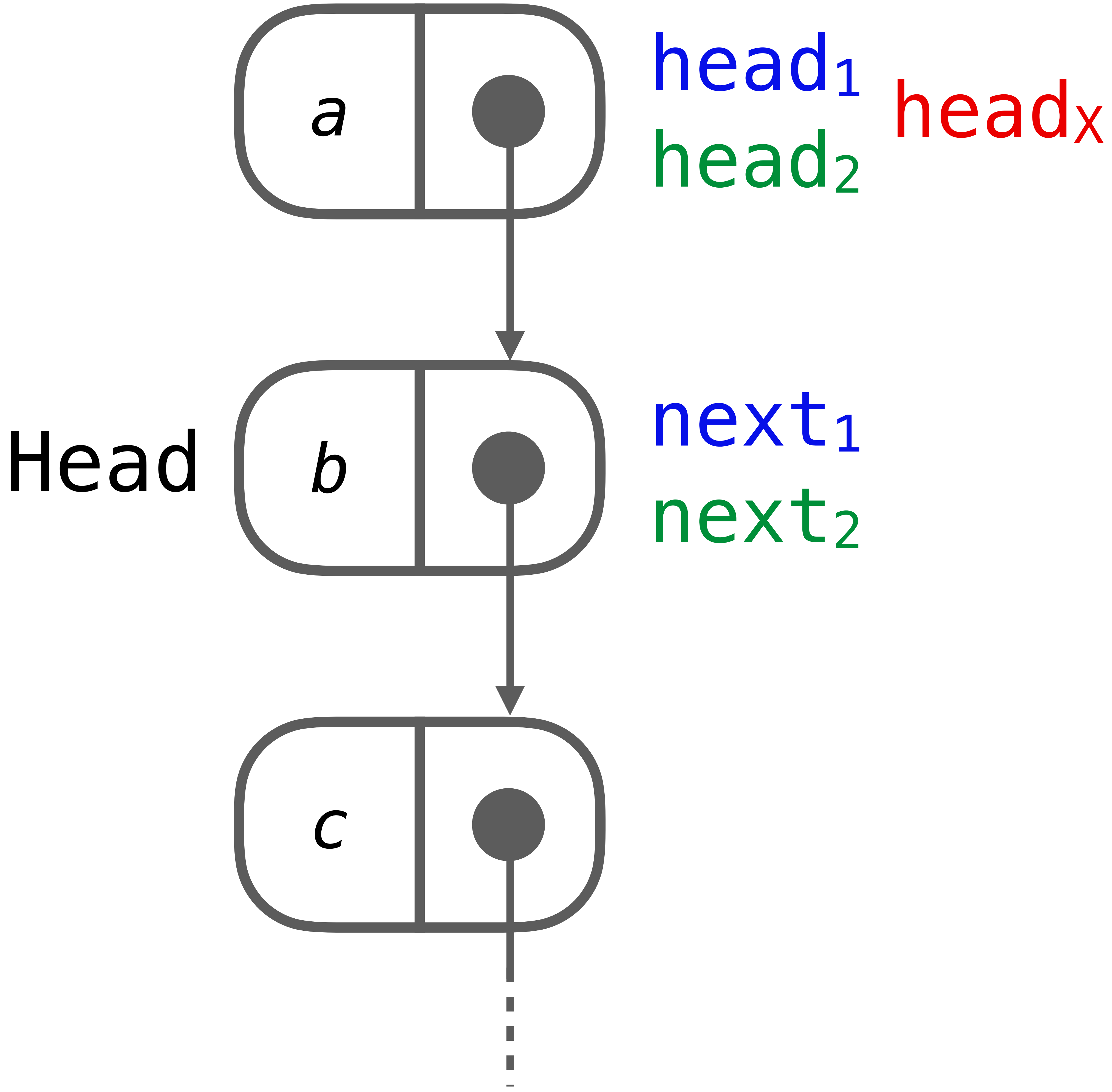
Lock-free Queue (Michael&Scott)

```
   void dequeue() {
     while (true) {
       head = Head;
X      next = head->next;
       // ...
1 2    if(CAS(Head,head,next)){
         // leak head?
         return;
   }}}
```



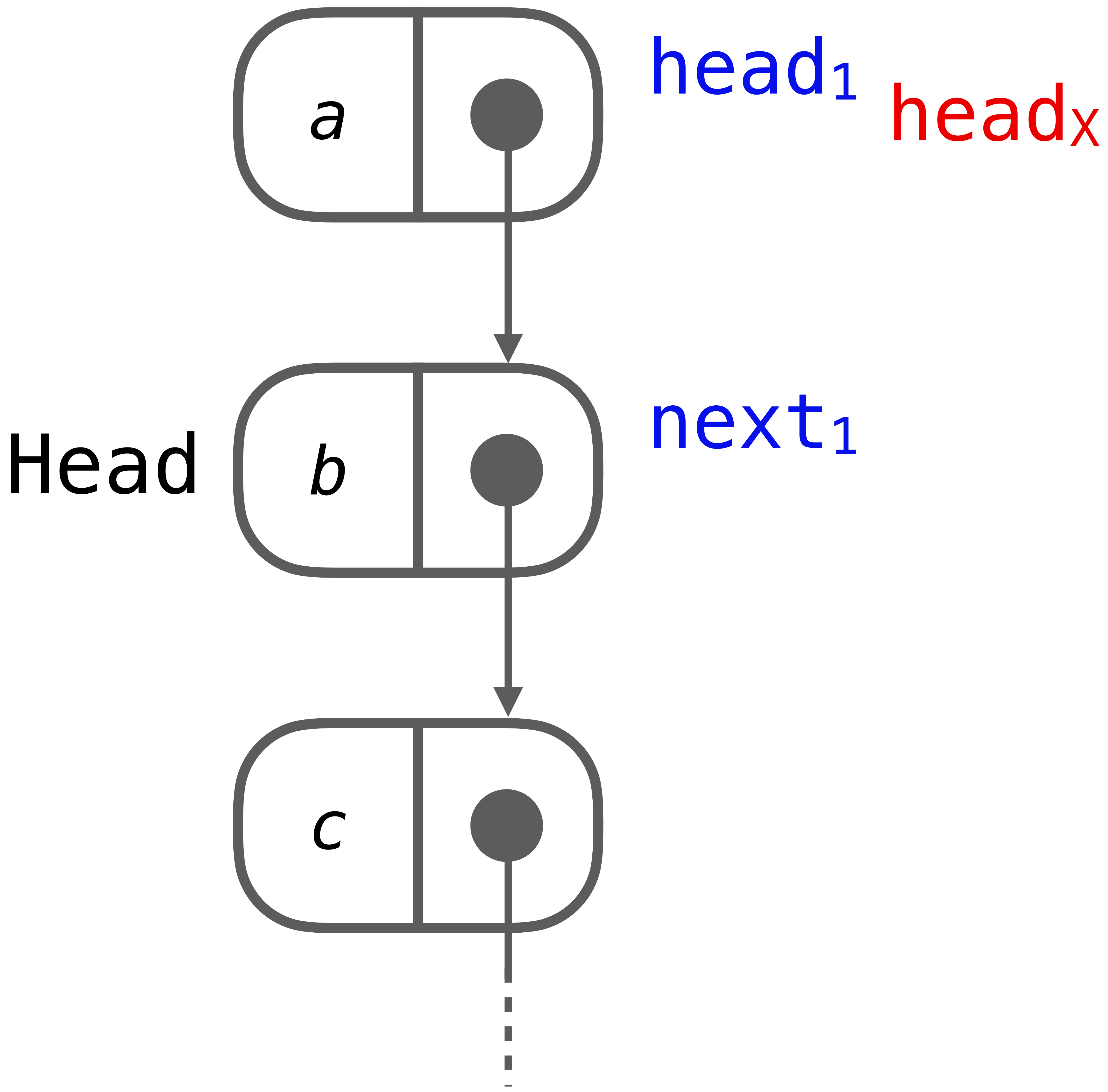
Lock-free Queue (Michael&Scott)

```
  void dequeue() {
    while (true) {
      head = Head;
 X    next = head->next;
      // ...
2     if(CAS(Head,head,next)){
1       // leak head?
        return;
  }}}
```



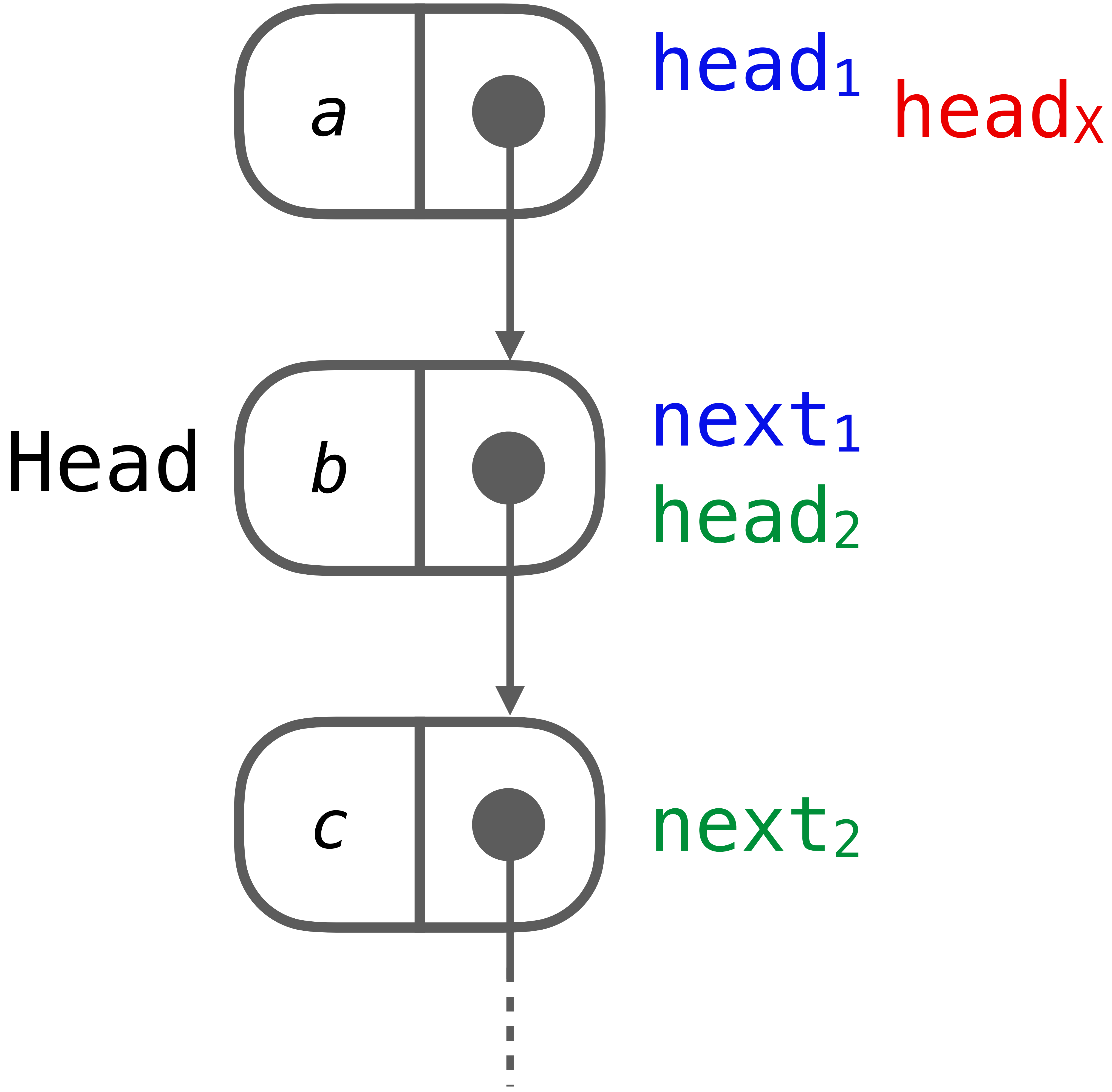
Lock-free Queue (Michael&Scott)

```
    void dequeue() {
     while (true) {
2      head = Head;
    X  next = head->next;
       // ...
       if(CAS(Head,head,next)){
1        // leak head?
         return;
    }}}
```



