Practical concurrent algorithms

Mihai Letia

Concurrent Algorithms 2012

Distributed Programming Laboratory

Slides by Aleksandar Dragojevic

Practical

- Course oriented towards theory
- Today: A glimpse of practice
  - Use practical operations
  - Implement practical data structure
    - Linked list
- Give a feeling of “real-world” concurrent algorithm

Outline

- Compare-and-swap
- Practical system model
- Lock-freedom
- Linked-list

Outline

- Compare-and-swap
- Practical system model
- Lock-freedom
- Linked-list

Compare-and-swap (CAS)

```
shared val = BOT;

CAS(old, new):
  ret = val;
  if val == old:
    val = new;
  return ret;
```

Why is it important?

- We can implement consensus among any number of processes

```
shared CAS cas;
Consensus(val):
  res = cas.CAS(BOT, val);
  if res != BOT:
    val = res;
  return val;
```

Why is it important? (cont)

- Consensus allows processes to agree on something
  - Intuitively this is powerful
- CAS allows any number of processes to agree on something

Why is it important also?

- It is available in hardware
  - On (probably) all architectures
- It is a staple of concurrent programming
  - Linked lists
  - Queues
  - Etc.

Outline

- Compare-and-swap
- Practical system model
- Lock-freedom
- Linked-list

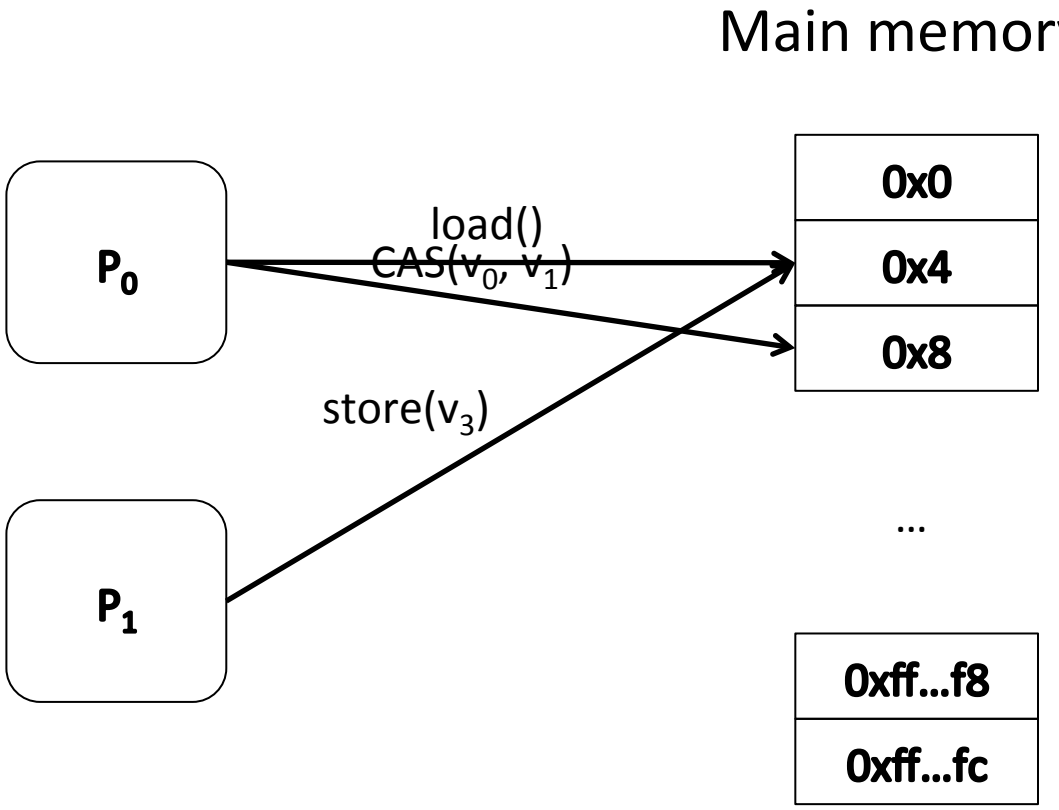
Practical system model

- So far: processes operate on shared objects
  - Object are linearizable
  - Sequential specification
- This lecture: processes operate on memory locations
  - Each location supports multiple instructions
  - Sequential specification

