Non-blocking algorithms

Master in computer science of IP Paris

Master CHPS of Paris Saclay

Gaël Thomas

Limit of lock algorithms

- Amdahl low
  - T: execution time of the application
  - p: percentage of code executable in parallel
  - ⇒ $T * (1 - p + p/n)$: execution time with n threads
  - ⇒ **$a = 1/(1 – p + p/n)$**: acceleration
  - ⇒ limit when $n \to \infty$: $a \to 1/(1 – p)$

- With numerical value:
  - $p = 75\% \Rightarrow a \to 1/0{,}25 = 4$ when $n \to \infty$ (3,7 with 32 cores)
  - $p = 95\% \Rightarrow a \to 1/0{,}05 = 20$ when $n \to \infty$ (12,55 with 32 cores, 17,42 with 128 cores)

⇒ we have to fight to make the last remaining percents parallel!

Non-blocking algorithms

Principle: build algorithm that “do not” block

- **Wait-free**: each operation terminates in a bounded number of steps
- **Lock-free**: if we call infinitely often an operation, the operation terminates infinitely often
(weakest than wait-free because no bound on the number of steps)
- **Obstruction-free**: at any point in the program, if a thread executes alone, it terminates its operation in a finite number of steps
(weakest than lock-free, imagine two threads that hamper each other)
- **Wait-free** => **Lock-free** => **Obstruction-free**, but the reverse is false

- *As soon as an operation takes a lock, it is not obstruction-free*

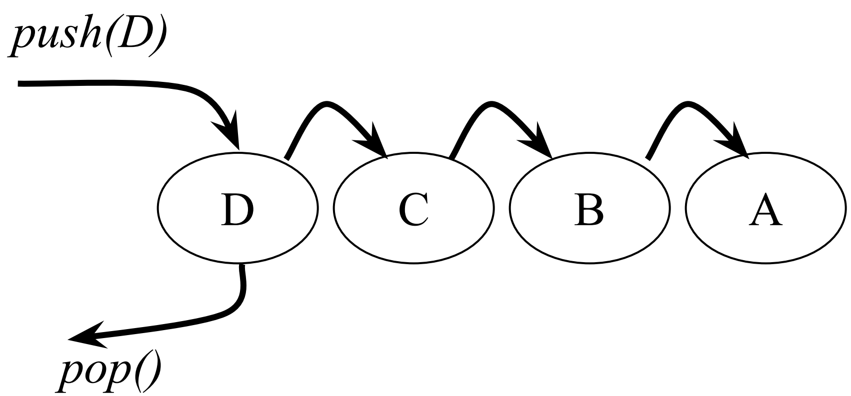
Non-blocking data structures

1. The stack
2. The queue
3. The linked list

The stack

Two operations:

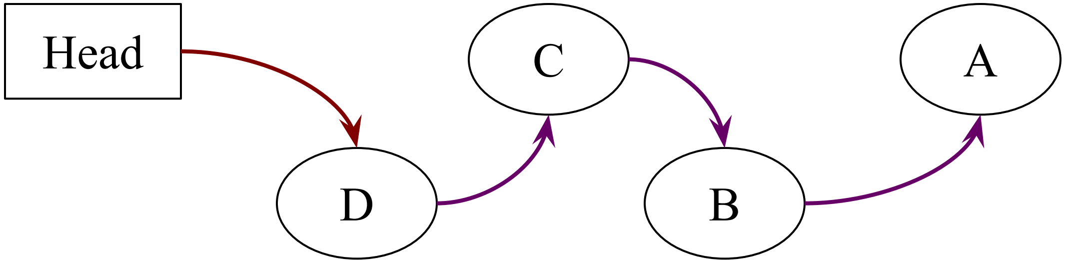
- push(Element e): push an element
- Element pop(): pop an element



A stack built with a lock

```
Class Stack {
  Node    head;
}
```

```
Class Node {
  Node    next
  Element element;
}
```



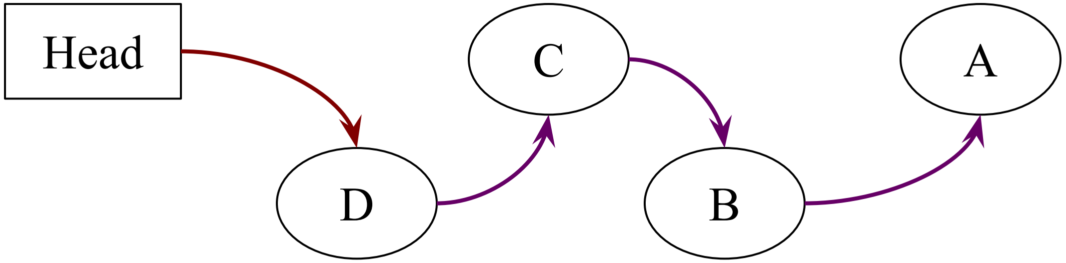
A stack built with a lock

```
Class Stack {
  Node     head;
}
```

```
Class Node {
  Node     next
  Element element;
}
```

```
sychronized void Stack.push(Element element) {
  Node n = new Node(head, element);
  head = n;
}
```

```
synchronized Element pop() {
  Node n = head;
  head = n.next;
  return n.element;
}
```

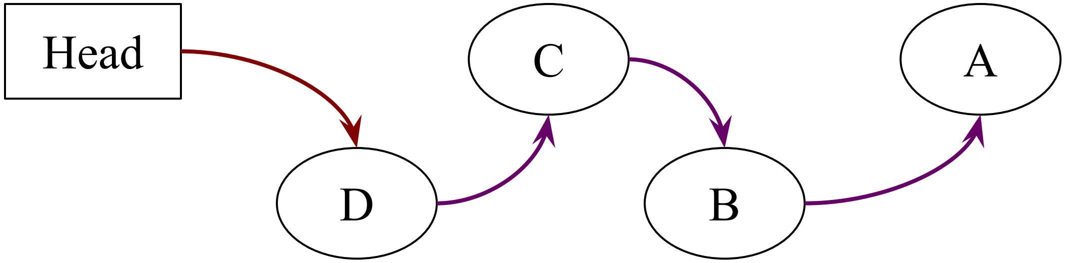


The lock-free stack (Scott'91)

```
Class Stack {
  Node    head;
}
```

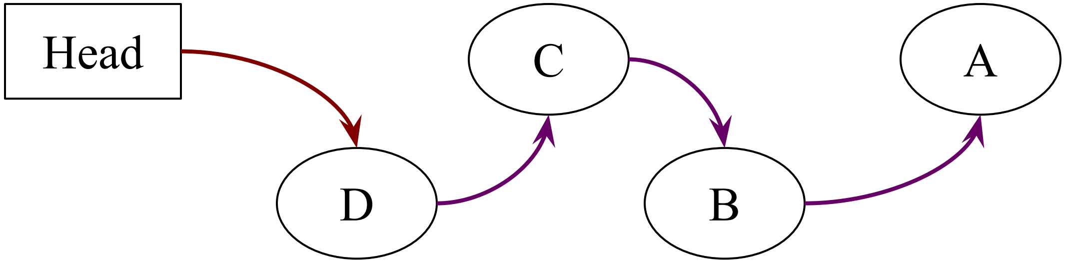
```
Class Node {
  Node    next
  Element element;
}
```

Principle: atomic compare and swap on the head



The lock-free stack (Scott'91)

```
Class Stack {                    Class Node {
  Node     head;                   Node     next
}                                  Element element;
                                 }

  void Stack.push(Element element) {
    do {
      Node n = new Node(head, element);
    } while(atomic-cas(&head, n.next, n)  != n.next);
  }
```



The lock-free stack (Scott'91)

```
Class Stack {
  Node    head;
}

Class Node {
  Node    next
  Element element;
}

void Stack.push(Element element) {
  do {
    Node n = new Node(head, element);
  } while(atomic-cas(&head, n.next, n) != n.next);
}


Element Stack.pop() {
  do {
    Node n = head;
  } while(atomic-cas(&head, n, n.next) != n);
  return n.element;
}
```

The lock-free stack (Scott'91)

Lock-free: if the threads call infinitely often push or pop, push or pop are executed infinitely often (proof: a push or a pop has to succeed to make the CAS of another push or pop fail)

Not wait-free: we can always delay a push with another push that makes the CAS fails

```
void Stack.push(Element element) {
  do {
    Node n = new Node(head, element);
  } while(atomic-cas(&head, n.next, n) != n.next);
}
Element Stack.pop() {
  do {
    Node n = head;
    if(n == null) error("No such element");
  } while(atomic-cas(&head, n, n.next) != n);
  return n.element;
}
```

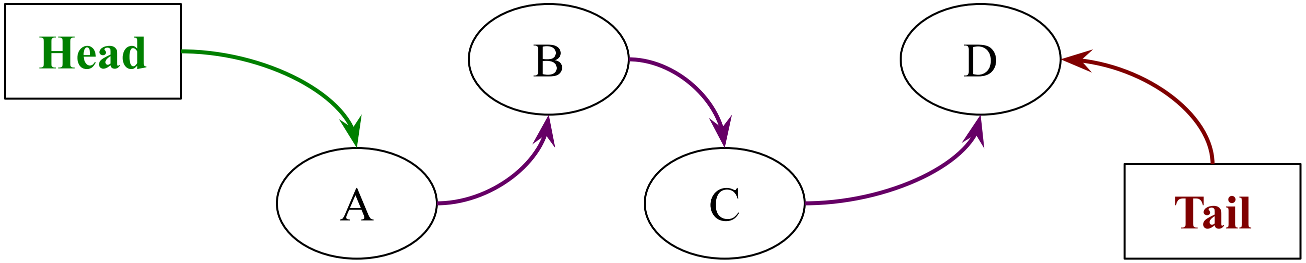
Non-blocking data structures

1. The stack
2. The queue
3. The linked list

The queue

Two operations :

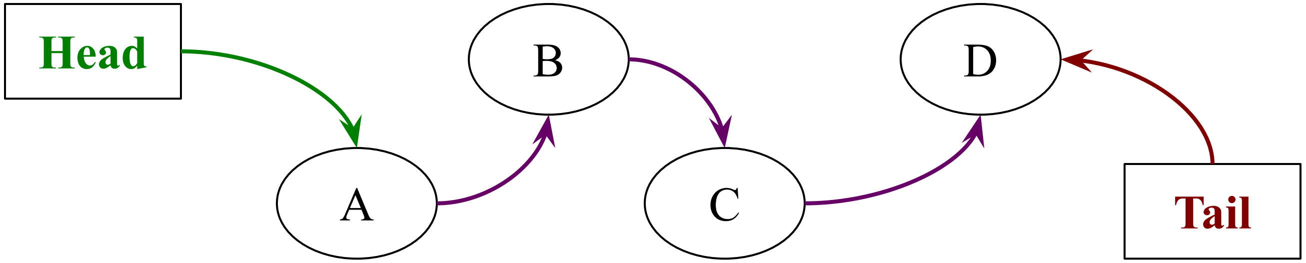
- void enqueue(Element e): adds the element at the tail of the queue
- Element dequeue(): remove the element at the head of the queue



The queue implemented with a lock

```
Class Queue {
  Node    head = null;
  Node    tail = null;
}
```

```
Class Node {
  Node    next;
  Element element;
}
```

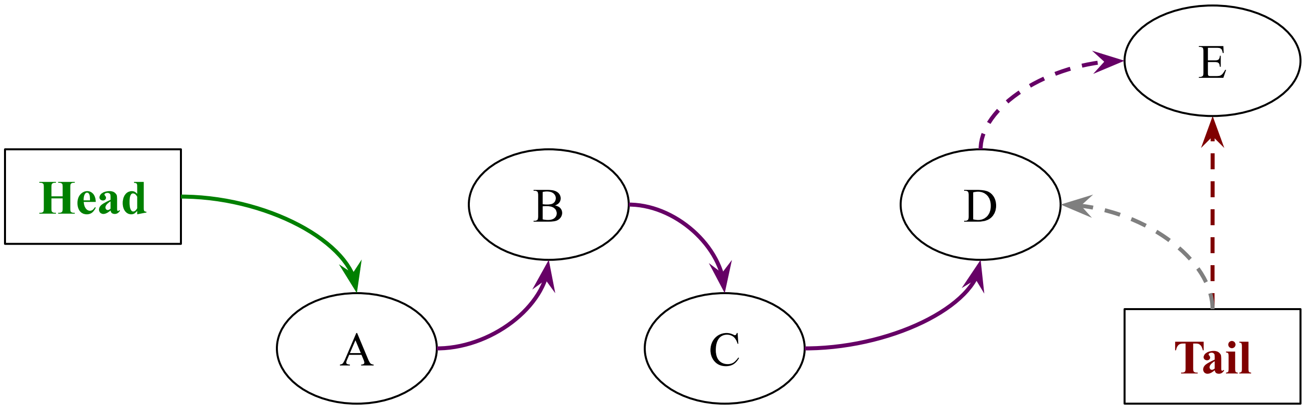


The queue implemented with a lock

```
Class Queue {
  Node     head = null;
  Node     tail = null;
}
```

```
Class Node {
  Node     next;
  Element element;
}
```

```
synchronized void Queue.enqueue(Element e) {
  Node n = new Node(null, e);
  if(tail != null) { tail.next = n; }
  else { head = n; }
  tail = n;
}
```

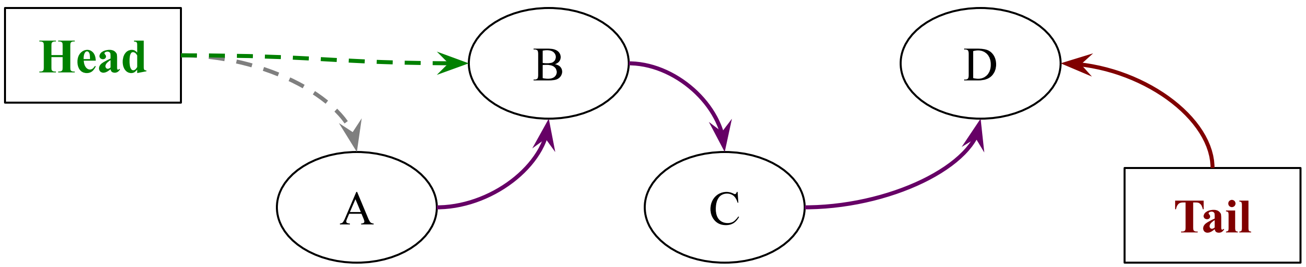


The queue implemented with a lock

```
Class Queue {
  Node    head = null;
  Node    tail = null;
}
```

```
Class Node {
  Node    next;
  Element element;
}
```