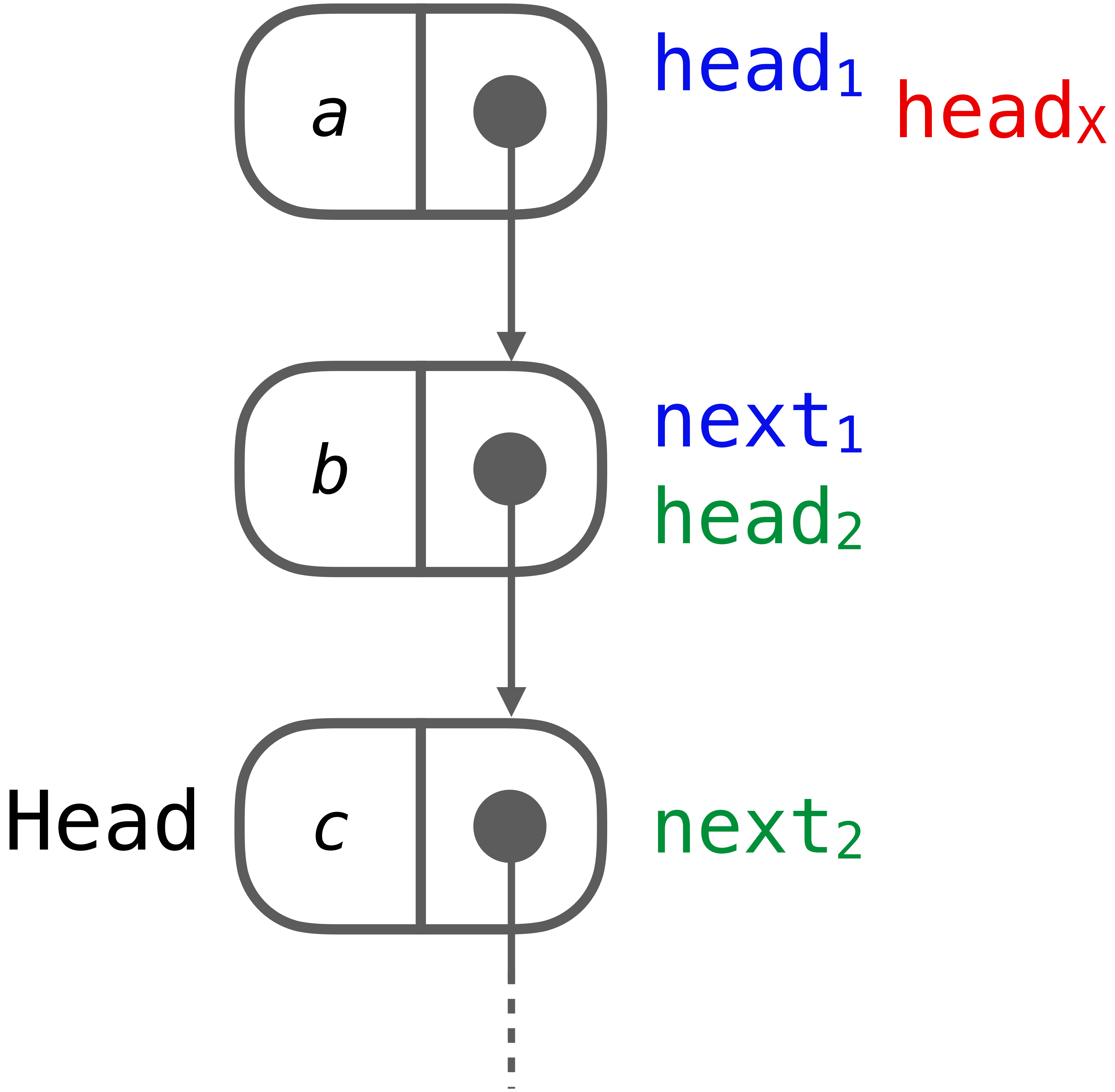
Lock-free Queue (Michael&Scott)

```
    void dequeue() {
      while (true) {
        head = Head;
      X next = head->next;
        // ...
  2     if(CAS(Head,head,next)){
1         // leak head?
          return;
    }}}
```



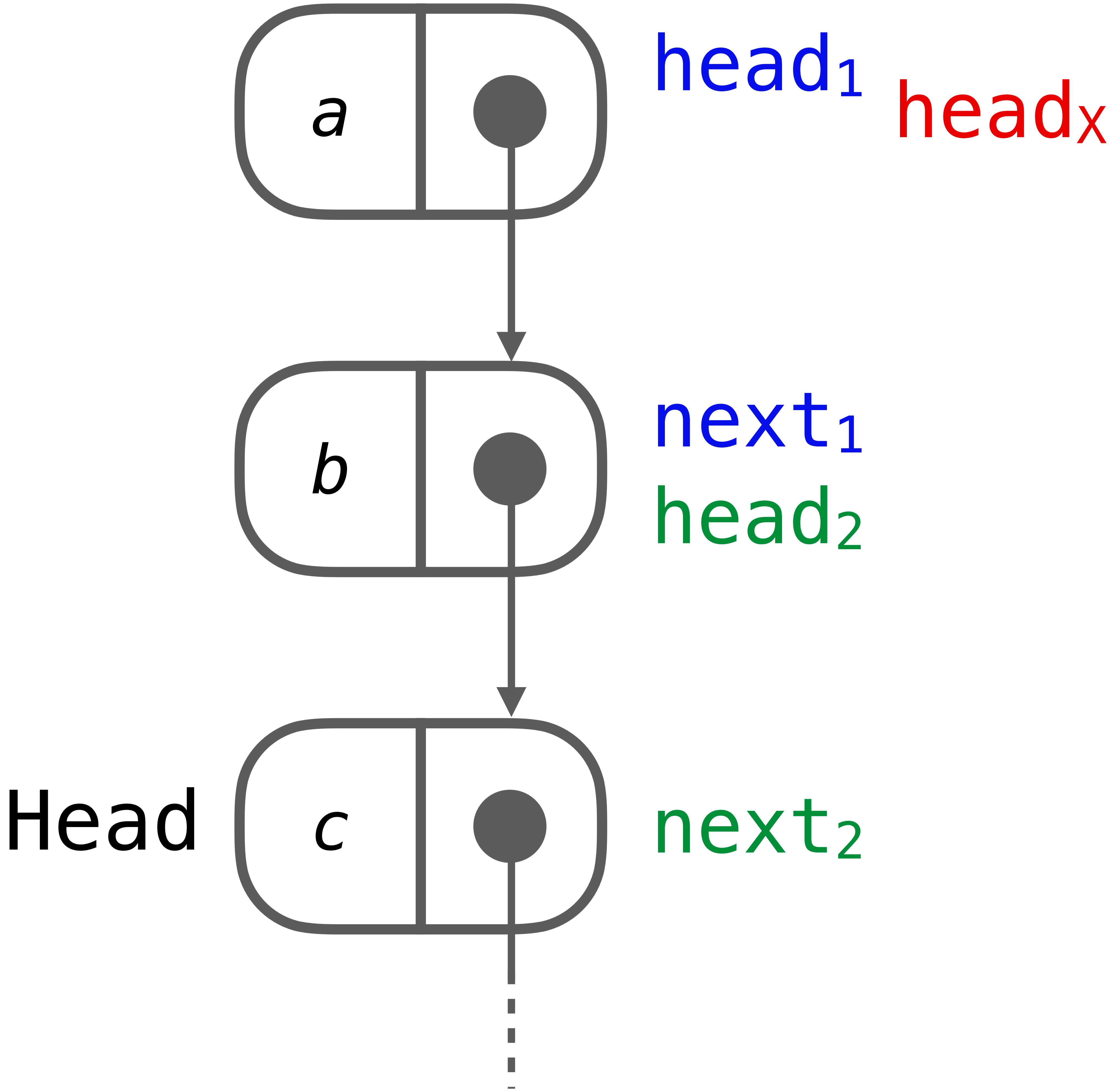
Lock-free Queue (Michael&Scott)

```
   void dequeue() {
    while (true) {
      head = Head;
    X next = head->next;
      // ...
      if(CAS(Head,head,next)){
12      // leak head?
        return;
   }}}
```



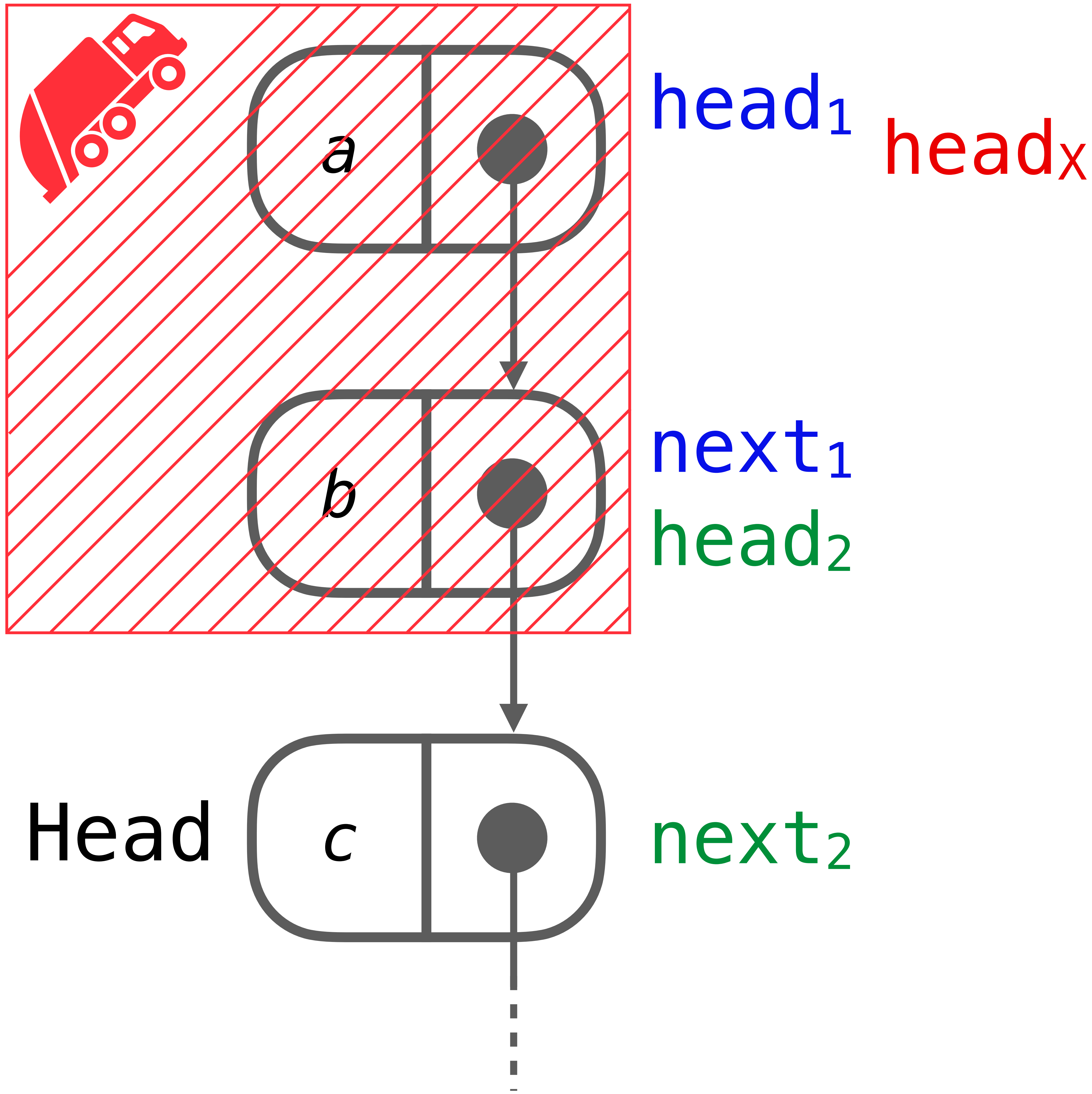
Lock-free Queue (Michael&Scott)

```
void dequeue() {
  while (true) {
     head = Head;
X    next = head->next;
     // ...
     if(CAS(Head,head,next)){
1 2    delete head;
       return;
}}}
```



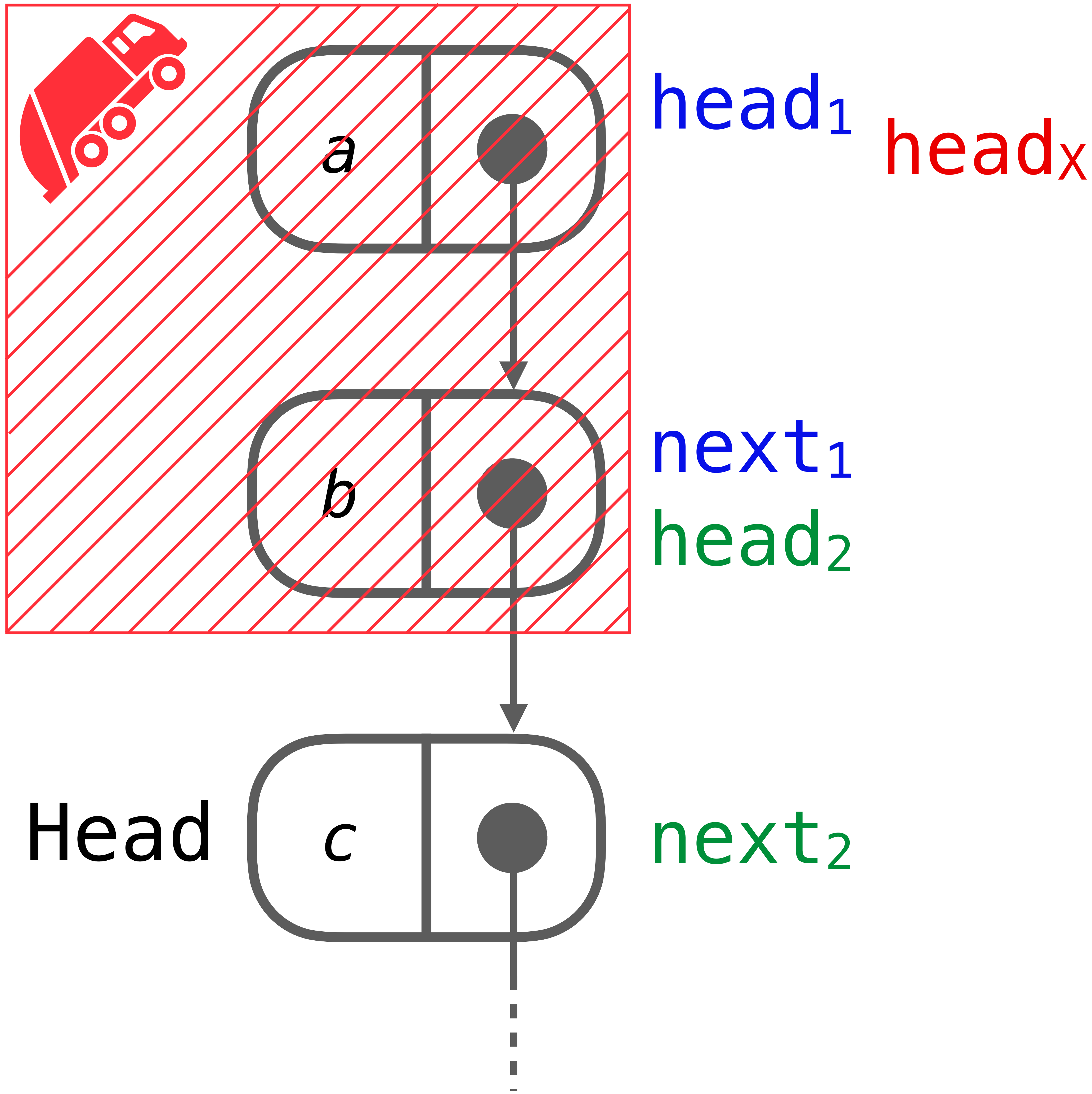
Lock-free Queue (Michael&Scott)

```
     void dequeue() {
       while (true) {
         head = Head;
       X next = head->next;
         // ...
         if(CAS(Head,head,next)){
           delete head;
 1 2       return;
     }}}
```