Practical system model (cont)

- Processes access memory by issuing atomic load, store and CAS instructions
- Processes can perform local computations on the loaded values
- Processes are equivalent to OS threads
  - All processes share the same address space

Practical system model (cont)



CAS in practice

```
w_t CAS(w_t *addr, w_t old, w_t new) {
  w_t curr = *addr;
  if(curr == old)
    *addr = new;
  return curr;
}
```

From practical model to other model

- It is straightforward to use practical model as the model used in the course
  - For operations supported by practical model
- Each location is treated as an object
- Only restricted operations are allowed on each location
  - Register: load/store
  - CAS: CAS

From practical model to other model (cont)

```
class CAS {
   w_t value = BOT;
   w_t CAS(w_t old, w_t new) {
      return CASInst(&value, old, new);
   }
}
```

CAS example

- Shared variable initialization
- We have a shared variable initialized to BOT
- Multiple processes want to initialize it to a value
  - Each process proposes its value (e.g. a problem it needs solved)
- Only one initialization should succeed
  - Processes need to agree on the initialized value

CAS example (cont)

```
w_t *prob = BOT;
process() {
   w_t *prop = init_problem(rnd());
   w_t *res = CAS(&prob, BOT, prop);
   if(res != BOT) uninit_problem(prop);
   work_on(prob);
}
```

Other available operations?

- Depends on the architecture
- Available on x86:
  - Test-and-set
  - Fetch-and-increment
  - Fetch-and-add

Outline

- Compare-and-swap
- Practical system model
- Lock-freedom
- Linked-list

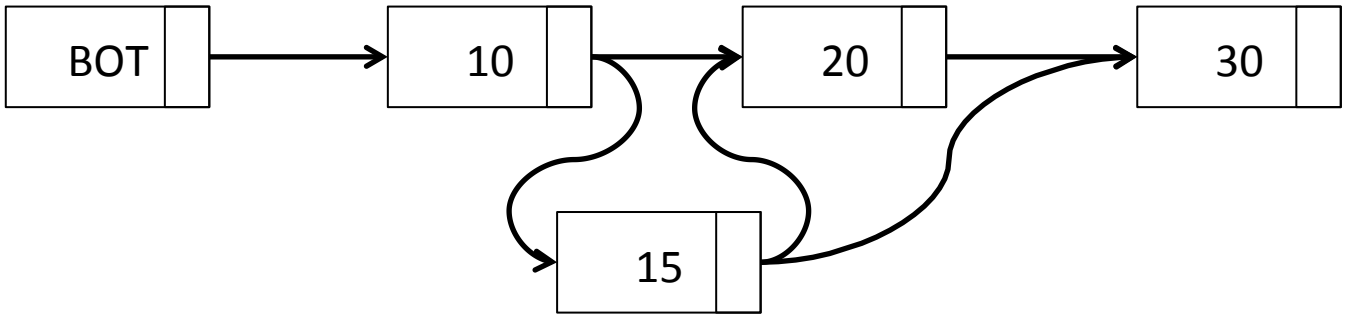
Linked list

- A list of nodes that are linked using references
- Dynamic data structure
  - Nodes can be allocated and deallocated
  - No need to know maximum size upfront
- Base for other data structures
  - Queue
  - Hash table
  - Skip list

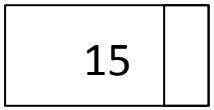
Linked list API

```
struct Node {
   int key;
   int val;
   Node *next;
};
bool Insert(Node *node);
Node *Remove(int key);
Node *Lookup(int key);
```

Linked list



Insert()



Remove(20)

Our goal today

- Implement the linked list in our model using load, store and CAS instructions

Progress

- We are not going to implement a wait-free linked list
  - We assumed wait-free implementations so far in the course
- We will implement a lock-free linked list
  - High performance
- Wait-free linked list is a hard problem
  - In next lectures

Outline

- Compare-and-swap
- Practical system model
- Lock-freedom
- Linked-list (we will return to this)

Lock-freedom

- If a process performs steps of an algorithm for sufficiently long time, some process will make progress
- What is difference to wait-freedom?