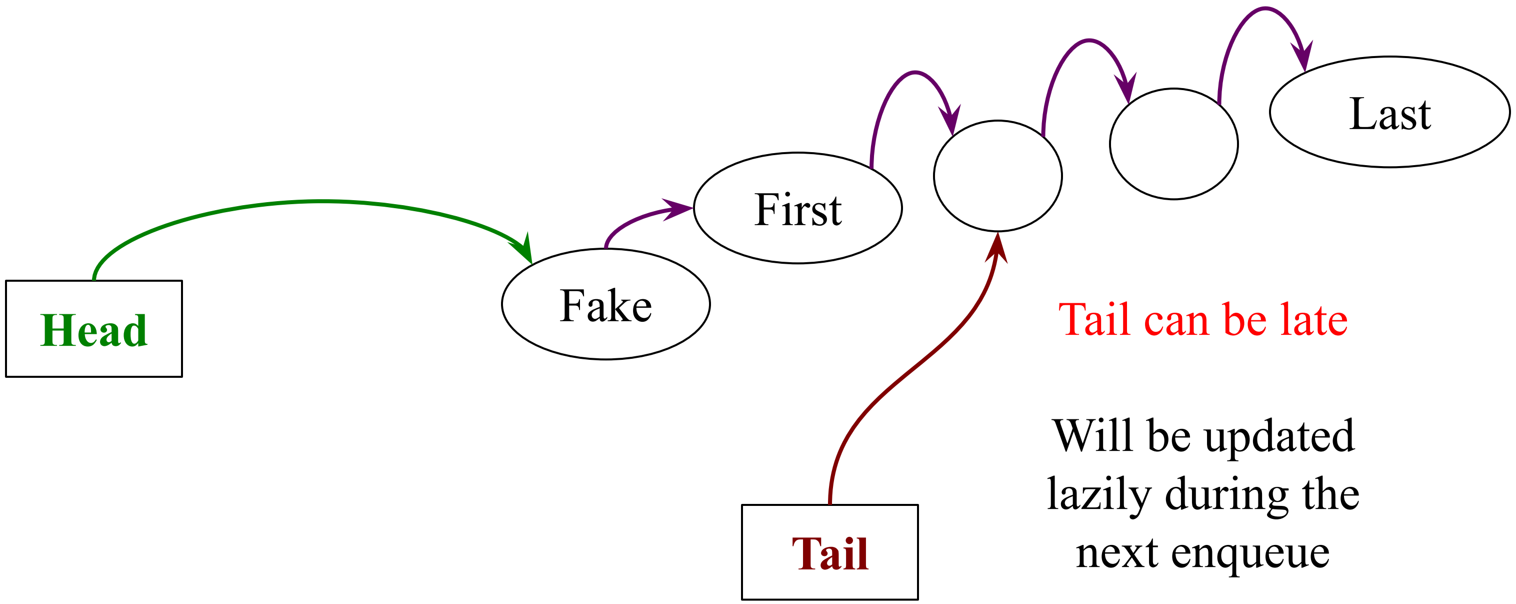
```
synchronized Element Queue.dequeue() {
  Node n = head;
  head = n.next;
  if(head == null) tail = null;
  return n.element;
}
```



The lock-free queue

Principles:

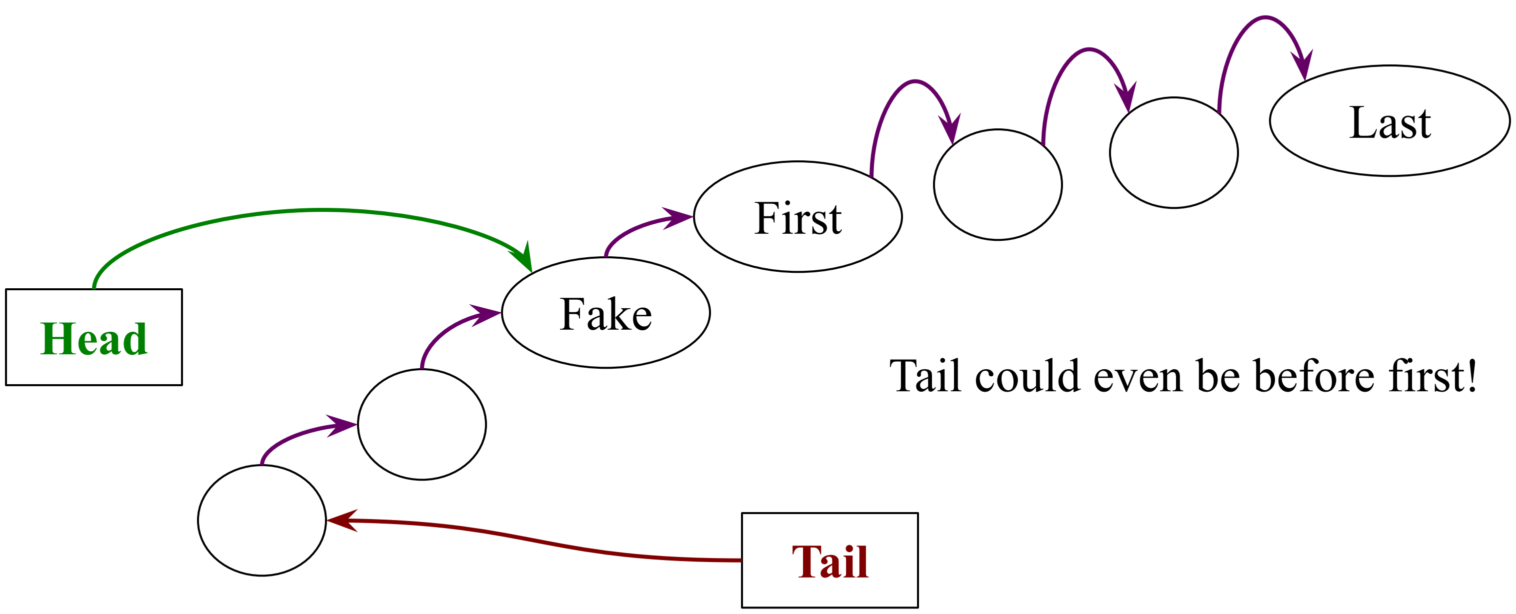
- Always keep a fake node in the list to simplify the initialization
- Update tail lazily
- Invariants: at each step
  - The node after the head, if it exists is the **first** enqueued node
  - The lists that start with head and tail have at least a common node
  - The last node of these lists is the **last** enqueued node



The lock-free queue

Principles:

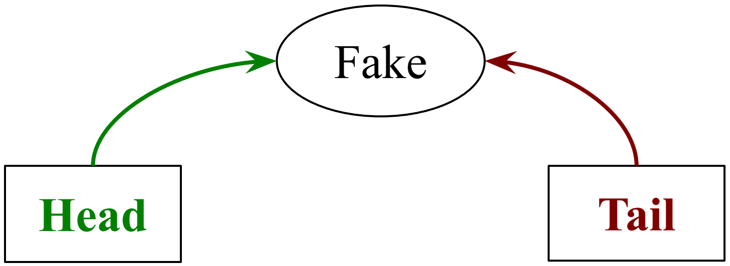
- Always keep a fake node in the list to simplify the initialization
- Update tail lazily
- Invariants: at each step
  - The node after the head, if it exists is the **first** enqueued node
  - The lists that start with head and tail have at least a common node
  - The last node of these lists is the **last** enqueued node



The lock-free queue

```
Class Queue {
  Node     head = new Node(null, null);
  Node     tail = head;
}
```

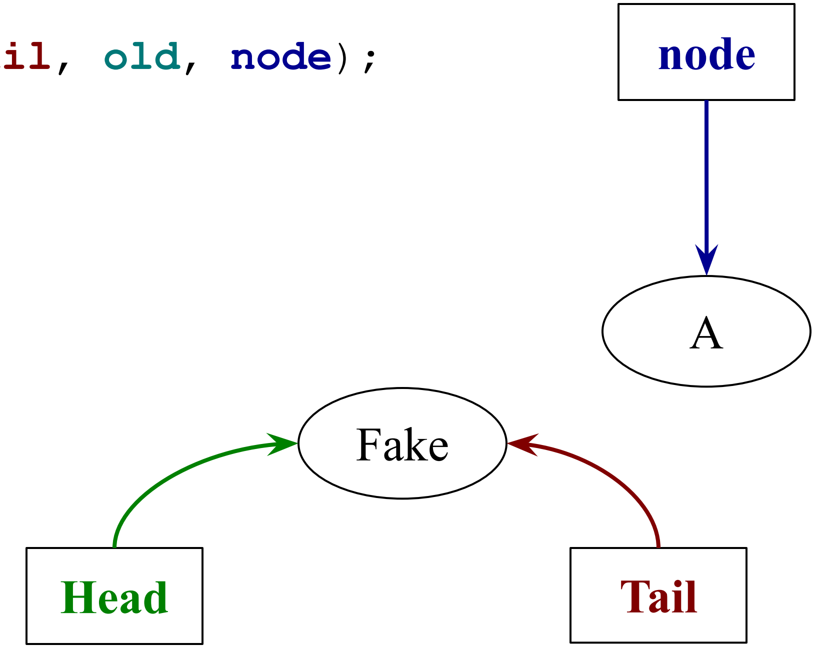
```
Class Node {
  Node     next;
  Element element;
}
```



The lock-free queue: enqueue

Enqueue A

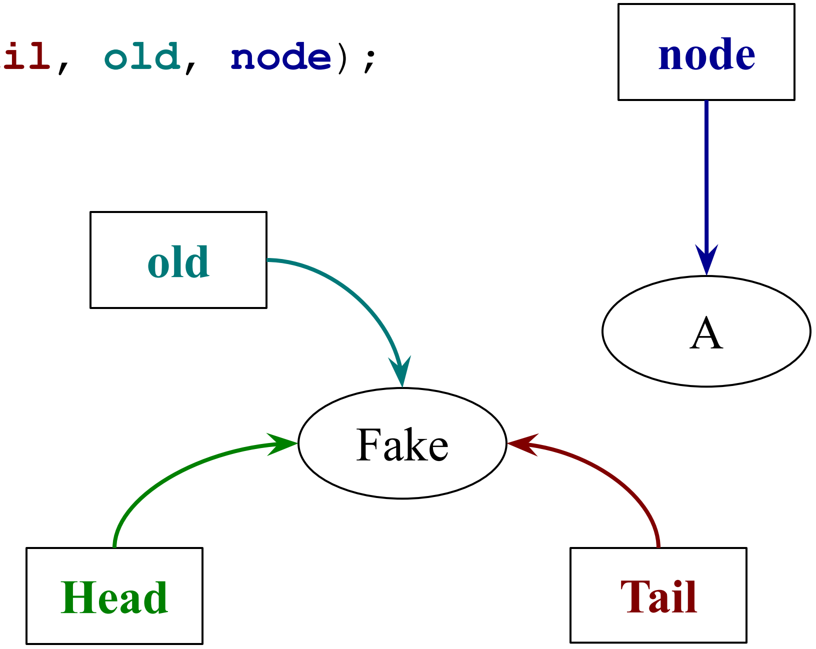
```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```



The lock-free queue: enqueue

Enqueue A

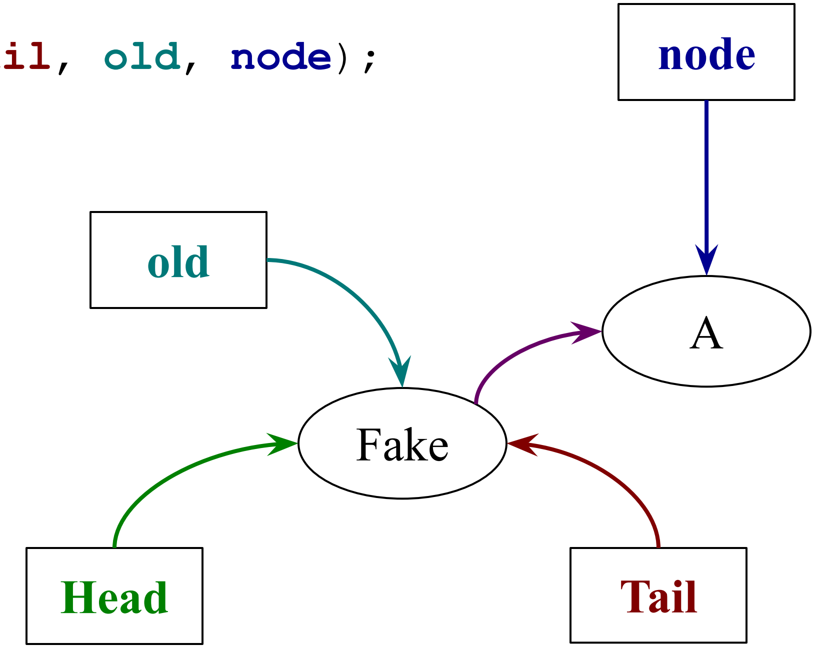
```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```