Lock-free Queue (Michael&Scott)

```
  void dequeue() {
   while (true) {
     head = Head;
     next = head->next;
     // ...
     if(CAS(Head,head,next)){
       **delete head;**
①②      return;
  }}}
```

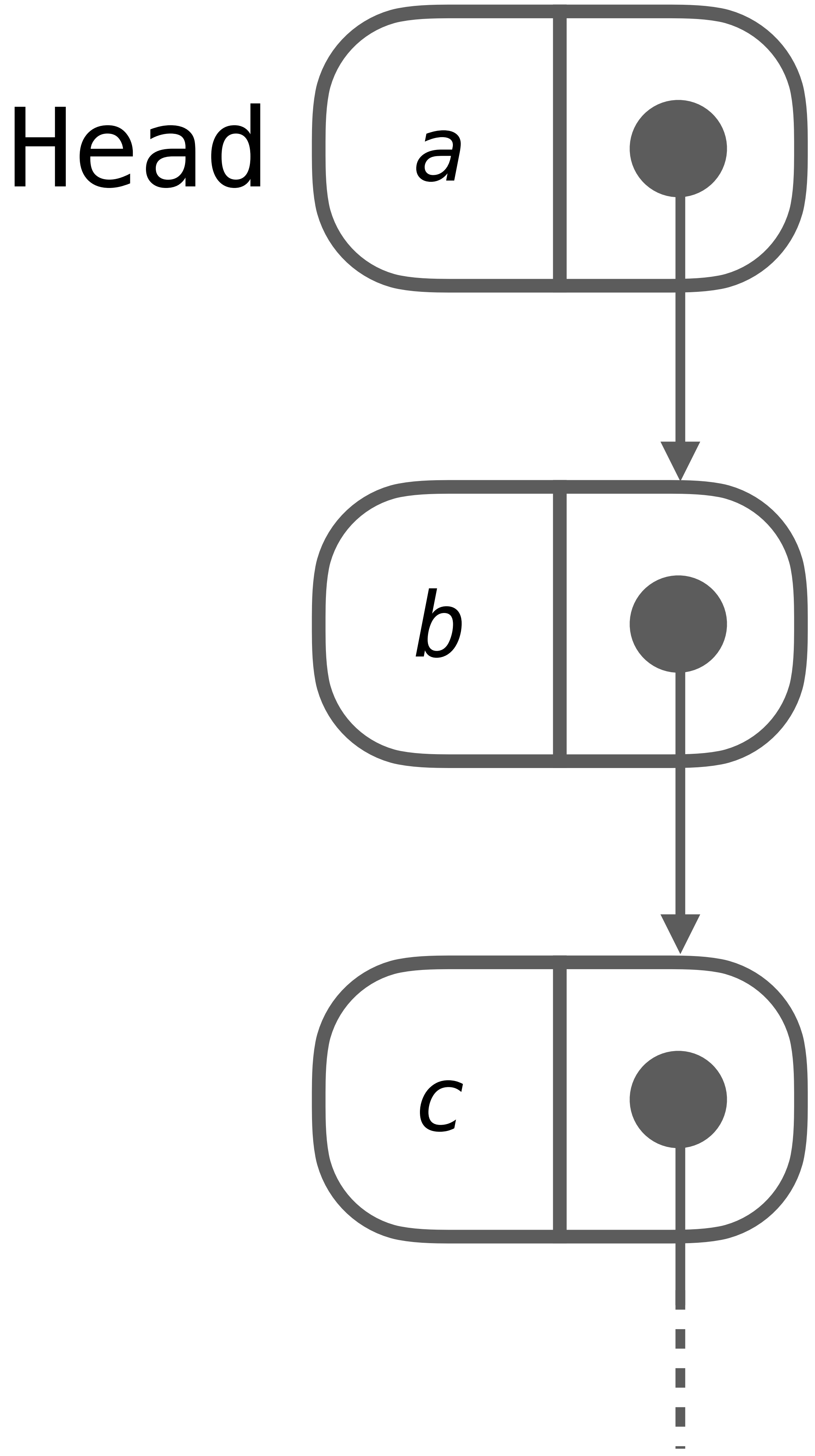


Reclamation

- Lock-free data structures (LFDS)
  - ➡ unsynchronized traversal
  - ➡ threads cannot detect whether a dereference is *safe*
- Safe memory reclamation (SMR)
  - ➡ defers deletion until it is safe
  - ➡ controlled by LFDS
  - ➡ various sophisticated techniques exist

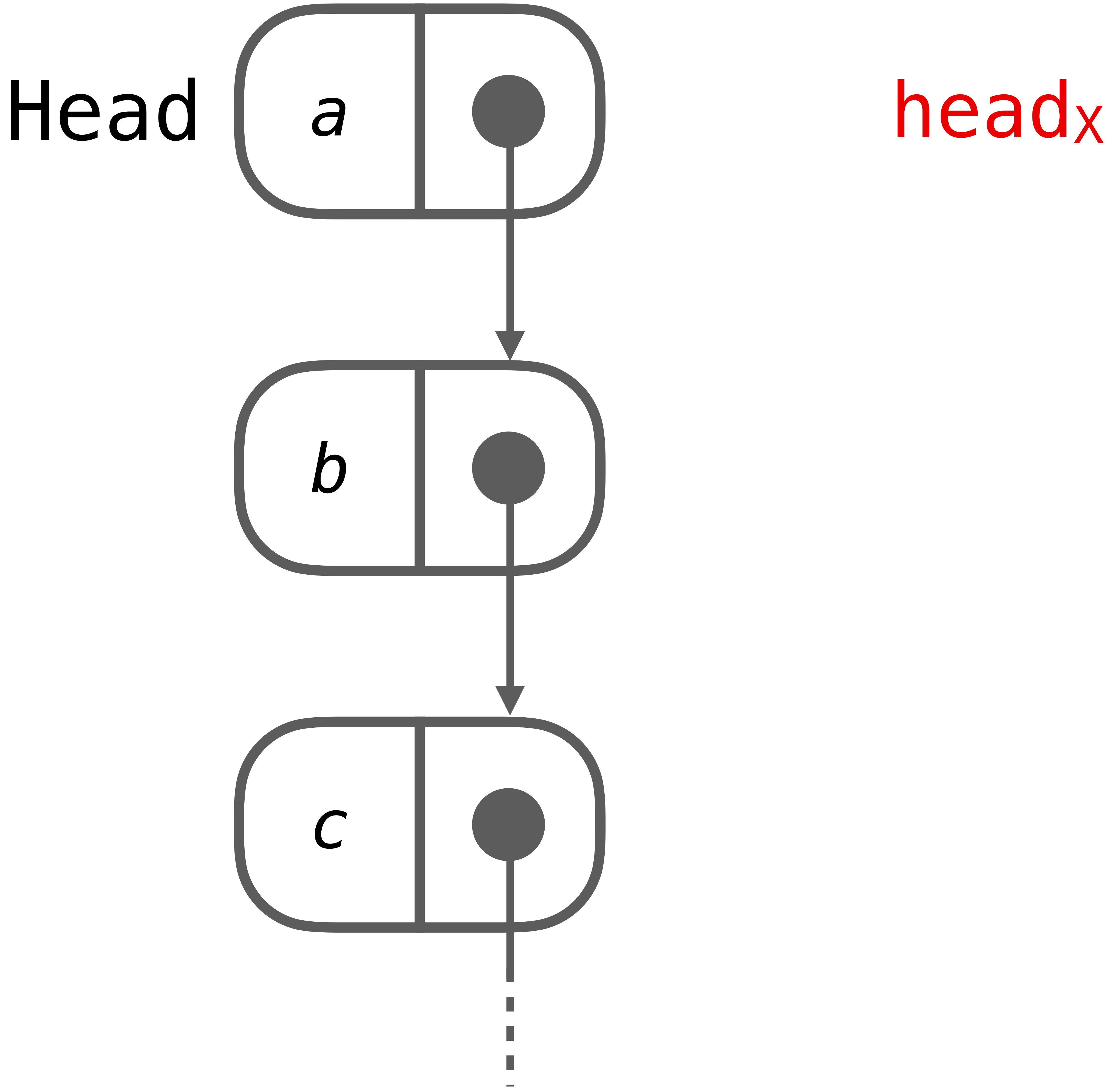
Lock-free Queue (Michael&Scott)

```
data_t dequeue() {
  while (true) {
1 X head = Head;
    protect(head);
    next = head->next;
    // ...
    if(CAS(Head,head,next)){
      retire(head);
      return;
}}}
```



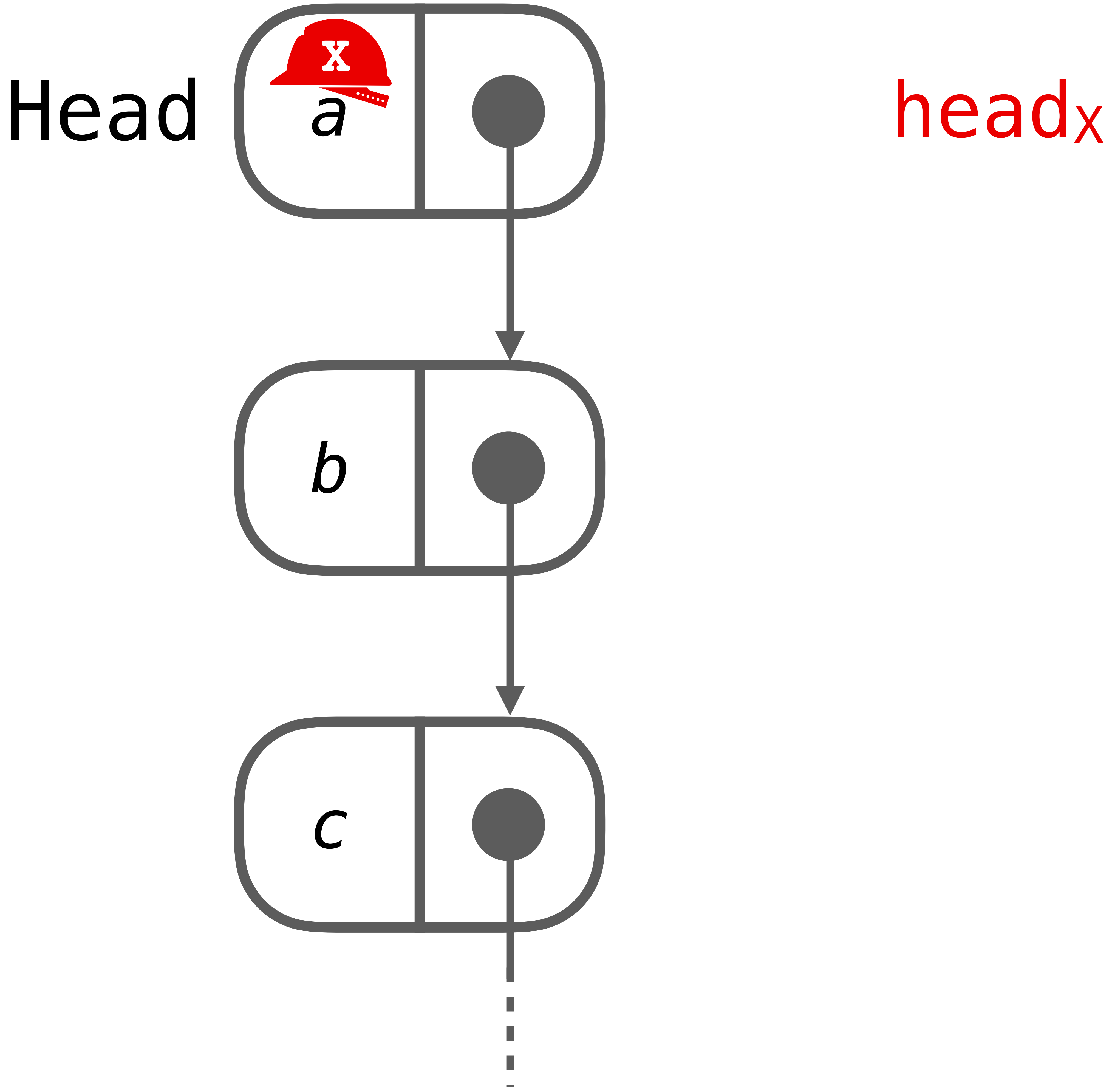
Lock-free Queue (Michael&Scott)

```
  data_t dequeue() {
    while (true) {
1      head = Head;
X      protect(head);
       next = head->next;
       // ...
       if(CAS(Head,head,next)){
         retire(head);
         return;
  }}}
```



