TECHNIQUES MODERNES DE PROGRAMMATION CONCURRENTE

COURS 3
ALGORITHMIQUE NON-BLOQUANTE

Par Jean-Pierre Lozi
Basé sur les cours de
Gaël Thomas

Université
Nice
Sophia Antipolis

RAPPEL : COMMENT MIEUX PARALLÉLISER…

- **Option 1 : réduire la taille des sections critiques**
  - Passer du coarse-grained locking au fined-grained locking, essentiel !

RAPPEL : COMMENT MIEUX PARALLÉLISER…

- **Option 1 : réduire la taille des sections critiques**
  - Passer du coarse-grained locking au fined-grained locking, essentiel !

- **Option 2 : améliorer les algorithmes de verrouillage**
  - À considérer, peut beaucoup améliorer les perfs si haute contention…

RAPPEL : COMMENT MIEUX PARALLÉLISER…

- **Option 1 : réduire la taille des sections critiques**
  - Passer du coarse-grained locking au fined-grained locking, essentiel !

- **Option 2 : améliorer les algorithmes de verrouillage**
  - À considérer, peut beaucoup améliorer les perfs si haute contention…

- **Option 3 : écrire des algorithmes sans verrous, a.k.a. non-bloquants !**
  - L'étape suivante, n'utiliser que des instructions atomiques !

RAPPEL : COMMENT MIEUX PARALLÉLISER…

- Option 1 : réduire la taille des sections critiques
  - Passer du coarse-grained locking au fined-grained locking, essentiel !
- Option 2 : améliorer les algorithmes de verrouillage
  - À considérer, peut beaucoup améliorer les perfs si haute contention…
- Option 3 : écrire des algorithmes sans verrous, a.k.a. non-bloquants !
  - L'étape suivante, n'utiliser que des instructions atomiques !
- Option 4 : changer complètement le modèle mémoire avec les mémoires transactionnelles
  - Le futur ?

OPTION 3 :
ÉCRIRE DES ALGORITHMES SANS VERROUS, A.K.A. « NON-BLOQUANTS »

ALGORITHMES SANS VERROUS

- **Idée : utiliser uniquement les instructions atomiques + modèle mémoire (barrières…)**
  - En fait, instructions atomiques, « micro sections critiques » faisant choses très spécifiques
  - Peu de choix : inverser deux cases mémoires, inverser si case mémoire vaut quelque chose…
  - **On peut quand même faire pas mal de choses !**
  - *Mais il faut être inventif, peut devenir très compliqué…*

ALGORITHMES SANS VERROUS

- **Idée : utiliser uniquement les instructions atomiques + modèle mémoire (barrières…)**
  - En fait, instructions atomiques, « micro sections critiques » faisant choses très spécifiques
  - Peu de choix : inverser deux cases mémoires, inverser si case mémoire vaut quelque chose…
  - On peut quand même faire pas mal de choses !
  - *Mais il faut être inventif, peut devenir très compliqué…*

- **On parle d'algorithmes « non-bloquants » !**

UN PEU DE THÉORIE…

- Trois classes d'algorithmes sans verrou : wait-free ⇒ lock-free ⇒ obstruction-free
  - Du plus « fort » au plus faible : algo wait-free est aussi lock-free et obstruction-free… Pas l'inverse !

UN PEU DE THÉORIE…

- Trois classes d'algorithmes sans verrou : wait-free ⇒ lock-free ⇒ obstruction-free
  - Du plus « fort » au plus faible : algo wait-free est aussi lock-free et obstruction-free… Pas l'inverse !
- Wait-free, le plus fort : *toute opération se termine en un nombre fini de pas*
  - Famine impossible : toute opération finira par se terminer.

UN PEU DE THÉORIE…

- **Trois classes d'algorithmes sans verrou : wait-free ⇒ lock-free ⇒ obstruction-free**
  - Du plus « fort » au plus faible : algo wait-free est aussi lock-free et obstruction-free… **Pas l'inverse !**
- **Wait-free, le plus fort :** *toute opération se termine en un nombre fini de pas*
  - Famine impossible : toute opération finira par se terminer.
- **Lock-free :** *si des opérations appelées infiniment souvent, se terminent infiniment souvent*
  - *Certains* threads progressent tout le temps, pas forcément tous
  - Du coup, famine possible
  - **Impliqué par wait-free :** « certains » au lieu de « tous »…

UN PEU DE THÉORIE…

- **Trois classes d'algorithmes sans verrou : wait-free ⇒ lock-free ⇒ obstruction-free**
  - Du plus « fort » au plus faible : algo wait-free est aussi lock-free et obstruction-free… **Pas l'inverse !**
- **Wait-free, le plus fort :** *toute opération se termine en un nombre fini de pas*
  - Famine impossible : toute opération finira par se terminer.
- **Lock-free :** *si des opérations appelées infiniment souvent, se terminent infiniment souvent*
  - *Certains* threads progressent tout le temps, pas forcément tous
  - Du coup, famine possible
  - **Impliqué par wait-free :** « certains » au lieu de « tous »…
- **Obstruction-free :** *tout thread s'exécutant seul termine son opération en un nombre fini de pas*
  - Si on arrête les autres threads qui peuvent nous ralentir, alors on arrive à terminer quand même
  - **Impliqué par lock-free :** n'appeler que des opérations d'un thread infiniment…

UN PEU DE THÉORIE…

- Si on utilise des verrous, on n'est même pas obstruction-free !

UN PEU DE THÉORIE…

- Si on utilise des verrous, on n'est même pas obstruction-free !
- Un verrou pris empêche une opération de se terminer en un nombre fini de pas…

UN PEU DE THÉORIE…

- Si on utilise des verrous, on n'est même pas obstruction-free !
- Un verrou pris empêche une opération de se terminer en un nombre fini de pas…
- Du coup, pas lock-free ni wait-free non plus.

UN PEU DE THÉORIE…

- Si on utilise des verrous, on n'est même pas obstruction-free !
- Un verrou pris empêche une opération de se terminer en un nombre fini de pas…
- Du coup, pas lock-free ni wait-free non plus.
- Obstruction-free « inventé » en 2003 par Herlihy et d'autres

UN PEU DE THÉORIE…

- Si on utilise des verrous, on n'est même pas obstruction-free !
- Un verrou pris empêche une opération de se terminer en un nombre fini de pas…
- Du coup, pas lock-free ni wait-free non plus.
- Obstruction-free « inventé » en 2003 par Herlihy et d'autres
- Avant ça, on avait non-blocking = lock-free… Donc parfois un peu confus !
  - On parle parfois d'algorithmique « lock-free » pour parler d'algorithmique sans verrous…

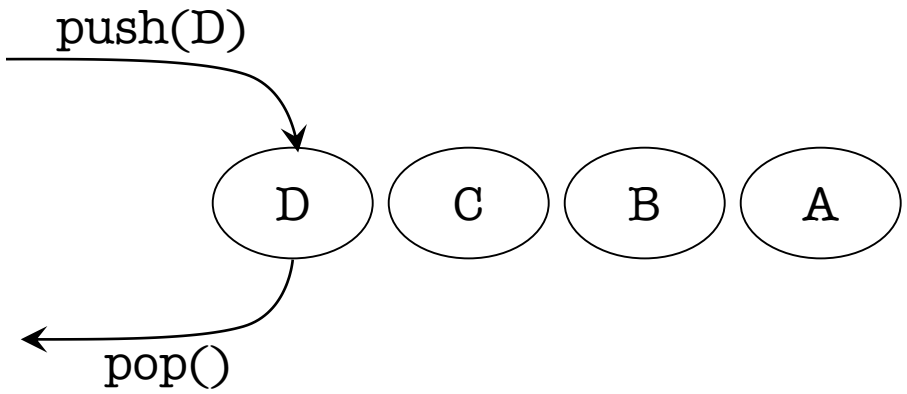
UN PEU DE THÉORIE…

- Si on utilise des verrous, on n'est même pas obstruction-free !
- Un verrou pris empêche une opération de se terminer en un nombre fini de pas…
- Du coup, pas lock-free ni wait-free non plus.
- Obstruction-free « inventé » en 2003 par Herlihy et d'autres
- **Avant ça, on avait non-blocking = lock-free… Donc parfois un peu confus !**
  - On parle parfois d'algorithmique « lock-free » pour parler d'algorithmique sans verrous…
- **On va voir des algos non-bloquants pour trois structures de données basiques :**
  - La pile
  - La file
  - La liste chaînée

LA PILE

LE B.A.-BA : LA PILE

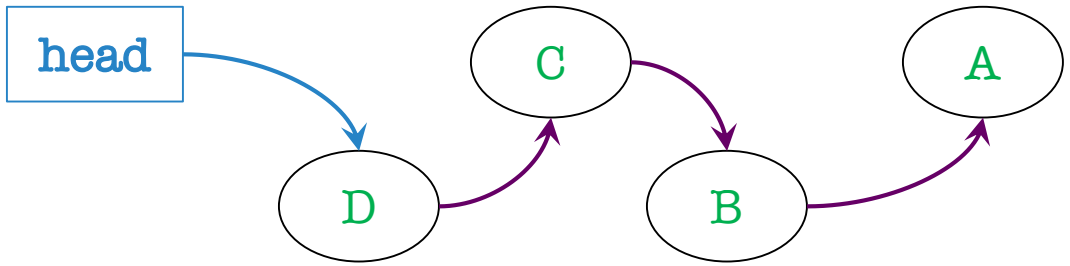
- **Deux opérations en mode LIFO (Last-In-First-Out) :**
  - `void push(Element e)` : empile un élément
  - `Element pop()` : dépile un élément



LA PILE AVEC VERROU

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

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

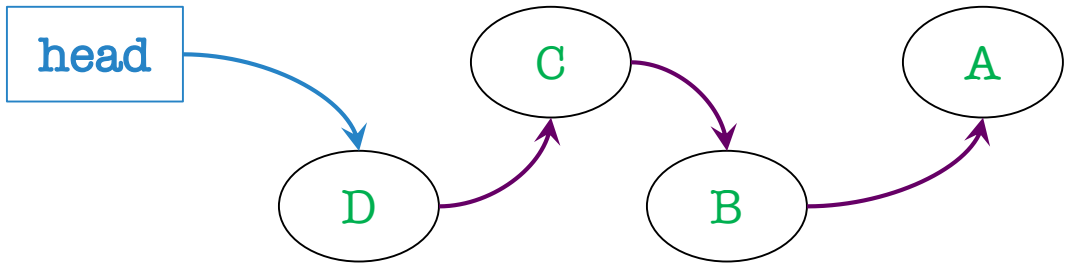


LA PILE AVEC VERROU

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

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

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

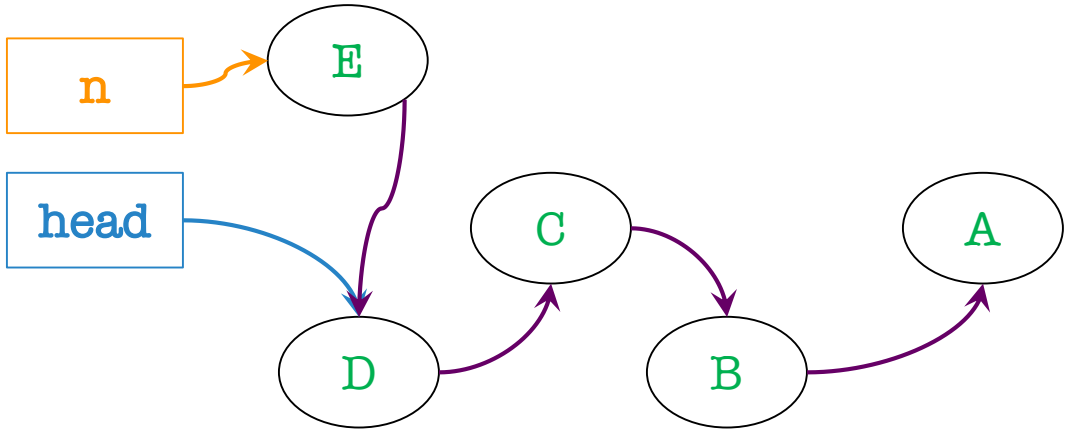


LA PILE AVEC VERROU

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

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

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

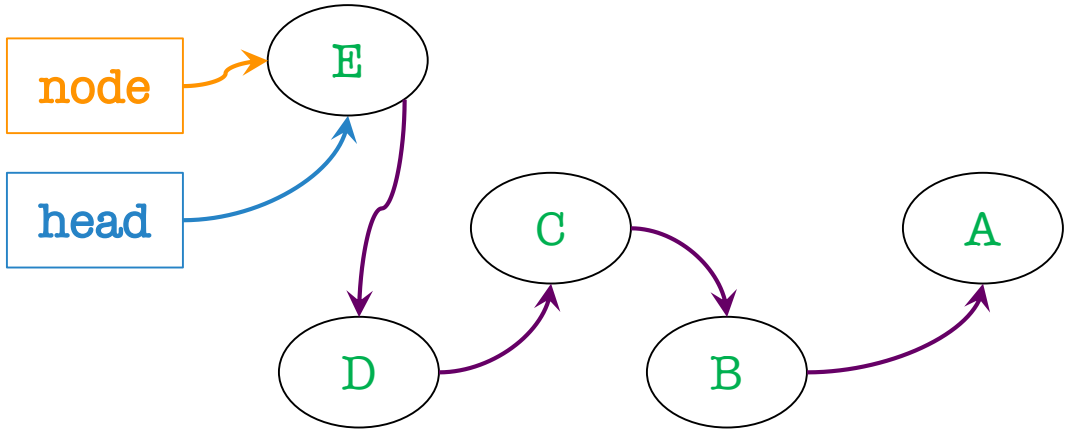


LA PILE AVEC VERROU

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

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

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

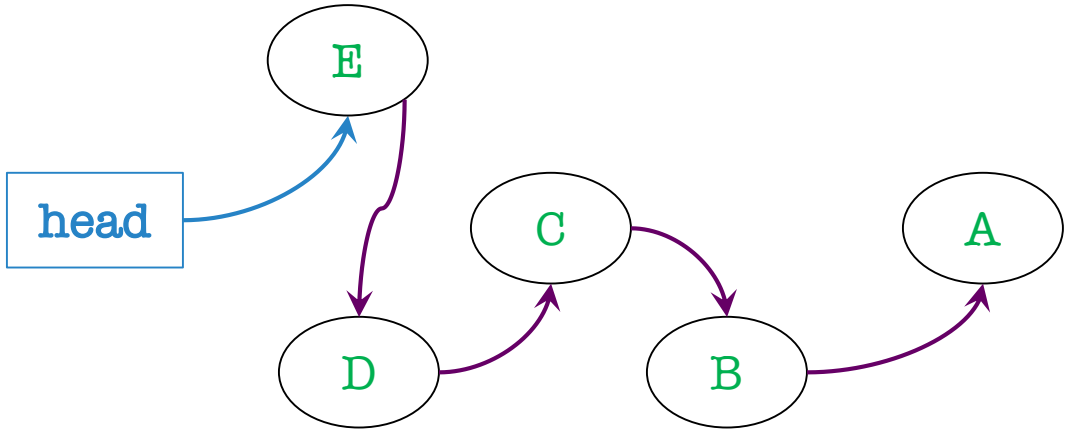


LA PILE AVEC VERROU

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

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

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



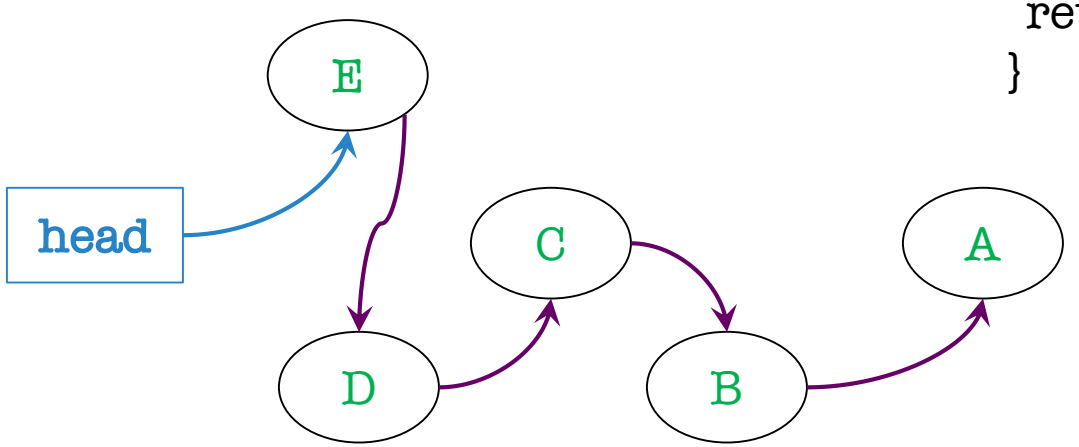
LA PILE AVEC VERROU

```
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;
}
```



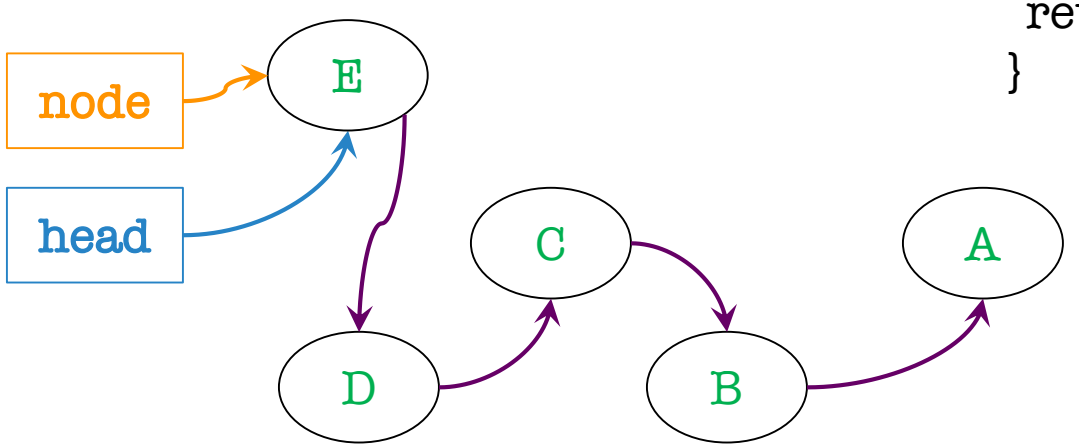
LA PILE AVEC VERROU

```
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;
}
```



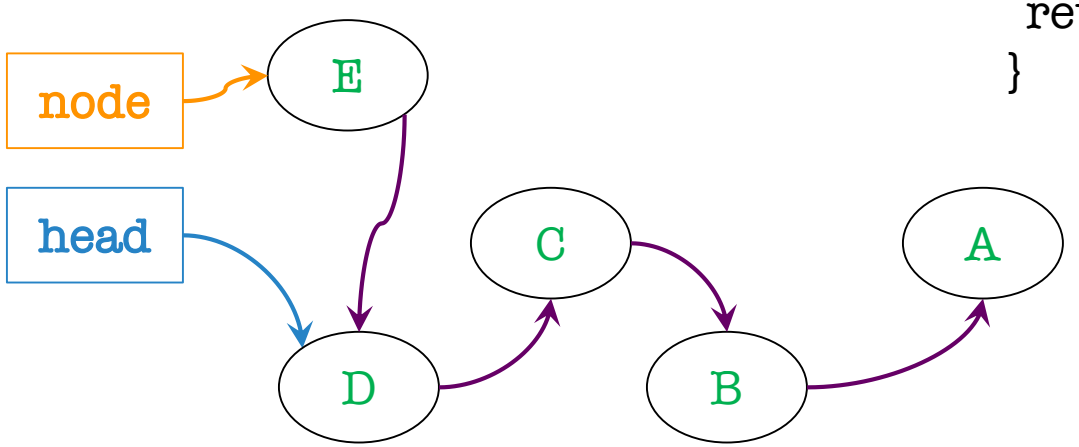
LA PILE AVEC VERROU

```
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;
}
```



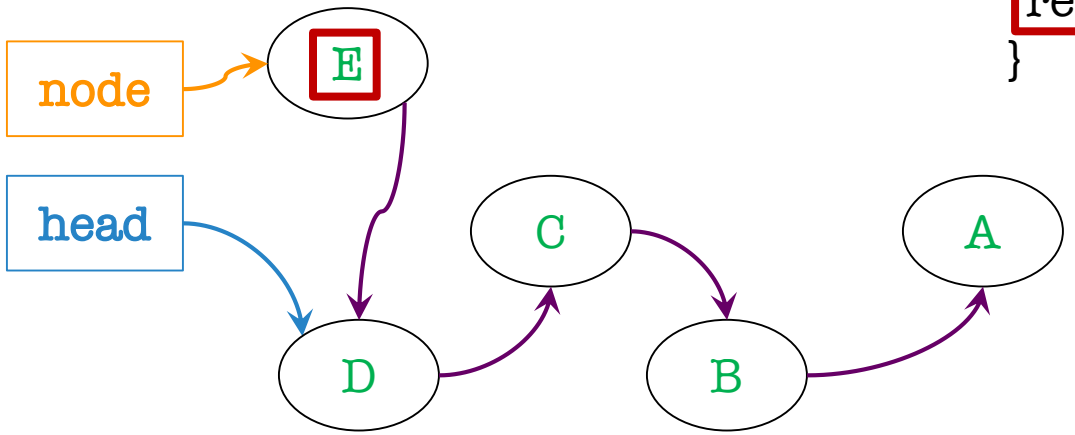
LA PILE AVEC VERROU

```
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;
}
```



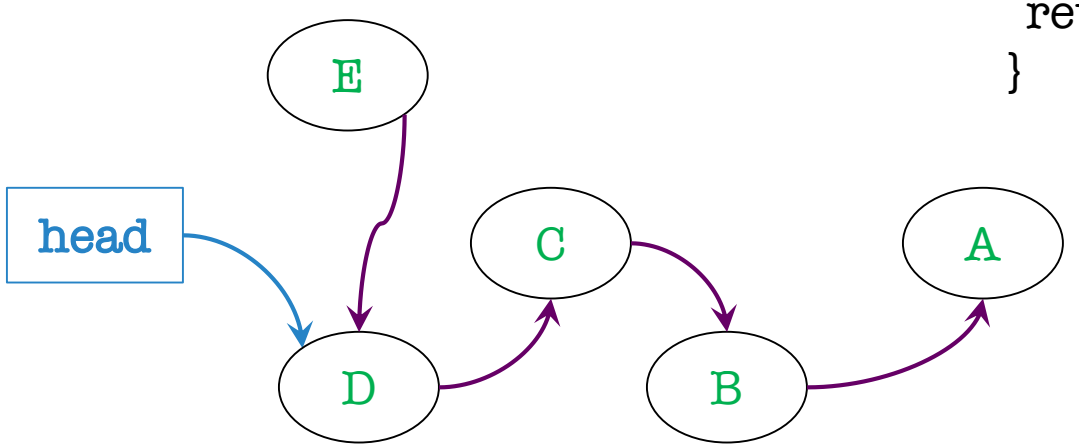
LA PILE AVEC VERROU

```
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;
}
```



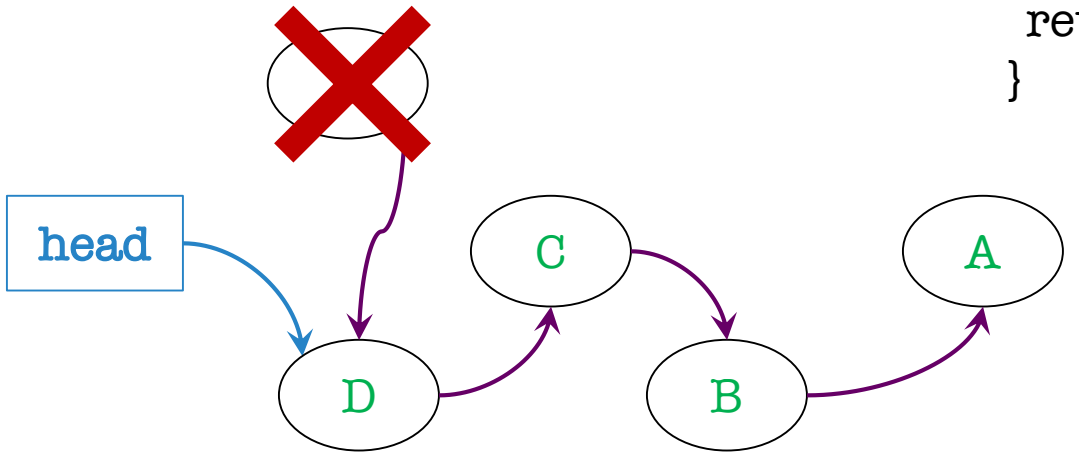
LA PILE AVEC VERROU

```
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;
}
```



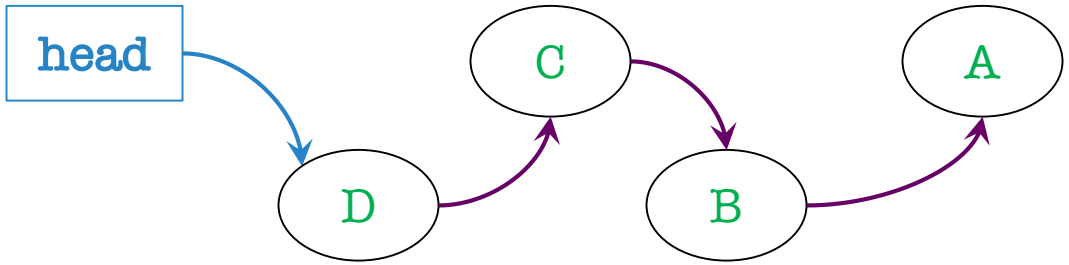
LA PILE NON-BLOQUANTE (SCOTT 91)

Principe : atomic Compare-And-Swap sur la tête !