Difference to wait-freedom

- With wait-free algorithms, when a process makes steps **it** is guaranteed to make progress
- With lock-free algorithms, **some process** is guaranteed to make progress
  - But not necessarily the process performing the steps

Intuition for lock-freedom

- Lock-freedom guarantees overall progress
  - No guarantee of overall progress is weaker than lock-freedom (more on this in next lectures)
- Eliminates live-lock
- It is usually fine in practice
  - In practice the “winner” is likely to go do something else and permit the “looser” to perform the operation later

Which is better?

- Wait-freedom is a stronger property
  - It is harder to achieve in efficient way
  - General wait-free algorithms in next lectures
- Lock-freedom is easier to achieve in practice efficiently
  - It is more often used for that reason
- If we had equally efficient algorithms $A_{\text{lock-free}}$ and $A_{\text{wait-free}}$, we would choose $A_{\text{wait-free}}$

Lock-free and Linearizable

- Having lock-free algorithms doesn't change the requirement for linearizable algorithms
- If the operation succeeds, there has to be a linearization point
- Lock-freedom just means that the operation will not necessarily succeed

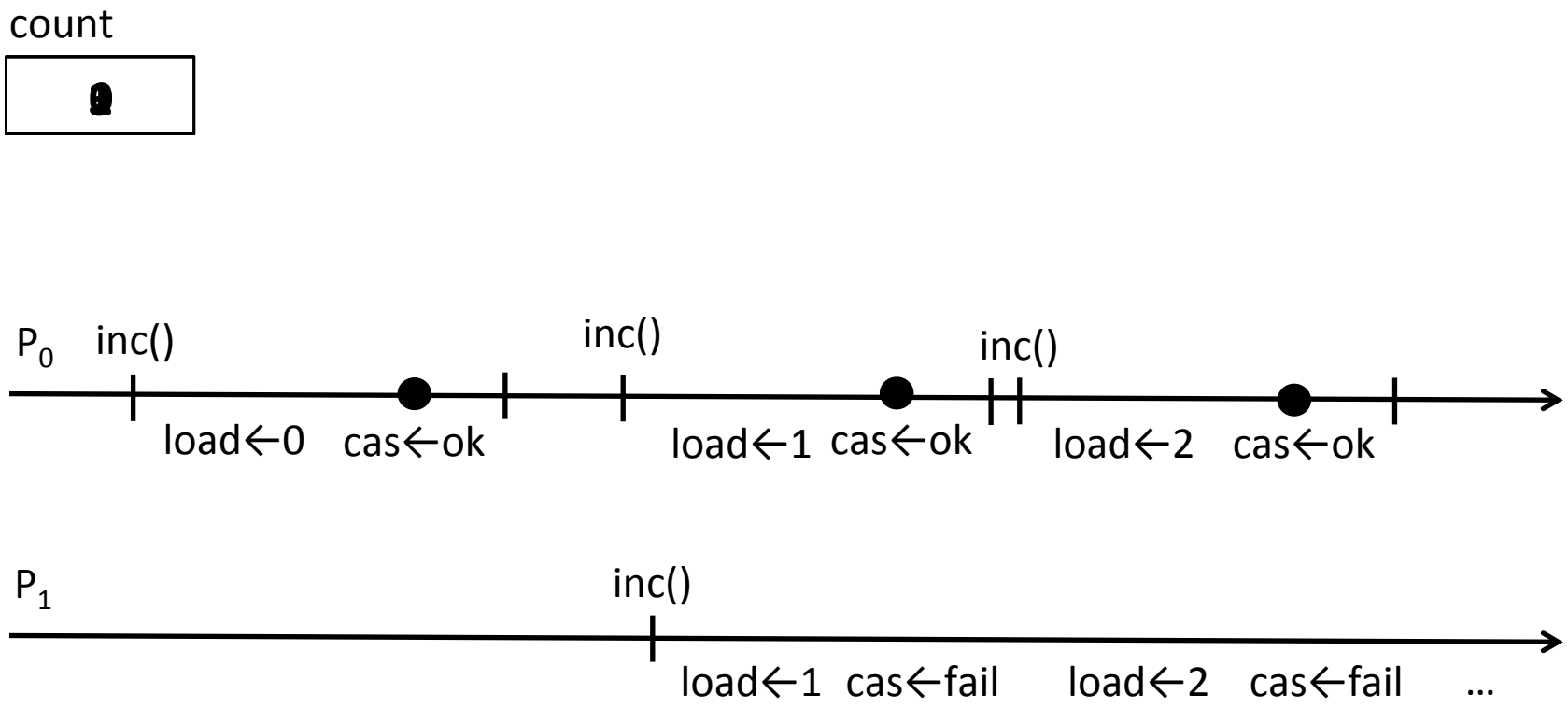
Lock-free strong counter

```
w_t count = 0;
w_t increment() {
  while(true) {
    w_t val = count; // implied load
    if(CAS(&count, val, val + 1) == val)
      return val;
  }
}
```