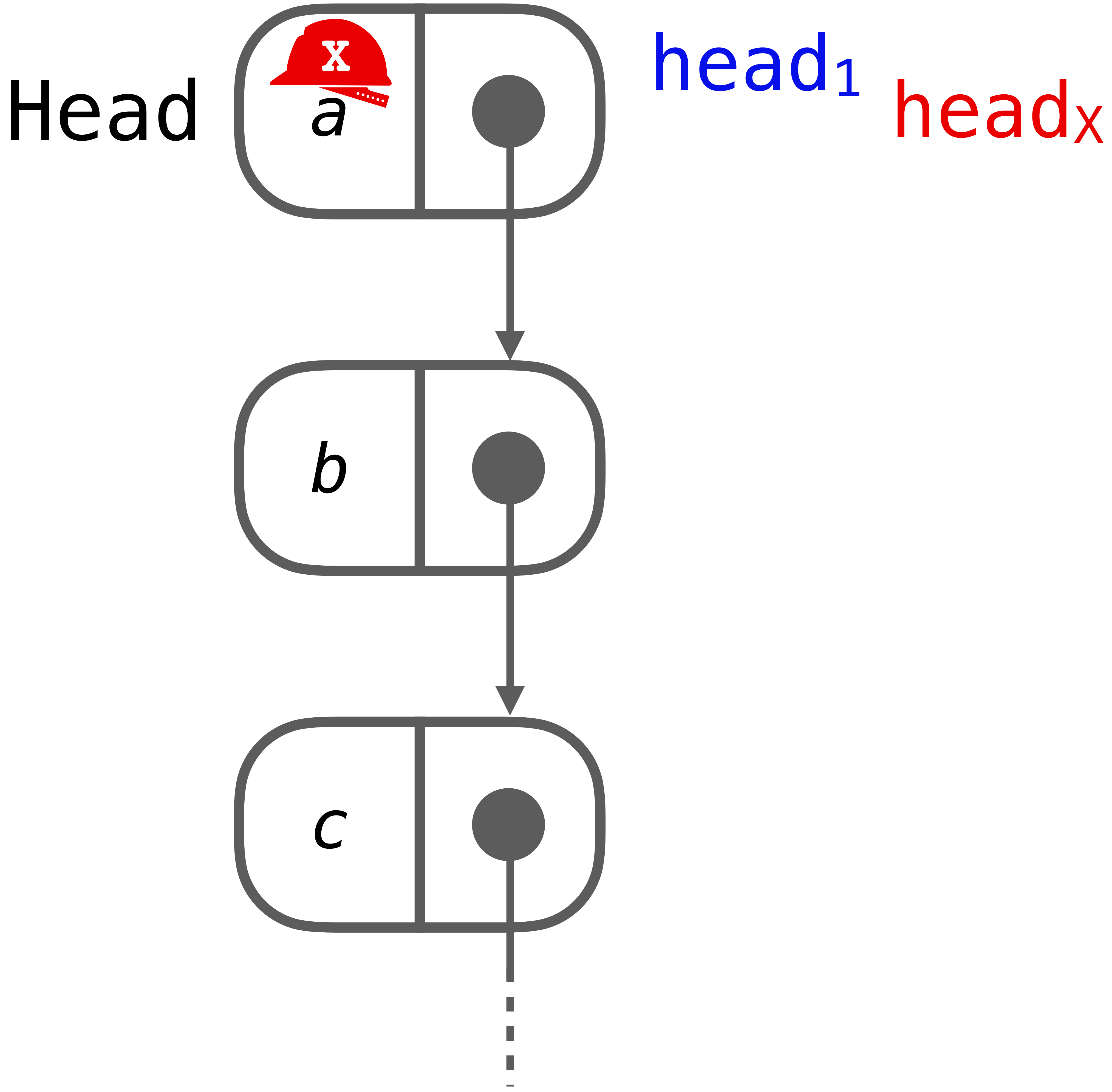
Lock-free Queue (Michael&Scott)

```
  data_t dequeue() {
    while (true) {
1     head = Head;
      protect(head);
X     next = head->next;
      // ...
      if(CAS(Head,head,next)){
        retire(head);
        return;
  }}}
```



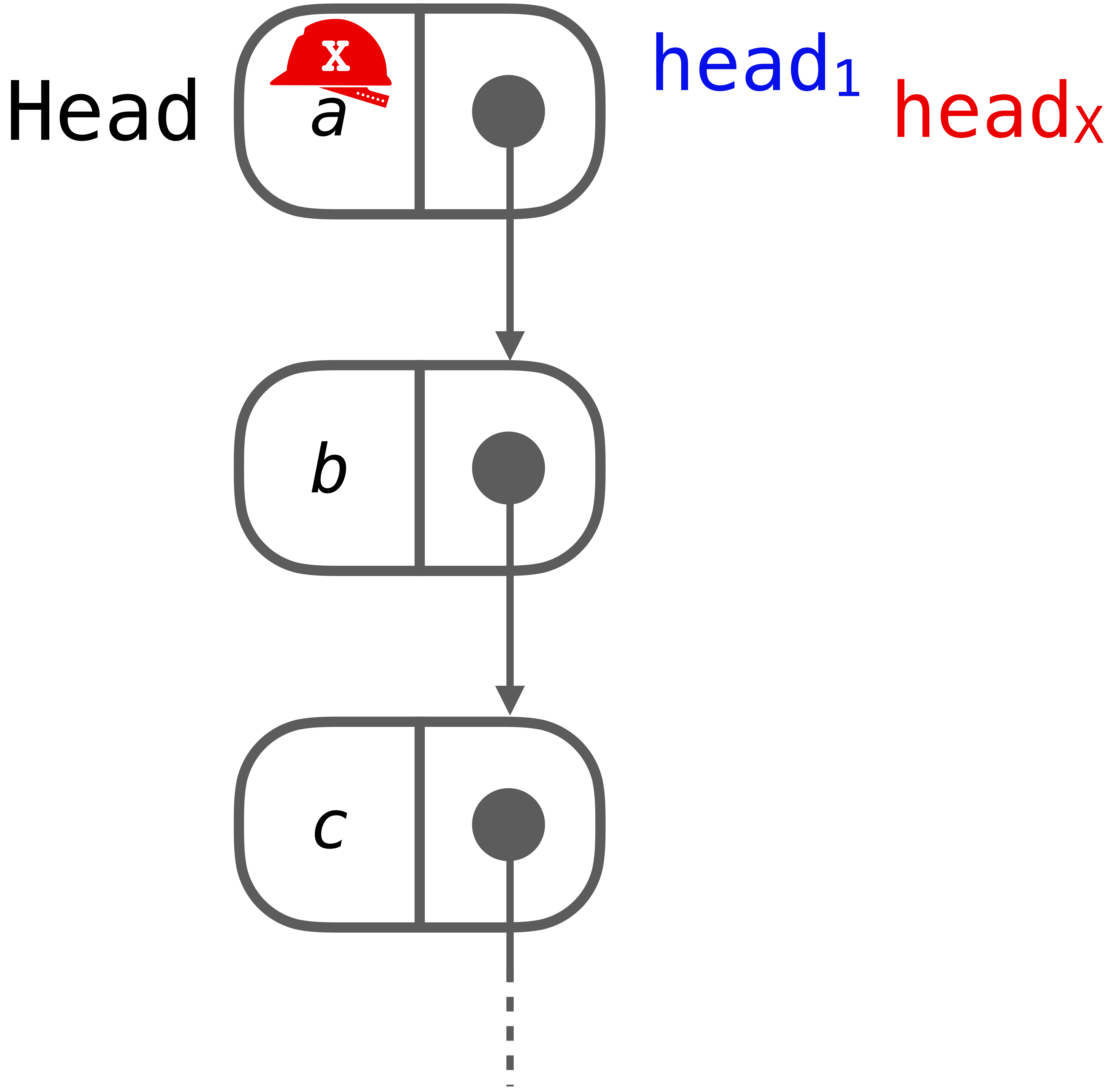
Lock-free Queue (Michael&Scott)

```
data_t dequeue() {
  while (true) {
    head = Head;
1   **protect(head);**
X   next = head->next;
    // ...
    if(CAS(Head,head,next)){
      **retire(head);**
      return;
}}}
```



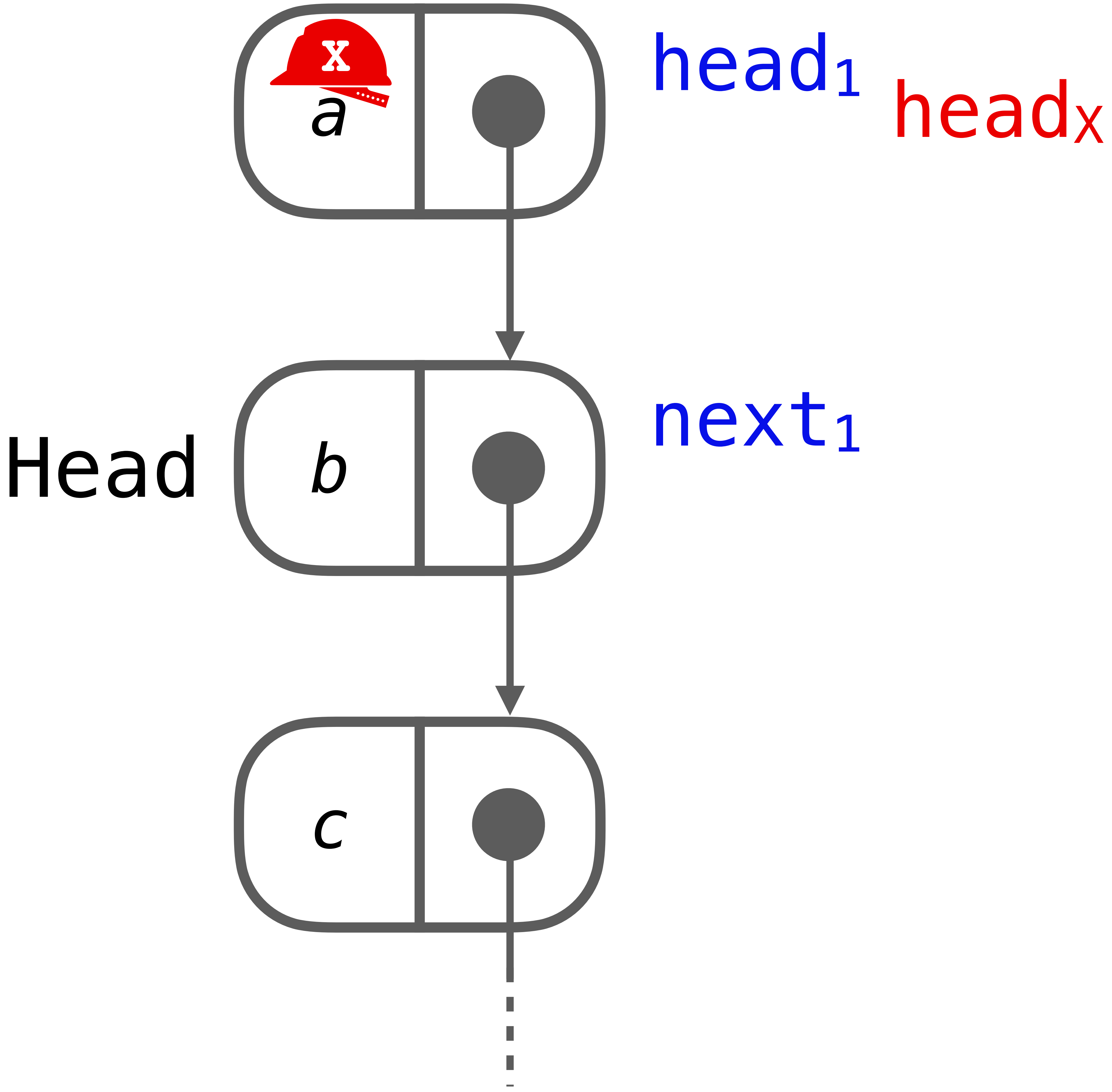
Lock-free Queue (Michael&Scott)

```
  data_t dequeue() {
    while (true) {
      head = Head;
      protect(head);
1 X   next = head->next;
      // ...
      if(CAS(Head,head,next)){
        retire(head);
        return;
  }}}
```