The lock-free queue: enqueue

Enqueue A

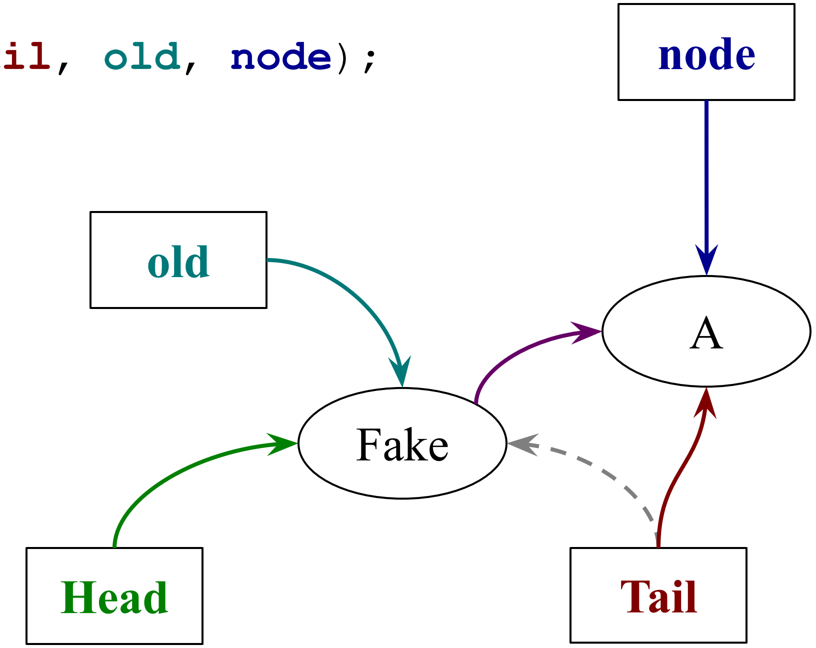
```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```



The lock-free queue: enqueue

Enqueue A

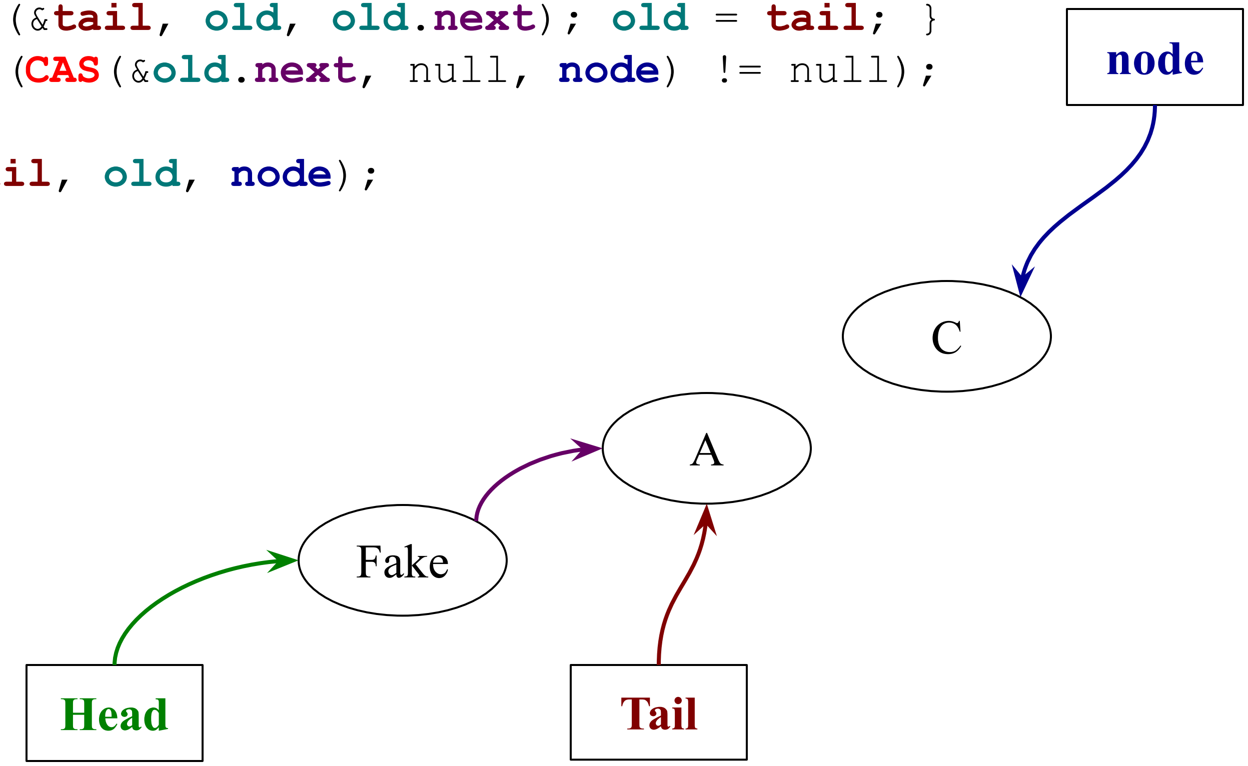
```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```



The lock-free queue: enqueue

Enqueue C and B in parallel

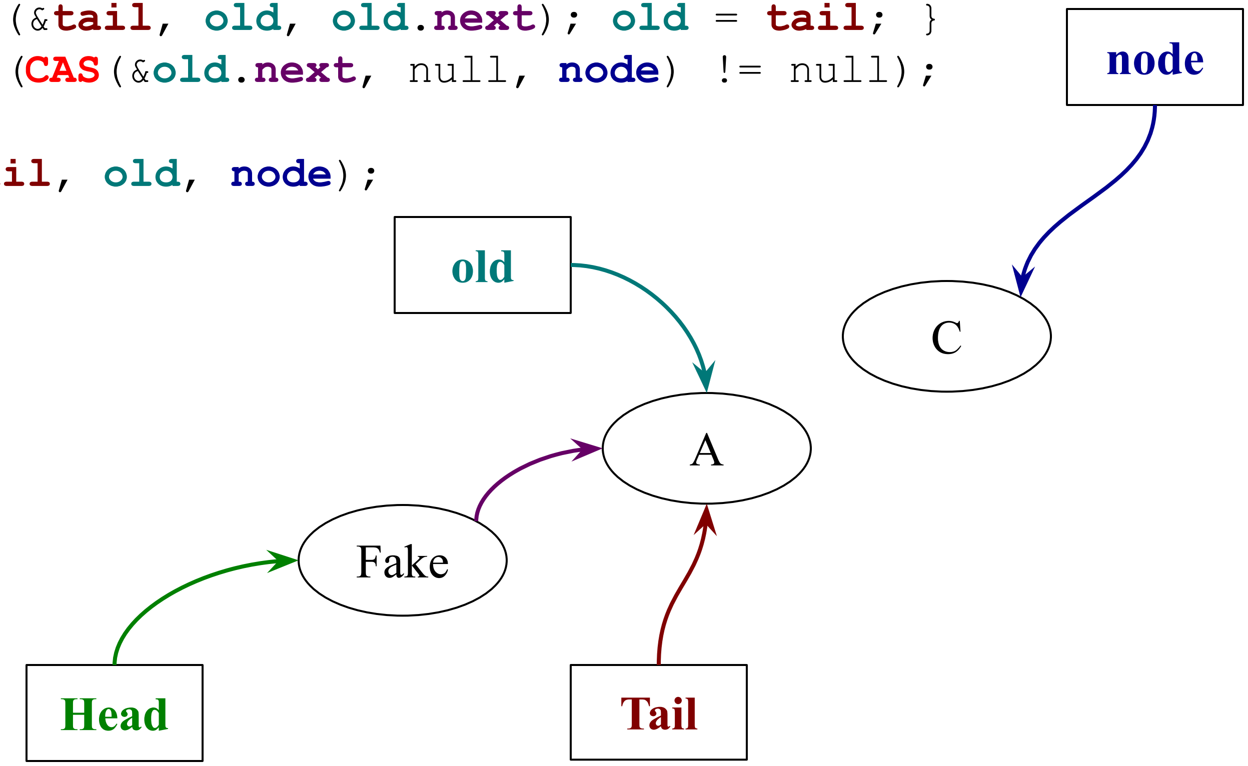
```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```



The lock-free queue: enqueue

Enqueue C and B in parallel

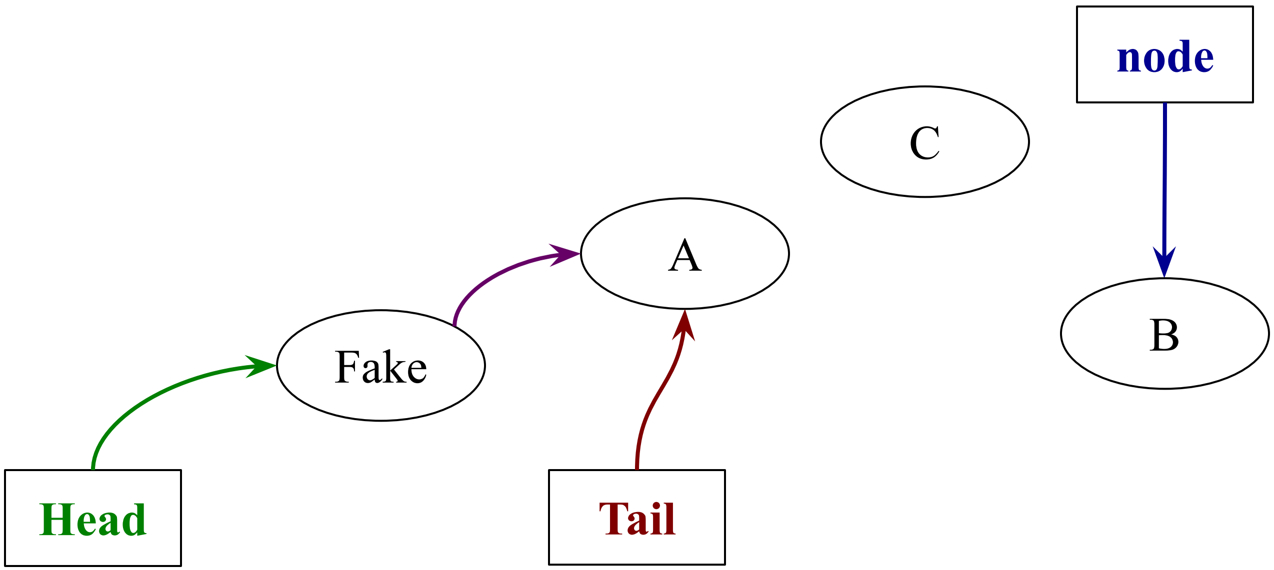
```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```



The lock-free queue: enqueue

Enqueue C and B in parallel

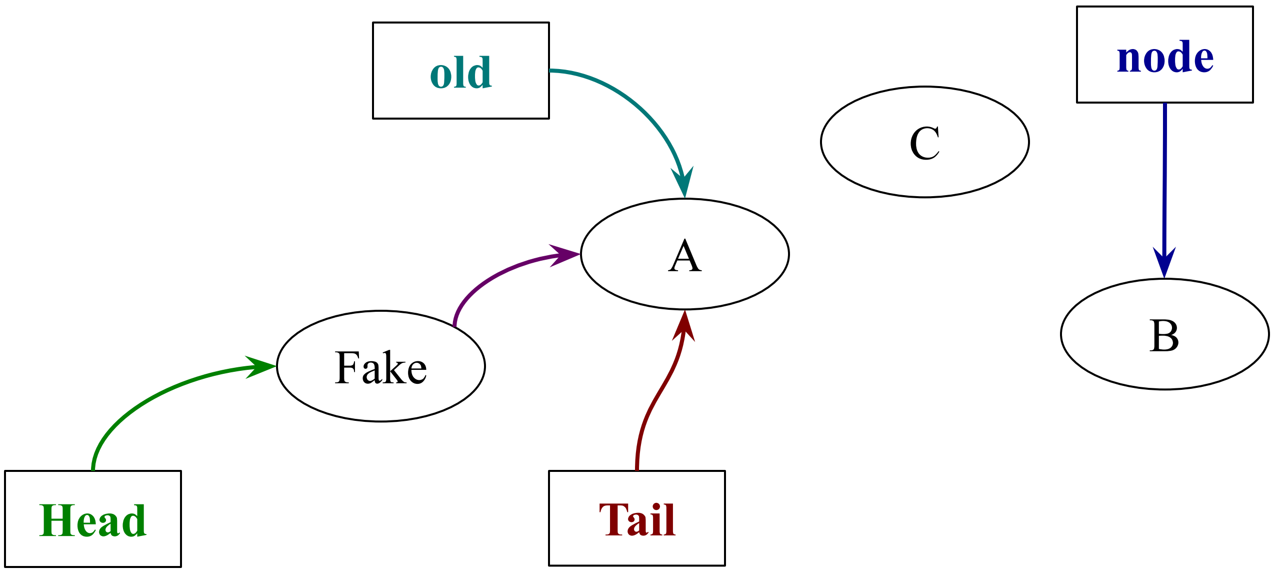
```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```



The lock-free queue: enqueue

Enqueue C and B in parallel

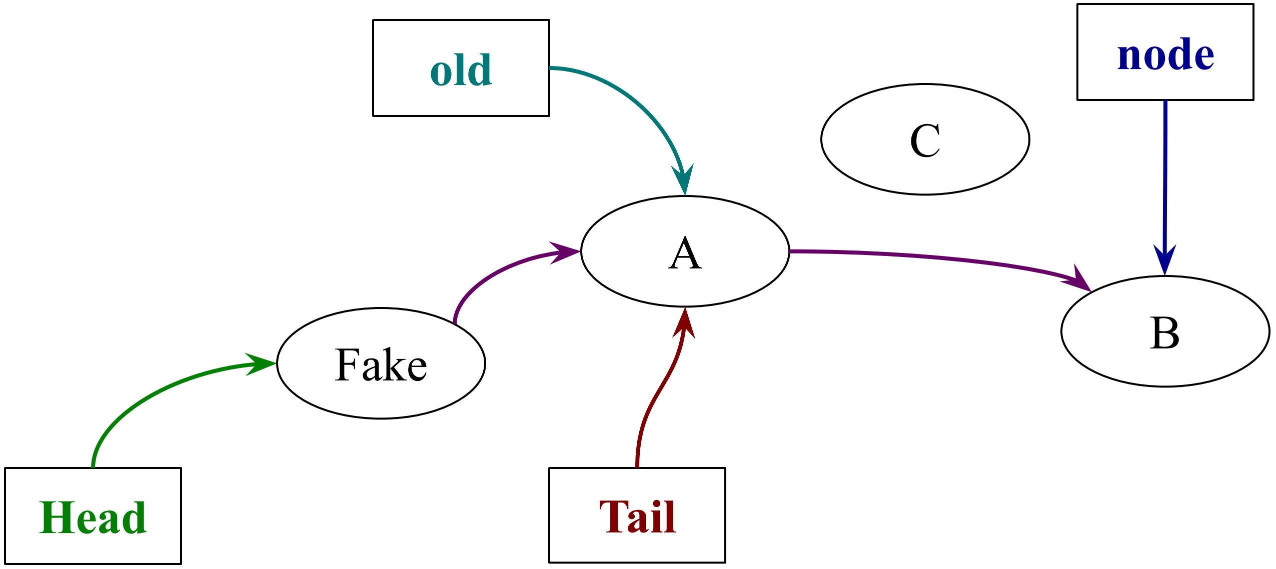
```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```