Lock-free strong counter (cont)

- Simple algorithm
- Starvation is possible
  - A process could be prevented from returning a value by another process that always succeeds
- Live-lock is not possible
  - A process is prevented from returning by a successful process
- In practice, starvation is not probable

Lock-free strong counter (cont)



CAS can continue to fail

Wait-free counter?

- General technique exists
  - More complex and less efficient
  - Next lecture
- Intuition
  - Processes have to help each other
  - A process that succeeds the CAS needs to help the other processes to finish their operations

Outline

- Compare-and-swap
- Practical system model
- Lock-freedom
- Linked-list

Our goal today

- Implement the lock-free linked list in our model using load, store and CAS instructions

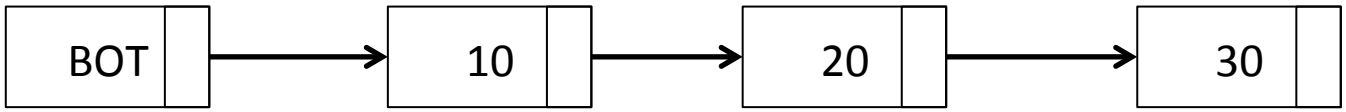
Lock-freedom

- If a process performs steps of an algorithm for sufficiently long, some process will make progress

Linked list API

```
struct Node {
   int key;
   int val;
   Node *next;
};
bool Insert(Node *node);
Node *Remove(int key);
Node *Lookup(int key);
```

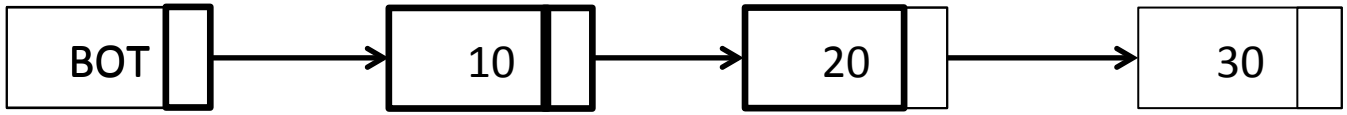
What should we do?



Overview

- Fields `key` and `val` are not changed after the element has been inserted into the list
- We only need to change `next` field concurrently
  - During insert and remove
  - We need to "swing" it from one node to another
- Keep the list sorted

Lookup (version 1)



Lookup(20)