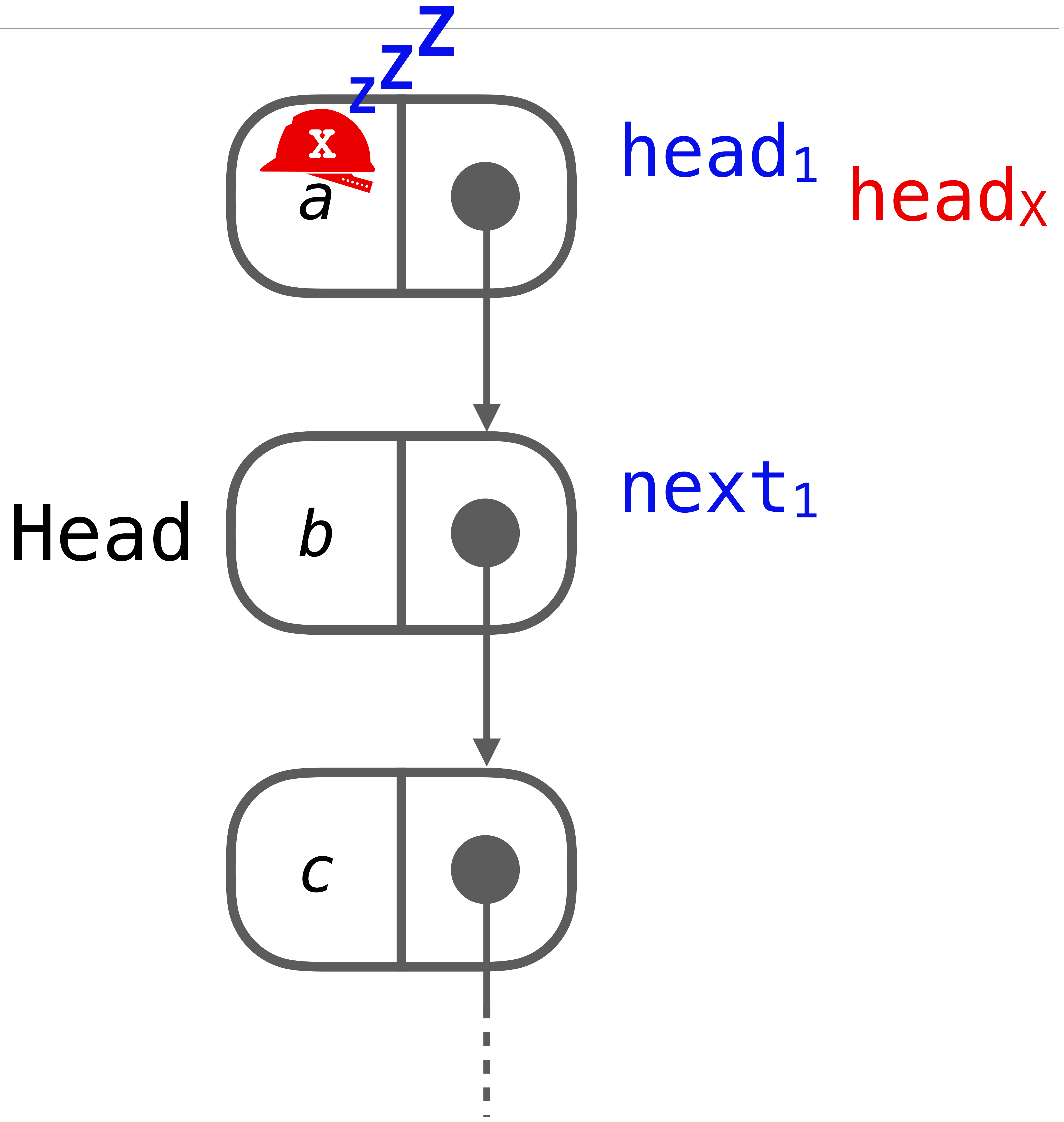
Lock-free Queue (Michael&Scott)

```
   data_t dequeue() {
     while (true) {
       head = Head;
       protect(head);
 X     next = head->next;
       // ...
       if(CAS(Head,head,next)){
1        retire(head);
         return;
   }}}
```



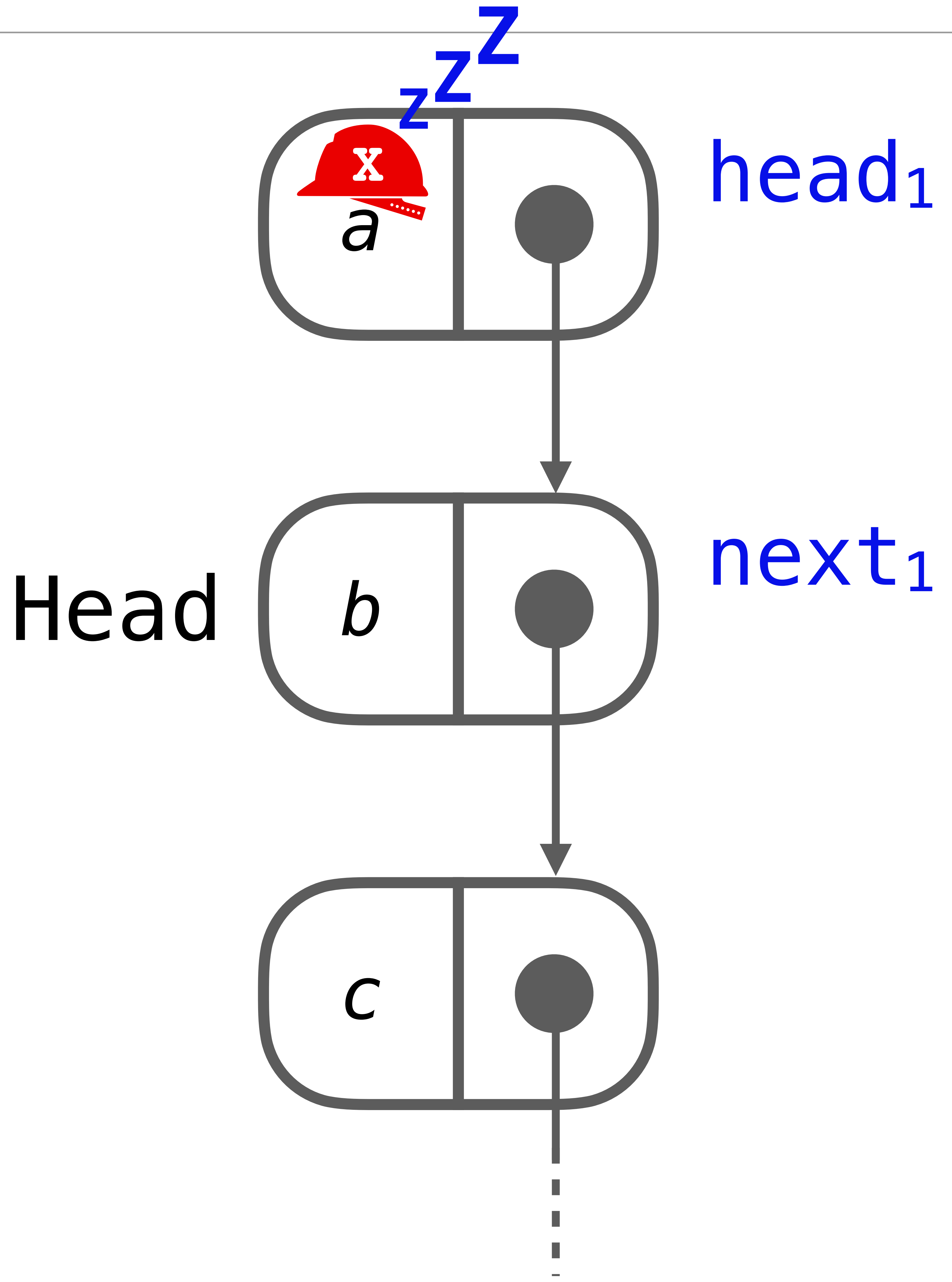
Lock-free Queue (Michael&Scott)

```
    data_t dequeue() {
      while (true) {
        head = Head;
        protect(head);
  X     next = head->next;
        // ...
        if(CAS(Head,head,next)){
          retire(head);
1         return;
    }}}
```



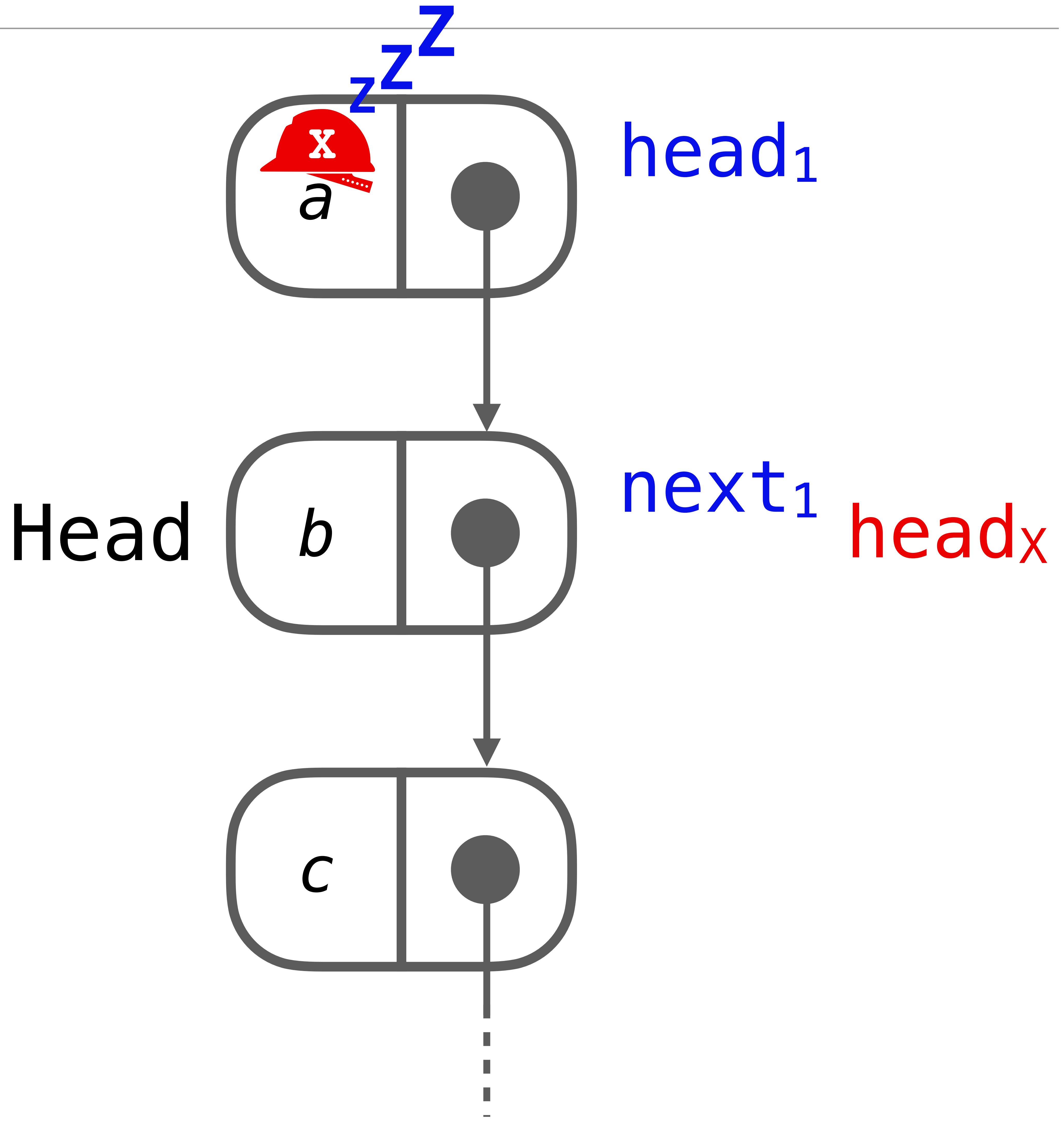
Lock-free Queue (Michael&Scott)

```
  data_t dequeue() {
    while (true) {
X     head = Head;
      protect(head);
      next = head->next;
      // ...
      if(CAS(Head,head,next)){
        retire(head);
1       return;
  }}}
```