The lock-free queue: enqueue

Enqueue C and B in parallel

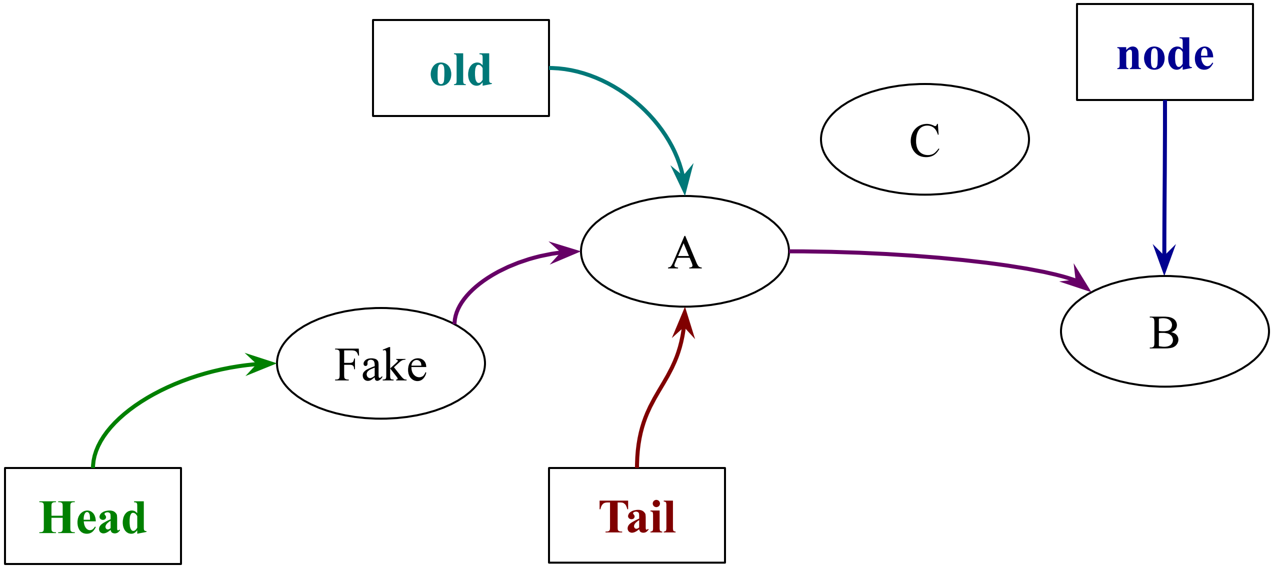
```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```



The lock-free queue: enqueue

Enqueue C and B in parallel

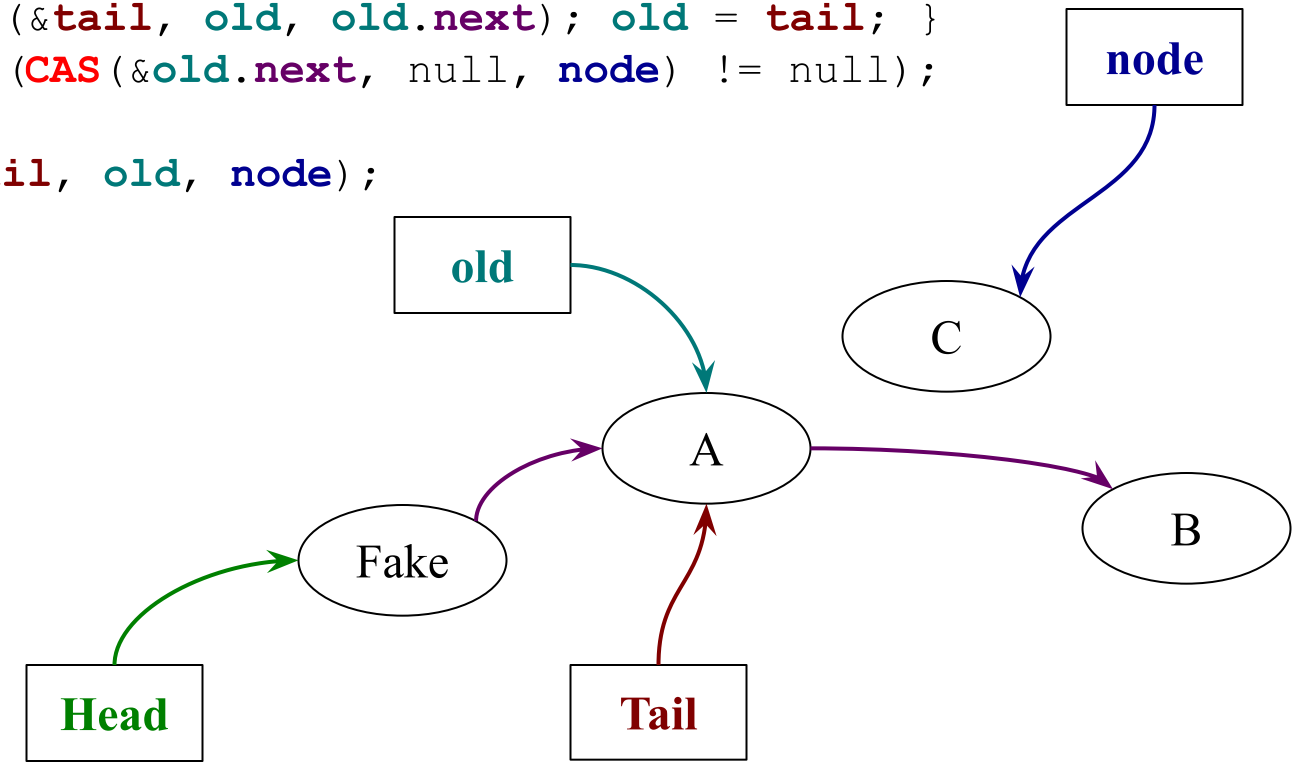
```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```



The lock-free queue: enqueue

Enqueue C and B in parallel

```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```

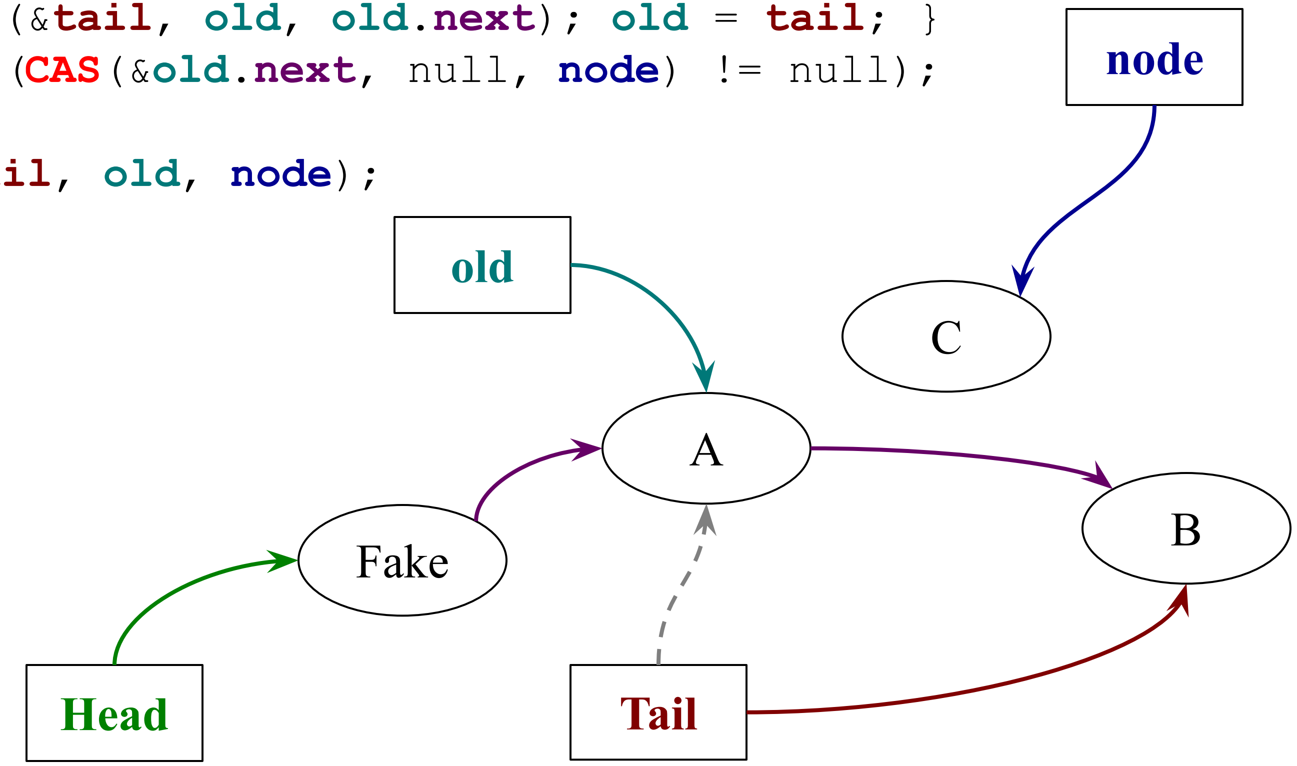


The lock-free queue: enqueue

Enqueue C and B in parallel

```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```

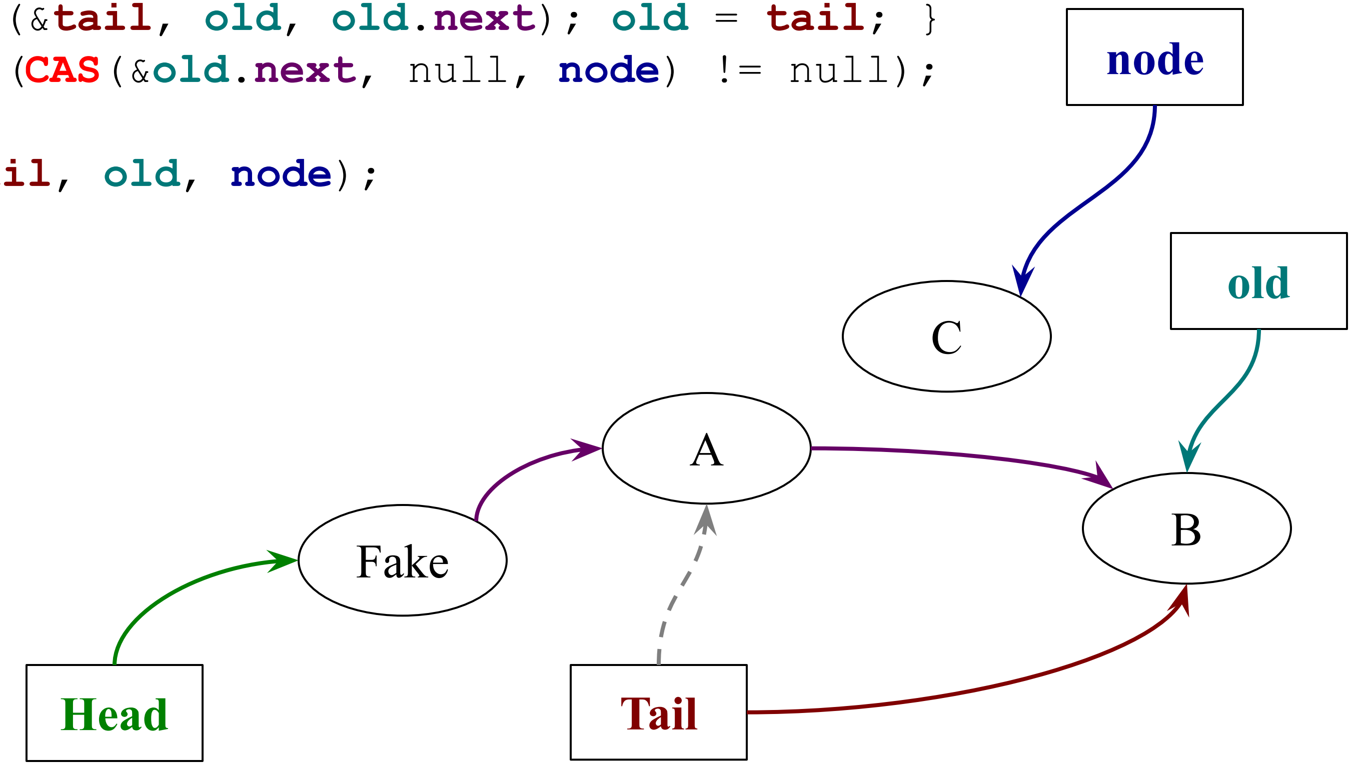
C pend en charge l'avancement de tail pour B



The lock-free queue: enqueue

Enqueue C and B in parallel

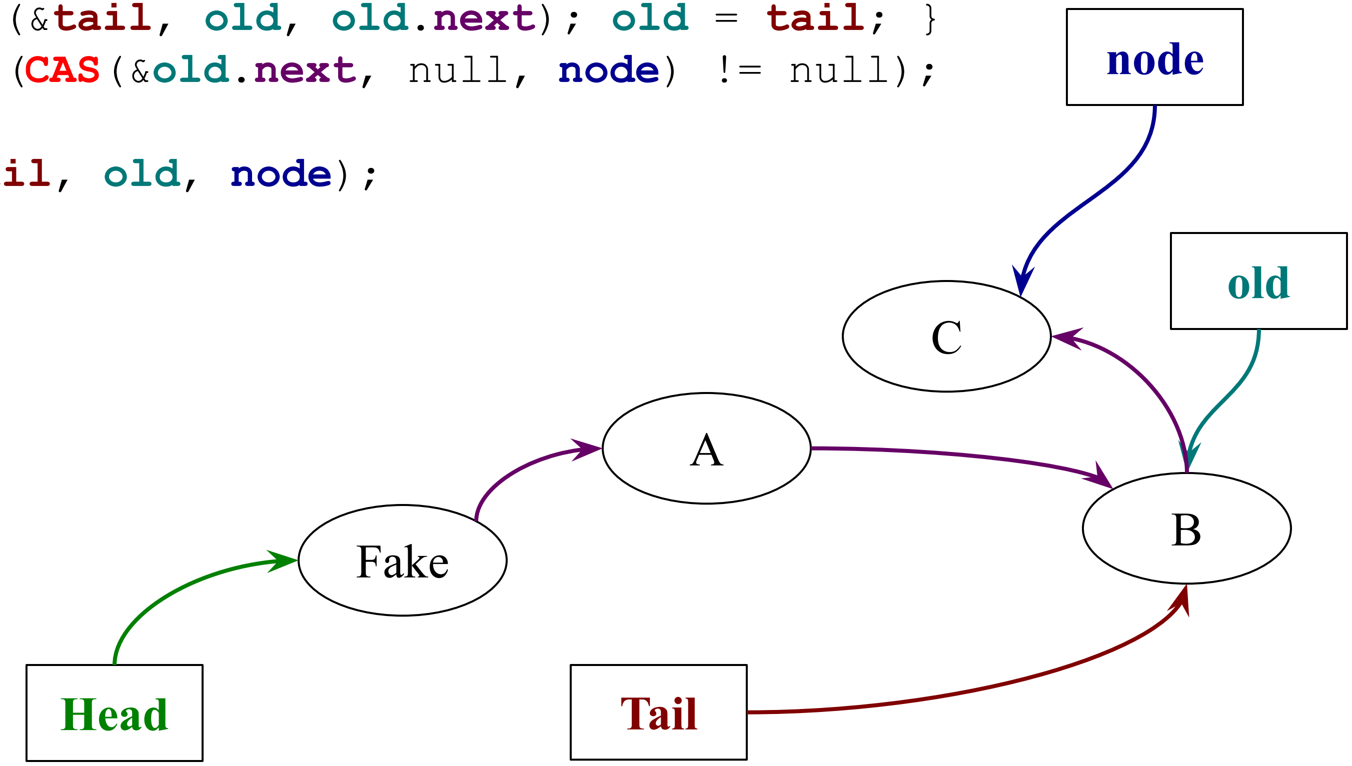
```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```



The lock-free queue: enqueue

Enqueue C and B in parallel

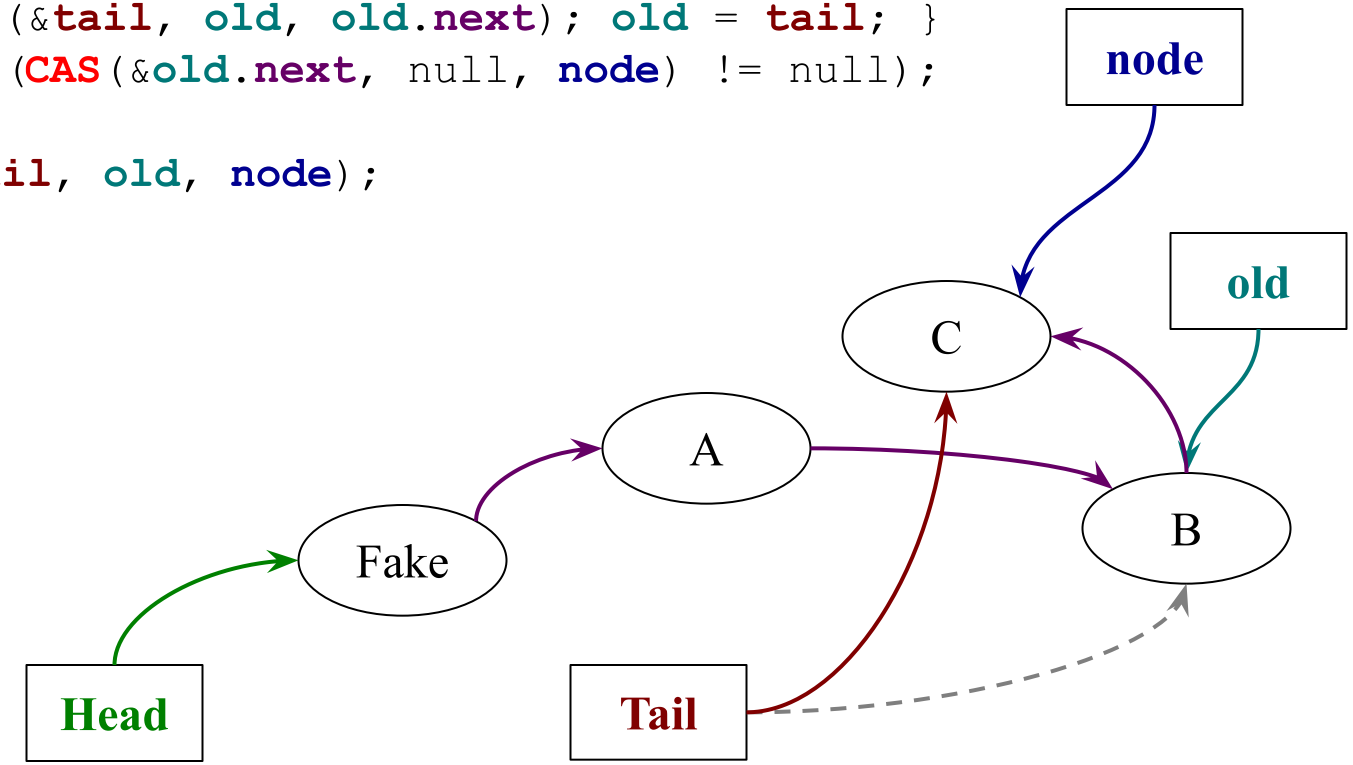
```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```