LA PILE NON-BLOQUANTE (SCOTT 91)

```
Class Stack {
 Node   head;
}

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

- **Idée du** push() **:**
  - On crée un nouveau nœud qui pointe vers le premier élément de la liste
  - Atomiquement, si premier élément toujours le même, on fait pointer la tête vers le nouvel élément !
  - Et si un autre élément a réussi à changer la tête entre les deux (push ou pop) ?
    - *On réessaie…*

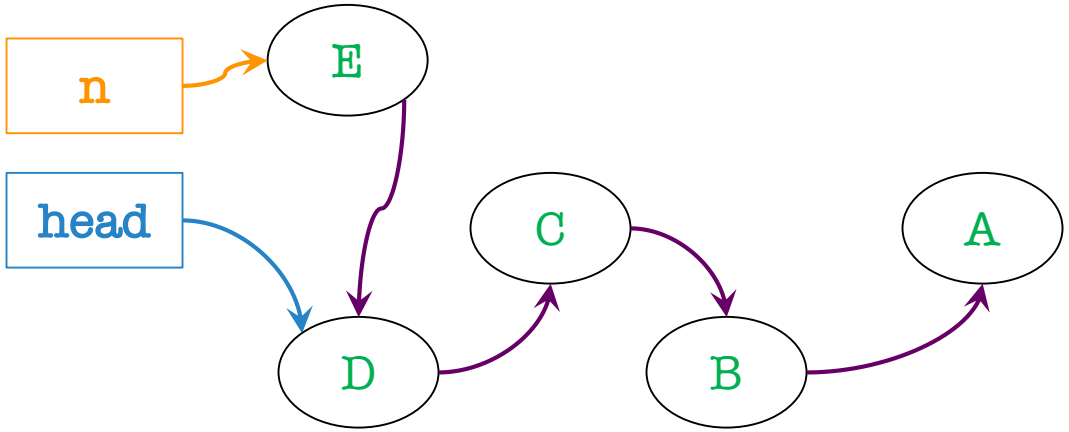



```
void Stack.push(Element element) {
 do {
  Node n = new Node(head, element);
 } while(CAS(&head, n.next, n) != n.next);
}
```

LA PILE NON-BLOQUANTE (SCOTT 91)

```
Class Stack {
 Node   head;
}

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

- Idée du push() :
  - On crée un nouveau nœud qui pointe vers le premier élément de la liste
  - Atomiquement, si premier élément toujours le même, on fait pointer la tête vers le nouvel élément !
  - Et si un autre élément a réussi à changer la tête entre les deux (push ou pop) ?
    - *On réessaie…*

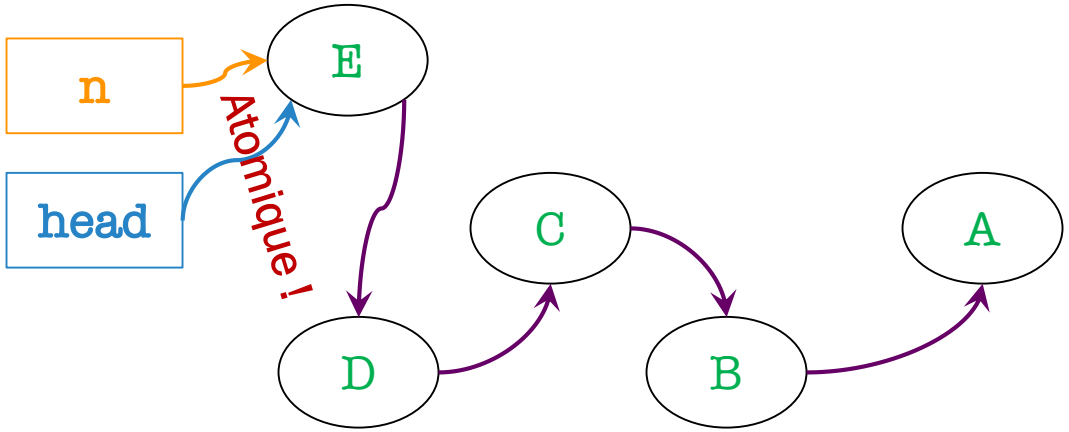



```
void Stack.push(Element element) {
 do {
  Node n = new Node(head, element);
 } while(CAS(&head, n.next, n) != n.next);
}
```

LA PILE NON-BLOQUANTE (SCOTT 91)

```
Class Stack {
 Node   head;
}

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

- **Idée du** push() :
  - On crée un nouveau nœud qui pointe vers le premier élément de la liste
  - Atomiquement, si premier élément toujours le même, on fait pointer la tête vers le nouvel élément !
  - Et si un autre élément a réussi à changer la tête entre les deux (push ou pop) ?
    - On réessaie…

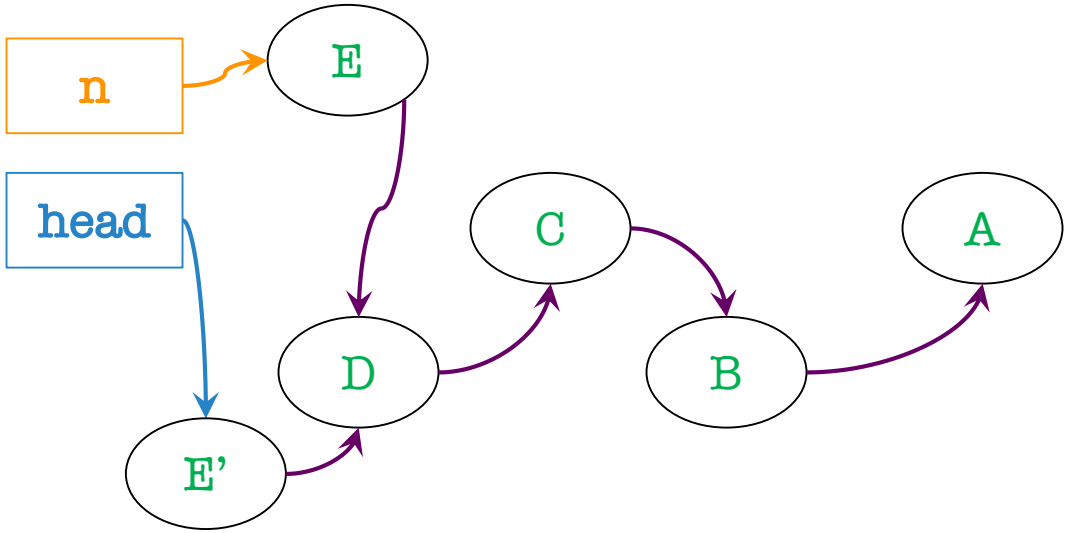



```
void Stack.push(Element element) {
 do {
  Node n = new Node(head, element);
 } while(CAS(&head, n.next, n) != n.next);
}
```

LA PILE NON-BLOQUANTE (SCOTT 91)

```
Class Stack {
 Node   head;
}

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

- **Idée du** push() :
  - On crée un nouveau nœud qui pointe vers le premier élément de la liste
  - Atomiquement, si premier élément toujours le même, on fait pointer la tête vers le nouvel élément !
  - Et si un autre élément a réussi à changer la tête entre les deux (push ou pop) ?
    - On réessaie…

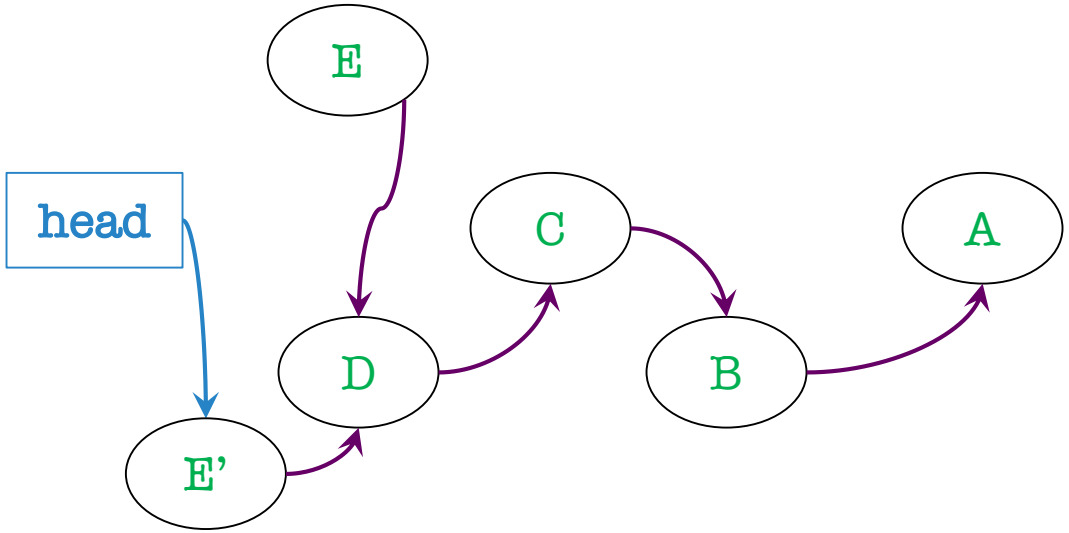



```
void Stack.push(Element element) {
 do {
  Node n = new Node(head, element);
 } while(CAS(&head, n.next, n) != n.next)
}
```

LA PILE NON-BLOQUANTE (SCOTT 91)

```
Class Stack {
 Node   head;
}

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

- **Idée du** push() :
  - On crée un nouveau nœud qui pointe vers le premier élément de la liste
  - Atomiquement, si premier élément toujours le même, on fait pointer la tête vers le nouvel élément !
  - Et si un autre élément a réussi à changer la tête entre les deux (push ou pop) ?
    - On réessaie…

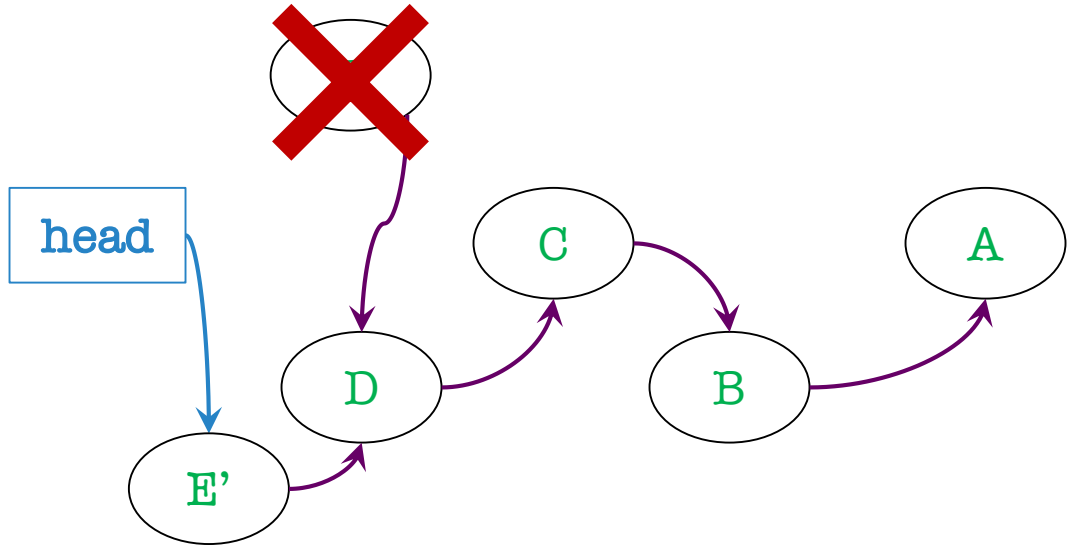



```
void Stack.push(Element element) {
 do {
   Node n = new Node(head, element);
  } while(CAS(&head, n.next, n) != n.next);
}
```

LA PILE NON-BLOQUANTE (SCOTT 91)

```
Class Stack {
 Node   head;
}

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

- **Idée du** push() :
  - On crée un nouveau nœud qui pointe vers le premier élément de la liste
  - Atomiquement, si premier élément toujours le même, on fait pointer la tête vers le nouvel élément !
  - Et si un autre élément a réussi à changer la tête entre les deux (push ou pop) ?
    - On réessaie…




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

LA PILE NON-BLOQUANTE (SCOTT 91)

```
Class Stack {
 Node   head;
}

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

- Idée du pop() : similaire !
  - On récupère le premier élément de la liste et on le stocke.
    - Si pas d'élément, erreur ou ou renvoie null.
  - Atomiquement, si le premier élément n'a pas changé, on le remplace par le suivant !
  - Si un autre élément a réussi à changer la tête entre les deux (push ou pop) échec, on réessaie…
  - Sinon, on renvoie le premier élément récupéré.

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

LA PILE NON-BLOQUANTE (SCOTT 91)

```
Class Stack {
 Node   head;
}

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

- Idée du pop() : similaire !
  - On récupère le premier élément de la liste et on le stocke.
    - Si pas d'élément, erreur ou ou renvoie null.
  - Atomiquement, si le premier élément n'a pas changé, on le remplace par le suivant !
  - Si un autre élément a réussi à changer la tête entre les deux (push ou pop) échec, on réessaie…
  - Sinon, on renvoie le premier élément récupéré.

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

LA PILE NON-BLOQUANTE (SCOTT 91)

```
Class Stack {
 Node   head;
}

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

- Idée du pop() : similaire !
  - On récupère le premier élément de la liste et on le stocke.
    - Si pas d'élément, erreur ou ou renvoie null.
  - Atomiquement, si le premier élément n'a pas changé, on le remplace par le suivant !
  - Si un autre élément a réussi à changer la tête entre les deux (push ou pop) échec, on réessaie…
  - Sinon, on renvoie le premier élément récupéré.

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

LA PILE NON-BLOQUANTE (SCOTT 91)

```
Class Stack {
 Node   head;
}

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

- Idée du pop() : similaire !
  - On récupère le premier élément de la liste et on le stocke.
    - Si pas d'élément, erreur ou ou renvoie null.
  - Atomiquement, si le premier élément n'a pas changé, on le remplace par le suivant !
  - Si un autre élément a réussi à changer la tête entre les deux (push ou pop) échec, on réessaie…
  - Sinon, on renvoie le premier élément récupéré.

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

LA PILE NON-BLOQUANTE (SCOTT 91)

```
Class Stack {
 Node   head;
}

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

- Idée du pop() : similaire !
  - On récupère le premier élément de la liste et on le stocke.
    - Si pas d'élément, erreur ou ou renvoie null.
  - Atomiquement, si le premier élément n'a pas changé, on le remplace par le suivant !
  - Si un autre élément a réussi à changer la tête entre les deux (push ou pop) échec, on réessaie…
  - Sinon, on renvoie le premier élément récupéré.

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

LA PILE NON-BLOQUANTE (SCOTT 91)

```
Class Stack {
 Node   head;
}

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

- Idée du pop() : similaire !
  - On récupère le premier élément de la liste et on le stocke.
    - Si pas d'élément, erreur ou ou renvoie null.
  - Atomiquement, si le premier élément n'a pas changé, on le remplace par le suivant !
  - Si un autre élément a réussi à changer la tête entre les deux (push ou pop) échec, on réessaie…
  - Sinon, on renvoie le premier élément récupéré.

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

LA PILE NON-BLOQUANTE (SCOTT 91)

- **Obstruction-free !**
  - *Quelque soit l'instant, si on interrompt les threads sauf un, dans le pire cas, un tour de boucle en plus suffit.*

LA PILE NON-BLOQUANTE (SCOTT 91)

- **Obstruction-free !**
  - *Quelque soit l'instant, si on interrompt les threads sauf un, dans le pire cas, un tour de boucle en plus suffit.*
- **Lock-free !**
  - *Si les threads appellent infiniment souvent push ou pop, elles seront exécutées infiniment souvent.*

LA PILE NON-BLOQUANTE (SCOTT 91)

- **Obstruction-free !**
  - *Quelque soit l'instant, si on interrompt les threads sauf un, dans le pire cas, un tour de boucle en plus suffit.*
- **Lock-free !**
  - *Si les threads appellent infiniment souvent push ou pop, elles seront exécutées infiniment souvent.*
  - **Preuve :** il faut réussir pop ou push pour bloquer un pop ou push
  - Si un pop ou push échoue, pas de progrès pour ce thread, mais progrès pour un autre
  - Certains threads progressent forcément

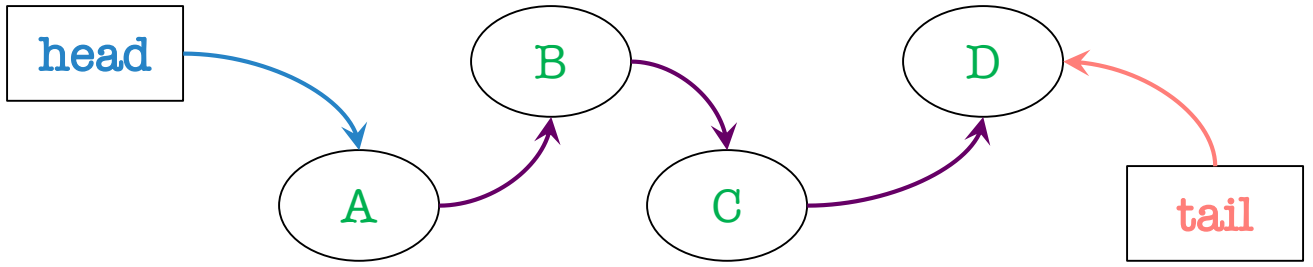
LA PILE NON-BLOQUANTE (SCOTT 91)

- **Obstruction-free !**
  - *Quelque soit l'instant, si on interrompt les threads sauf un, dans le pire cas, un tour de boucle en plus suffit.*
- **Lock-free !**
  - *Si les threads appellent infiniment souvent push ou pop, elles seront exécutées infiniment souvent.*
  - **Preuve :** il faut réussir pop ou push pour bloquer un pop ou push
  - Si un pop ou push échoue, pas de progrès pour ce thread, mais progrès pour un autre
  - Certains threads progressent forcément
- **Pas wait-free !**
  - *Un push/pop peut être retardé indéfiniment par d'autres push/pop !*

LA FILE

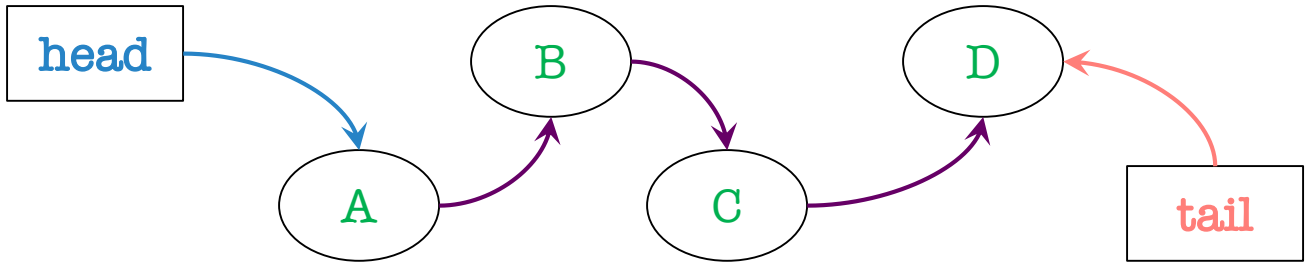
LA FILE

- **Rappel :** file, file d'attente, queue… Même chose, FIFO (« First In First Out »).



LA FILE

- **Rappel :** file, file d'attente, queue… Même chose, FIFO (« First In First Out »).
- **Deux opérations :**
  - `void enqueue(Element e)` **:** ajoute l'élément en queue
  - `Element dequeue()` **:** dépile l'élément en tête

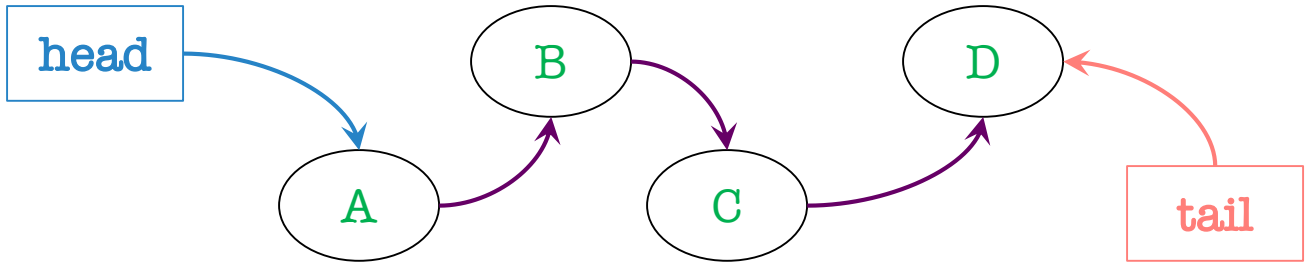


LA FILE AVEC VERROU

```
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;
}
```

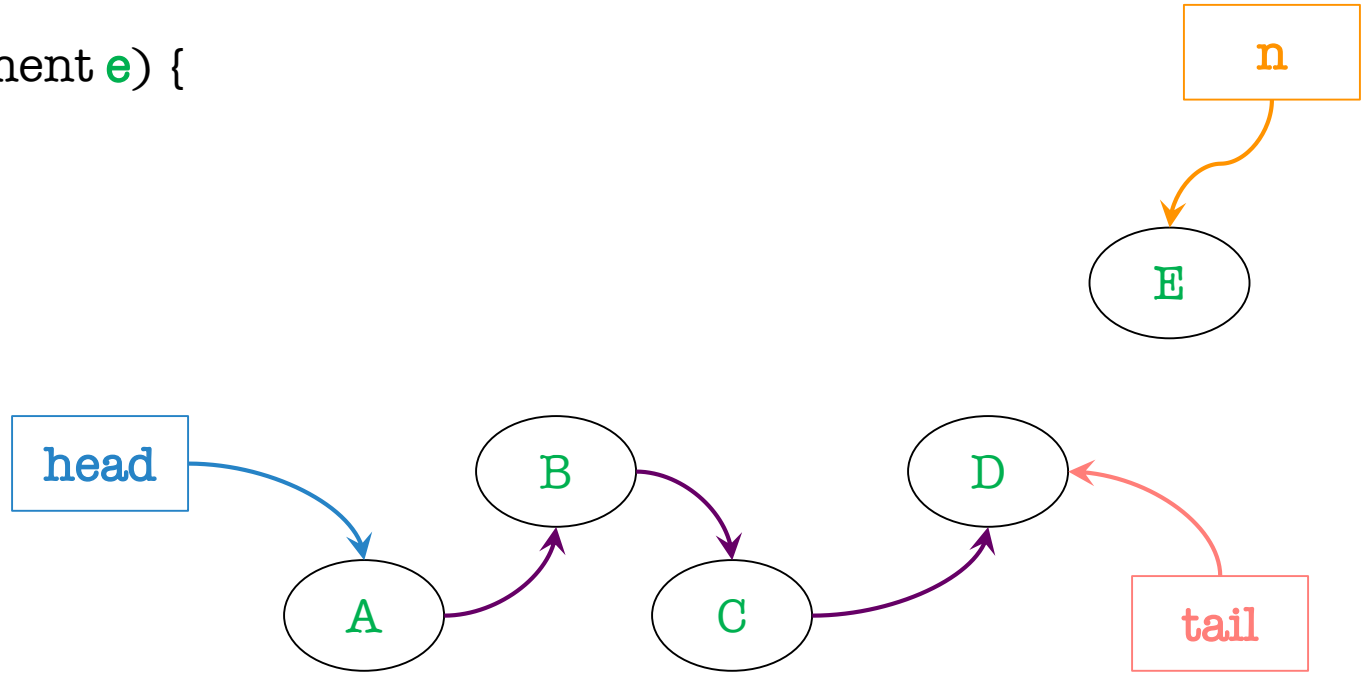


LA FILE AVEC VERROU

```
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;
}
```

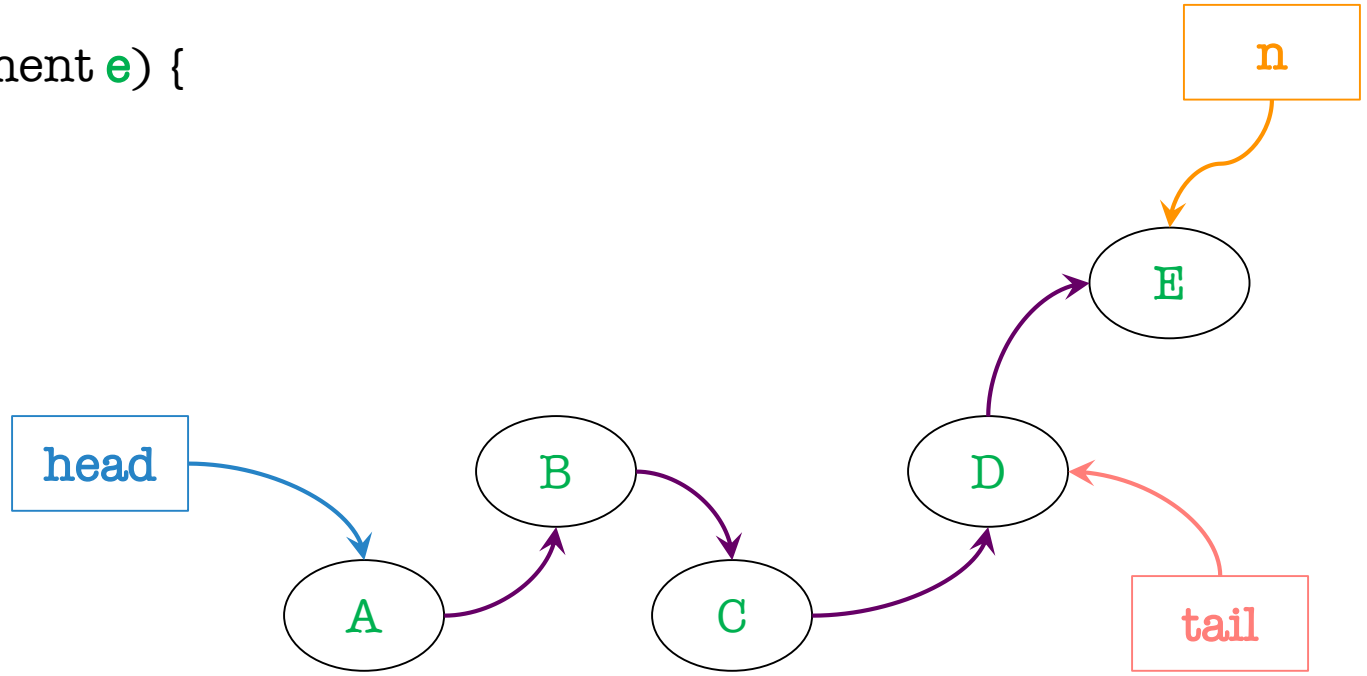


LA FILE AVEC VERROU

```
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;
}
```