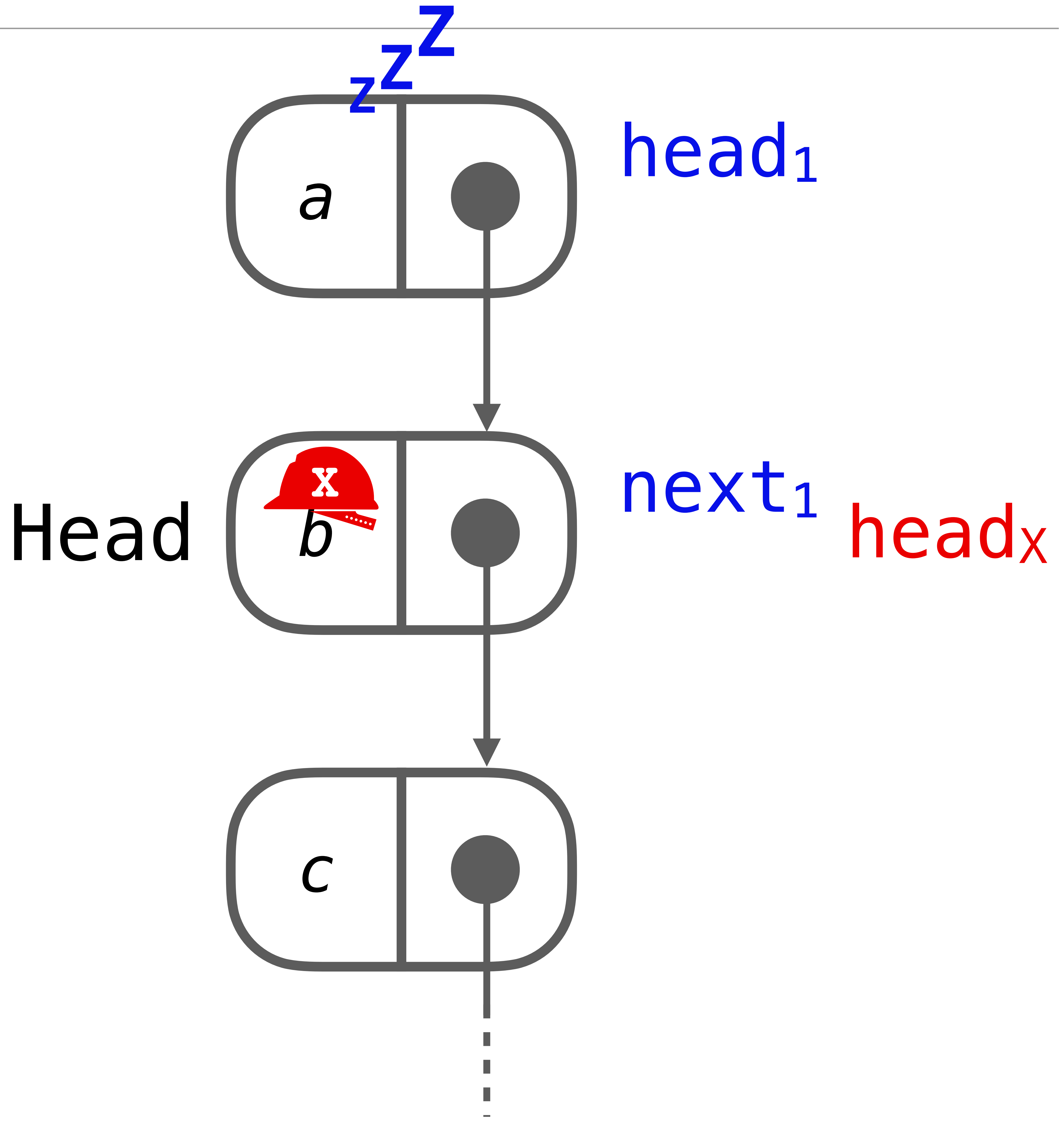
Lock-free Queue (Michael&Scott)

```
data_t dequeue() {
  while (true) {
    head = Head;
X   protect(head);
    next = head->next;
    // ...
    if(CAS(Head,head,next)){
      retire(head);
1     return;
}}}
```



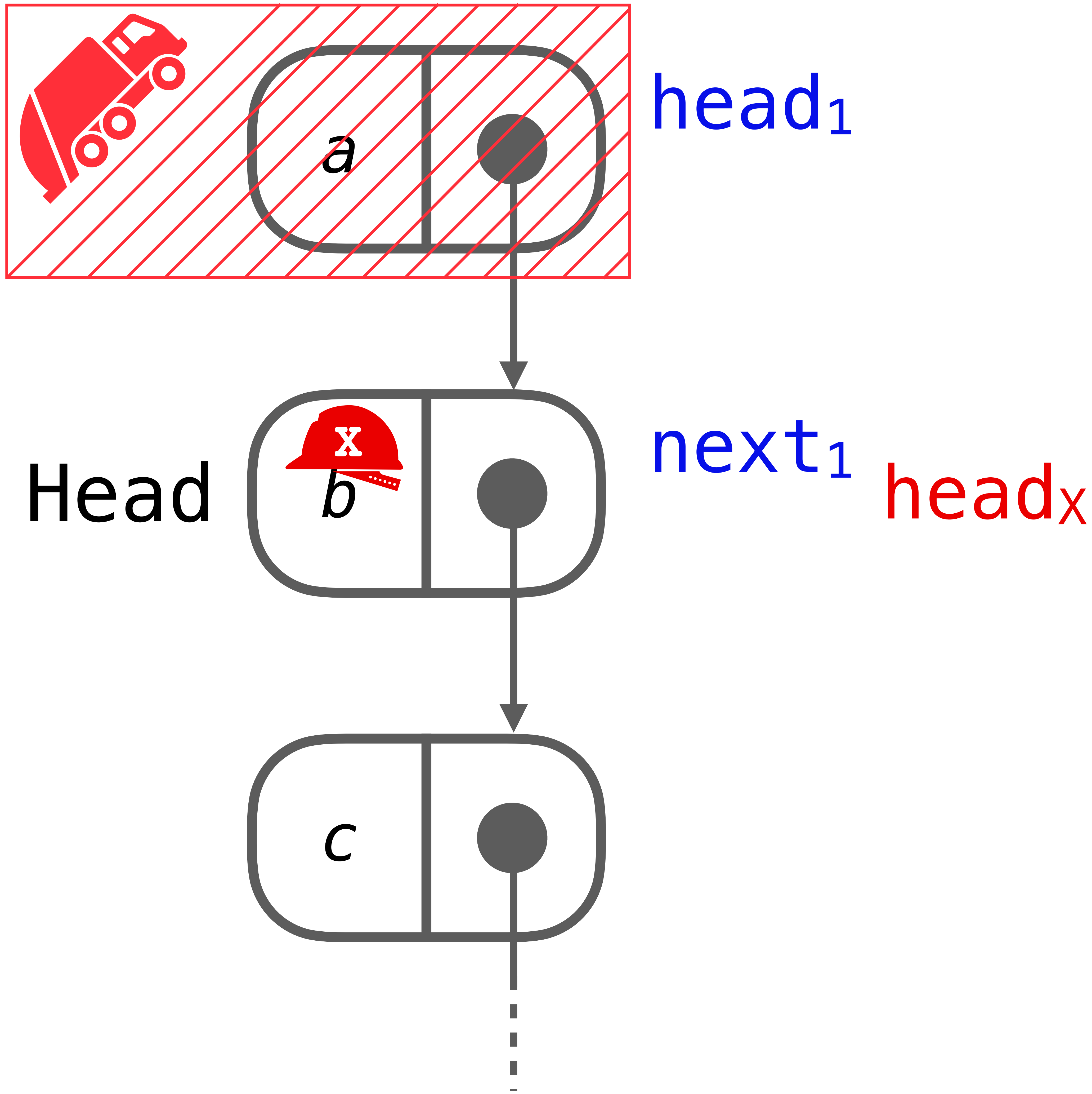
Lock-free Queue (Michael&Scott)

```
   data_t dequeue() {
    while (true) {
     head = Head;
     protect(head);
X    next = head->next;
     // ...
     if(CAS(Head,head,next)){
      retire(head);
1     return;
   }}}
```



Lock-free Queue (Michael&Scott)

```
  data_t dequeue() {
    while (true) {
      head = Head;
      protect(head);
X     next = head->next;
      // ...
      if(CAS(Head,head,next)){
        retire(head);
1       return;
  }}}
```



State-of-the-art Verification of Data Structures

- Pen&paper, mechanized/tool-supported
  - ➡ require deep understanding of proof technique, LFDS, and SMR
  - ➡ few works consider reclamation
- **Automated** (model-checking)
  - ➡ only done for GC
  - ➡ or custom semantics (allowing accesses of deleted memory)
  - ➡ **no works consider SMR**

Verification LFDS+SMR

```
struct Node {
    data_t data;
    Node* node;
}
```

```
shared:
    Node* Head;
    Node* Tail;
```

```
void init() {
    Head = new Node();
    Head->next = null;
    Tail = Head;
}
```

```
void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;


        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}
```

46 LOC

```
data_t dequeue() {
    while (true) {
        Node* head = Head;


        Node* tail = Tail;
        Node* next = head->next;

        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {

                return data;
            }
        }
    }
}
```

GC Implementation

(automated verification possible)

Verification LFDS+SMR

```
struct Node {
    data_t data;
    Node* node;
}
```

```
shared:
    Node* Head;
    Node* Tail;
```

```
void init() {
    Head = new Node();
    Head->next = null;
    Tail = Head;
}
```

```
void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect(tail, 0);
        if (Tail != tail) continue;
        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}
```

```
data_t dequeue() {
    while (true) {
        Node* head = Head;
        protect(head, 0);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        protect(next, 1);
        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {
                retire(head);
                return data;
            }
        }
    }
}
```

46+6 LOC

Verification LFDS+SMR

```
struct Node {
    data_t data;
    Node* node;
}
```

```
shared:
    Node* Head;
    Node* Tail;
```

```
void init() {
    Head = new Node();
    Head->next = null;
    Tail = Head;
}
```

```
void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect(tail, 0);
        if (Tail != tail) continue;
        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}
```

```
data_t dequeue() {
    while (true) {
        Node* head = Head;
        protect(head, 0);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        protect(next, 1);
        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {
                retire(head);
                return data;
            }
        }
    }
}
```

46+6 LOC

```
struct Rec {
    Rec* next;
    Node* hp0;
    Node* hp1;
}

shared:
    Rec* HPRecs;

thread-local:
    Rec* myRec;
    List<Node*> retiredList;

void join() {
    myRec = new HPRec();
    while (true) {
        Rec* tmp = HPRecs;
        myRec->next = tmp;
        if (CAS(HPRecs, tmp, myRec)) {
            break;
        }
    }
}

void part() {
    unprotect(0);
    unprotect(1);
}
```

+52 LOC

```
void protect(Node* ptr, int i) {
    if (i == 0) myRec->hp0 = ptr;
    if (i == 1) myRec->hp1 = ptr;
    assert(false);
}

void unprotect(int i) {
    protect(null, i);
}

void retire(Node* ptr) {
    retiredList.add(ptr);
    if (*) reclaim();
}

void reclaim() {
    List<Node*> protectedList;
    Rec* tmp = HPRecs;
    while (tmp != null) {
        Node* hp0 = cur->hp0;
        Node* hp1 = cur->hp1;
        protectedList.add(hp0);
        protectedList.add(hp1);
        cur = cur->next;
    }
    for (Node* ptr : retiredList) {
        if (!protectedList.contains(ptr)) {
            retiredList.remove(ptr);
            delete ptr;
    }
}
```

Verification LFDS+SMR

```
struct Node {
    data_t data;
    Node∗ node;
}
```

```
shared:
    Node∗ Head;
    Node∗ Tail;
```

```
void init() {
    Head = new Node();
    Head->next = null;
    Tail = Head;
}
```

```
void enqueue(data_t val) {
    Node∗ node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node∗ tail = Tail;
        protect(tail, 0);
        if (Tail != tail) continue;
        Node∗ next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}
```

```
data_t dequeue() {
    while (true) {
        Node∗ head = Head;
        protect(head, 0);
        if (Head != head) continue;
        Node∗ tail = Tail;
        Node∗ next = head->next;
        protect(next, 1);
        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {
                retire(head);
                return data;
            }
        }
    }
}
```

46+6 LOC

```
struct Rec {
    Rec∗ next;
    Node∗ hp0;
    Node∗ hp1;
}

shared:
    Rec∗ HPRecs;

thread-local:
    Rec∗ myRec;
    List<Node∗> retiredList;

void join() {
    myRec = new HPRec();
    while (true) {
        Rec∗ tmp = HPRecs;
        myRec->next = tmp;
        if (CAS(HPRecs, tmp, myRec)) {
            break;
        }
    }
}

void part() {
    unprotect(0);
    unprotect(1);
}
```

```
void protect(Node∗ ptr, int i) {
    if (i == 0) myRec->hp0 = ptr;
    if (i == 1) myRec->hp1 = ptr;
    assert(false);
}

void unprotect(int i) {
    protect(null, i);
}

void retire(Node∗ ptr) {
    retiredList.add(ptr);
    if (∗) reclaim();
}

void reclaim() {
    List<Node∗> protectedList;
    Rec∗ tmp = HPRecs;
    while (tmp != null) {
        Node∗ hp0 = cur->hp0;
        Node∗ hp1 = cur->hp1;
        protectedList.add(hp0);
        protectedList.add(hp1);
        cur = cur->next;
    }
    for (Node∗ ptr : retiredList) {
        if (!protectedList.contains(ptr)) {
            retiredList.remove(ptr);
            delete ptr;
        }
    }
}
```

+52 LOC

It is a **second** lock-free data structure!

Verification LFDS+SMR

```
struct Node {           shared:           void init() {
    data_t data;            Node* Head;       Head = new Node();
    Node* node;             Node* Tail;       Head->next = null;
}                                             Tail = Head;
                                          }
```

```
void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect(tail, 0);
        if (Tail != tail) continue;
        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}
```

```
data_t dequeue() {
    while (true) {
        Node* head = Head;
        protect(head, 0);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        protect(next, 1);
        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {
                retire(head);
                return data;
            }
        }
    }
}
```

LFDS

46+6 LOC

```
struct Rec {
    Rec* next;
    Node* hp0;
    Node* hp1;
}

shared:
    Rec* HPRecs;

thread-local:
    Rec* myRec;
    List<Node*> retiredList;

void join() {
    myRec = new HPRec();
    while (true) {
        Rec* tmp = HPRecs;
        myRec->next = tmp;
        if (CAS(HPRecs, tmp, myRec)) {
            break;
        }
    }
}

void part() {
    unprotect(0);
    unprotect(1);
}
```

```
void protect(Node* ptr, int i) {
    if (i == 0) myRec->hp0 = ptr;
    if (i == 1) myRec->hp1 = ptr;
    assert(false);
}

void unprotect(int i) {
    protect(null, i);
}

void retire(Node* ptr) {
    retiredList.add(ptr);
    if (*) reclaim();
}

void reclaim() {
    List<Node*> protectedList;
    Rec* tmp = HPRecs;
    while (tmp != null) {
        Node* hp0 = cur->hp0;
        Node* hp1 = cur->hp1;
        protectedList.add(hp0);
        protectedList.add(hp1);
        cur = cur->next;
    }
    for (Node* ptr : retiredList) {
        if (!protectedList.contains(ptr)) {
            retiredList.remove(ptr);
            delete ptr;
        }
    }
}
```

It is a **second** lock-free data structure!