The lock-free queue: enqueue

Enqueue C and B in parallel

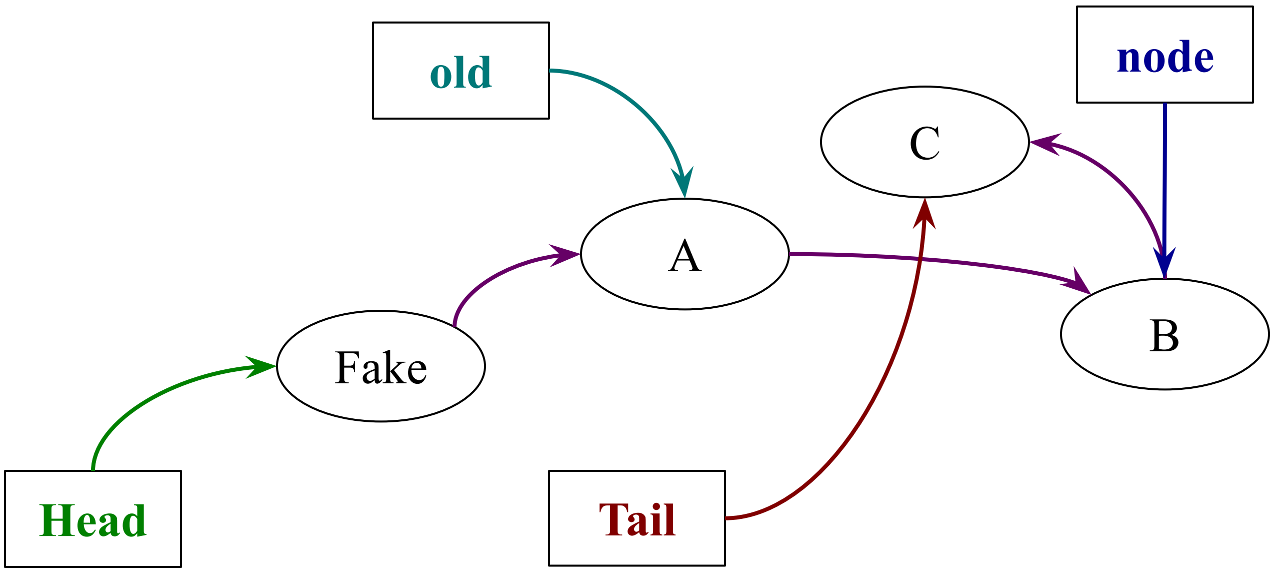
```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```



The lock-free queue: enqueue

Enqueue C and B in parallel

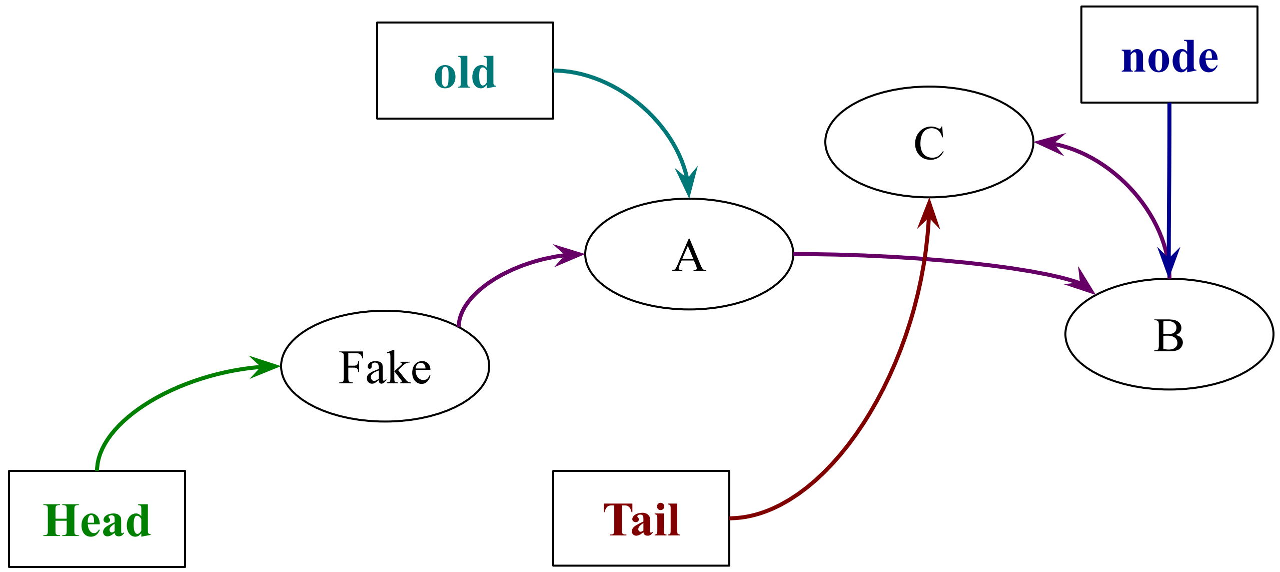
```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

  CAS(&tail, old, node);
}
```



The lock-free queue: enqueue

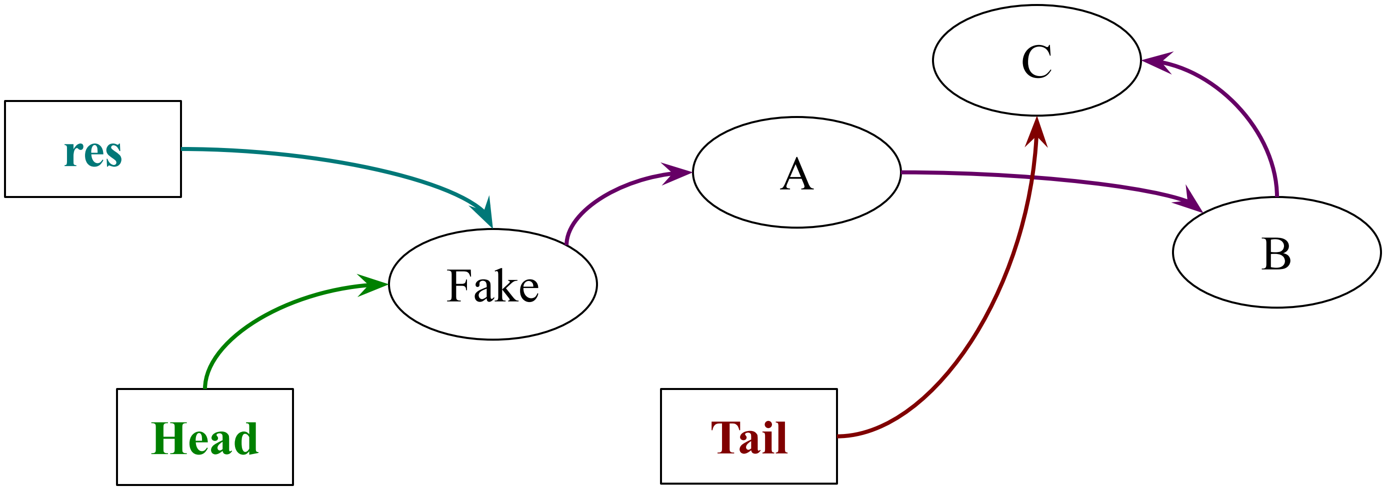
Enqueue C and B in parallel

```
void Queue.enqueue(Element e) {
  Node node = new Node(null, e);
  do {
    Node old = tail;
    while(old.next != NULL) {
      CAS(&tail, old, old.next); old = tail; }
  } while(CAS(&old.next, null, node) != null);

→ CAS(&tail, old, node);
}
```



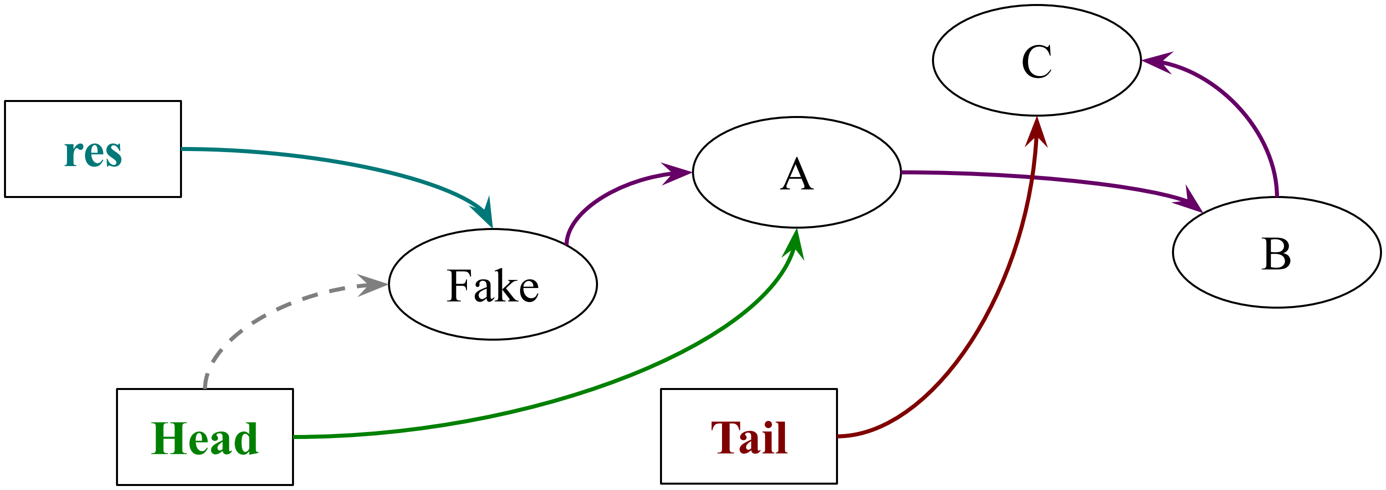
The lock-free queue: dequeue

```
Element Queue.dequeue() {
  do {
    Node res = head;
    if(res.next == null) return null;
  } while(CAS(&head, res, res.next) != res);

  return res.next.value;
}
```



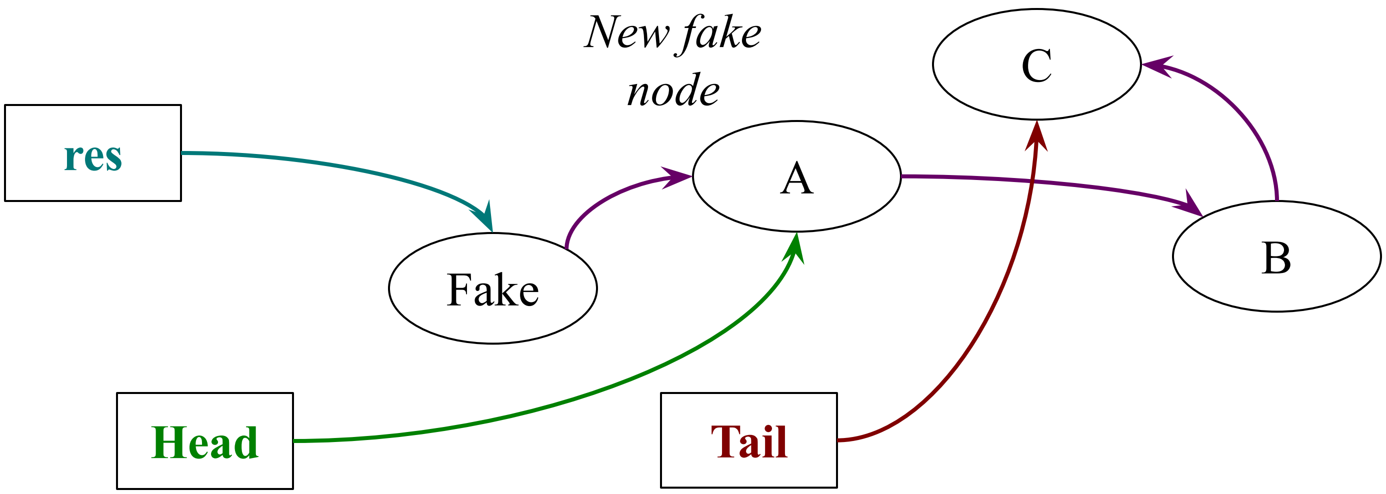
The lock-free queue: dequeue