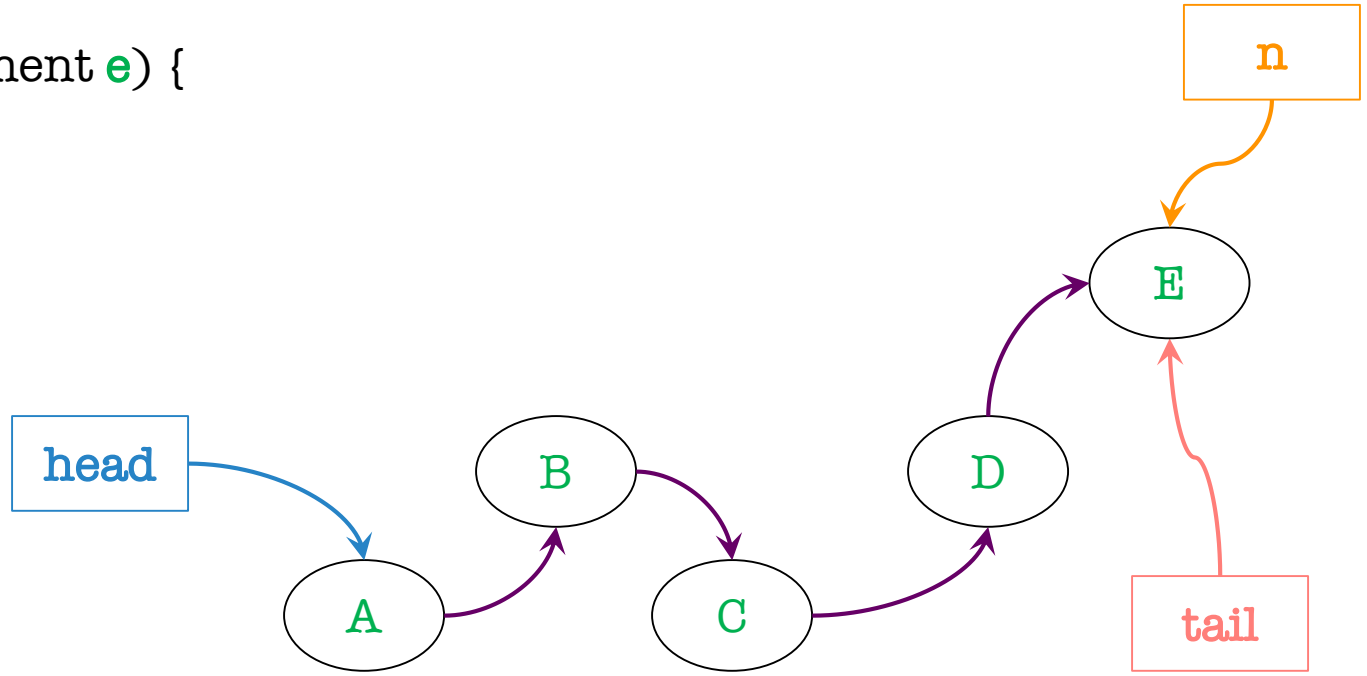


LA FILE AVEC VERROU

```
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;
}
```

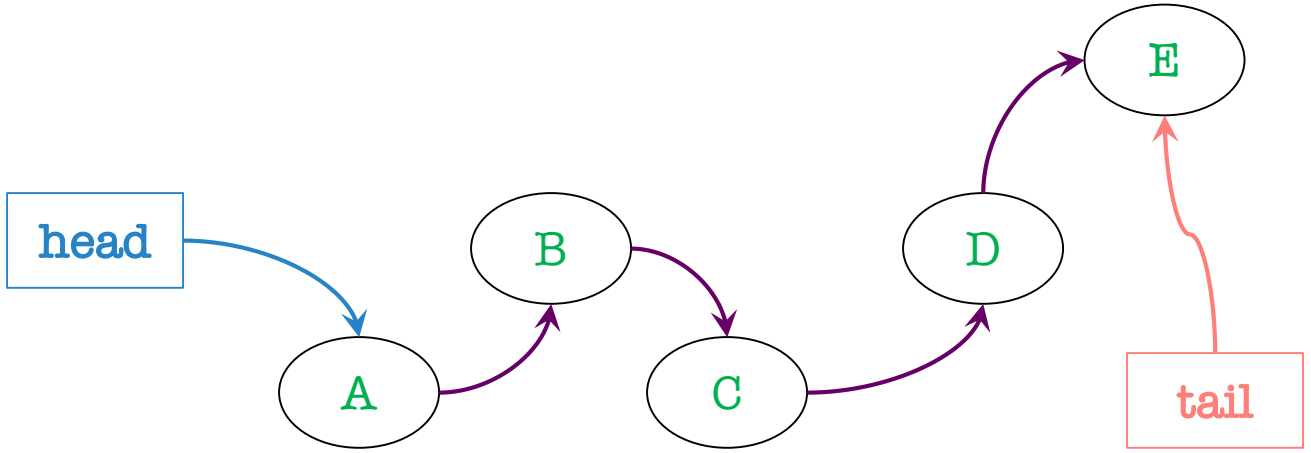


LA FILE AVEC VERROU

```
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;
}
```

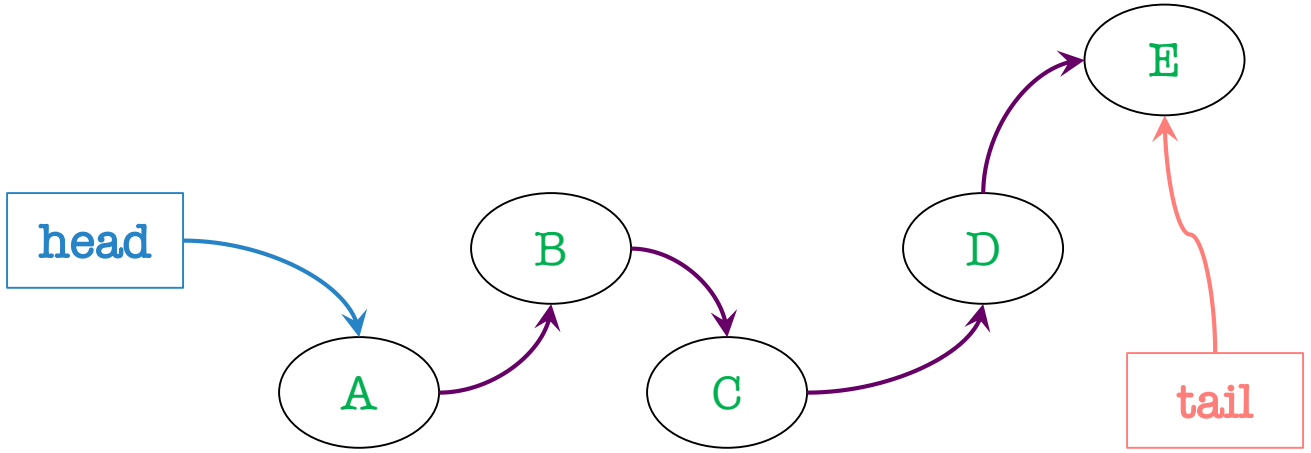


LA FILE AVEC VERROU

```
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;
}
```

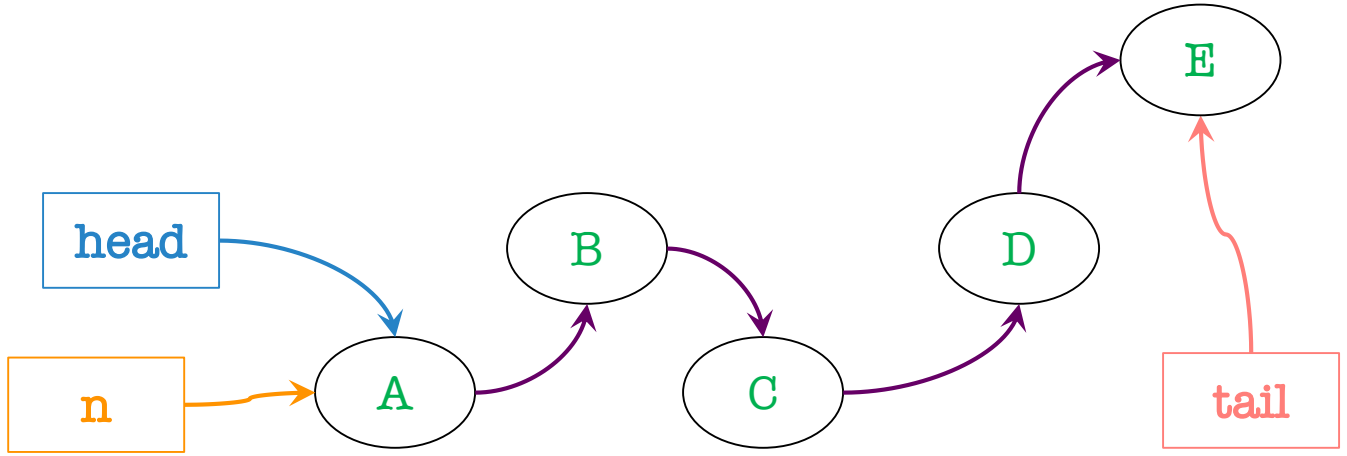


LA FILE AVEC VERROU

```
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;
}
```

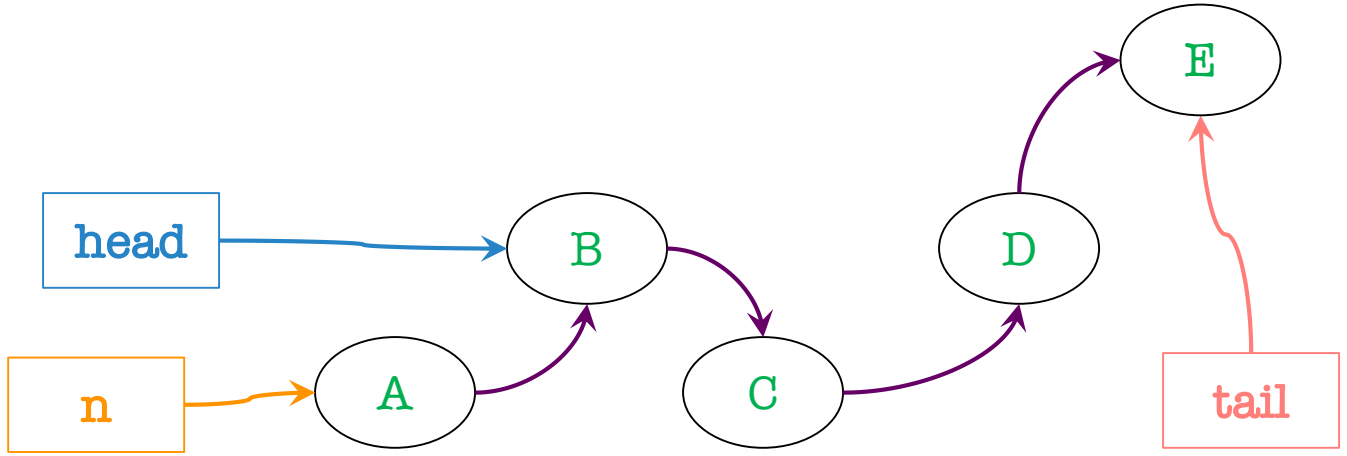


LA FILE AVEC VERROU

```
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;
}
```

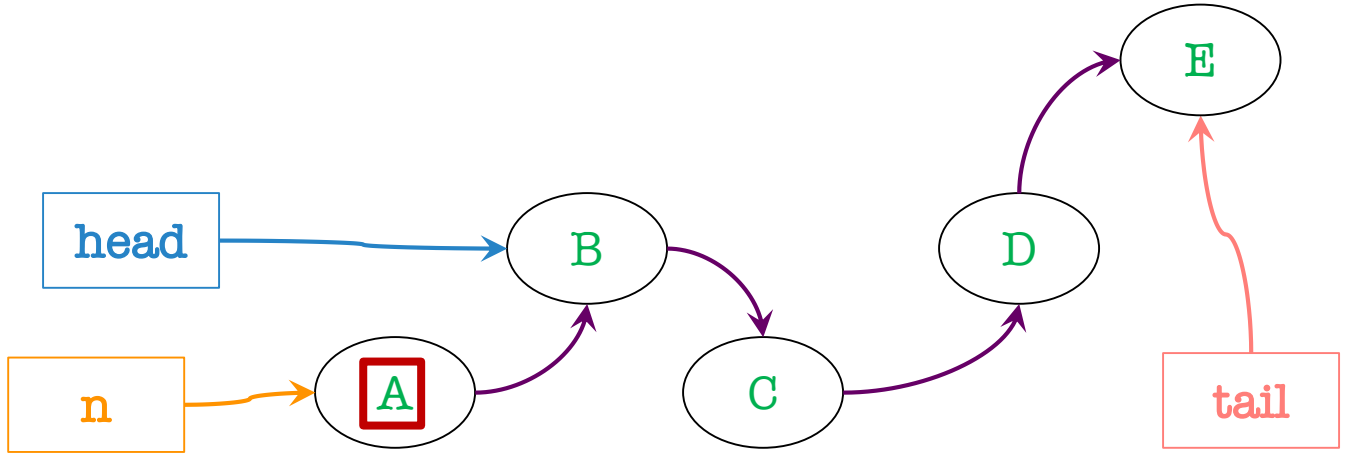


LA FILE AVEC VERROU

```
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;
}
```

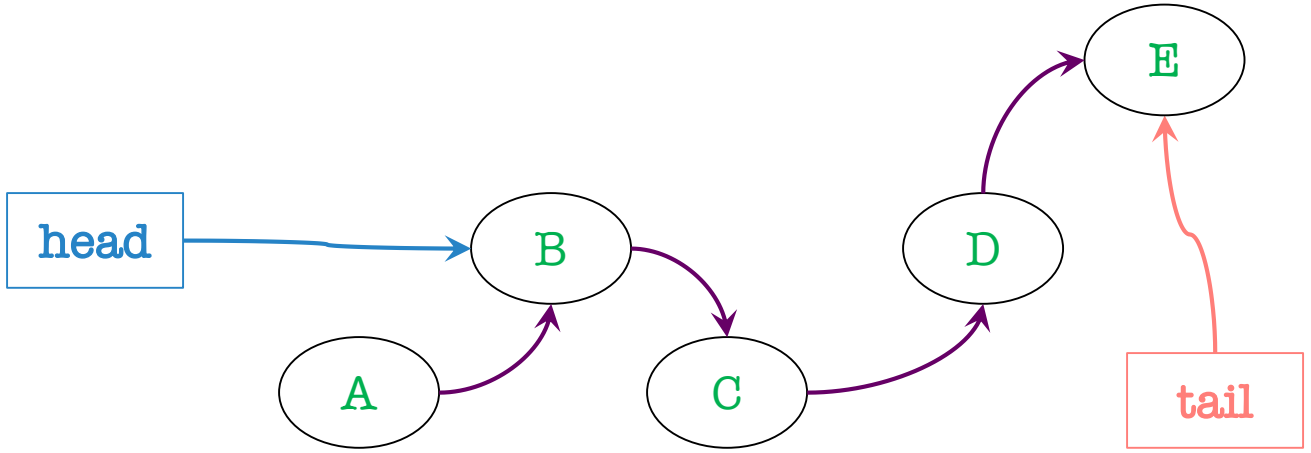


LA FILE AVEC VERROU

```
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;
}
```

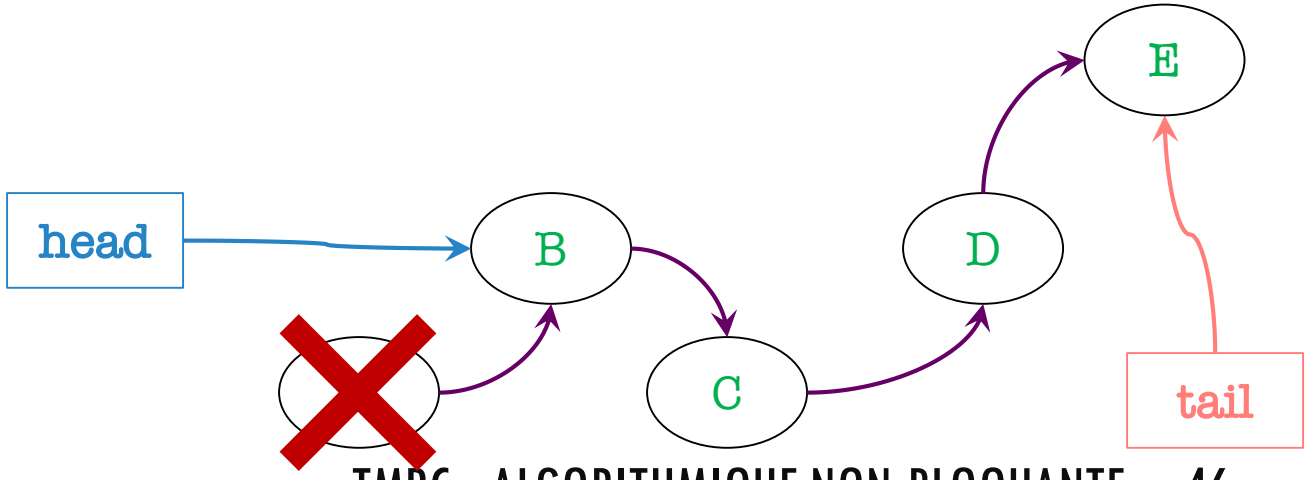


LA FILE AVEC VERROU

```
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;
}
```



LA FILE NON-BLOQUANTE

- **Avec la pile, c'était facile !**
  - Seulement un pointeur head sur lequel on fait des CAS() en boucle dans tous les cas…

LA FILE NON-BLOQUANTE

- **Avec la pile, c'était facile !**
  - Seulement un pointeur head sur lequel on fait des CAS() en boucle dans tous les cas…
  - Pour `push()`, on prépare le chaînage (nœud pas encore accorché donc OK), puis CAS().

LA FILE NON-BLOQUANTE

- **Avec la pile, c'était facile !**
  - Seulement un pointeur head sur lequel on fait des CAS() en boucle dans tous les cas…
  - Pour `push()`, on prépare le chaînage (nœud pas encore accorché donc OK), puis CAS().
  - Pour `pop()`, on décroche le nœud avec un CAS(),

LA FILE NON-BLOQUANTE

- **Avec la pile, c'était facile !**
  - Seulement un pointeur head sur lequel on fait des CAS() en boucle dans tous les cas…
  - Pour push(), on prépare le chaînage (nœud pas encore accorché donc OK), puis CAS().
  - Pour pop(), on décroche le nœud avec un CAS(),
- **Pour une file, insertion beaucoup plus compliquée…**
  - Il faudrait préparer le chaînage sur la liste car sens inverse, PUIS faire un CAS() sur la queue…

LA FILE NON-BLOQUANTE

- **Avec la pile, c'était facile !**
  - Seulement un pointeur head sur lequel on fait des CAS() en boucle dans tous les cas…
  - Pour push(), on prépare le chaînage (nœud pas encore accorché donc OK), puis CAS().
  - Pour pop(), on décroche le nœud avec un CAS(),
- **Pour une file, insertion beaucoup plus compliquée…**
  - Il faudrait préparer le chaînage sur la liste car sens inverse, PUIS faire un CAS() sur la queue…
  - Impossible de rendre ces deux opérations atomiques !
    - Dernier nœud / pointeur de queue peuvent être modifiés entre temps…

LA FILE NON-BLOQUANTE

- **Avec la pile, c'était facile !**
  - Seulement un pointeur head sur lequel on fait des CAS() en boucle dans tous les cas…
  - Pour push(), on prépare le chaînage (nœud pas encore accorché donc OK), puis CAS().
  - Pour pop(), on décroche le nœud avec un CAS(),
- **Pour une file, insertion beaucoup plus compliquée…**
  - Il faudrait préparer le chaînage sur la liste car sens inverse, PUIS faire un CAS() sur la queue…
  - **Impossible de rendre ces deux opérations atomiques !**
    - Dernier nœud / pointeur de queue peuvent être modifiés entre temps…
  - **Il faut donc ruser !**
    - On va voir un algo qui marche, chez vous « jouez » avec sur de petits exemples.
    - Vous verrez qu'il est difficile de faire plus simple…

LA FILE NON-BLOQUANTE : PRINCIPES

- Un faux nœud (fake) pour gérer le cas où la liste est vide !
  - *Evite un cas particulier, astuce courante.*

LA FILE NON-BLOQUANTE : PRINCIPES

- Un faux nœud (fake) pour gérer le cas où la liste est vide !
  - *Evite un cas particulier, astuce courante.*
  - On ne peux pas ajouter « juste un if » dans une instruction atomique.

LA FILE NON-BLOQUANTE : PRINCIPES

- Un faux nœud (fake) pour gérer le cas où la liste est vide !
  - *Evite un cas particulier, astuce courante.*
  - On ne peux pas ajouter « juste un if » dans une instruction atomique.
- Pointeur tail mis à jour de façon opportuniste, pas forcément à jour !
  - *Vraie queue : fin de la liste.*

LA FILE NON-BLOQUANTE : PRINCIPES

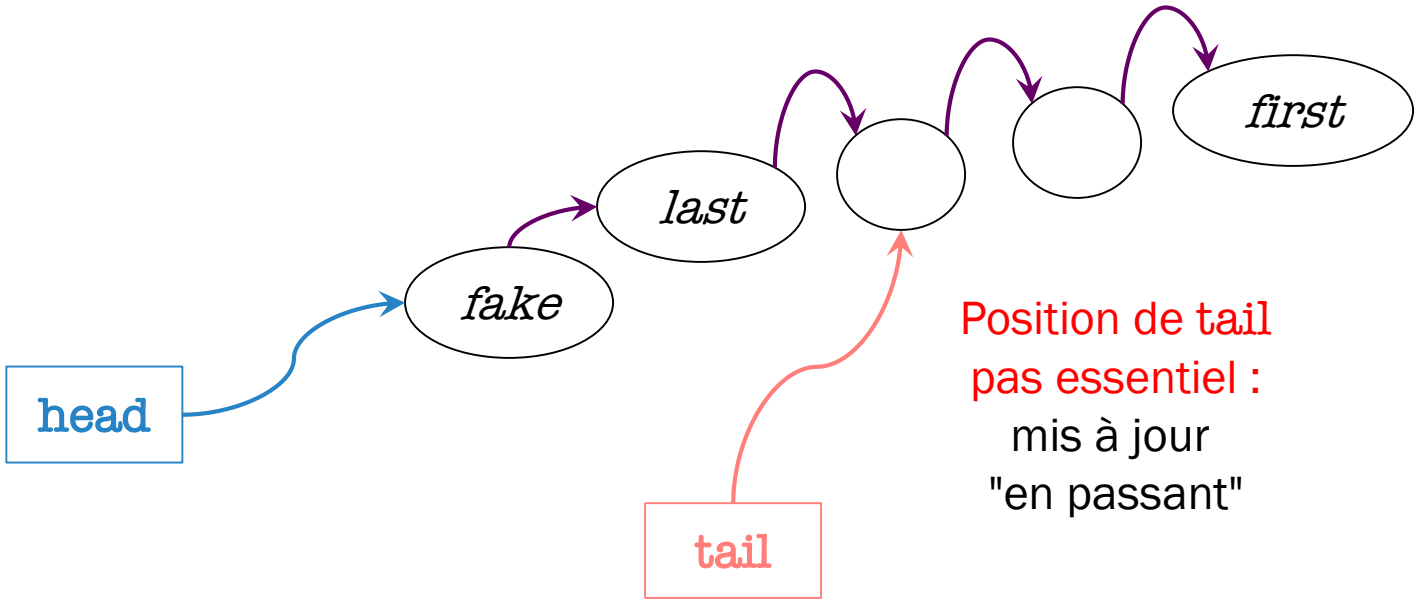
- Un faux nœud (fake) pour gérer le cas où la liste est vide !
  - *Evite un cas particulier, astuce courante.*
  - On ne peux pas ajouter « juste un if » dans une instruction atomique.
- Pointeur tail mis à jour de façon opportuniste, pas forcément à jour !
  - *Vraie queue : fin de la liste.*
  - **Idée pour l'insertion :** on part de tail et on fait avancer tail avec des CAS() le plus loin possible
  - On tente atomiquement un chaînage, si quelqu'un d'autre nous a devancé on recommence tout
  - Sinon succès : on tente de faire pointer tail vers nous atomiquement (pas grave si échec)

LA FILE NON-BLOQUANTE : PRINCIPES

- **Un faux nœud (fake) pour gérer le cas où la liste est vide !**
  - *Evite un cas particulier, astuce courante.*
  - On ne peux pas ajouter « juste un if » dans une instruction atomique.
- **Pointeur tail mis à jour de façon opportuniste, pas forcément à jour !**
  - *Vraie queue : fin de la liste.*
  - **Idée pour l'insertion :** on part de tail et on fait avancer tail avec des CAS() le plus loin possible
  - On tente atomiquement un chaînage, si quelqu'un d'autre nous a devancé on recommence tout
  - Sinon succès : on tente de faire pointer tail vers nous atomiquement (pas grave si échec)
- **À chaque instant (invariants) :**
  - Le second nœud après head est le nœud de tête
  - Les listes head et tail se rejoignent
  - Le dernier nœud de ces listes est le nœud en queue