+52 LOC

Verification LFDS+SMR

LFDS

```
struct Node {           shared:            void init() {
    data_t data;            Node* Head;        Head = new Node();
    Node* node;             Node* Tail;        Head->next = null;
}                                              Tail = Head;
                                           }

void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect(tail, 0);
        if (Tail != tail) continue;
        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}

data_t dequeue() {
    while (true) {
        Node* head = Head;
        protect(head, 0);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        protect(next, 1);
        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {
                retire(head);
                return data;
            }
        }
    }
}
```

46+6 LOC

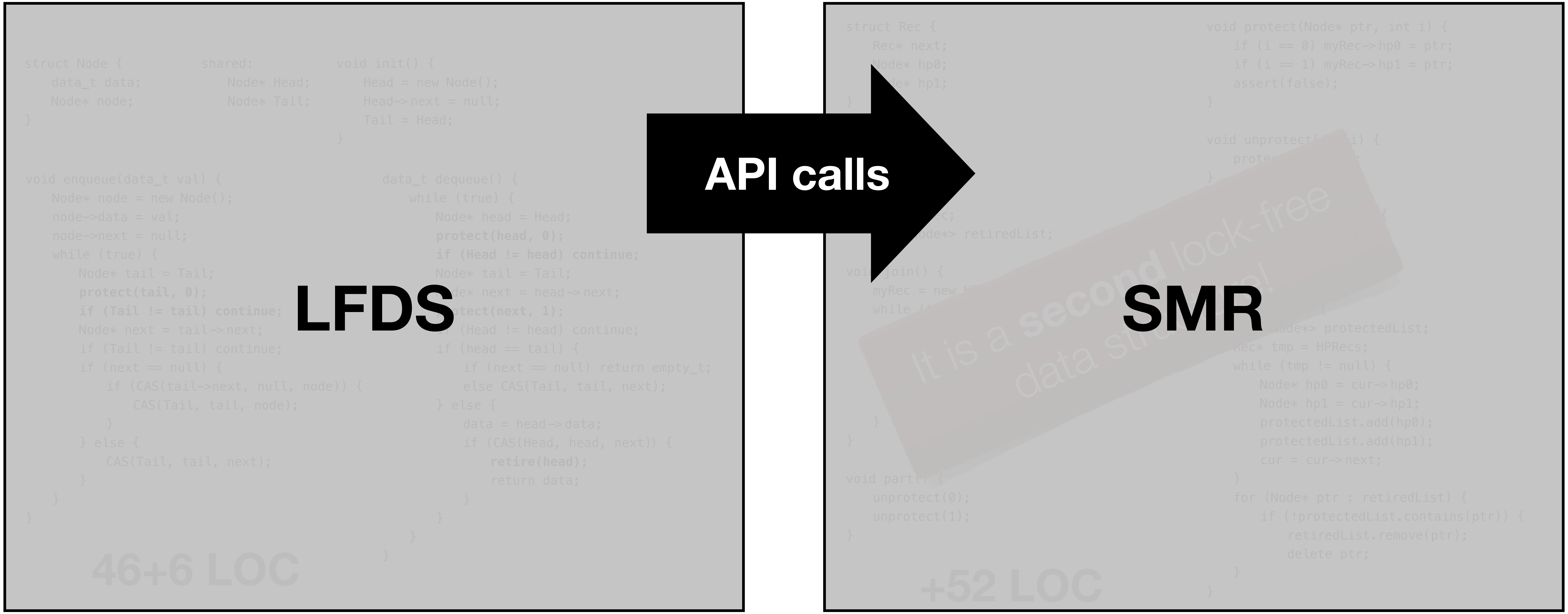
SMR

```
struct Rec {
    Rec* next;
    Node* hp0;
    Node* hp1;
}

shared:
    Rec* HPRecs;

thread-local:
    Rec* myRec;
    List<Node*> retiredList;

void join() {
    myRec = new HP[illegible]
    while ([illegible]

    }
}

void part() {
    unprotect(0);
    unprotect(1);
}

void protect(Node* ptr, int i) {
    if (i == 0) myRec->hp0 = ptr;
    if (i == 1) myRec->hp1 = ptr;
    assert(false);
}

void unprotect([illegible] i) {
    protec[illegible];
}

[illegible] {

[illegible] {
    [illegible]Node*> protectedList;
    Rec* tmp = HPRecs;
    while (tmp != null) {
        Node* hp0 = cur->hp0;
        Node* hp1 = cur->hp1;
        protectedList.add(hp0);
        protectedList.add(hp1);
        cur = cur->next;
    }
    for (Node* ptr : retiredList) {
        if (!protectedList.contains(ptr)) {
            retiredList.remove(ptr);
            delete ptr;
        }
    }
}
```

+52 LOC

It is a **second** lock-free data structure!

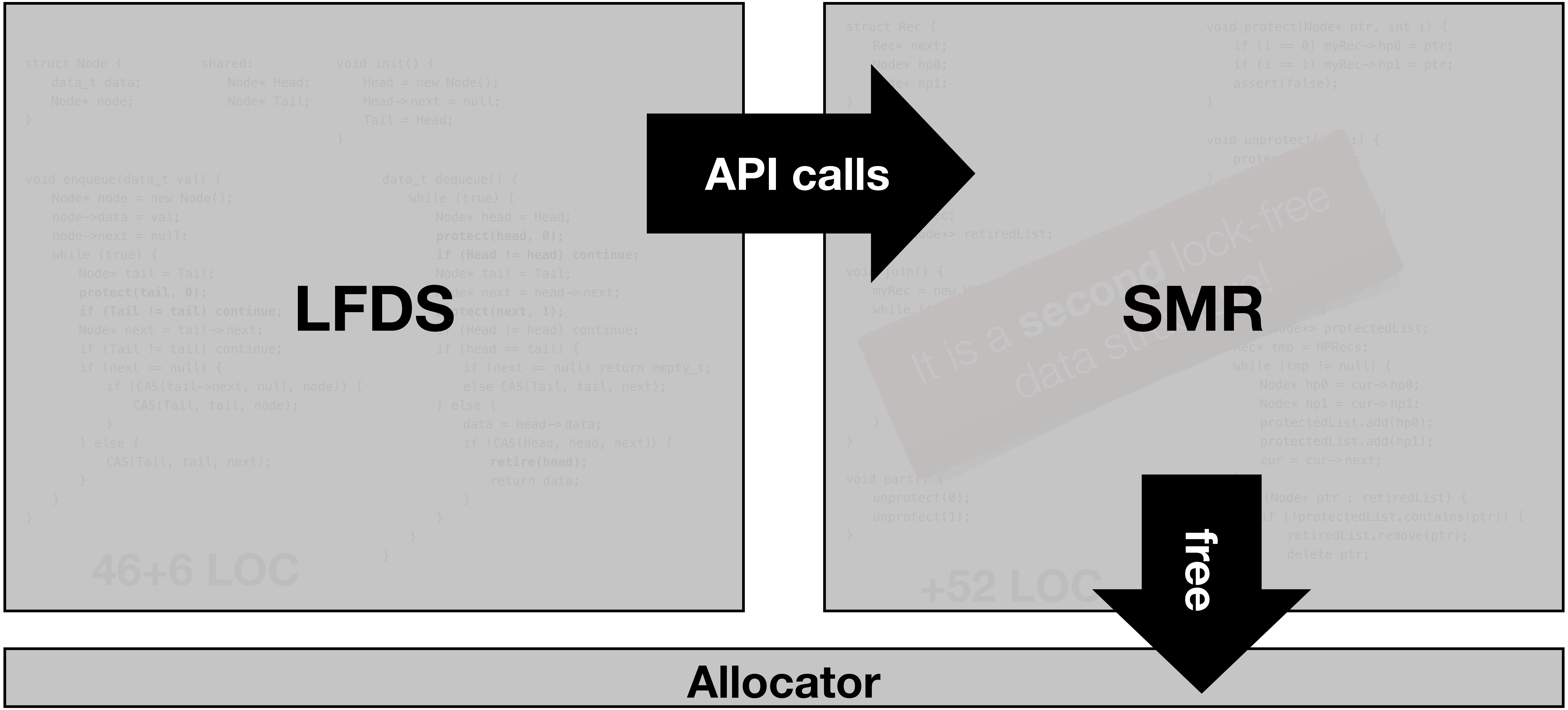
Verification LFDS+SMR



```
struct Node {          shared:          void init() {
    data_t data;           Node* Head;      Head = new Node();
    Node* node;            Node* Tail;      Head->next = null;
}                                           Tail = Head;
                                        }

void enqueue(data_t val) {                      data_t dequeue() {
    Node* node = new Node();                        while (true) {
    node->data = val;                                   Node* head = Head;
    node->next = null;                                  protect(head, 0);
    while (true) {                                      if (Head != head) continue;
        Node* tail = Tail;                              Node* tail = Tail;
        protect(tail, 0);                               Node* next = head->next;
        if (Tail != tail) continue;                     protect(next, 1);
        Node* next = tail->next;                        if (Head != head) continue;
        if (Tail != tail) continue;                     if (head == tail) {
        if (next == null) {                                 if (next == null) return empty_t;
            if (CAS(tail->next, null, node)) {              else CAS(Tail, tail, next);
                CAS(Tail, tail, node);                  } else {
            }                                               data = head->data;
        } else {                                            if (CAS(Head, head, next)) {
            CAS(Tail, tail, next);                              retire(head);
        }                                                       return data;
    }                                                       }
}                                                       }
                                                    }
                                                }
```

```
struct Rec {                                void protect(Node* ptr, int i) {
    Rec* next;                                  if (i == 0) myRec->hp0 = ptr;
    Node* hp0;                                  if (i == 1) myRec->hp1 = ptr;
      de* hp1;                                  assert(false);
}                                           }

                                            void unprotect(   i) {
                                                protec          ;
                                            }

         ec;                                                   {
       ode*> retiredList;

voi  join() {
    myRec = new H
    while (
                                                             {
                                                Node*> protectedList;
                                                Rec* tmp = HPRecs;
                                                while (tmp != null) {
                                                    Node* hp0 = cur->hp0;
                                                    Node* hp1 = cur->hp1;
                                                    protectedList.add(hp0);
    }                                               protectedList.add(hp1);
}                                                   cur = cur->next;
                                                }
void part() {                                   for (Node* ptr : retiredList) {
    unprotect(0);                                   if (!protectedList.contains(ptr)) {
    unprotect(1);                                       retiredList.remove(ptr);
}                                                       delete ptr;
                                                    }
                                            }
```