```
Element Queue.dequeue() {
  do {
    Node res = head;
    if(res.next == null) return null;
  } while(CAS(&head, res, res.next) != res);

  return res.next.value;
}
```



The lock-free queue: dequeue

```
Element Queue.dequeue() {
  do {
    Node res = head;
    if(res.next == null) return null;
  } while(CAS(&head, res, res.next) != res);

→ return res.next.value;
}
```



The lock-free queue

Lock-free: if the threads call infinitely often enqueue or dequeue, enqueue or dequeue are executed infinitely often (proof: an enqueue or a queue has to succeed to make the CAS of another enqueue or dequeue fail)

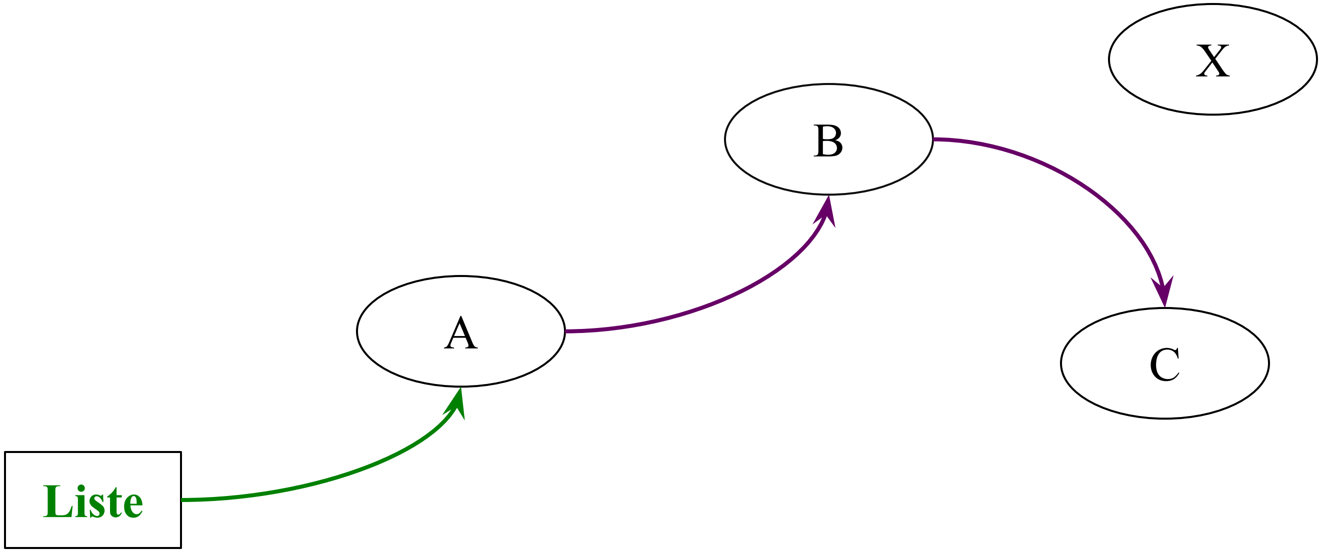
Not wait-free: we can always delay an enqueue with another enqueue that makes the CAS fails

Non-blocking data structures

1. The stack
2. The queue
3. The linked list

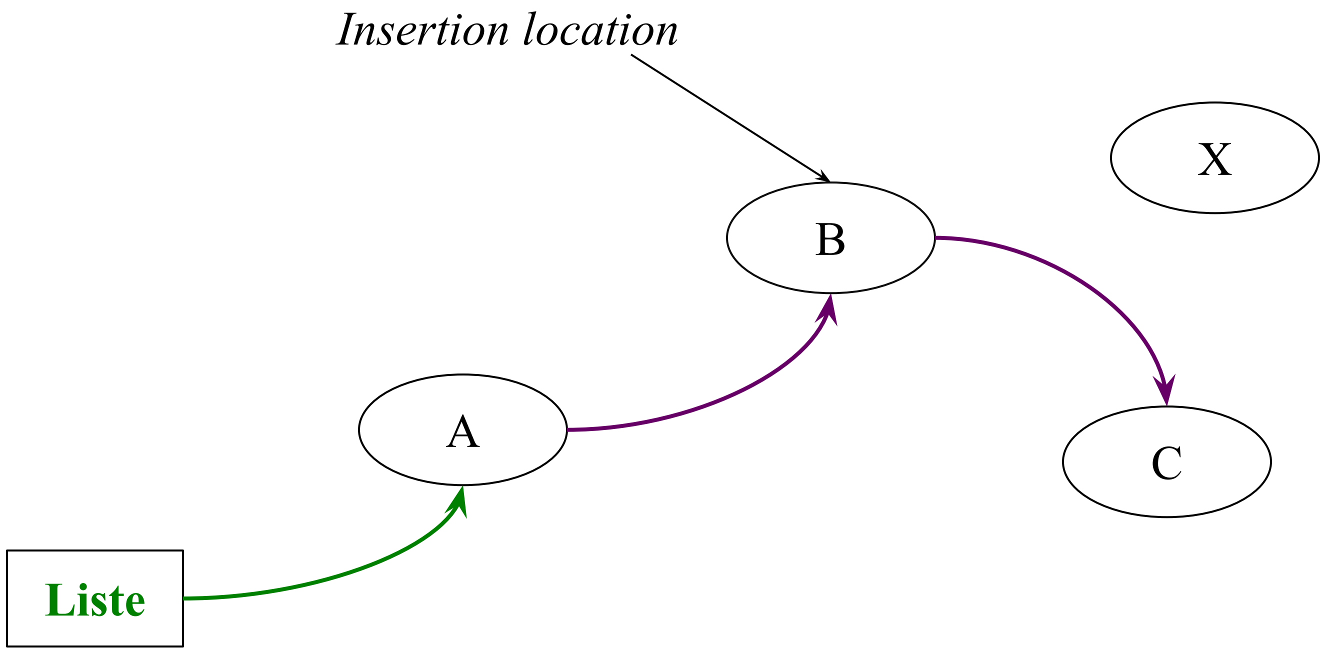
The linked list

Main problem: insert and remove at the same place



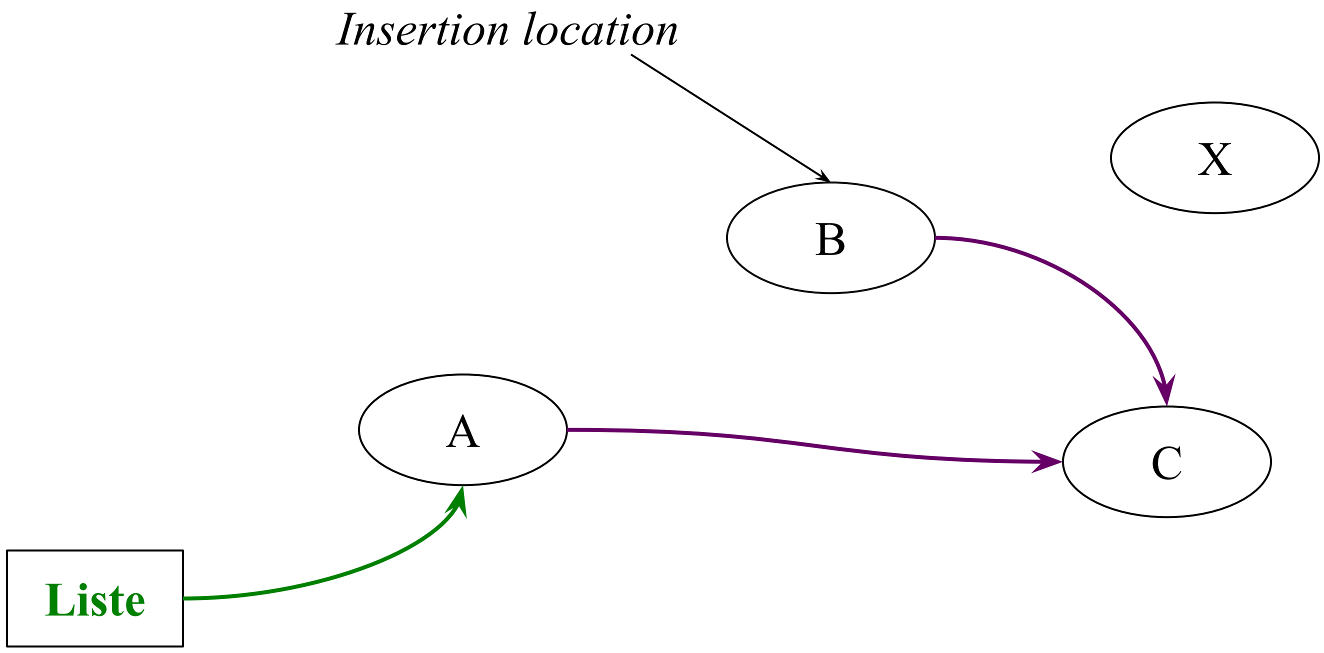
The linked list

Main problem: insert and remove at the same place



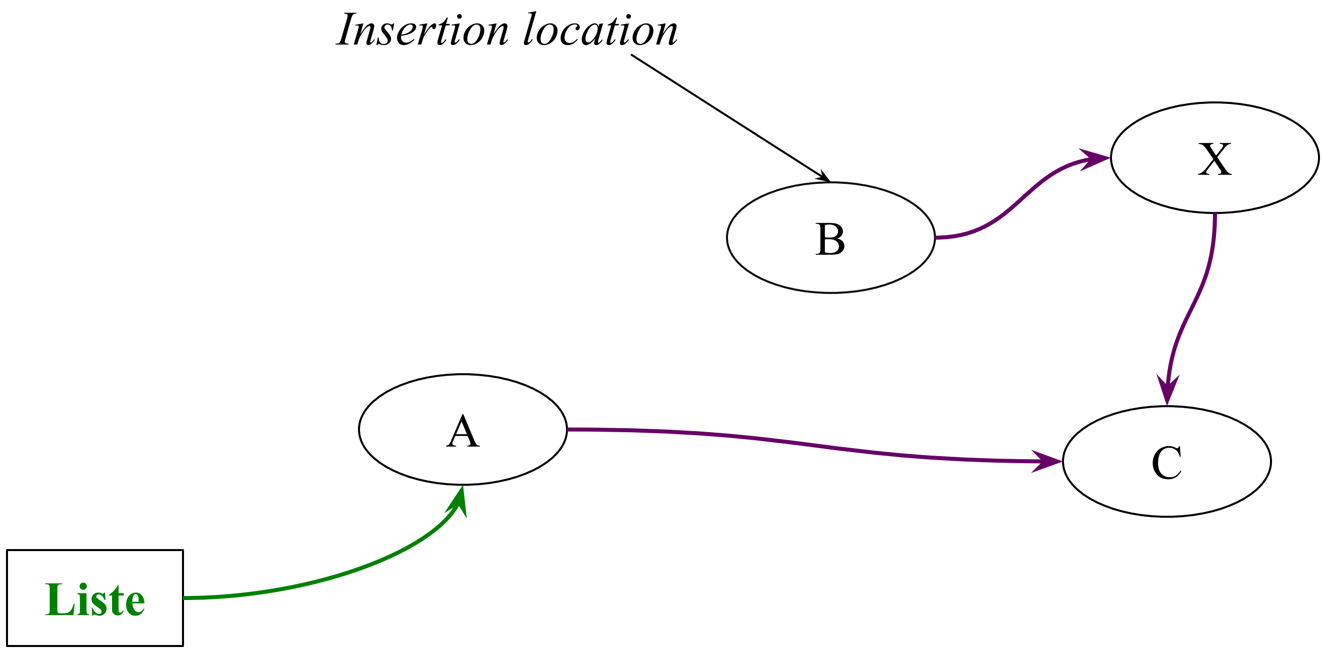
The linked list

Main problem: insert and remove at the same place



The linked list

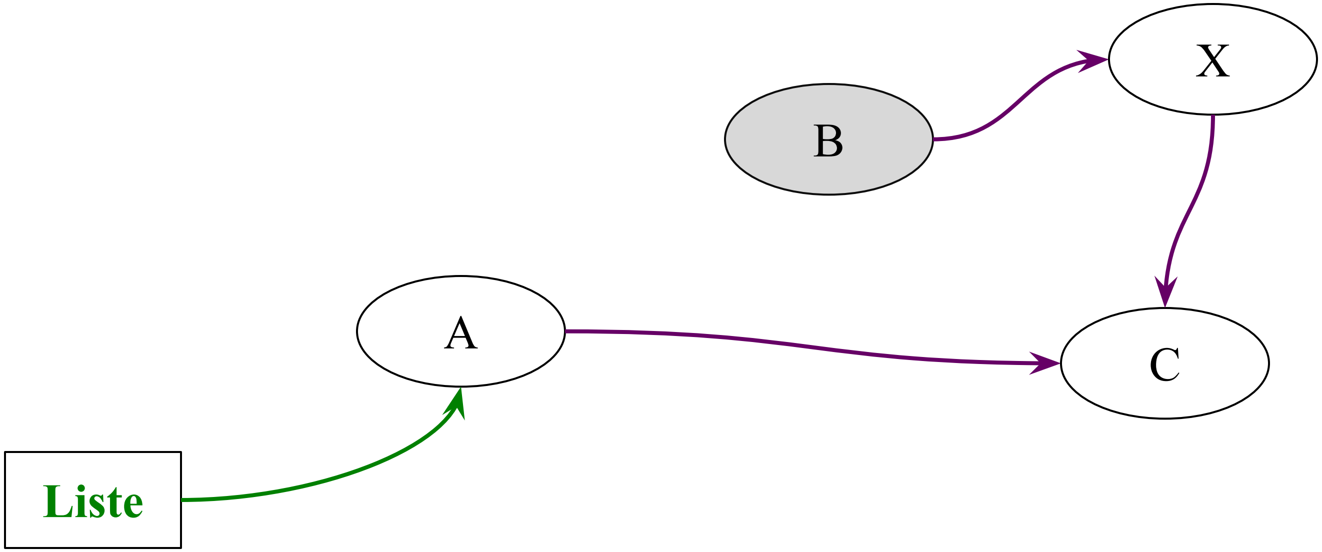
Main problem: insert and remove at the same place



The linked list

Principle (Tim Harris, DISC 2001)

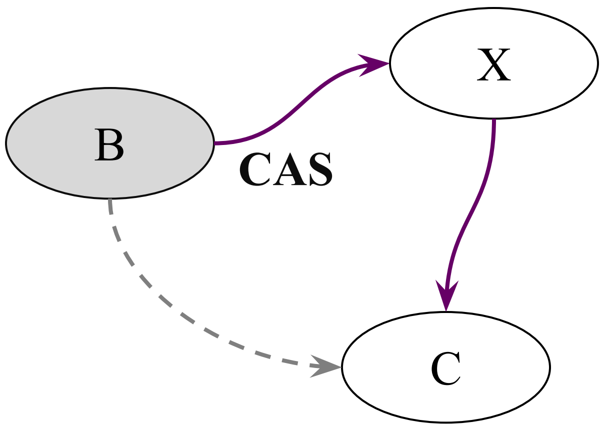
- Each node has a color
- A white node is present, a grey node is deleted
- If the color of a node changes during an insertion, restart



The linked list

Principle (Tim Harris, DISC 2001)

- Each node has a color
- A white node is present, a grey node is deleted
- If the color of a node changes during an insertion, restart



Problem:

- If we CAS the next pointer, we can not see if a color changes

Solution:

- Embeds the color in the next pointer

The linked list

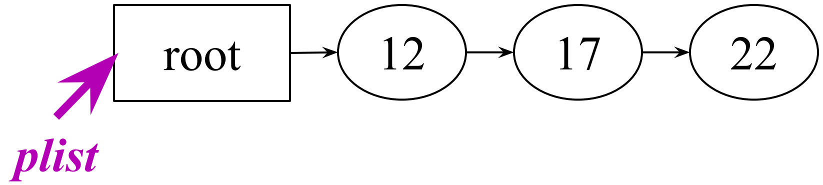
```
typedef uintptr_t coloredPointer;