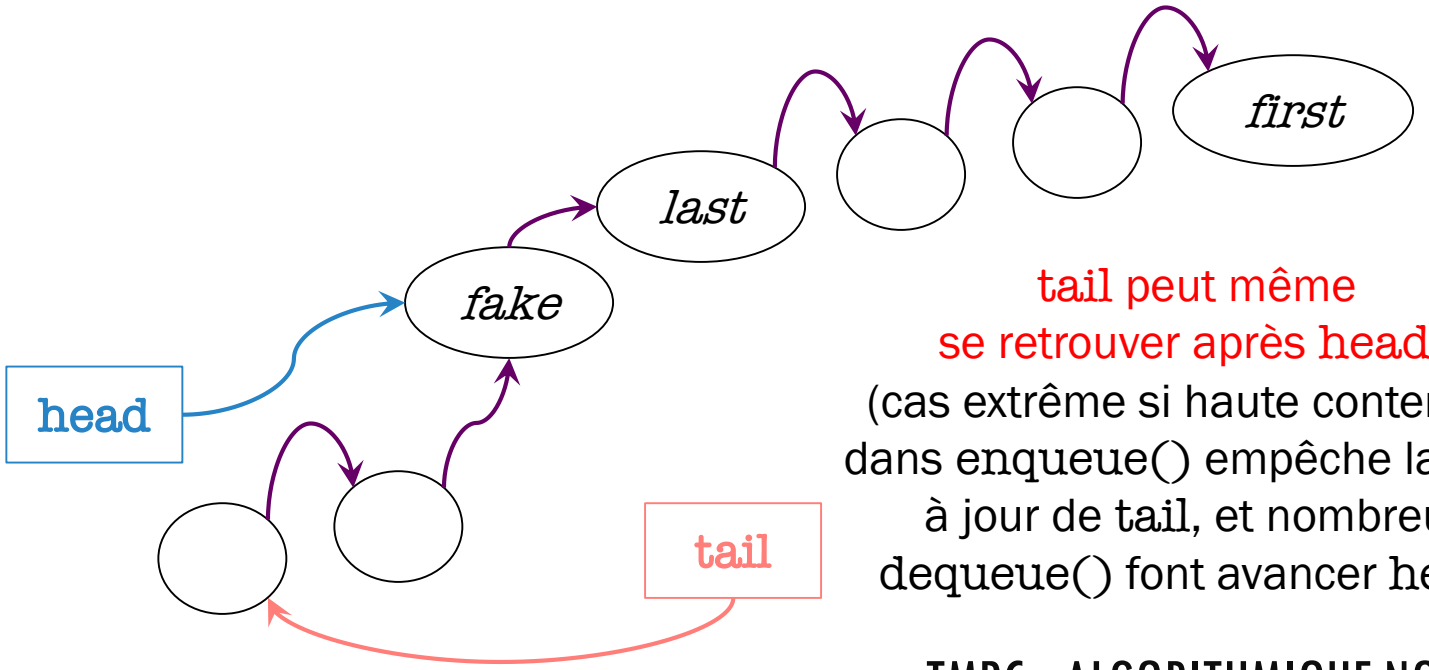
LA FILE NON-BLOQUANTE : PRINCIPES

- À chaque instant (invariants) :
  - Le second nœud après head est le nœud de tête
  - Les listes head et tail se rejoignent
  - Le dernier nœud de ces listes est le nœud en queue



LA FILE NON-BLOQUANTE : PRINCIPES

- À chaque instant (invariants) :
  - Le second nœud après head est le nœud de tête
  - Les listes head et tail se rejoignent
  - Le dernier nœud de ces listes est le nœud en queue

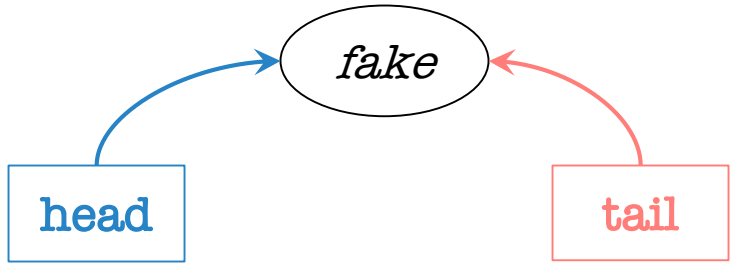


tail peut même
se retrouver après head !
(cas extrême si haute contention
dans enqueue() empêche la mise
à jour de tail, et nombreux
dequeue() font avancer head)

LA FILE NON-BLOQUANTE

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

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



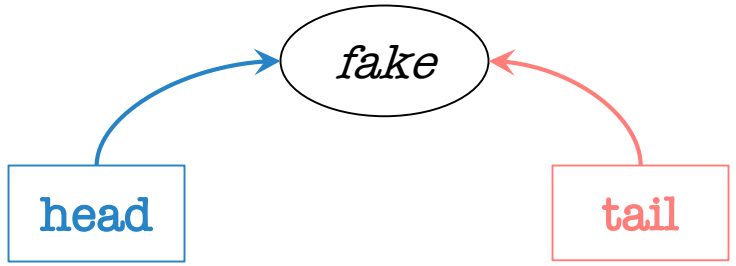
LA FILE NON-BLOQUANTE

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

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

Insertion de A sans concurrence.

```
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);
}
```



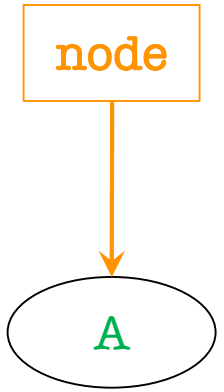
LA FILE NON-BLOQUANTE

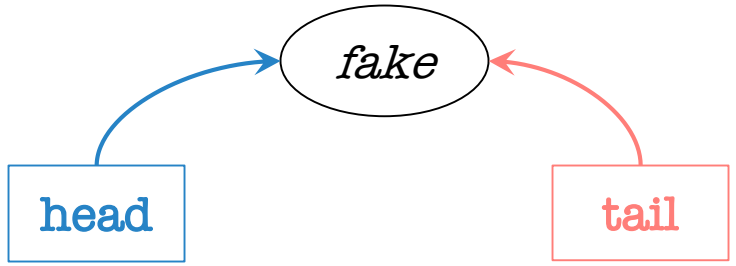
```
Class Queue {
  Node   head = new Node(null, null);
  Node   tail = head;
}
```

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

Insertion de A sans concurrence.

```
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);
}
```





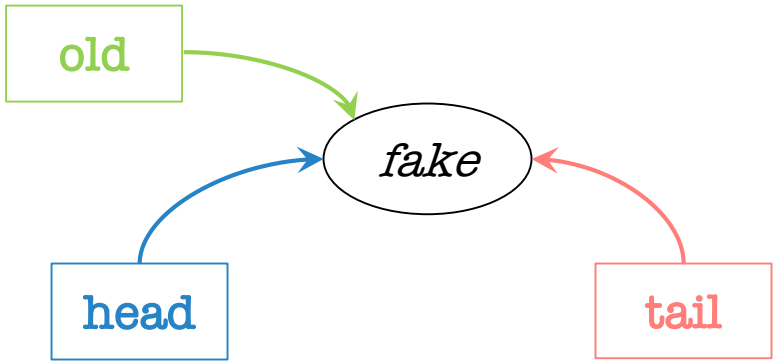
LA FILE NON-BLOQUANTE

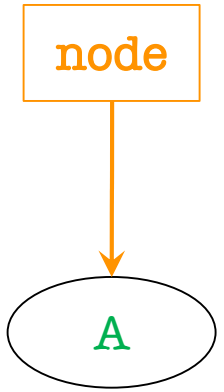
```
Class Queue {
  Node   head = new Node(null, null);
  Node   tail = head;
}
```

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

Insertion de A sans concurrence.

```
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);
}
```





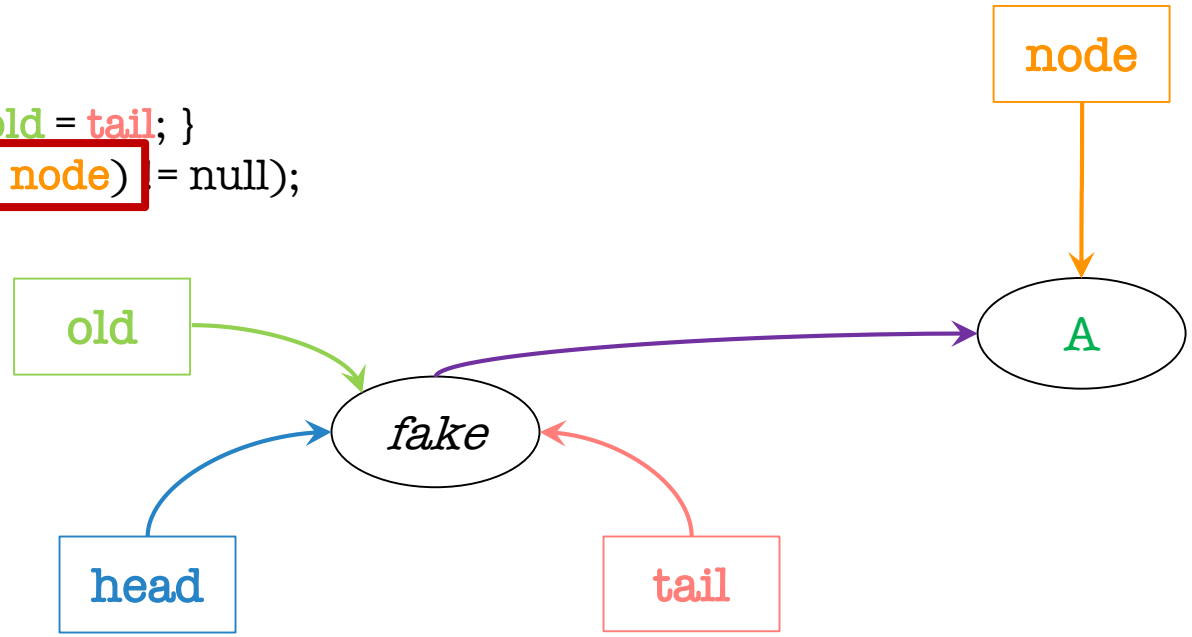
LA FILE NON-BLOQUANTE

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

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

Insertion de A sans concurrence.

```
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);
}
```



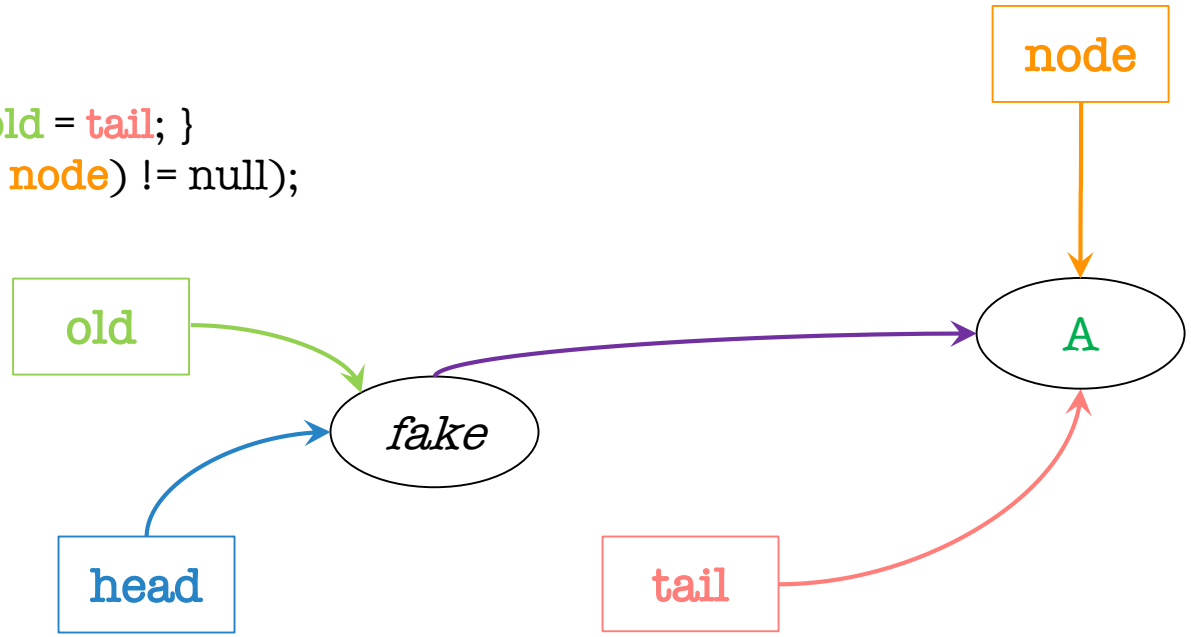
LA FILE NON-BLOQUANTE

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

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

Insertion de A sans concurrence.

```
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);
}
```



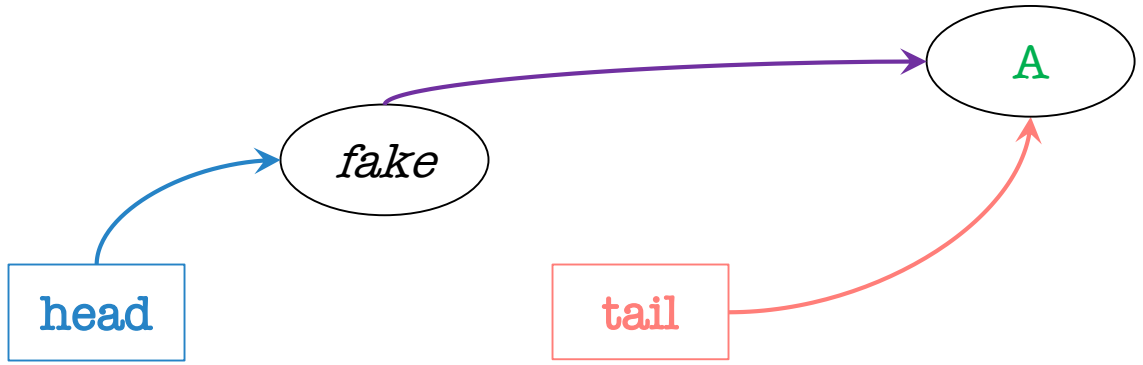
LA FILE NON-BLOQUANTE

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

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

Insertion de C en concurrence avec l'insertion de B.

```
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);
}
```



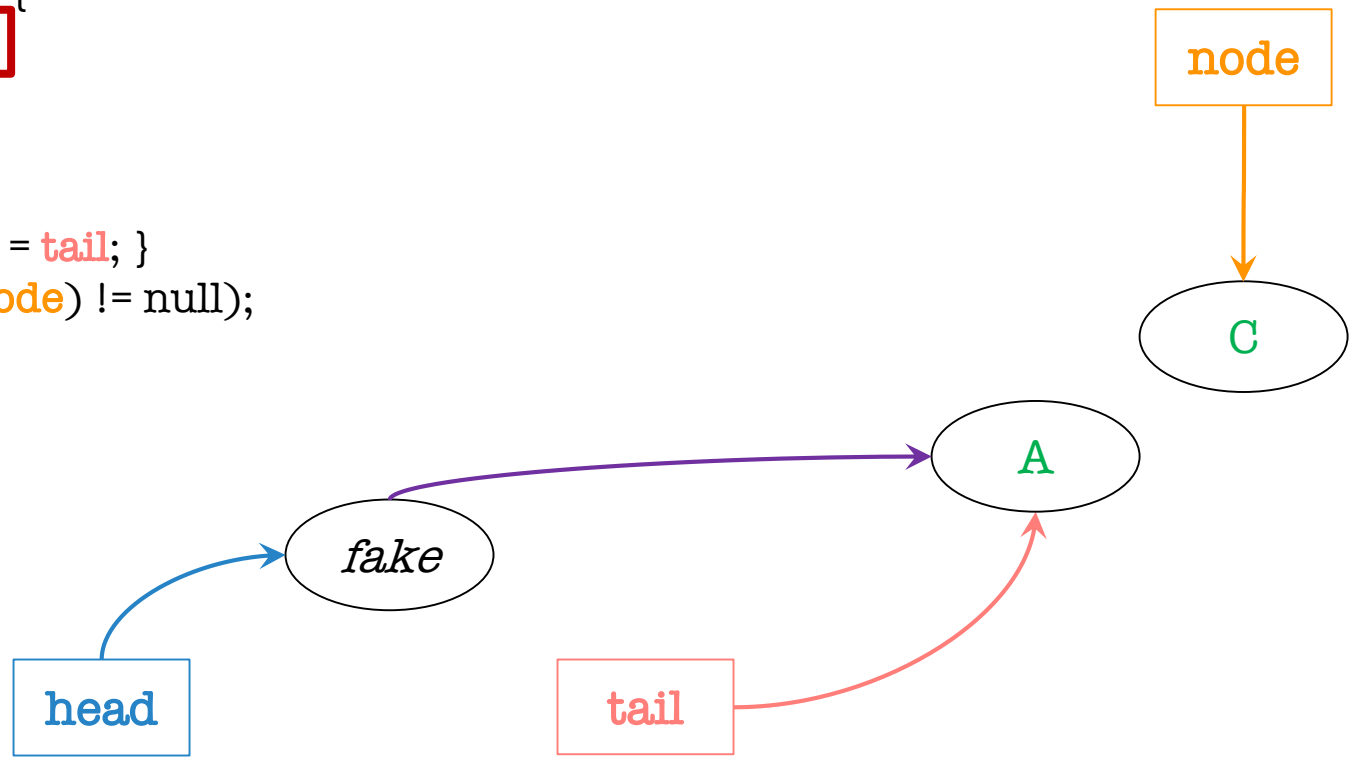
LA FILE NON-BLOQUANTE

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

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

Insertion de C en concurrence avec l'insertion de B.

```
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);
}
```



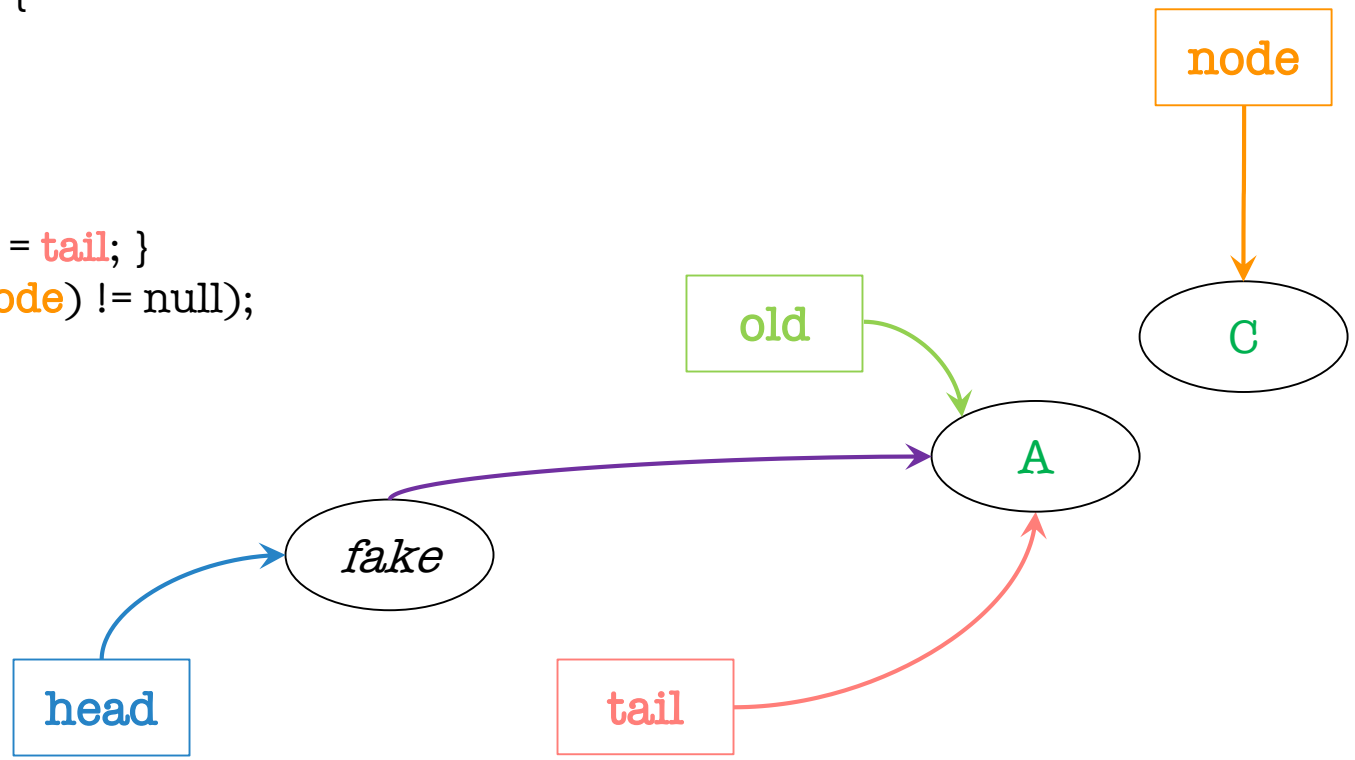
LA FILE NON-BLOQUANTE

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

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

Insertion de C en concurrence avec l'insertion de B.

```
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);
}
```



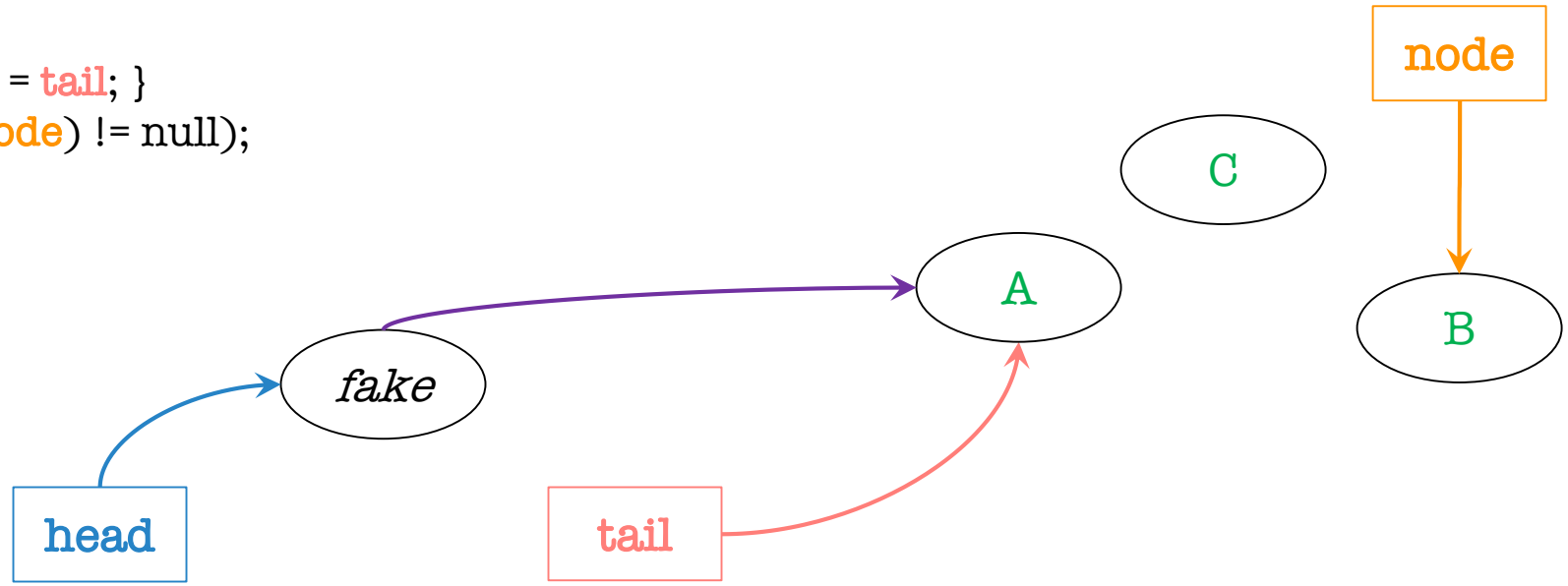
LA FILE NON-BLOQUANTE

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

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

Insertion de C en concurrence avec l'insertion de B.

```
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);
}
```



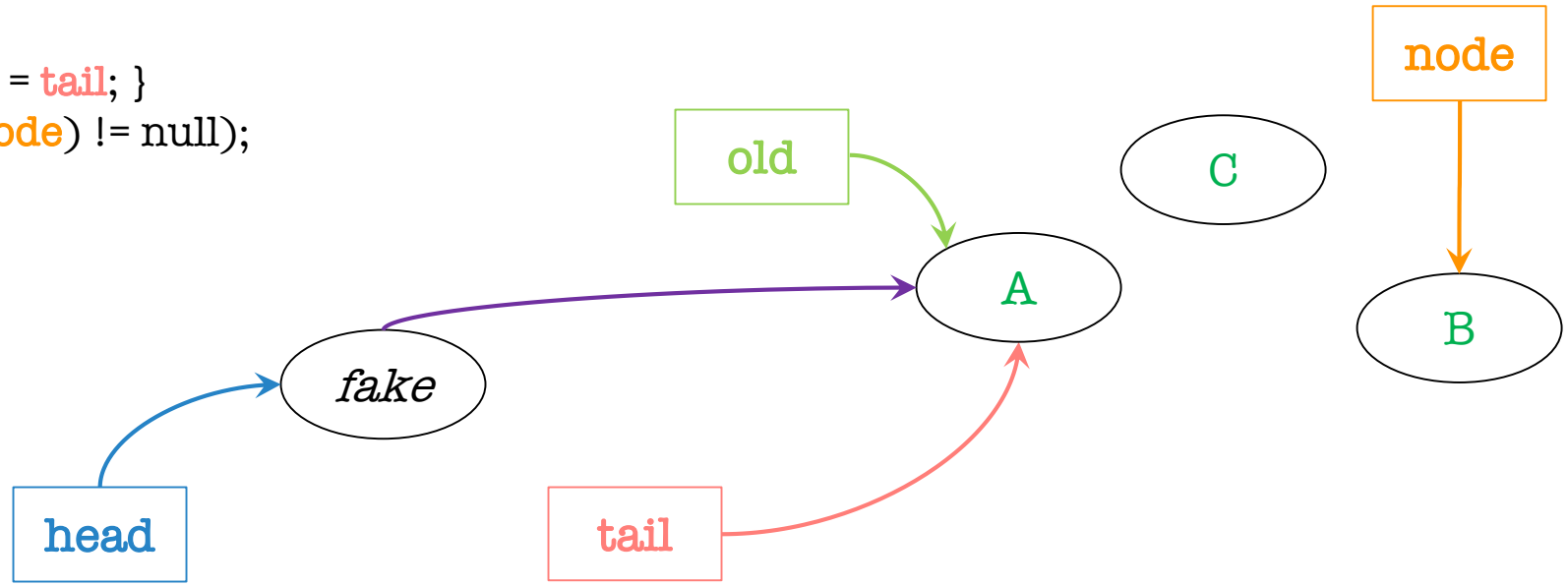
LA FILE NON-BLOQUANTE

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

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

Insertion de C en concurrence avec l'insertion de B.

```
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);
}
```