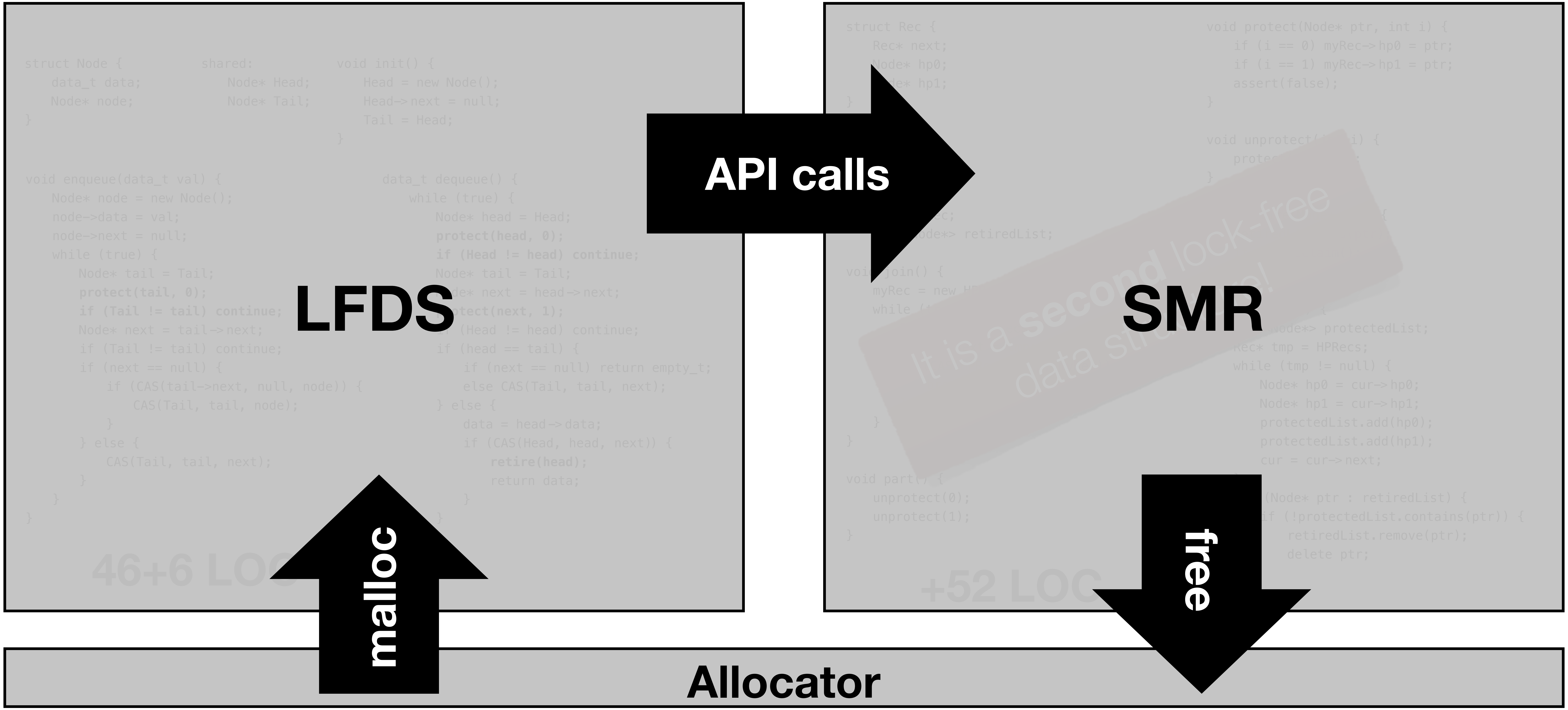
Verification LFDS+SMR



```
struct Node {          shared:          void init() {
    data_t data;           Node* Head;      Head = new Node();
    Node* node;            Node* Tail;      Head->next = null;
}                                           Tail = Head;
                                        }

void enqueue(data_t val) {                      data_t dequeue() {
    Node* node = new Node();                        while (true) {
    node->data = val;                                   Node* head = Head;
    node->next = null;                                  protect(head, 0);
    while (true) {                                      if (Head != head) continue;
        Node* tail = Tail;                              Node* tail = Tail;
        protect(tail, 0);                               de* next = head->next;
        if (Tail != tail) continue;                     otect(next, 1);
        Node* next = tail->next;                        f (Head != head) continue;
        if (Tail != tail) continue;                     if (head == tail) {
        if (next == null) {                                 if (next == null) return empty_t;
            if (CAS(tail->next, null, node)) {              else CAS(Tail, tail, next);
                CAS(Tail, tail, node);                  } else {
            }                                               data = head->data;
        } else {                                            if (CAS(Head, head, next)) {
            CAS(Tail, tail, next);                              retire(head);
        }                                                       return data;
    }                                                       }
}                                                       }
                                                    }
                                                }
```

```
struct Rec {                                  void protect(Node* ptr, int i) {
    Rec* next;                                    if (i == 0) myRec->hp0 = ptr;
    Node* hp0;                                    if (i == 1) myRec->hp1 = ptr;
     de* hp1;                                     assert(false);
}                                             }

                                              void unprotect(    i) {
                                                  protec
                                              }

          ec;
       ode*> retiredList;

voi join() {
    myRec = new HP
    while (
                                                  Node*> protectedList;
                                                  Rec* tmp = HPRecs;
                                                  while (tmp != null) {
                                                      Node* hp0 = cur->hp0;
                                                      Node* hp1 = cur->hp1;
                                                      protectedList.add(hp0);
                                                      protectedList.add(hp1);
    }                                                 cur = cur->next;
}
                                                    (Node* ptr : retiredList) {
void part() {                                       if (!protectedList.contains(ptr)) {
    unprotect(0);                                       retiredList.remove(ptr);
    unprotect(1);                                       delete ptr;
}
```

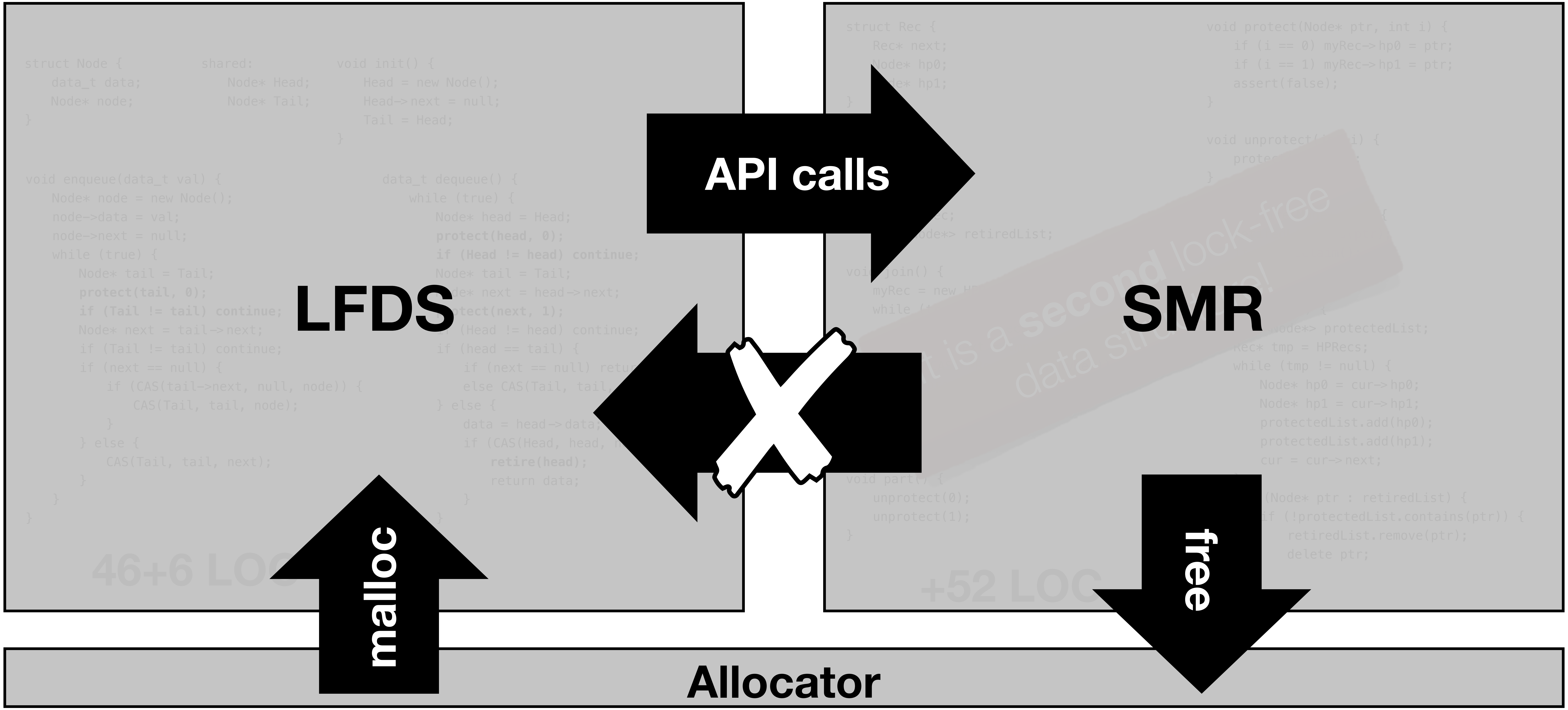
Verification LFDS+SMR



```
struct Node {           shared:            void init() {
    data_t data;            Node* Head;        Head = new Node();
    Node* node;             Node* Tail;        Head->next = null;
}                                              Tail = Head;
                                           }

void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect(tail, 0);
        if (Tail != tail) continue;
        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}

data_t dequeue() {
    while (true) {
        Node* head = Head;
        protect(head, 0);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        protect(next, 1);
        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {
                retire(head);
                return data;
            }
        }
    }
}
```

```
struct Rec {
    Rec* next;
    Node* hp0;
    Node* hp1;
}

[illegible]ec;
[illegible]ode*> retiredList;

void join() {
    myRec = new H[illegible]
    while ([illegible]
    }
}

void part() {
    unprotect(0);
    unprotect(1);
}

void protect(Node* ptr, int i) {
    if (i == 0) myRec->hp0 = ptr;
    if (i == 1) myRec->hp1 = ptr;
    assert(false);
}

void unprotect([illegible] i) {
    protec[illegible];
}

[illegible] {
    [illegible]Node*> protectedList;
    Rec* tmp = HPRecs;
    while (tmp != null) {
        Node* hp0 = cur->hp0;
        Node* hp1 = cur->hp1;
        protectedList.add(hp0);
        protectedList.add(hp1);
        cur = cur->next;
    }
    [illegible] (Node* ptr : retiredList) {
        if (!protectedList.contains(ptr)) {
            retiredList.remove(ptr);
            delete ptr;
```

Verification LFDS+SMR



```
struct Node {            shared:            void init() {
    data_t data;             Node* Head;        Head = new Node();
    Node* node;              Node* Tail;        Head->next = null;
}                                               Tail = Head;
                                            }

void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect(tail, 0);
        if (Tail != tail) continue;
        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}

data_t dequeue() {
    while (true) {
        Node* head = Head;
        protect(head, 0);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        protect(next, 1);
        if (Head != head) continue;
        if (head == tail) {
            if (next == null) retu
            else CAS(Tail, tail,
        } else {
            data = head->data;
            if (CAS(Head, head, n
                retire(head);
                return data;
            }
        }
```

```
struct Rec {                        void protect(Node* ptr, int i) {
    Rec* next;                          if (i == 0) myRec->hp0 = ptr;
    Node* hp0;                          if (i == 1) myRec->hp1 = ptr;
    de* hp1;                            assert(false);
}                                   }

                                    void unprotect(   i) {
                                        protec
                                    }
        ec;
        ode*> retiredList;

void join() {
    myRec = new H
    while (
                                           {
                                        ode*> protectedList;
                                        Rec* tmp = HPRecs;
                                        while (tmp != null) {
                                            Node* hp0 = cur->hp0;
                                            Node* hp1 = cur->hp1;
                                            protectedList.add(hp0);
                                            protectedList.add(hp1);
                                            cur = cur->next;

void part() {                              (Node* ptr : retiredList) {
    unprotect(0);                          if (!protectedList.contains(ptr)) {
    unprotect(1);                              retiredList.remove(ptr);
}                                              delete ptr;
```

Contribution 1: Compositional Verification for LFDS + SMR

Compositional Verification

- API between LFDS and SMR
  - ➡ give a formal specification SPEC
  - ➡ SPEC states *which&when* addresses are freed

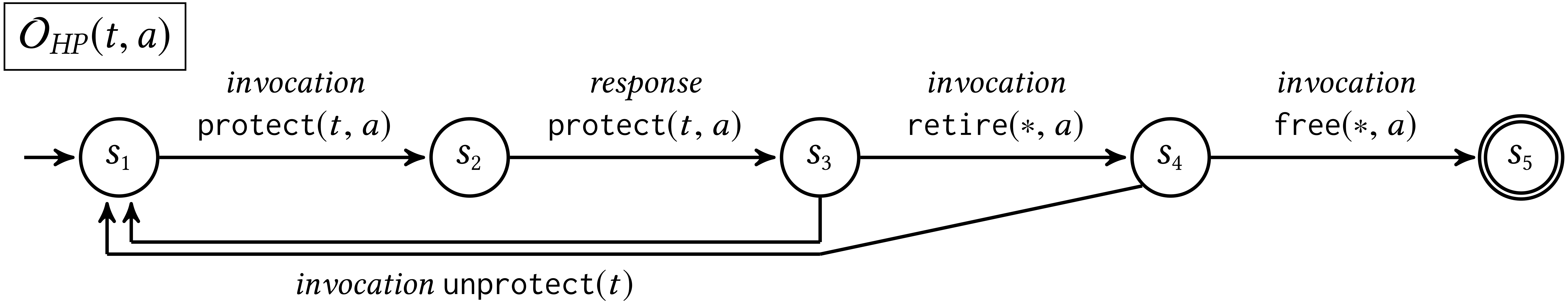
- **Compositional Verification**
  1) verify SMR against SPEC
  2) verify LFDS, using SPEC to over-approximate SMR

SPEC Example

- Hazard pointers:

a retired node is not reclaimed if it has been protected continuously since before the retire

- Programmers rely on this guarantee, not on the actual implementation
- Formalized:


$O_{HP}(t, a)$
invocation protect(t, a)
response protect(t, a)
invocation retire$(*, a)$
invocation free$(*, a)$
s_1
s_2
s_3
s_4
s_5
invocation unprotect(t)

Experiments

- SMR against SPEC:

SMR implementation	SPEC size	Time	Correct?
Hazard Pointers (HP)	3x5x5	**1.5s**	yes
Epoch-based Reclamation (EBR)	3x5	**11.2s**	yes

Experiments

- SMR against SPEC:

SMR implementation	SPEC size	Time	Correct?
Hazard Pointers (HP)	3x5x5	**1.5s**	yes
Epoch-based Reclamation (EBR)	3x5	**11.2s**	yes

- Linearizability of LFDS+SPEC

Infeasible: severe state space explosion due to re-allocations!

Contribution 2: State Space Reduction

State Space Reduction

- Theorem:

For verification, it is sound to restrict re-allocations to a single address

- Two requirements:

  1) SPEC invariant to re-allocations

  2) LFDS free from ABAs

State Space Reduction

- Theorem:

For verification, it is sound to restrict re-allocations to a single address

- Two requirements:

1) SPEC invariant to re-allocations ➡ check on SPEC automaton

2) LFDS free from ABAs ➡ check on reduced (!) LFDS state space

Experiments cont.

LFDS	SPEC	Time	Linearizable?
Michael&Scott's queue	NoReclaim	**7m**	yes
Michael&Scott's queue	EBR	**44m**	yes
Michael&Scott's queue	HP	**120m**	yes
Treiber's stack	EBR	**16s**	yes
Treiber's stack	HP	**19s**	yes
DGLM queue	EBR	**63m**	yes*
DGLM queue	HP	**117m**	yes*

* with hint for heap abstraction

Fin.

Questions?