Node pointer(coloredPointer ptr) { return (Node)(ptr & -2); }
int mark(coloredPointer ptr) { return ptr & 1; }
```

```
Class Node {
  coloredPointer next;
  Element        element;
};
```

- Ideas:
  - In order to delete a node, marks it grey (modify its coloredPointer)
  - When a thread find a deleted node during an insert or a delete, try to remove it from the list (garbage collect)

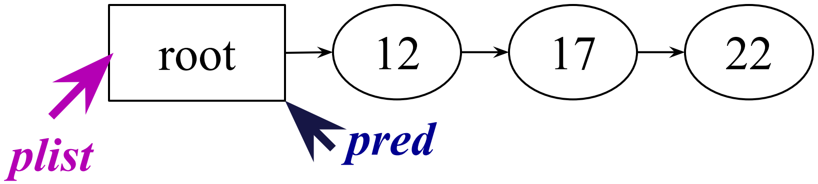
Delete: the traversal

```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);






    pred = &cur->next;
  } }
```



Delete: the traversal

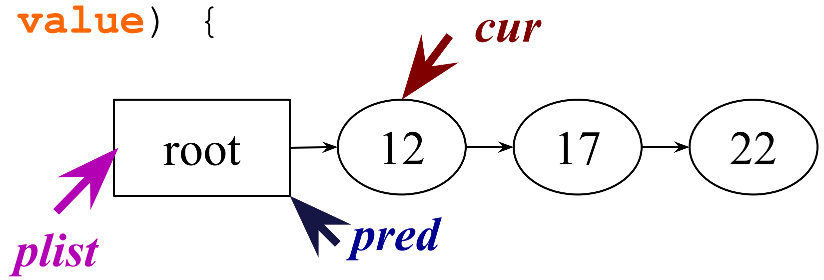
```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```




```
    pred = &cur->next;
  } }
```

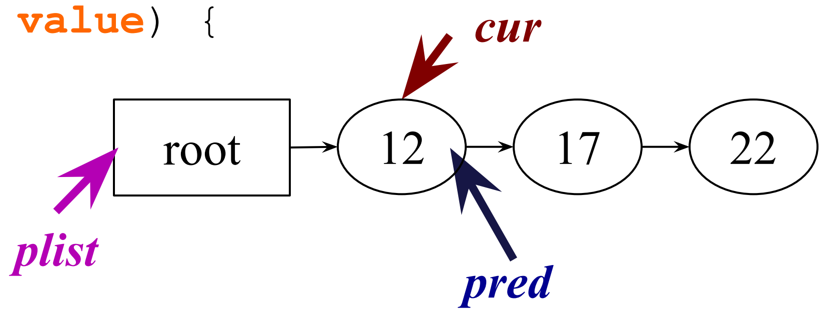
Delete: the traversal

```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
➤   Node cur = pointer(*pred);







    pred = &cur->next;
  } }
```



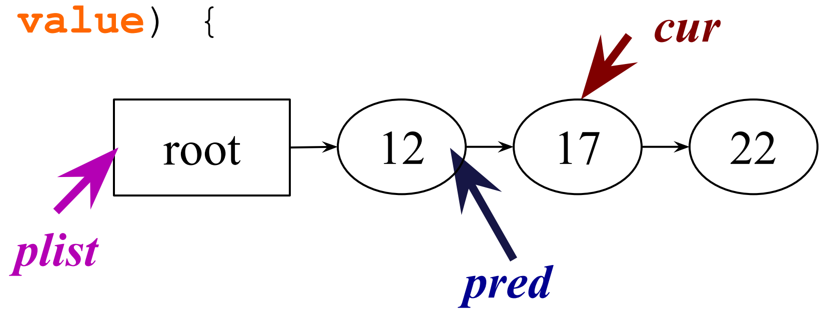
Delete: the traversal

```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);




    pred = &cur->next;
  } }
```



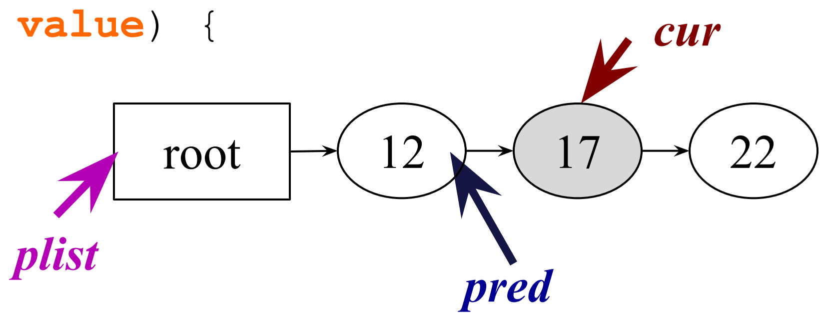
Delete: the traversal

```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
➤   Node cur = pointer(*pred);











    pred = &cur->next;
  } }
```



Delete: remove the deleted nodes

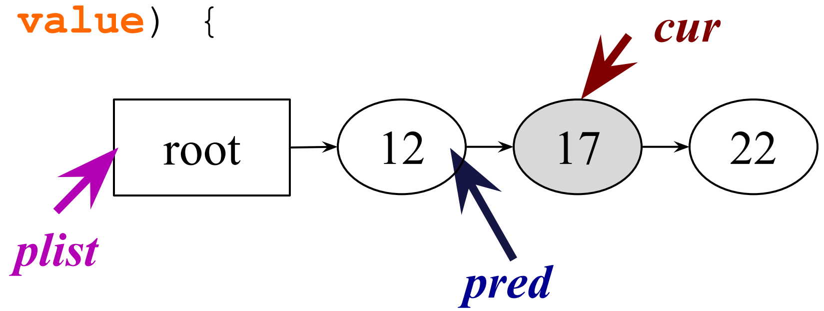
```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
➤   Node cur = pointer(*pred);

    if(mark(cur->next)) {                 /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```



Principle: opportunistically remove the deleted nodes

Delete: remove the deleted nodes

```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```

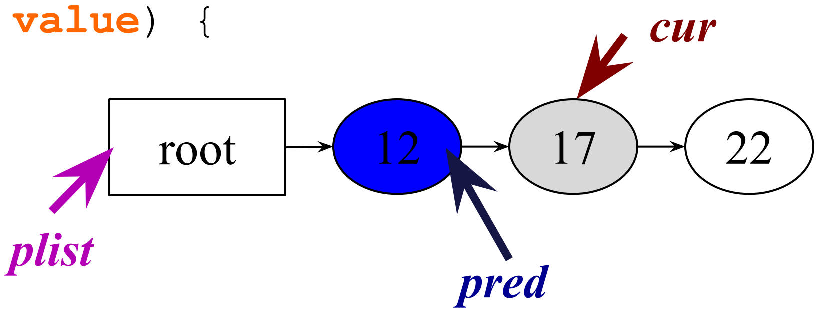


Principle: opportunistically remove the deleted nodes

```
➤   if(mark(cur->next)) {                    /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```

Delete: remove the deleted nodes

```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



Principle: opportunistically remove the deleted nodes

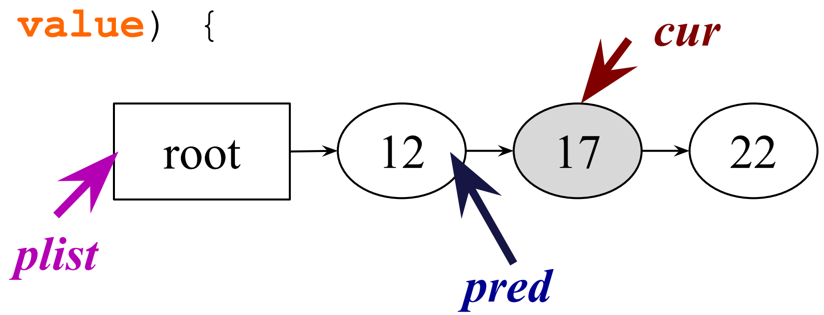
```
    if(mark(cur->next)) {                  /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```

The CAS can fail because

- pred is concurrently marked grey
- A new node is inserted after pred

Delete: remove the deleted nodes

```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



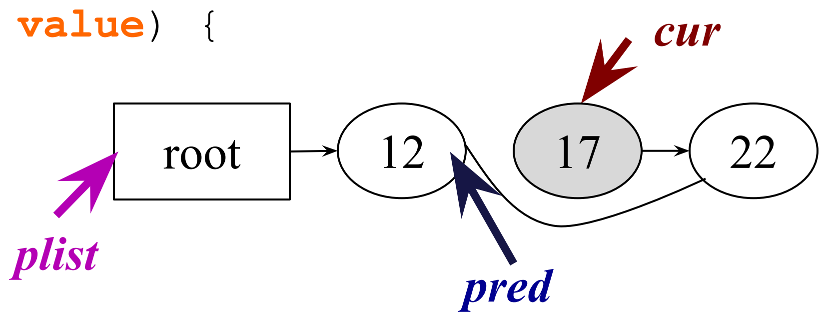
Principle: opportunistically remove the deleted nodes

```
    if(mark(cur->next)) {                /* cur is deleted */
➤     if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```

Suppose that the CAS succeeds

Delete: remove the deleted nodes

```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```

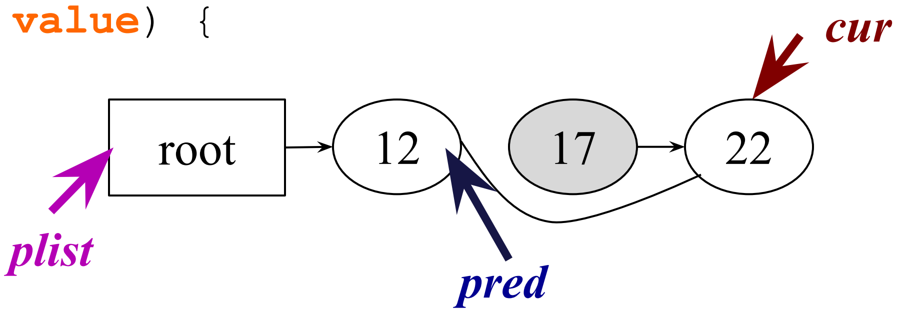


Principle: opportunistically remove the deleted nodes

```
    if(mark(cur->next)) {                    /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```

Delete: remove the deleted nodes

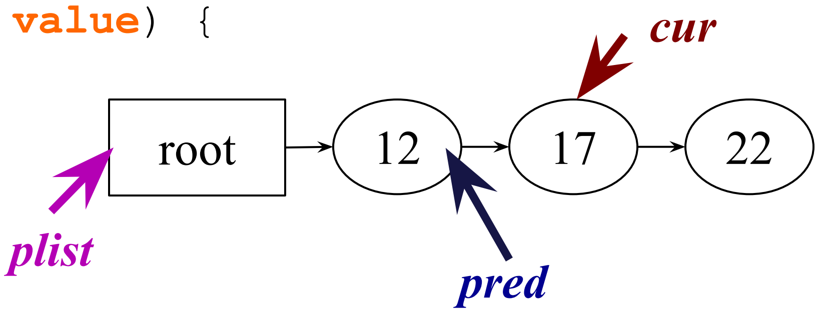
```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



Principle: opportunistically remove the deleted nodes

```
    if(mark(cur->next)) {                /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```

Delete: marks a node deleted



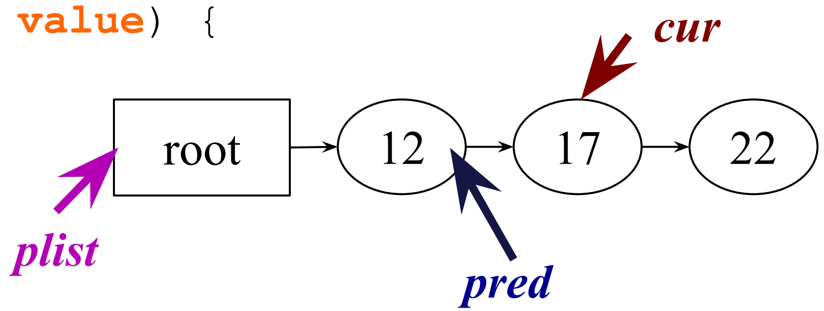

```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
➤   Node cur = pointer(*pred);
```

We suppose value = 17

```
    if(cur->value == value) {                /* found! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1; }

    if(mark(cur->next)) {                    /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```

Delete: marks a node deleted

```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```

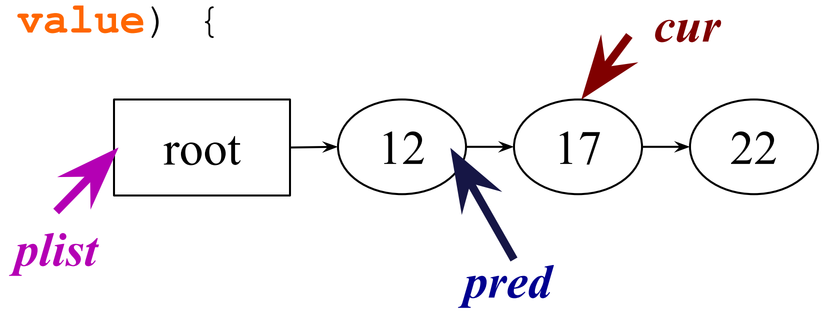


We suppose value = 17

```
    if(cur->value == value) {                 /* found! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1; }

    if(mark(cur->next)) {                     /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```

Delete: marks a node deleted

```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



We suppose value = 17

```
    if(cur->value == value) {                    /* found! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1; }

    if(mark(cur->next)) {                        /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```