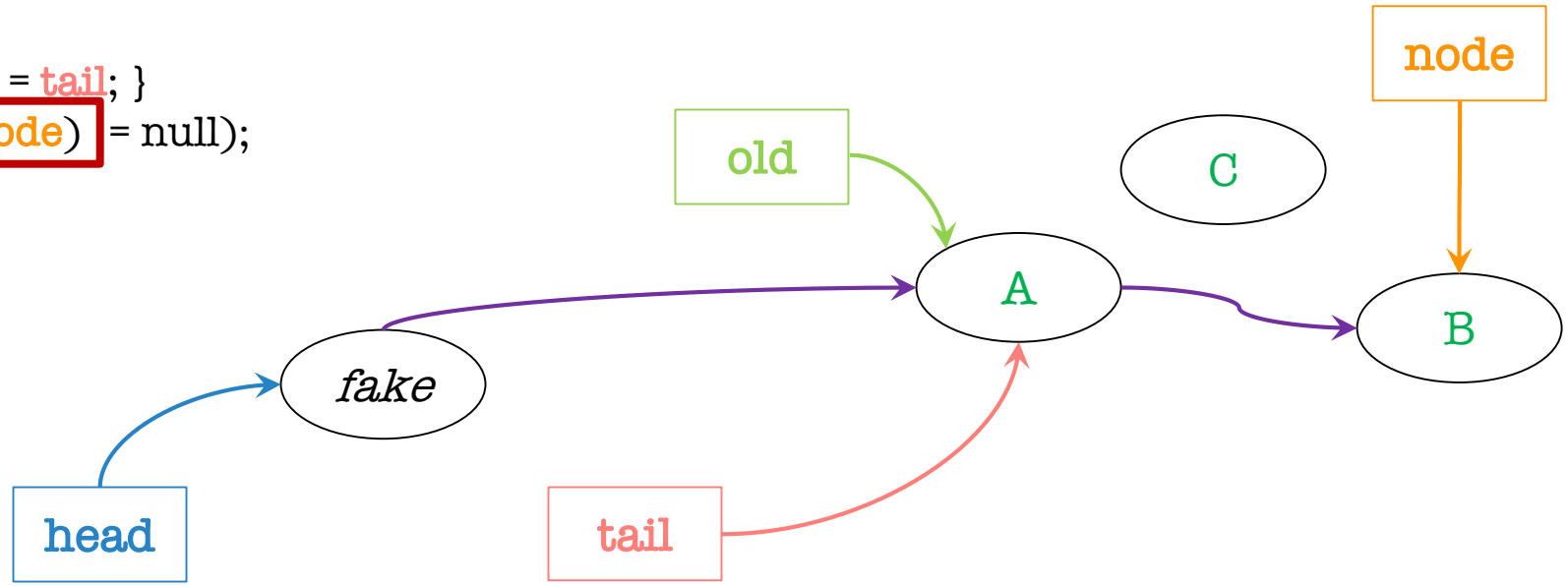
LA FILE NON-BLOQUANTE

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

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

Insertion de C en concurrence avec l'insertion de B.

```
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);
}
```



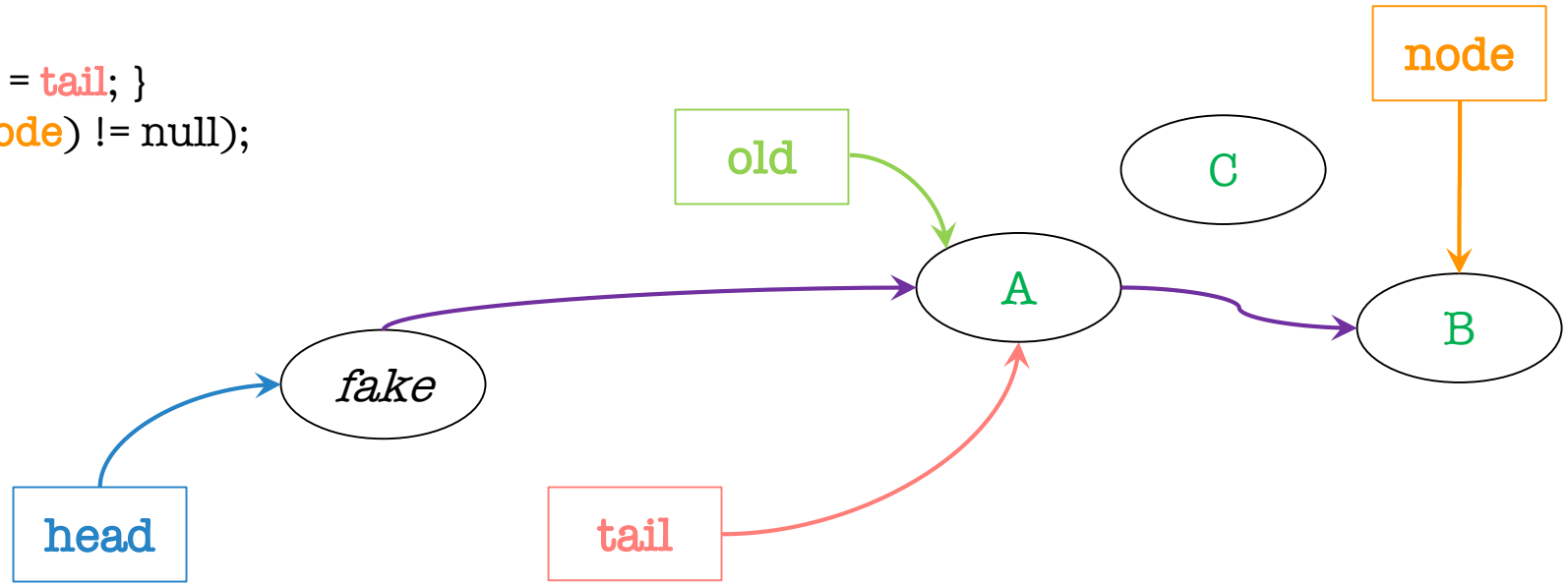
LA FILE NON-BLOQUANTE

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

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

Insertion de C en concurrence avec l'insertion de B.

```
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);
}
```



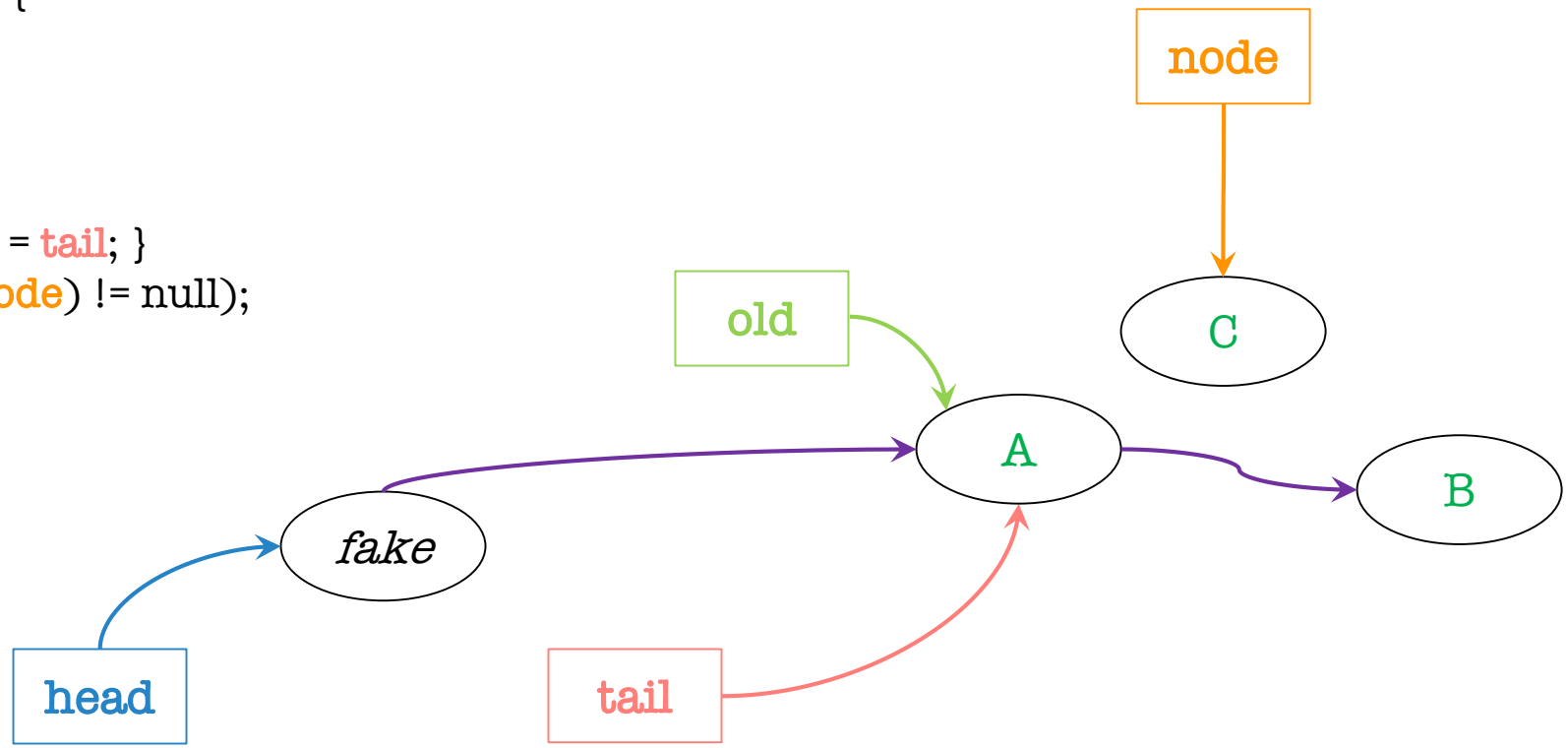
LA FILE NON-BLOQUANTE

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

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

Insertion de C en concurrence avec l'insertion de B.

```
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);
}
```



LA FILE NON-BLOQUANTE

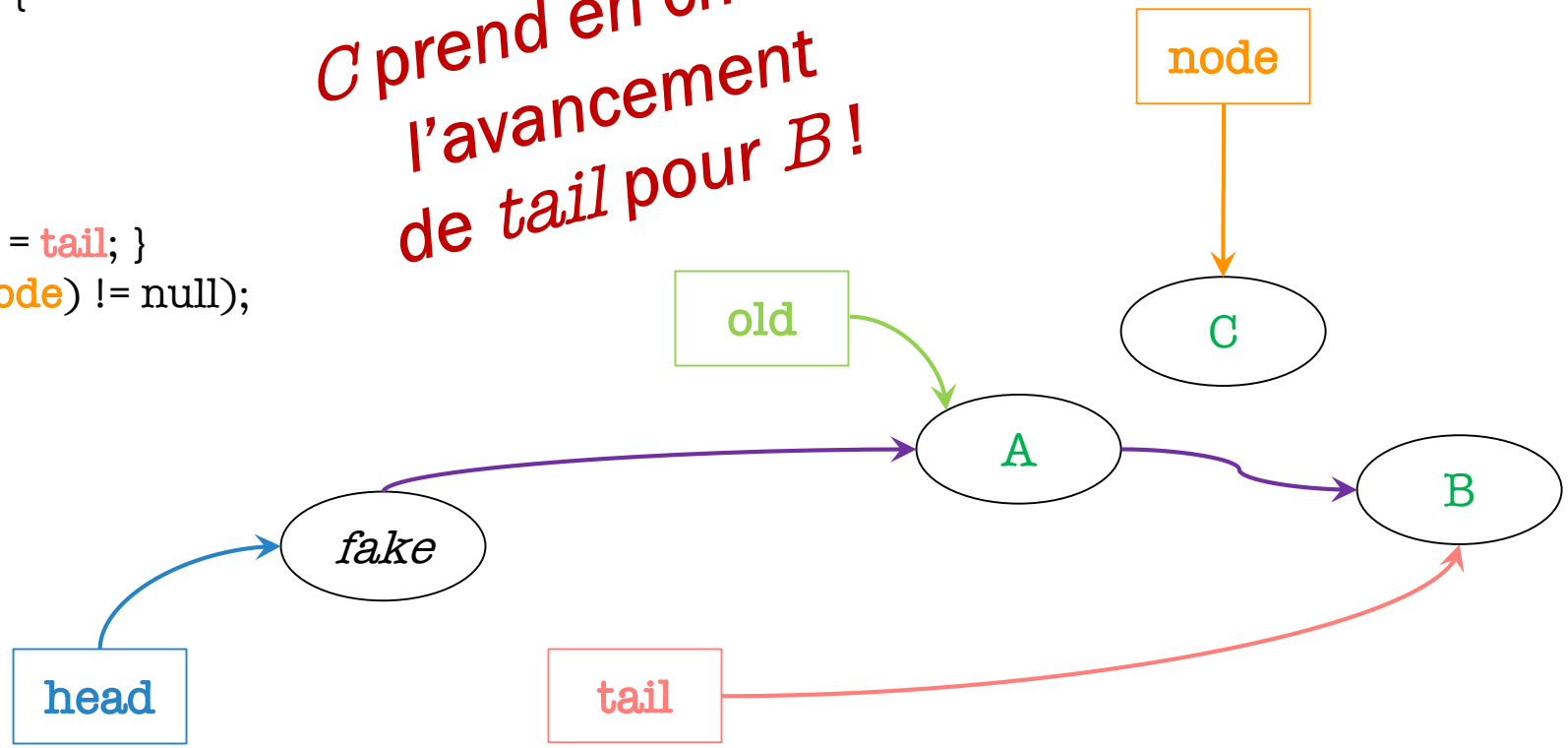
```
Class Queue {
  Node   head = new Node(null, null);
  Node   tail = head;
}
```

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

Insertion de C en concurrence avec l'insertion de B.

```
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 prend en charge l'avancement de tail pour B !



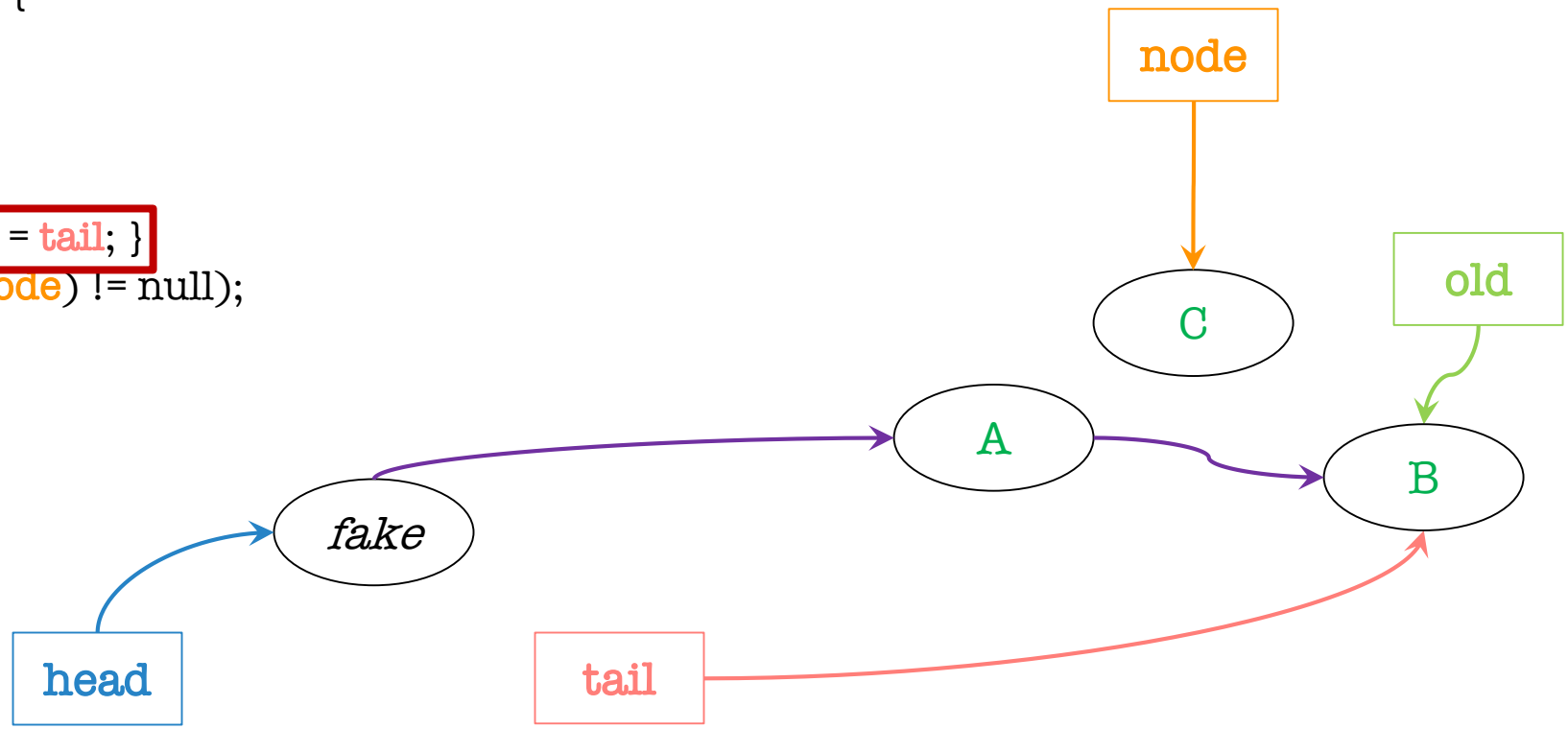
LA FILE NON-BLOQUANTE

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

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

Insertion de C en concurrence avec l'insertion de B.

```
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);
}
```



LA FILE NON-BLOQUANTE

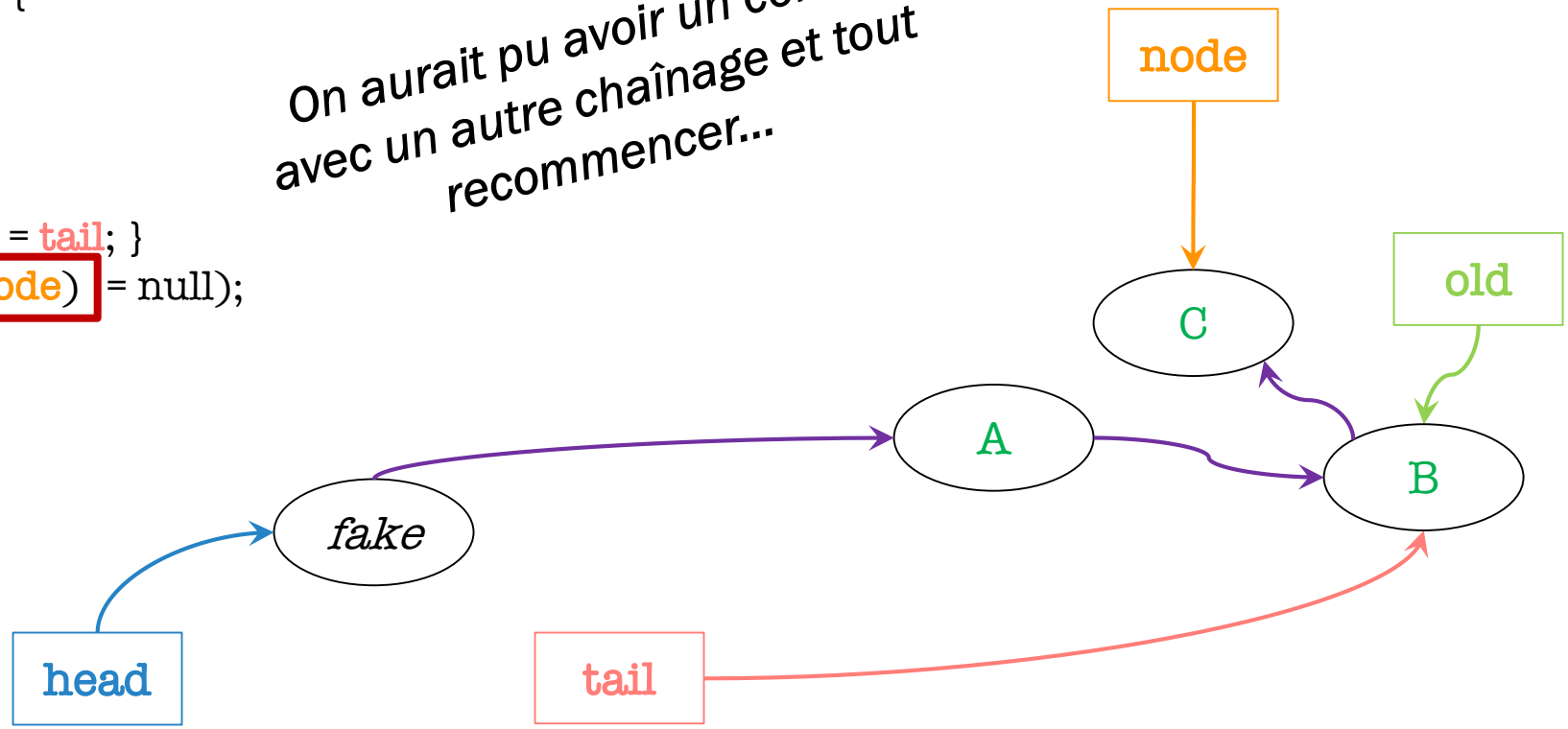
```
Class Queue {
  Node   head = new Node(null, null);
  Node   tail = head;
}
```

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

Insertion de C en concurrence avec l'insertion de B.

```
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);
}
```

On aurait pu avoir un conflit avec un autre chaînage et tout recommencer...



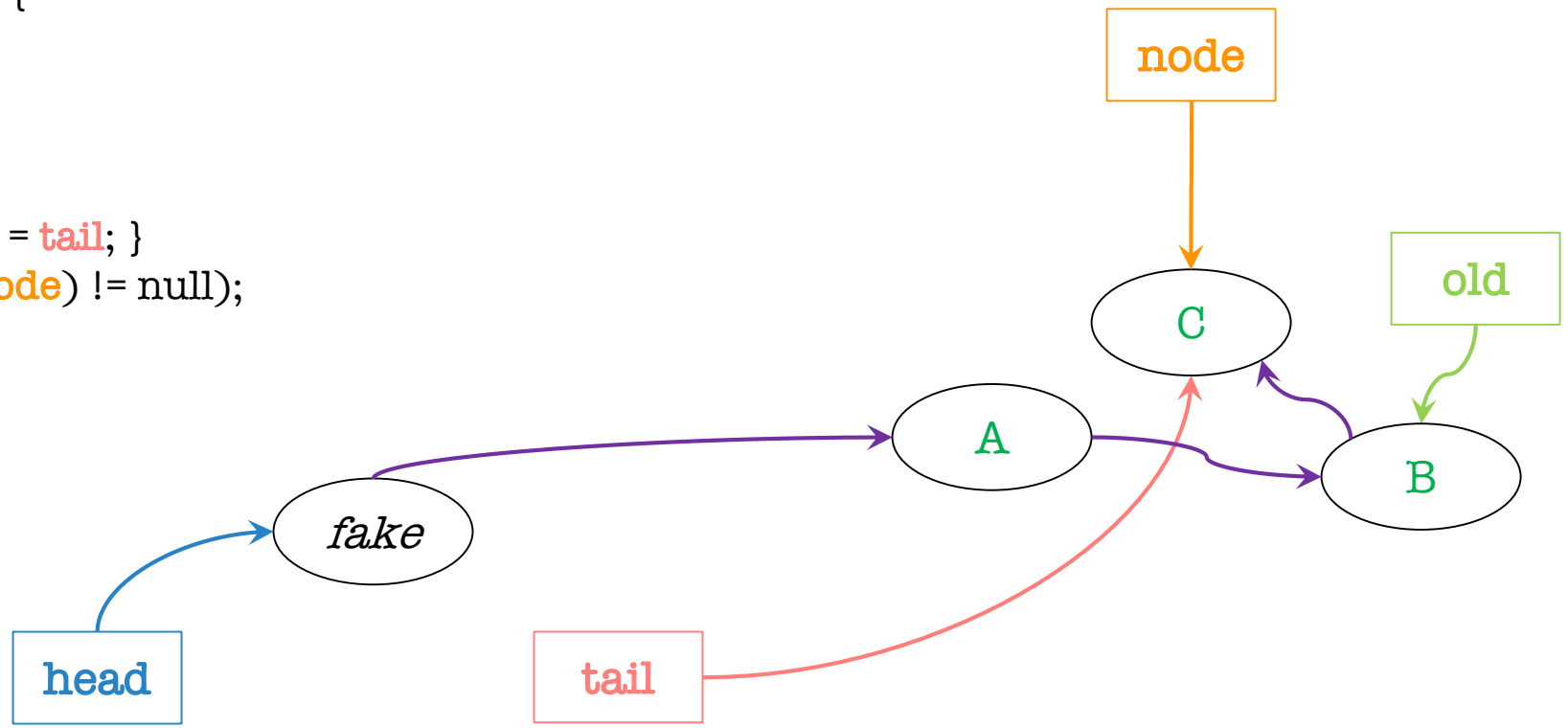
LA FILE NON-BLOQUANTE

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

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

Insertion de C en concurrence avec l'insertion de B.

```
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);
}
```



LA FILE NON-BLOQUANTE

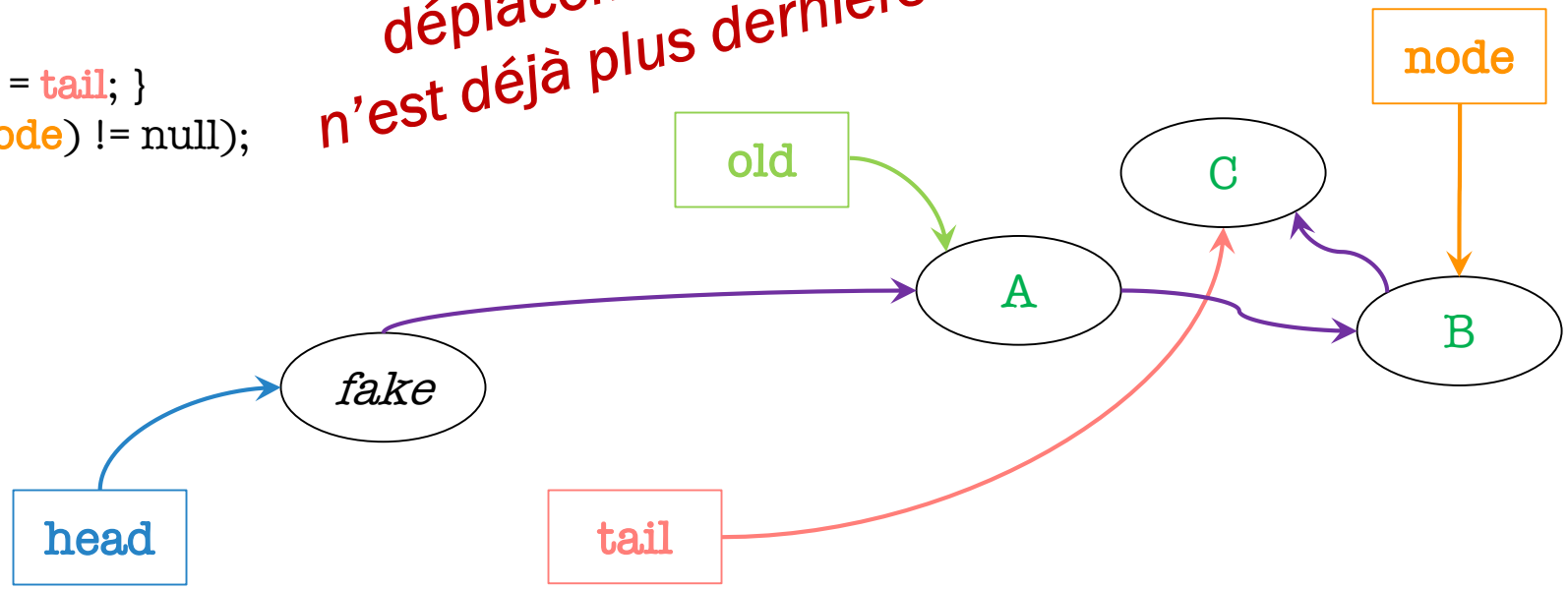
```
Class Queue {
  Node   head = new Node(null, null);
  Node   tail = head;
}
```

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

Insertion de C en concurrence avec l'insertion de B.

```
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);
}
```

Echec, car C a pris en charge le déplacement... On n'est déjà plus derniers !



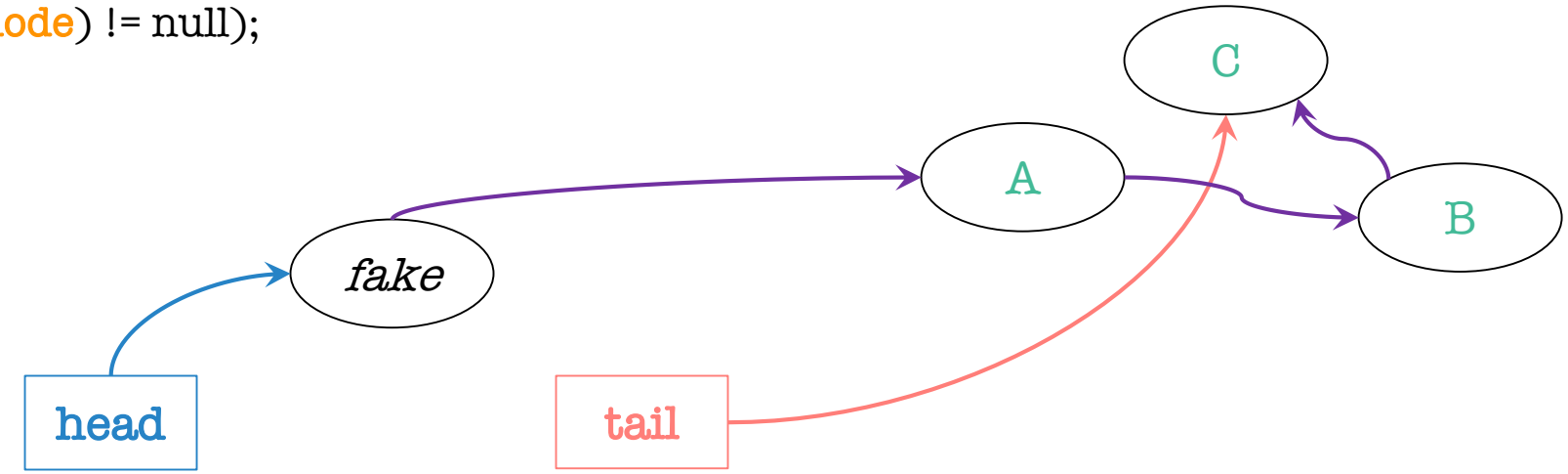
LA FILE NON-BLOQUANTE

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

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

Insertion de C en concurrence avec l'insertion de B.

```
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);
}
```



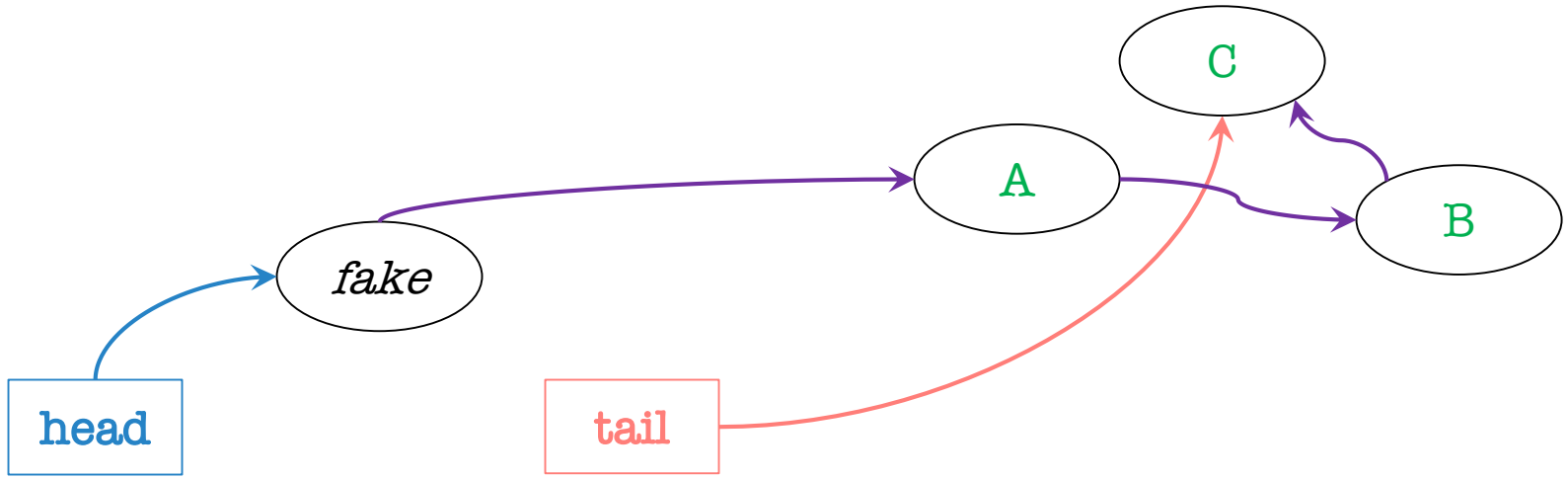
LA FILE NON-BLOQUANTE

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

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

Suppression de la tête sans concurrence.

```
Element Queue.dequeue() {
 do {
   Node res = head;
   if(res.next == null) error("No such element");
 } while(CAS(&head, res, res.next) != res);

 return res.next.value;
}
```



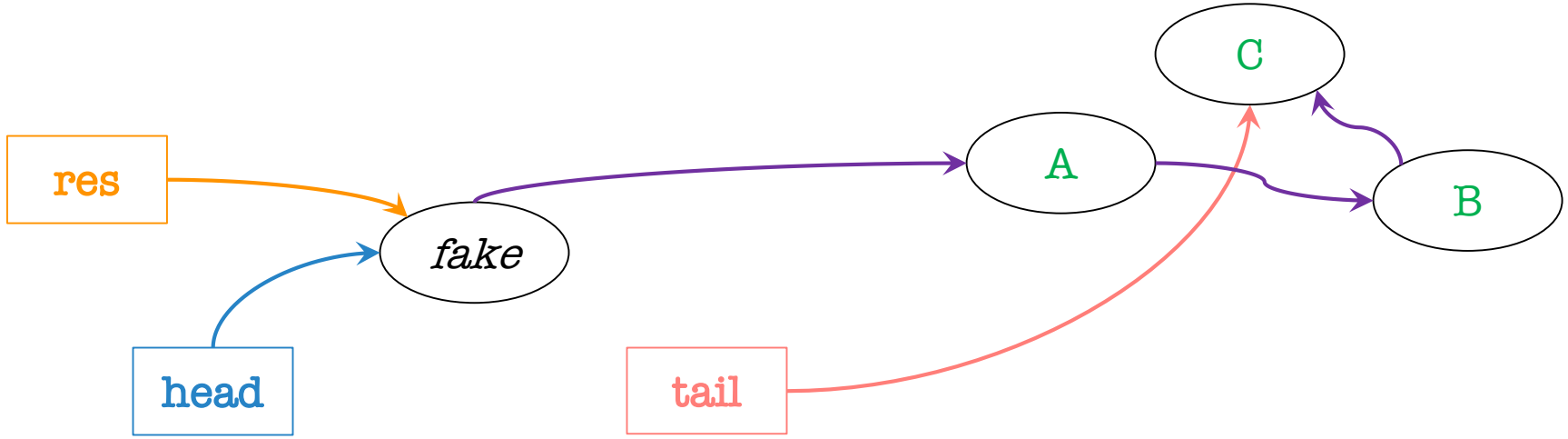
LA FILE NON-BLOQUANTE

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

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

Suppression de la tête sans concurrence.

```
Element Queue.dequeue() {
  do {
    Node res = head;
    if(res.next == null) error("No such element");
  } while(CAS(&head, res, res.next) != res);

  return res.next.value;
}
```



LA FILE NON-BLOQUANTE

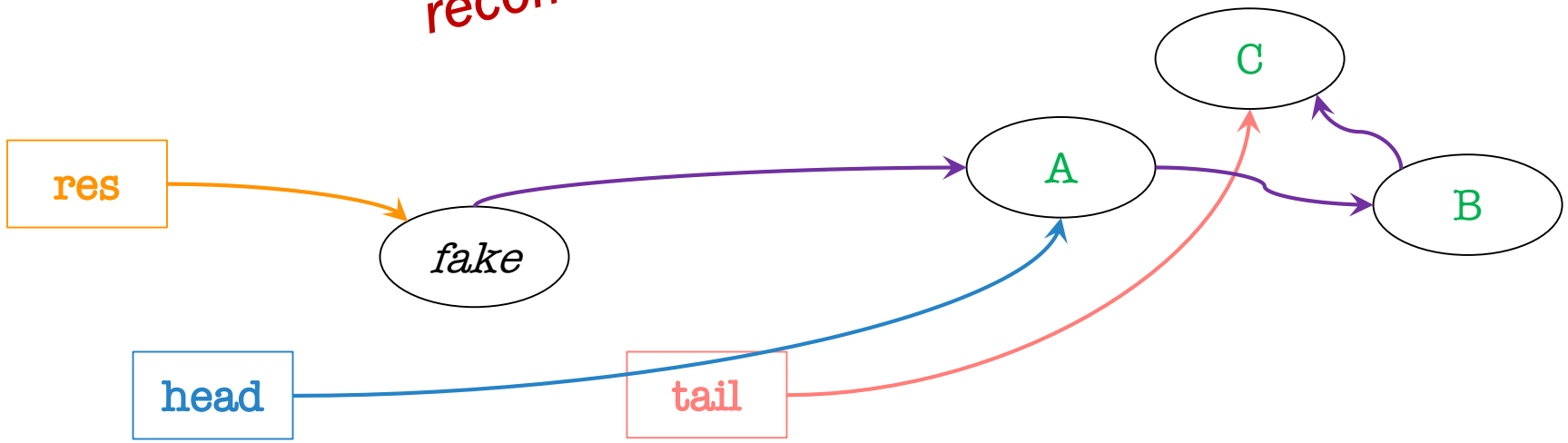
```
Class Queue {
  Node   head = new Node(null, null);
  Node   tail = head;
}
```

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

Suppression de la tête sans concurrence.

```
Element Queue.dequeue() {
 do {
   Node res = head;
   if(res.next == null) error("No such element");
 } while(CAS(&head, res, res.next) != res);

 return res.next.value;
}
```

Pourrait ne pas marcher si conflit entre deux *dequeue()* ici: on recommencerait alors...



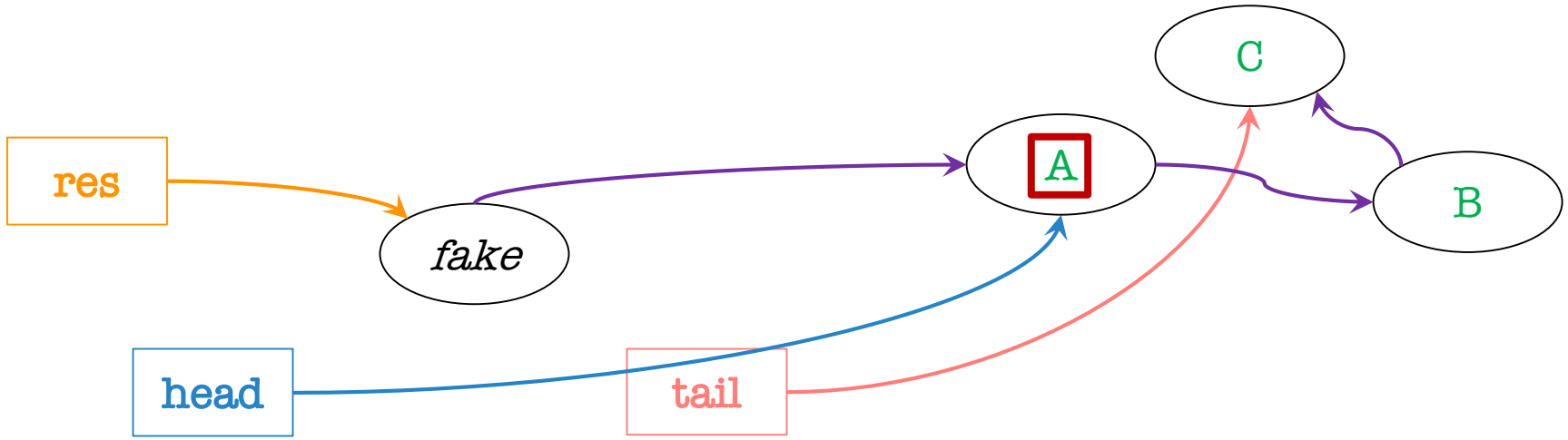
LA FILE NON-BLOQUANTE

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

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

Suppression de la tête sans concurrence.

```
Element Queue.dequeue() {
 do {
   Node res = head;
   if(res.next == null) error("No such element");
 } while(CAS(&head, res, res.next) != res);

 return res.next.value;
}
```



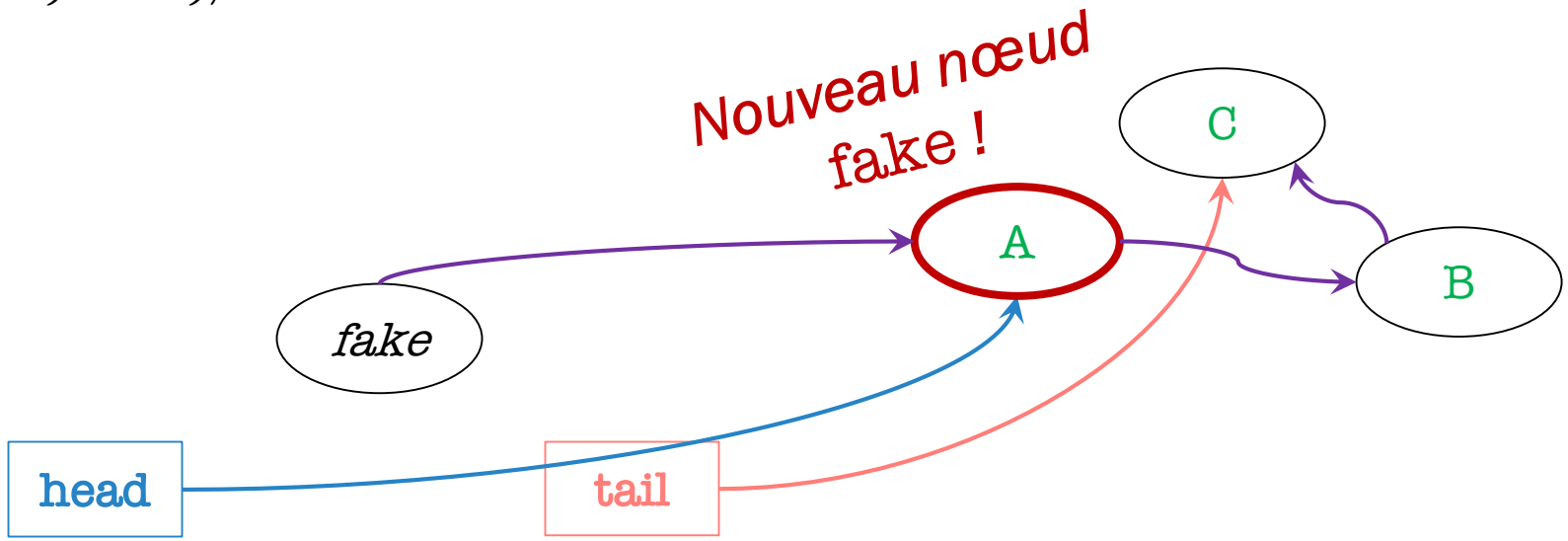
LA FILE NON-BLOQUANTE

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

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

Suppression de la tête sans concurrence.

```
Element Queue.dequeue() {
  do {
    Node res = head;
    if(res.next == null) error("No such element");
  } while(CAS(&head, res, res.next) != res);