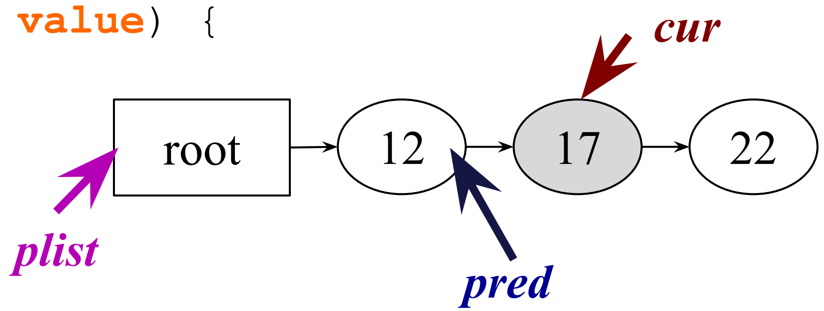
The CAS may fail because of

node 17 is already removed by another thread

an insert between 17 and 22

Delete: marks a node deleted

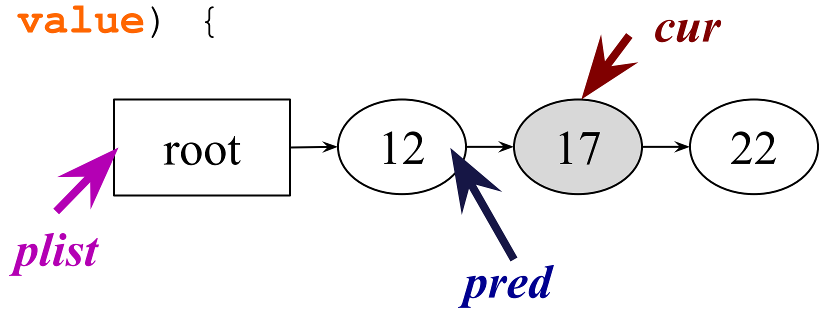
```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);

    if(cur->value == value) {                  /* found! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1; }

    if(mark(cur->next)) {                      /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```



We suppose value = 17

Delete: marks a node deleted

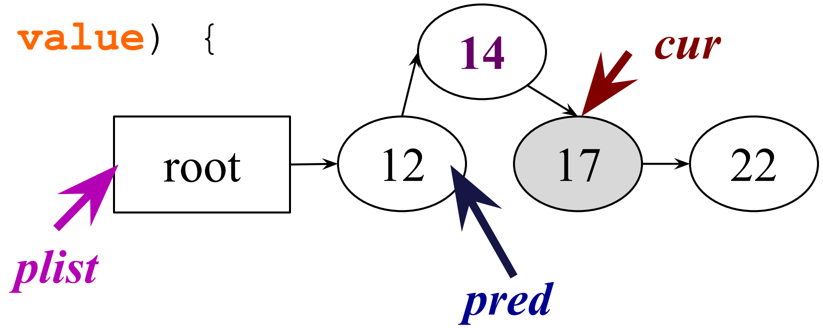
```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



We suppose value = 17

```
    if(cur->value == value) {                /* found! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1; }

➤   if(mark(cur->next)) {                    /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```

Delete: marks a node deleted



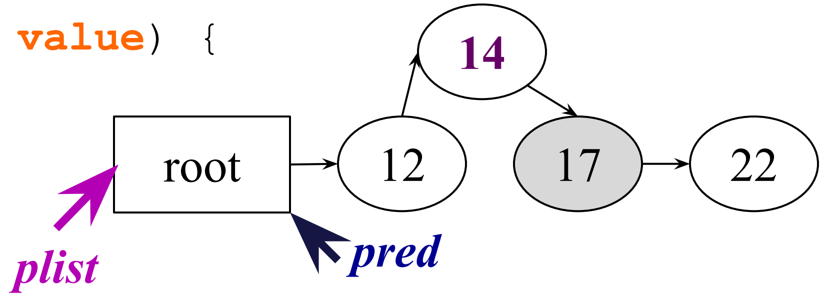

```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```

We suppose value = 17

```
    if(cur->value == value) {                    /* found! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1; }

    if(mark(cur->next)) {                        /* cur is deleted */
➤     if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```

Imagine that another thread inserts the node 14 concurrently
=> goto restart

Delete: marks a node deleted




```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```

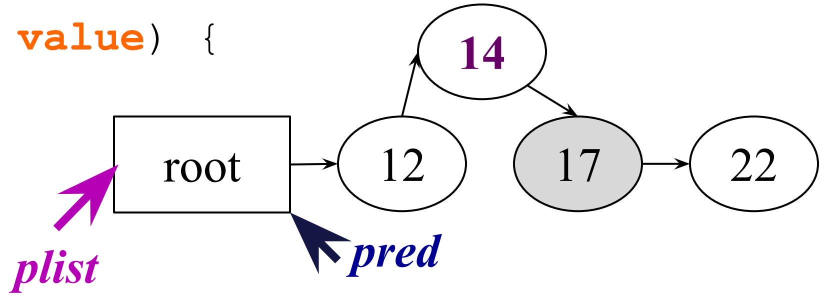
We suppose value = 17

```
    if(cur->value == value) {                /* found! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1; }

    if(mark(cur->next)) {                    /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```

Imagine that another thread inserts the node 14 concurrently
=> goto restart
=> not removed from the list since found = 1

Delete: marks a node deleted

```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



We suppose value = 17

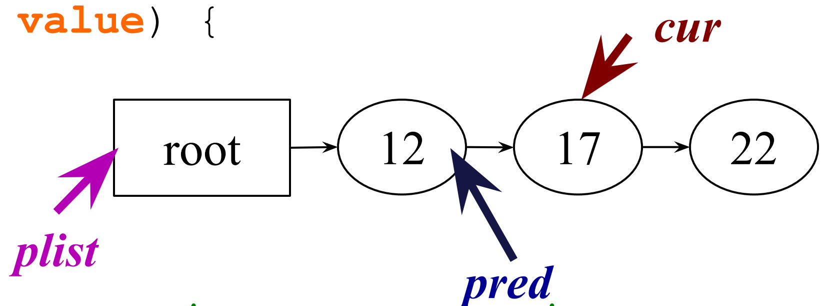
```
    if(cur->value == value) {                /* found! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1; }

    if(mark(cur->next)) {
      if(CAS(pred, cur, pointe
      else continue;
    }
    pred = &cur->next;
  } }
```

Not a problem because the next thread that will traverse the list will remove the node 17!

=> not removed from the list since found = 1

Delete: not found

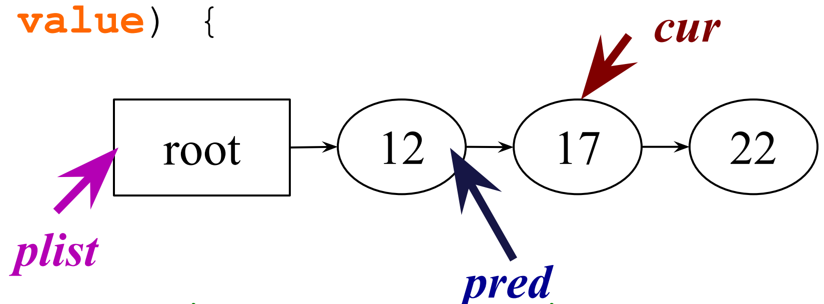
```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
    if(cur == null || value < cur->value) /* not found */
      return 0;

    if(cur->value == value) {               /* found! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1; }

    if(mark(cur->next)) {                   /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```



We suppose value = 13

Delete: not found

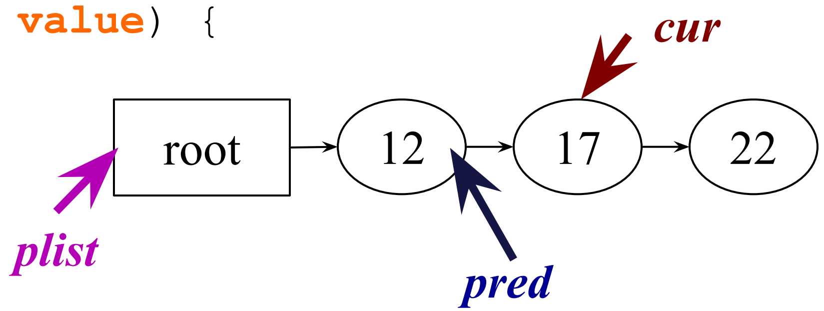
```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
    if(cur == null || value < cur->value) /* not found */
      return 0;

    if(cur->value == value) {             /* found! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1; }

    if(mark(cur->next)) {                 /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  } }
```



We suppose value = 13 => leave the function when cur reaches the node 17 (13 < 17)

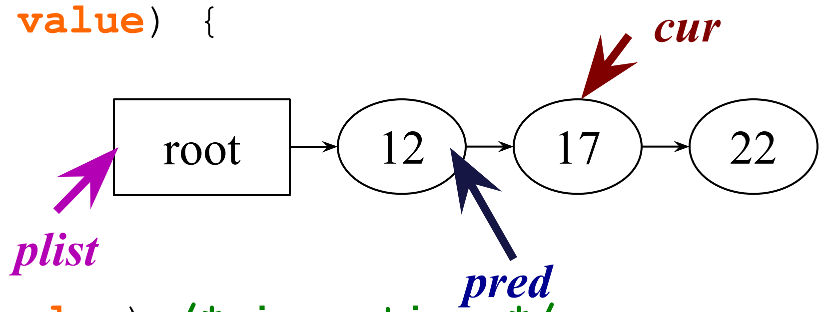
Add: the traversal

```
void add(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(true) {
    Node cur = pointer(*pred);



    if(mark(cur->next)) {                     /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }

    pred = &cur->next;
  }
}
```



Insert: same principle, we remove the deleted node during the traversal

Add: the insertion

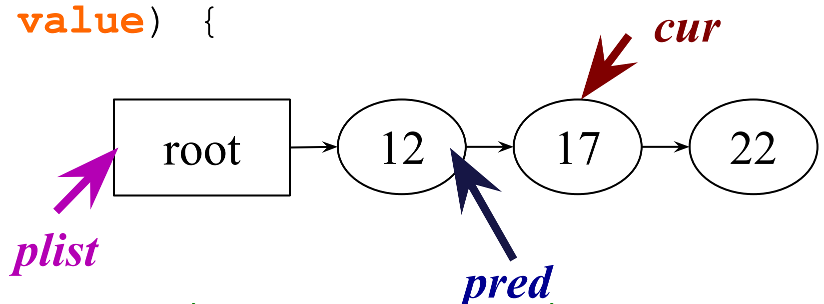



```
void add(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(true) {
    Node cur = pointer(*pred);
    if(cur == null || value < cur->value) /* insertion */
      if(CAS(pred, cur, new Node(cur, value)) != cur) goto restart;
      else return;

    if(mark(cur->next)) {                    /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }

    pred = &cur->next;
  }
}
```

Exemple: value = 14

Add: the insertion

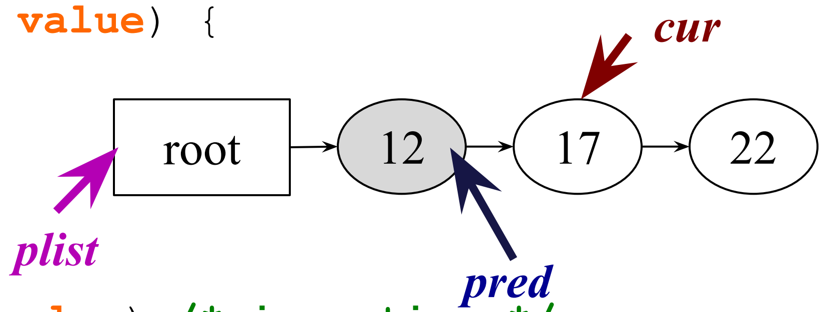
```
void add(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(true) {
    Node cur = pointer(*pred);
    if(cur == null || value < cur->value) /* insertion */
      if(CAS(pred, cur, new Node(cur, value)) != cur) goto restart;
      else return;

    if(mark(cur->next)) {                    /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }

    pred = &cur->next;
  }
}
```



Exemple: value = 14

Add: the insertion

```
void add(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(true) {
    Node cur = pointer(*pred);
    if(cur == null || value < cur->value) /* insertion */
      if(CAS(pred, cur, new Node(cur, value)) != cur) goto restart;
      else return;

    if(mark(cur->next)) {                    /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }

    pred = &cur->next;
  }
}
```

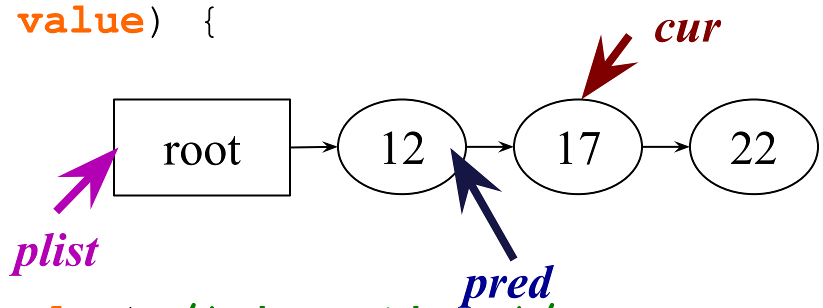


The CAS may fail for two reasons:

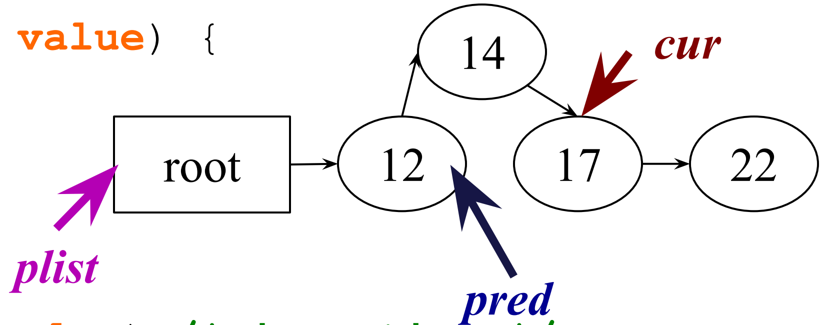
- 12 becomes grey (deleted)
- another node is inserted between 12 and 17

Add: the insertion

```
void add(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(true) {
    Node cur = pointer(*pred);
    if(cur == null || value < cur->value) /* insertion */
      if(CAS(pred, cur, new Node(cur, value)) != cur) goto restart;
      else return;

    if(mark(cur->next)) {                 /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }

    pred = &cur->next;
  }
}
```



Add: the insertion




```
void add(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(true) {
    Node cur = pointer(*pred);
    if(cur == null || value < cur->value) /* insertion */
      if(CAS(pred, cur, new Node(cur, value)) != cur) goto restart;
      else return;

    if(mark(cur->next)) {                    /* cur is deleted */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }

    pred = &cur->next;
  }
}
```

To take away

- Three levels of non blocking algorithms from the strongest to the weakest
  - Wait-free
  - Lock-free
  - Obstruction-free

- Three lock-free algorithms
  - The stack: especially simple
  - The queue: enforces invariants
  - The linked list: enforces invariants and helps to remove deleted node during a traversal

- For each lock-free algorithm, we have a linearization point, i.e., a point in the program where the operation succeeds and becomes visible