  return res.next.value;
}
```



LA FILE NON-BLOQUANTE

- **Obstruction-free !**
  - *Si un seul thread élu, finira par réussir à faire son `enqueue()` ou `dequeue()`.*

LA FILE NON-BLOQUANTE

- **Obstruction-free !**
  - *Si un seul thread élu, finira par réussir à faire son `enqueue()` ou `dequeue()`.*
- **Lock-free !**
  - *Si les threads appellent infiniment souvent `enqueue()` ou `dequeue()`…*
  - *…au moins un passera de temps en temps !*

LA FILE NON-BLOQUANTE

- **Obstruction-free !**
  - *Si un seul thread élu, finira par réussir à faire son* `enqueue()` *ou* `dequeue()`*.*
- **Lock-free !**
  - *Si les threads appellent infiniment souvent* `enqueue()` *ou* `dequeue()`*…*
  - *…au moins un passera de temps en temps !*
  - **Preuve :** pour faire tourner un thread dans enqueue() ou dequeue(),
  - …il faut qu'il y ait des enqueue() et dequeue() qui terminent !

LA FILE NON-BLOQUANTE

- **Obstruction-free !**
  - *Si un seul thread élu, finira par réussir à faire son* `enqueue()` *ou* `dequeue()`*.*
- **Lock-free !**
  - *Si les threads appellent infiniment souvent* `enqueue()` *ou* `dequeue()`*…*
  - *…au moins un passera de temps en temps !*
  - **Preuve :** pour faire tourner un thread dans enqueue() ou dequeue(),
  - …il faut qu'il y ait des enqueue() et dequeue() qui terminent !
- **Pas wait-free !**
  - *On peut faire tourner un thread indéfiniment dans une des boucles !*
  - Avec suffisamment d'opérations concurrentes.

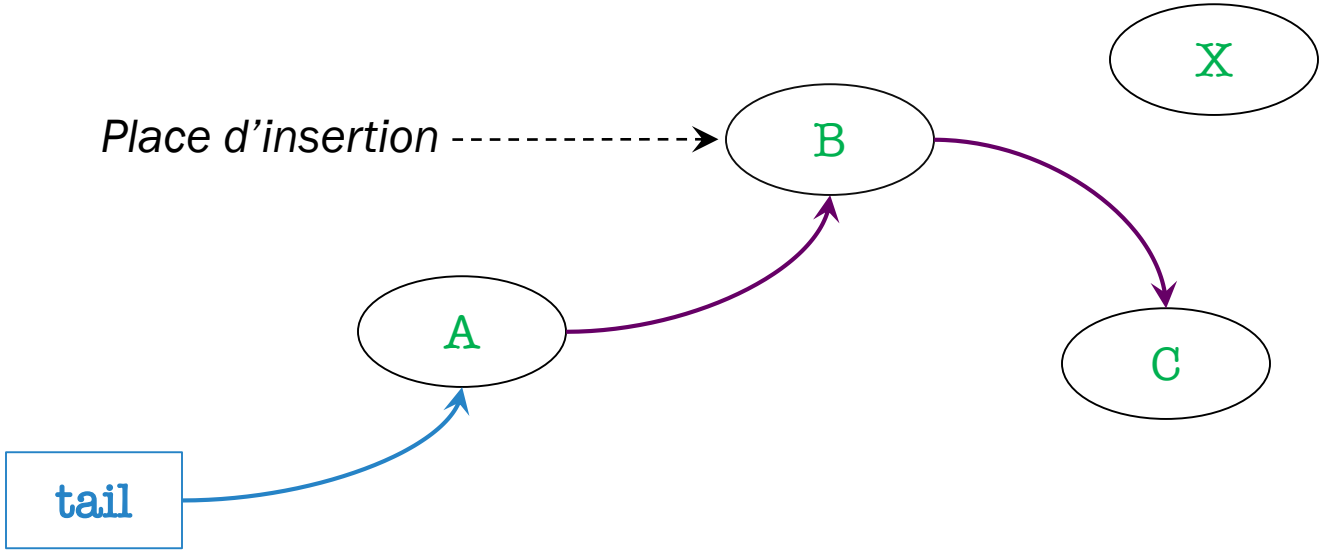
LA LISTE CHAÎNÉE

LA LISTE CHAÎNÉE

- **Le gros problème :** insertion et suppression au même endroit !

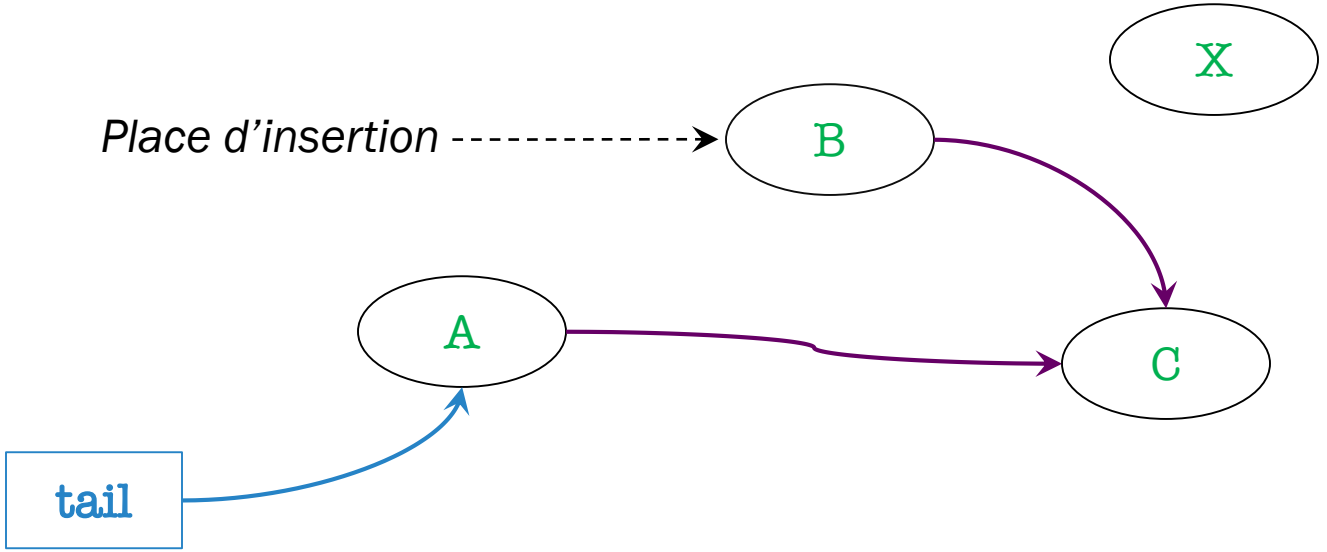
LA LISTE CHAÎNÉE

- **Le gros problème :** insertion et suppression au même endroit !
- **Exemple :** suppression de `B` et insertion de `X` après `B` concurrentes.



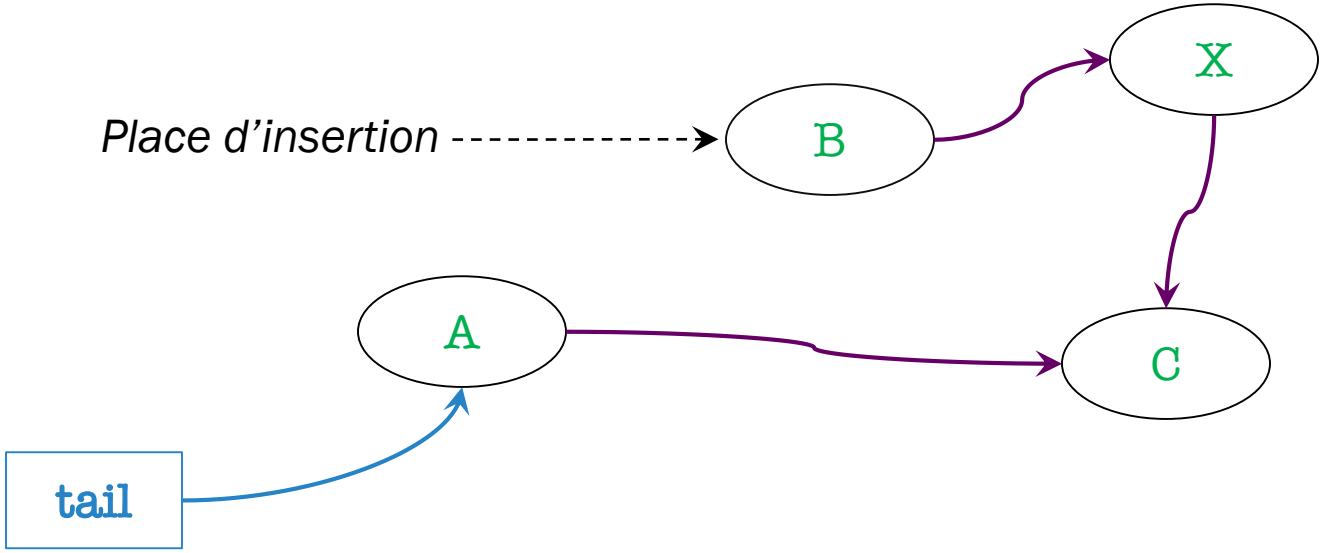
LA LISTE CHAÎNÉE

- **Le gros problème :** insertion et suppression au même endroit !
- **Exemple :** suppression de `B` et insertion de `X` après `B` concurrentes.



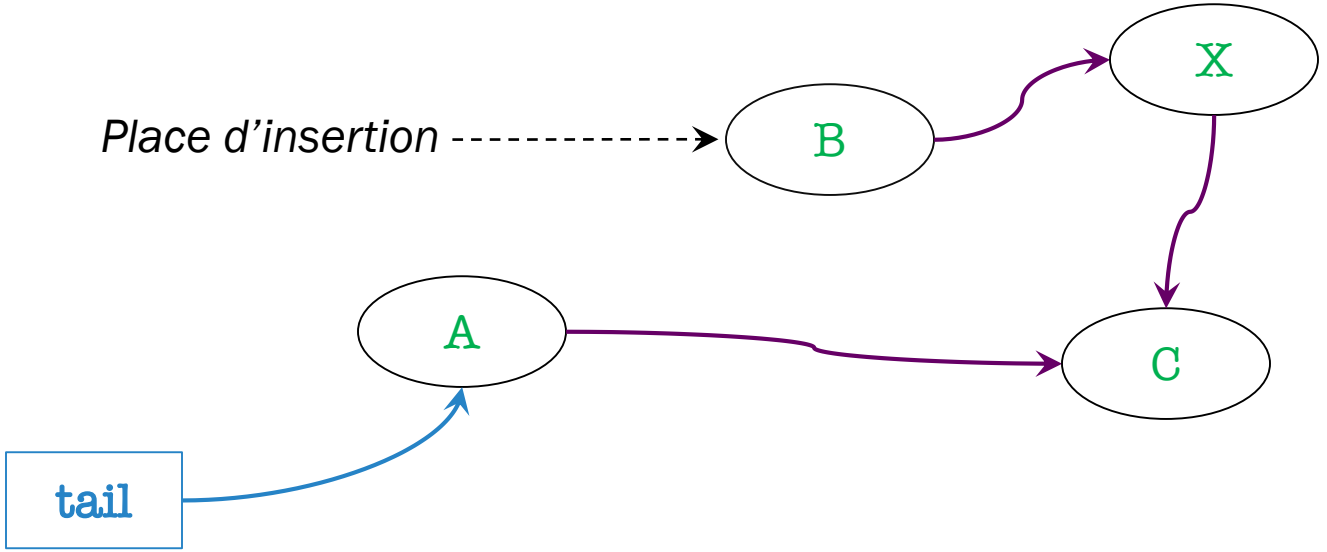
LA LISTE CHAÎNÉE

- **Le gros problème :** insertion et suppression au même endroit !
- **Exemple :** suppression de `B` et insertion de `X` après `B` concurrentes.



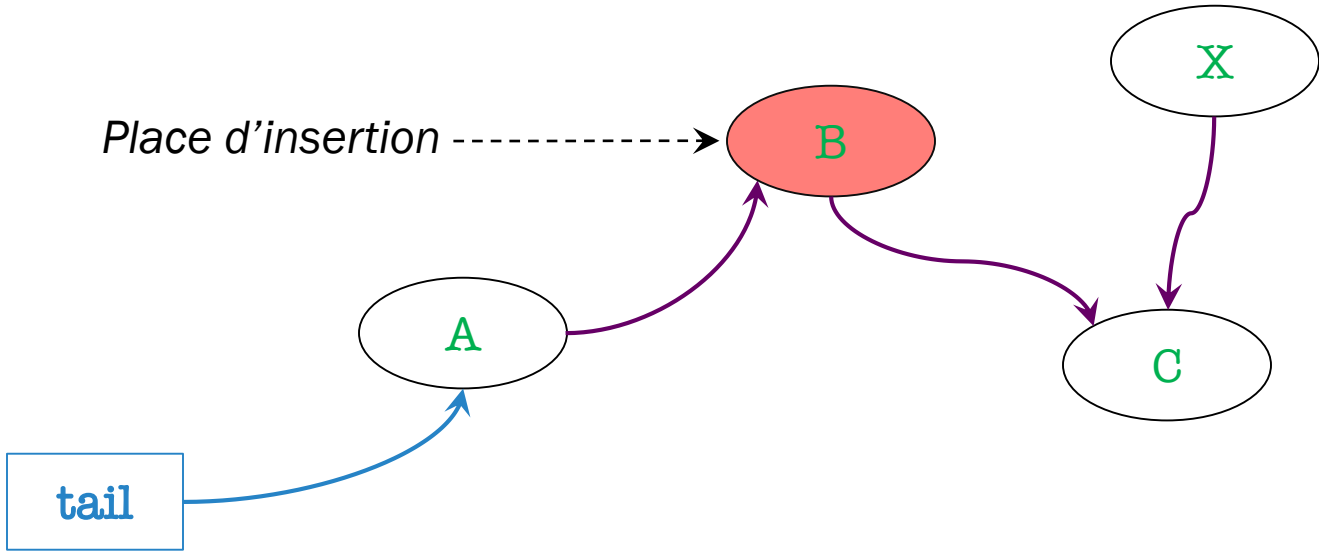
LA LISTE CHAÎNÉE

- **Le gros problème :** insertion et suppression au même endroit !
- **Exemple :** suppression de `B` et insertion de `X` après `B` concurrentes.
  - **Résultat :** un graphe au lieu d'une liste…



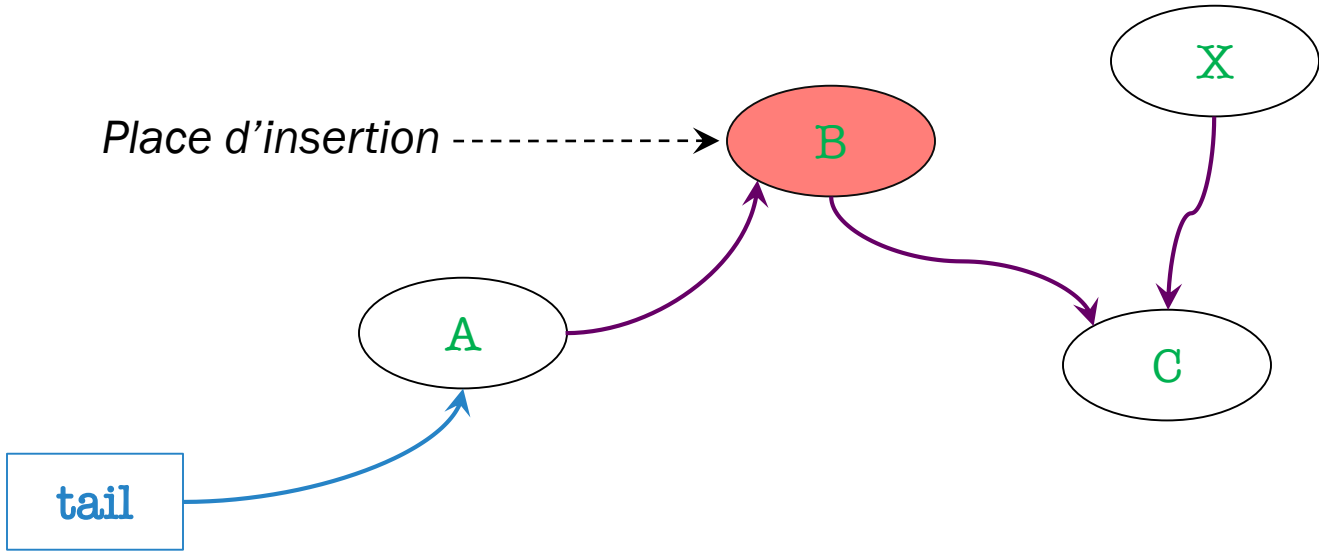
LA LISTE CHAÎNÉE

- **Le gros problème :** insertion et suppression au même endroit !



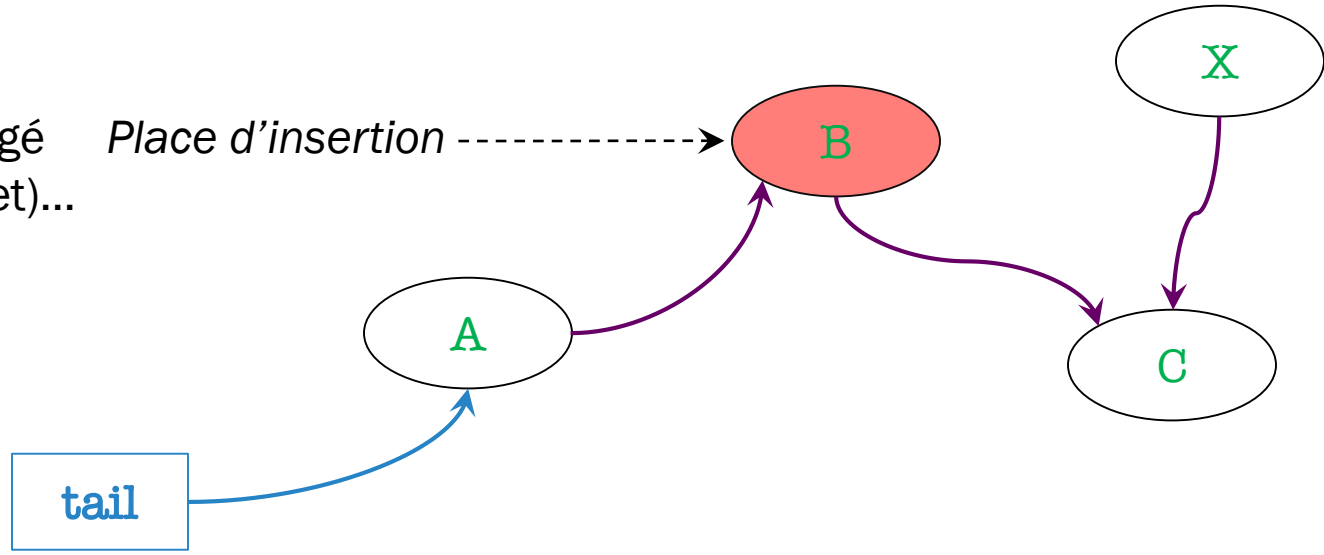
LA LISTE CHAÎNÉE

- **Le gros problème :** insertion et suppression au même endroit !
- **Principe de solution [Tim Harris, DISC 2011] :**
  - On change la couleur du nœud qui va être supprimé.
  - Pendant insertion, si couleur du nœud d'insertion a changé, le supprimer et recommencer.
    - Suppressions aussi pendant parcours.



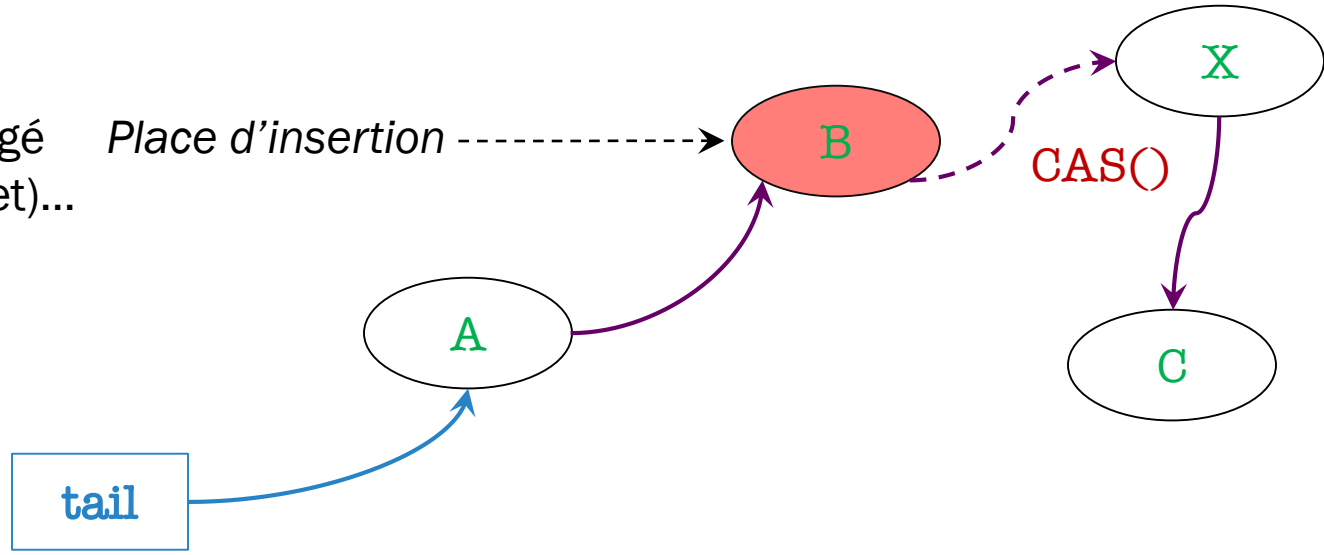
LA LISTE CHAÎNÉE

- **Le gros problème :** insertion et suppression au même endroit !
- **Principe de solution [Tim Harris, DISC 2011] :**
  - On change la couleur du nœud qui va être supprimé.
  - Pendant insertion, si couleur du nœud d'insertion a changé, le supprimer et recommencer.
    - Suppressions aussi pendant parcours.
- **Problème :**
  - On doit atomiquement tester si couleur a changé et faire le chaînage (test `next` pas changé + set)…



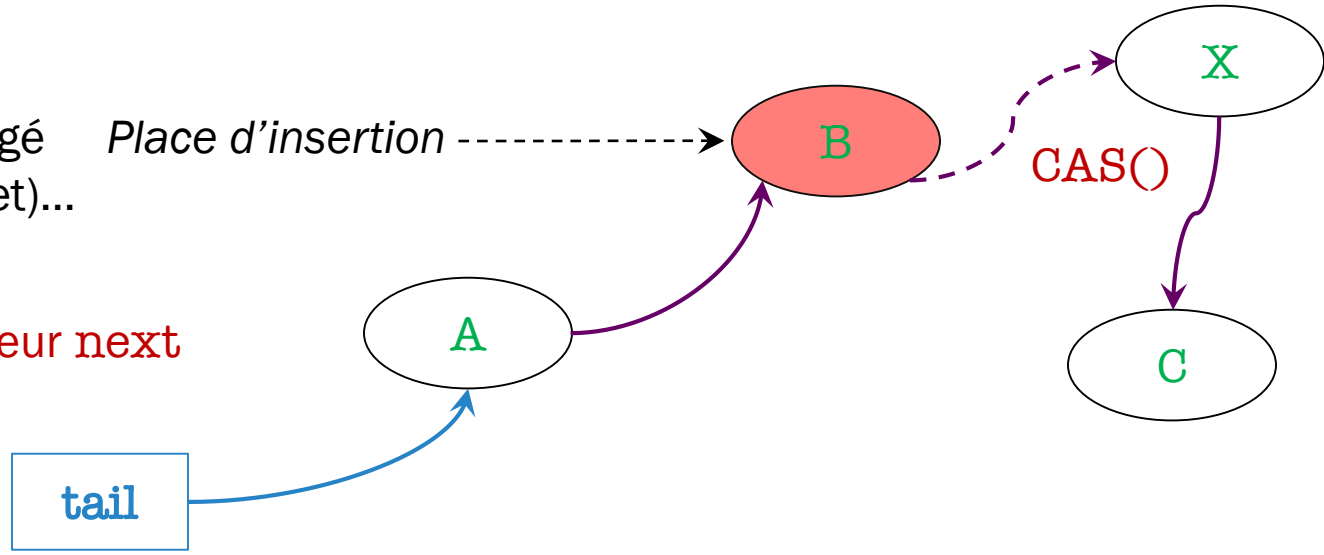
LA LISTE CHAÎNÉE

- **Le gros problème :** insertion et suppression au même endroit !
- **Principe de solution [Tim Harris, DISC 2011] :**
  - On change la couleur du nœud qui va être supprimé.
  - Pendant insertion, si couleur du nœud d'insertion a changé, le supprimer et recommencer.
    - Suppressions aussi pendant parcours.
- **Problème :**
  - On doit atomiquement tester si couleur a changé et faire le chaînage (test `next` pas changé + set)…
  - *On n'a que des CAS !*



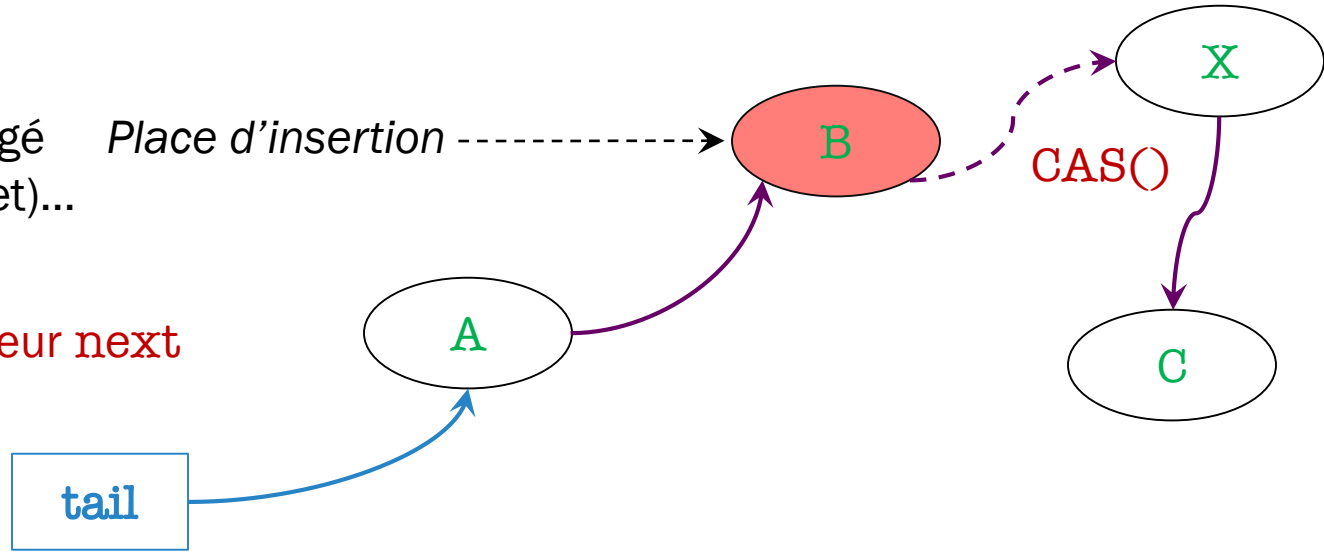
LA LISTE CHAÎNÉE

- **Le gros problème :** insertion et suppression au même endroit !
- **Principe de solution [Tim Harris, DISC 2011] :**
  - On change la couleur du nœud qui va être supprimé.
  - Pendant insertion, si couleur du nœud d'insertion a changé, le supprimer et recommencer.
    - Suppressions aussi pendant parcours.
- **Problème :**
  - On doit atomiquement tester si couleur a changé et faire le chaînage (test `next` pas changé + set)…
  - *On n'a que des CAS !*
  - **Solution :** utiliser le bit de poids faible du pointeur `next` pour stocker la couleur !



LA LISTE CHAÎNÉE

- **Le gros problème :** insertion et suppression au même endroit !
- **Principe de solution [Tim Harris, DISC 2011] :**
  - On change la couleur du nœud qui va être supprimé.
  - Pendant insertion, si couleur du nœud d'insertion a changé, le supprimer et recommencer.
    - Suppressions aussi pendant parcours.
- **Problème :**
  - On doit atomiquement tester si couleur a changé et faire le chaînage (test `next` pas changé + set)…
  - *On n'a que des CAS !*
  - **Solution :** utiliser le bit de poids faible du pointeur `next` pour stocker la couleur !
  - *Pas utilisé car alignement… (au pire le forcer)*

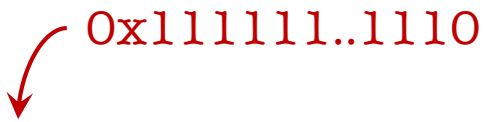


LA LISTE CHAÎNÉE

```
typedef uintptr_t coloredPointer;

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

LA LISTE CHAÎNÉE

```
typedef uintptr_t coloredPointer;

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



LA LISTE CHAÎNÉE

```
typedef uintptr_t coloredPointer;

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

0x111111..1110

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

LA LISTE CHAÎNÉE

```
typedef uintptr_t coloredPointer;

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

0x111111..1110

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

- **Idée :** avant de supprimer un nœud n, marquer n.next
  - **Liste cohérente :** tout nœud non supprimé toujours dans la liste et liste toujours connectée.
  - Des nœuds marqués « à supprimer » peuvent être dans la liste

LA LISTE CHAÎNÉE

- On va étudier un peu l'algorithme… « À gros grain », pas besoin de tout comprendre.

LA LISTE CHAÎNÉE

- **On va étudier un peu l'algorithme… « À gros grain », pas besoin de tout comprendre.**
- Parfois assez « prise de tête », car il faut avoir toutes les opérations bien en tête pour imaginer enchaînements

LA LISTE CHAÎNÉE

- **On va étudier un peu l'algorithme… « À gros grain », pas besoin de tout comprendre.**
- Parfois assez « prise de tête », car il faut avoir toutes les opérations bien en tête pour imaginer enchaînements
- Et pourtant seulement une liste chaînée ici ! Imaginez une structure plus complexe…

LA LISTE CHAÎNÉE

- **On va étudier un peu l'algorithme… « À gros grain », pas besoin de tout comprendre.**
- Parfois assez « prise de tête », car il faut avoir toutes les opérations bien en tête pour imaginer enchaînements
- Et pourtant seulement une liste chaînée ici ! Imaginez une structure plus complexe…
- **Idée :** voir quelques astuces que ces algos utilisent, et comprendre pourquoi c'est compliqué ☺

LA LISTE CHAÎNÉE

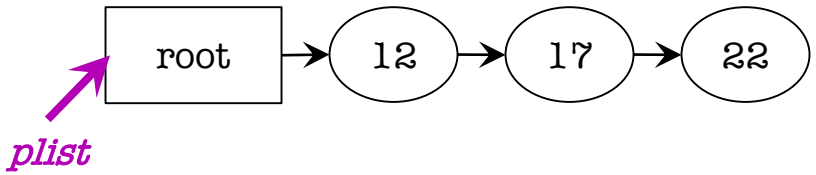
- **On va étudier un peu l'algorithme… « À gros grain », pas besoin de tout comprendre.**
- Parfois assez « prise de tête », car il faut avoir toutes les opérations bien en tête pour imaginer enchaînements
- Et pourtant seulement une liste chaînée ici ! Imaginez une structure plus complexe…
- **Idée :** voir quelques astuces que ces algos utilisent, et comprendre pourquoi c'est compliqué ☺

- **Pour les détails, voir :** https://timharris.uk/papers/2001-disc.pdf

LA LISTE CHAÎNÉE TRIÉE D'ENTIERS : PARCOURS ET COMPACTION

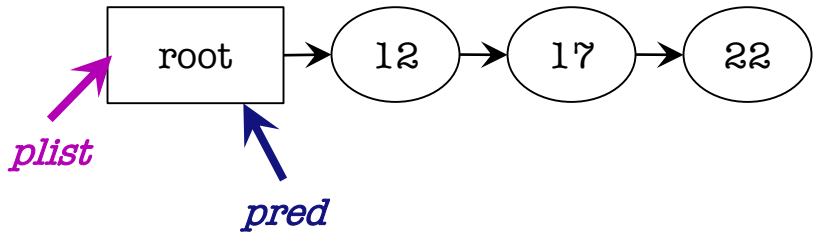
LA LISTE CHAÎNÉE : PARCOURS

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



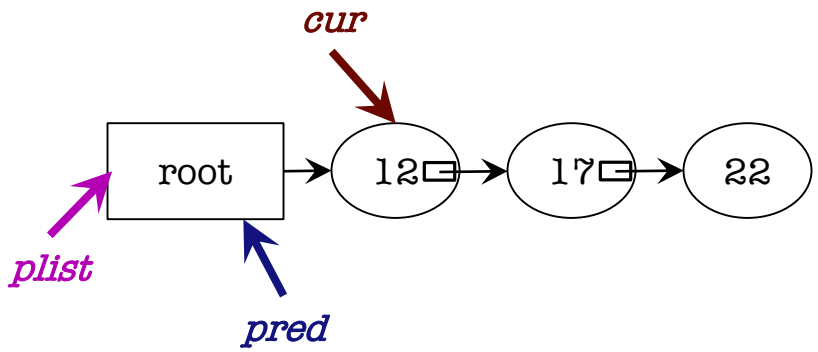
LA LISTE CHAÎNÉE : PARCOURS

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



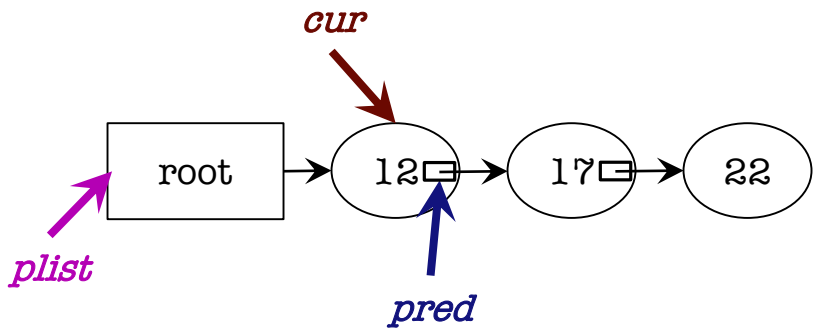
LA LISTE CHAÎNÉE : PARCOURS

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



LA LISTE CHAÎNÉE : PARCOURS

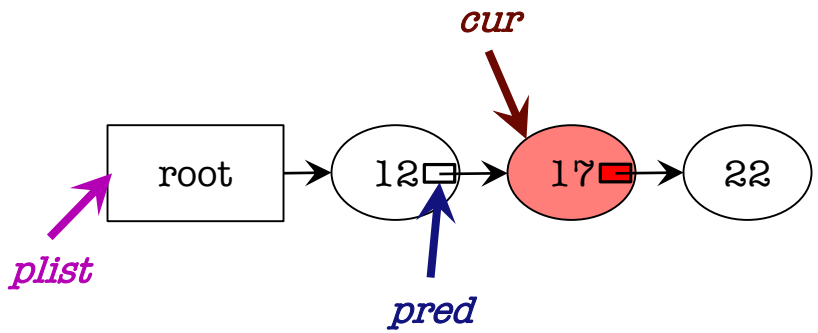
```
void del(coloredPointer* plist, int value) {
restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```




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

LA LISTE CHAÎNÉE : COMPACTION

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



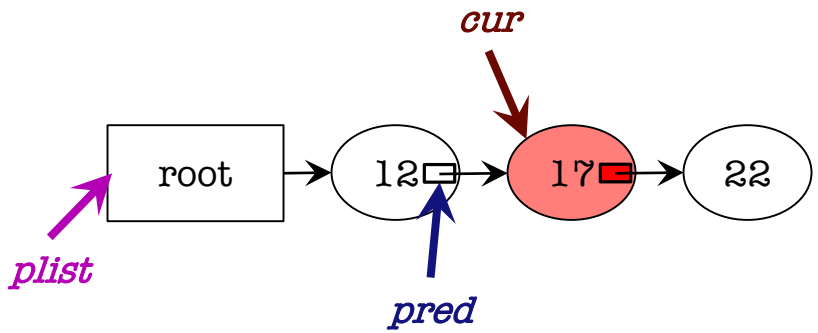
Principe : supprime les nœuds marqués, même si pas le « propriétaire »

On suppose que 17 est marqué !

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

LA LISTE CHAÎNÉE : COMPACTION

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



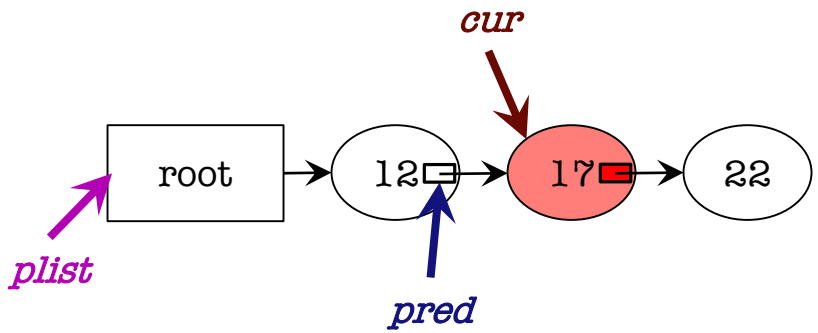
Principe : supprime les nœuds marqués, même si pas le « propriétaire »

On suppose que 17 est marqué !

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

LA LISTE CHAÎNÉE : COMPACTION

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



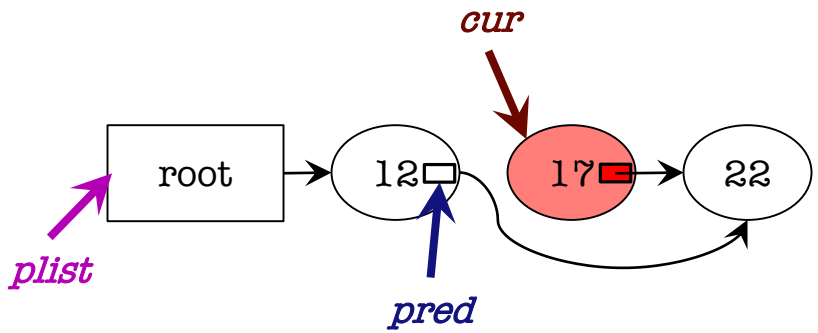
Le cas peut échouer dans deux cas :
(1) Si 12 s'est fait colorer (plist devenu impair)
(2) Si un nœud inséré après 12

(2) problématique : on peut perdre un nœud.
Dans le doute, on repart de zéro !

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

LA LISTE CHAÎNÉE : COMPACTION

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

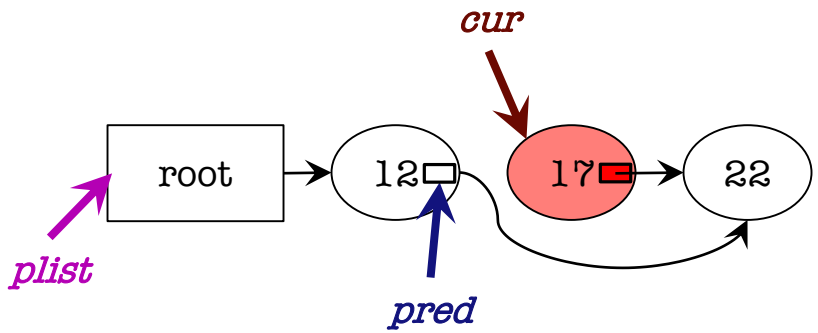


On suppose que le CAS a réussi ici.

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

LA LISTE CHAÎNÉE : COMPACTION

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

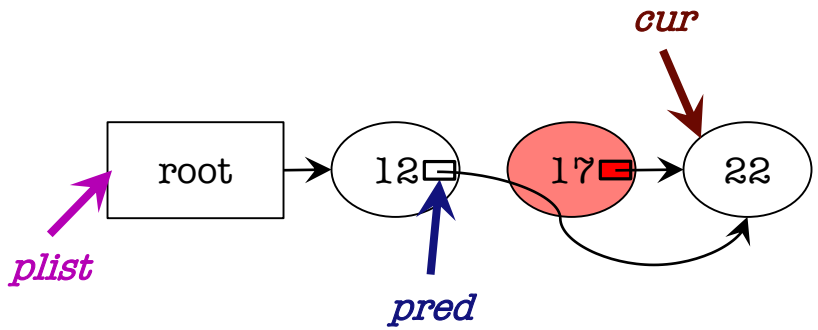


On suppose que le CAS a réussi ici.

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

LA LISTE CHAÎNÉE : COMPACTION

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



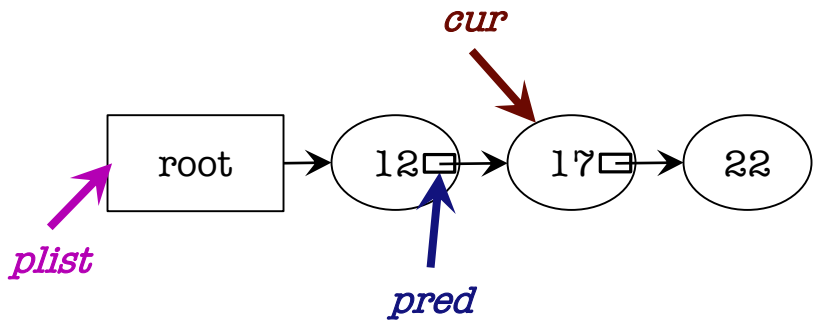
On suppose que le CAS a réussi ici.

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

LA LISTE CHAÎNÉE TRIÉE D'ENTIERS : SUPPRESSION

LA LISTE CHAÎNÉE : SUPPRESSION

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

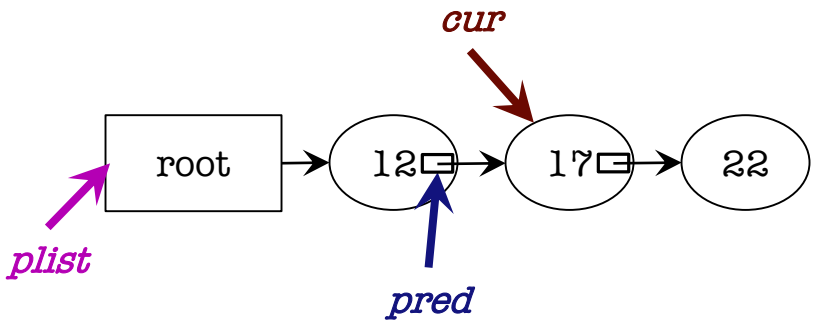


On suppose que value = 17.

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

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

LA LISTE CHAÎNÉE : SUPPRESSION

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

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

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

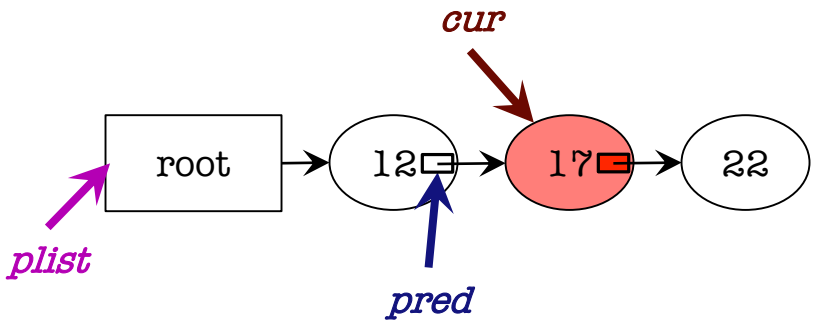


Le CAS() peut échouer si :
- 17 déjà coloré par un autre thread
- Nœud inséré avant 22.
- 22 supprimé.

Peu importe, on insiste.

LA LISTE CHAÎNÉE : SUPPRESSION

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

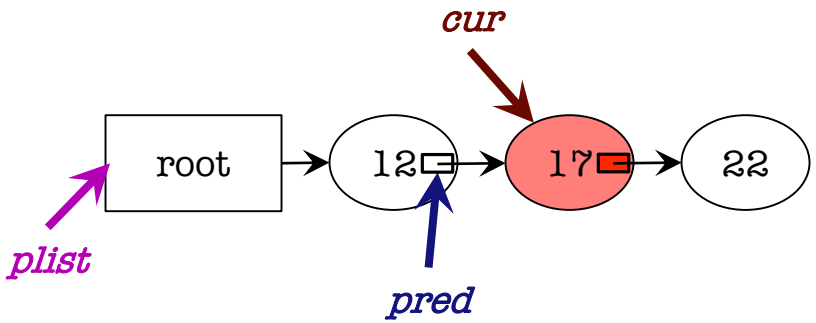


Le CAS() finit par fonctionner.

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

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

LA LISTE CHAÎNÉE : SUPPRESSION

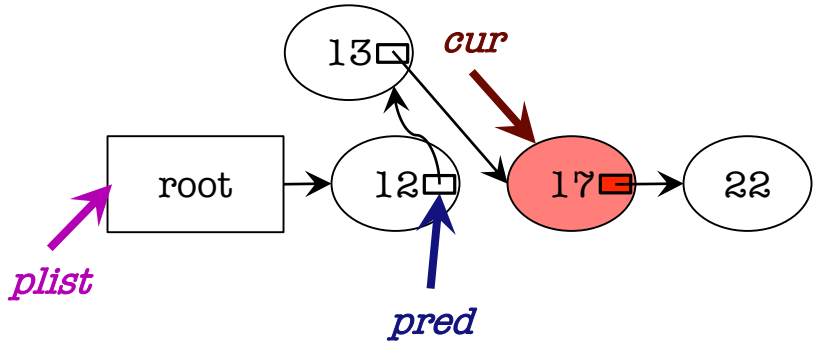
```
void del(coloredPointer* plist, int value) {
restart:
 coloredPointer* pred = plist;
 while(!found) {
  Node cur = pointer(*pred);
```




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

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

LA LISTE CHAÎNÉE : SUPPRESSION

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

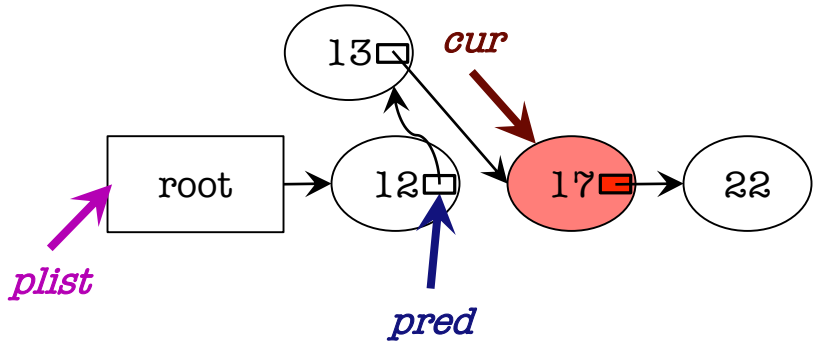


Imaginons qu'un autre thread a inséré 13.
Du coup, le CAS() échoue.

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

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

LA LISTE CHAÎNÉE : SUPPRESSION

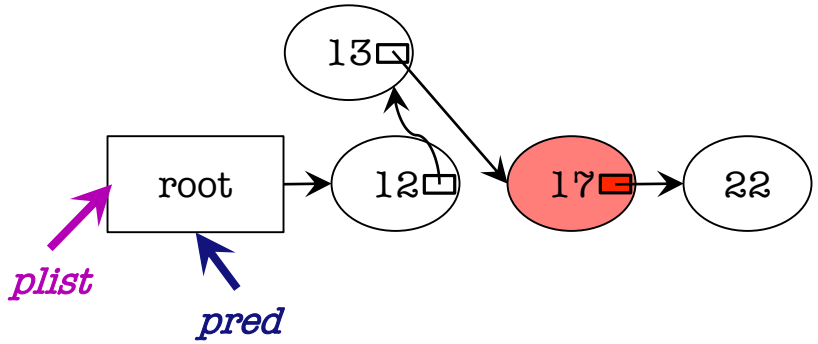
```
void del(coloredPointer* plist, int value) {
restart:
 coloredPointer* pred = plist;
 while(!found) {
  Node cur = pointer(*pred);

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

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



Imaginons qu'un autre thread a inséré 13.
Du coup, le CAS() échoue.
Et on quitte car found == 1.

LA LISTE CHAÎNÉE : SUPPRESSION

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



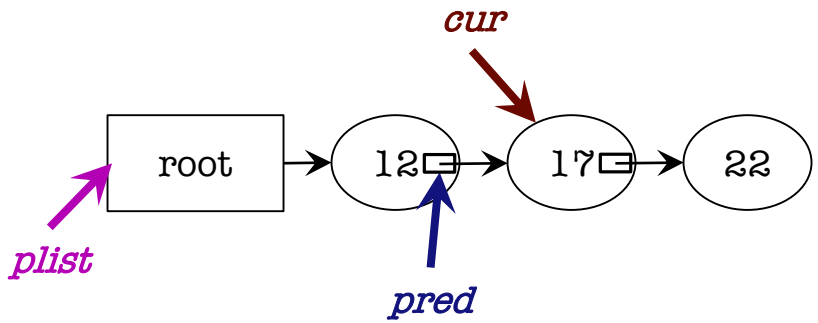
Le thread suivant va supprimer 17.
On en est revenus au cas compaction.

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

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

LA LISTE CHAÎNÉE TRIÉE D'ENTIERS : SUPPRESSION D'ÉLÉMENT DÉJÀ SUPPRIMÉ ?

LA LISTE CHAÎNÉE : SUPPRESSION

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

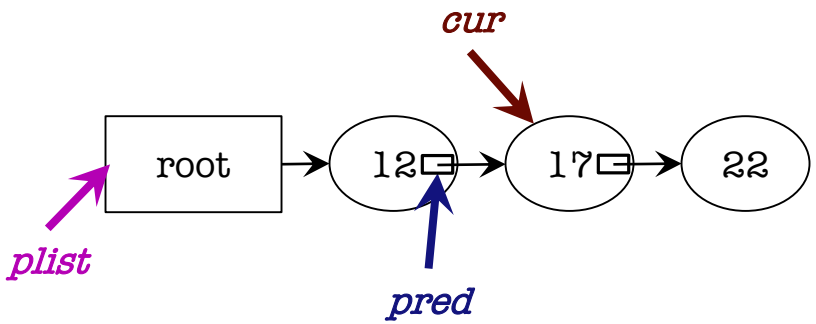


On suppose que value = 13.

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

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

LA LISTE CHAÎNÉE : SUPPRESSION

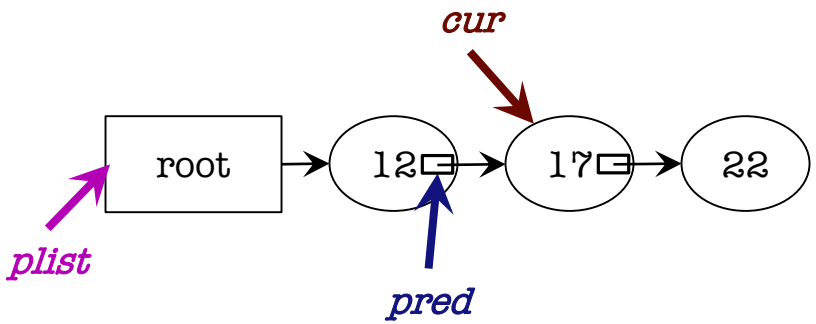
```
void del(coloredPointer* plist, int value) {
restart:
 coloredPointer* pred = plist;
 while(!found) {
  Node cur = pointer(*pred);
  if(cur == null || value < cur->value) /* pas trouvé */
   return 0;

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

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



On suppose que value = 13.
⇒ On sort de la fonction.

LA LISTE CHAÎNÉE : SUPPRESSION

```
void del(coloredPointer* plist, int value) {
restart:
 coloredPointer* pred = plist;
 while(!found) {
  Node cur = pointer(*pred);
  if(cur == null || value < cur->value) /* pas trouvé */
   return 0;

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

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



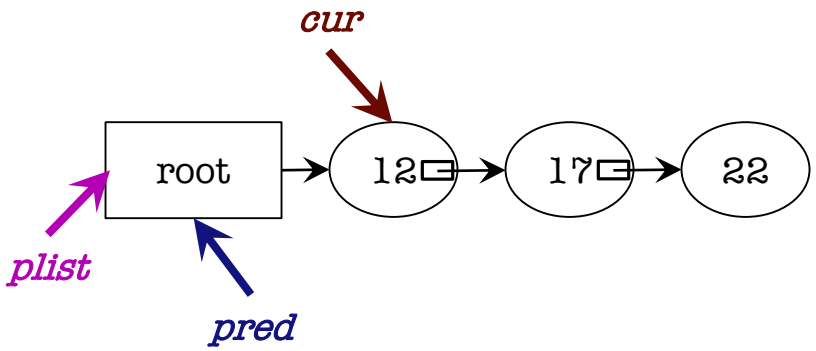
On suppose que value = 13.
⇒ On sort de la fonction.

On a la fonction del() complète.

LA LISTE CHAÎNÉE TRIÉE D'ENTIERS : INSERTION

LA LISTE CHAÎNÉE : INSERTION

```
void add(coloredPointer* plist, int value) {
restart:
 coloredPointer* pred = plist;
 while(!found) {
  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 doit être supprimé */
    if(CAS(pred, cur, cur->next) != cur) goto restart;
    else continue;
   }
   pred = &cur->next;
 } }
```

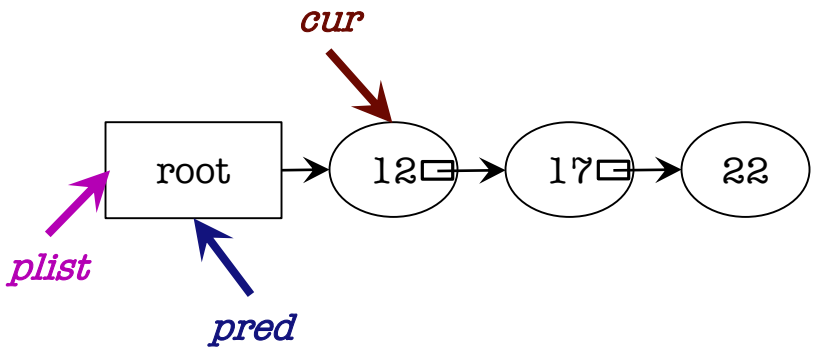


Même principe que pour parcours et compaction…

On suppose que `value = 11`.

LA LISTE CHAÎNÉE : INSERTION

```
void add(coloredPointer* plist, int value) {
restart:
 coloredPointer* pred = plist;
 while(!found) {
  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 doit être supprimé */
   if(CAS(pred, cur, cur->next) != cur) goto restart;
   else continue;
  }
  pred = &cur->next;
 } }
```

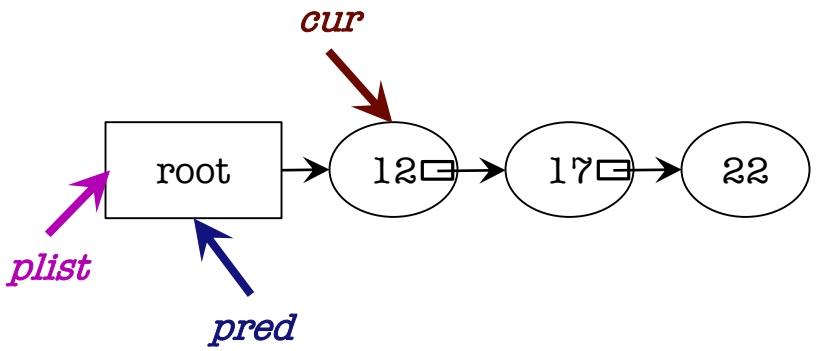


Même principe que pour parcours et compaction…

On suppose que value = 11.
Il faut s'insérer juste avant cur !

LA LISTE CHAÎNÉE : INSERTION

```
void add(coloredPointer* plist, int value) {
restart:
 coloredPointer* pred = plist;
 while(!found) {
  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 doit être supprimé */
   if(CAS(pred, cur, cur->next) != cur) goto restart;
   else continue;
  }
  pred = &cur->next;
 } }
```



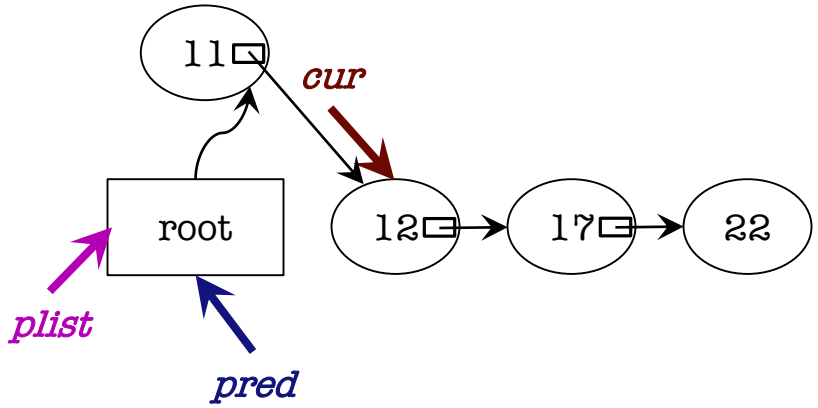
On suppose que value = 11.

Le CAS() peut échouer si :
(1) 12 a été coloré.
(2) Un nœud ajouté avant 12.

(2) problématique : perte d'un nœud.
Dans le doute, on repart de zéro !

LA LISTE CHAÎNÉE : INSERTION

```
void add(coloredPointer* plist, int value) {
restart:
 coloredPointer* pred = plist;
 while(!found) {
  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 doit être supprimé */
   if(CAS(pred, cur, cur->next) != cur) goto restart;
   else continue;
  }
  pred = &cur->next;
 } }
```



On suppose que value = 11.

On suppose que le CAS a réussi ici.

CONCLUSION ALGORITHMIQUE NON-BLOQUANTE

- Avec quelques opérations atomiques, on peut faire pas mal de choses.

CONCLUSION ALGORITHMIQUE NON-BLOQUANTE

- Avec quelques opérations atomiques, on peut faire pas mal de choses.
- **Problème :** ça devient vite très compliqué !

CONCLUSION ALGORITHMIQUE NON-BLOQUANTE

- Avec quelques opérations atomiques, on peut faire pas mal de choses.
- **Problème :** ça devient vite très compliqué !
- Équivalent à utiliser des sections critiques minuscules…

CONCLUSION ALGORITHMIQUE NON-BLOQUANTE

- Avec quelques opérations atomiques, on peut faire pas mal de choses.
- **Problème :** ça devient vite très compliqué !
- Équivalent à utiliser des sections critiques minuscules…
- Généralement très efficace, mais à très haute contention, on risque de rééssayer souvent.

CONCLUSION ALGORITHMIQUE NON-BLOQUANTE

- Avec quelques opérations atomiques, on peut faire pas mal de choses.
- **Problème :** ça devient vite très compliqué !
- Équivalent à utiliser des sections critiques minuscules…
- Généralement très efficace, mais à très haute contention, on risque de rééssayer souvent.
- On a juste vu quelques algos, toute une théorie !

CONCLUSION ALGORITHMIQUE NON-BLOQUANTE

- Avec quelques opérations atomiques, on peut faire pas mal de choses.
- **Problème :** ça devient vite très compliqué !
- Équivalent à utiliser des sections critiques minuscules…
- Généralement très efficace, mais à très haute contention, on risque de rééssayer souvent.
- On a juste vu quelques algos, toute une théorie !
- **Par exemple :** quand peut-on composer des algos non-bloquants de manière correcte ?

CONCLUSION ALGORITHMIQUE NON-BLOQUANTE

- Avec quelques opérations atomiques, on peut faire pas mal de choses.
- **Problème :** ça devient vite très compliqué !
- Équivalent à utiliser des sections critiques minuscules…
- Généralement très efficace, mais à très haute contention, on risque de rééssayer souvent.
- On a juste vu quelques algos, toute une théorie !
- **Par exemple :** quand peut-on composer des algos non-bloquants de manière correcte ?
- Voir notion de linéarisabilité…

CONCLUSION ALGORITHMIQUE NON-BLOQUANTE

- Avec quelques opérations atomiques, on peut faire pas mal de choses.
- **Problème :** ça devient vite très compliqué !
- Équivalent à utiliser des sections critiques minuscules…
- Généralement très efficace, mais à très haute contention, on risque de rééssayer souvent.
- On a juste vu quelques algos, toute une théorie !
- **Par exemple :** quand peut-on composer des algos non-bloquants de manière correcte ?
- Voir notion de linéarisabilité…
- *Pour plus d'infos, lire « The Art of Multiprocessor Programming » de Herlihy et Shavit.*