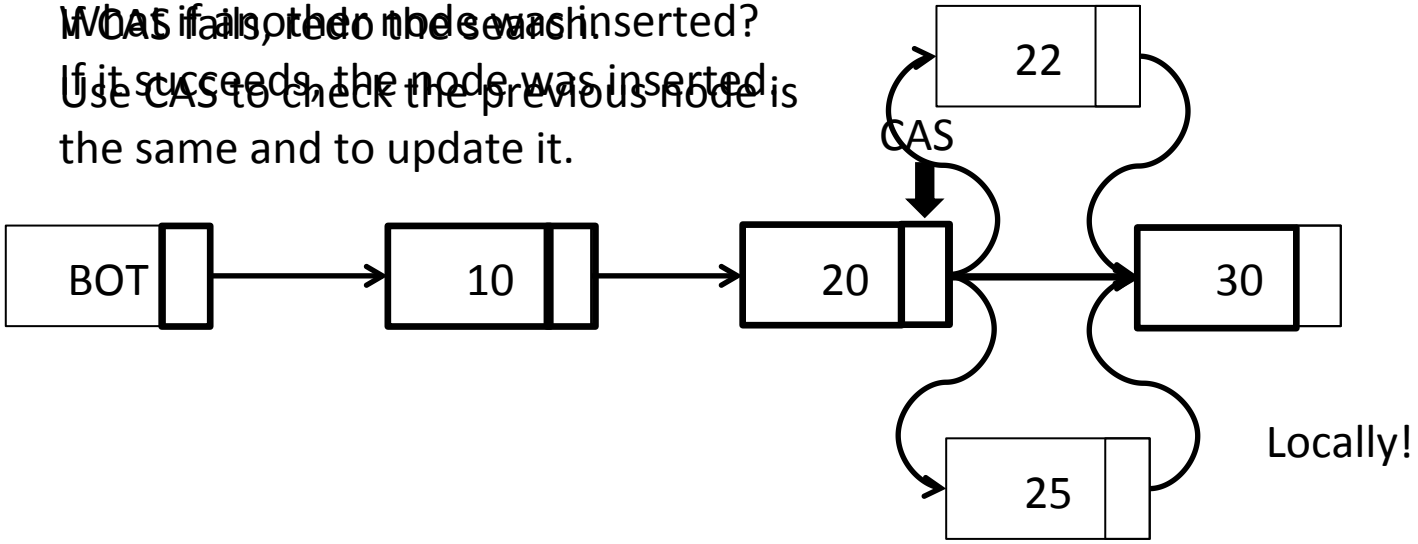
Insert

- First lookup an element
- If the node with the same key exists, return false
- If not, we found the position for element insertion
- Locally prepare node's `next` field
- Use CAS to make sure previous node hasn't changed and to update it to the new node

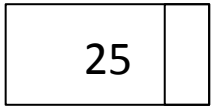
Insert (version 1)

What if another node was inserted?
If it succeeds, the node was inserted.
If CAS fails, redo the search.

Use CAS to check the previous node is the same and to update it.



Insert(25)



Remove

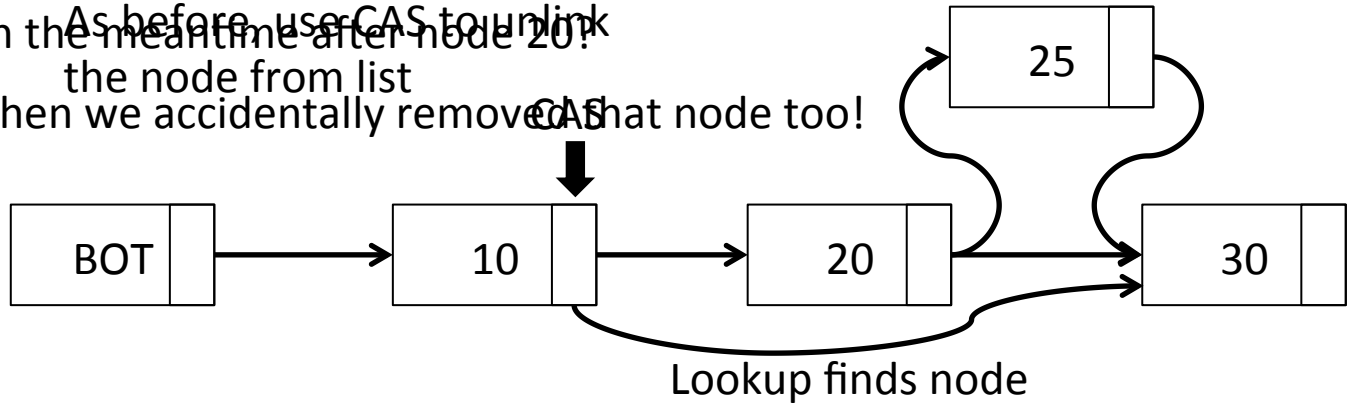
- First lookup an element
- If node with the key doesn’t exist, return false
- If it does, we found the node to remove and the previous node
- Use CAS to make sure previous node hasn’t changed and to remove the found node

Remove (version 1)

What if another node was inserted in the meantime after node 20?

As before, use CAS to unlink the node from list

Then we accidentally removed that node too!



Remove(20)

Is Remove (version 1) correct then?

- It is not linerizable (lost operation)
- Which means that it is not correct
- What should we do about it?

Why is there a problem?

- We update the previous node's next field and make it point to what we saw was the next field of the removed node
- But that doesn't mean this is still the next field of the removed node
- CAS doesn't ensure the next field of the removed node is unchanged
- It ensures the next field of the previous node is unchanged (this is also necessary)

Why is there a problem? (cont)

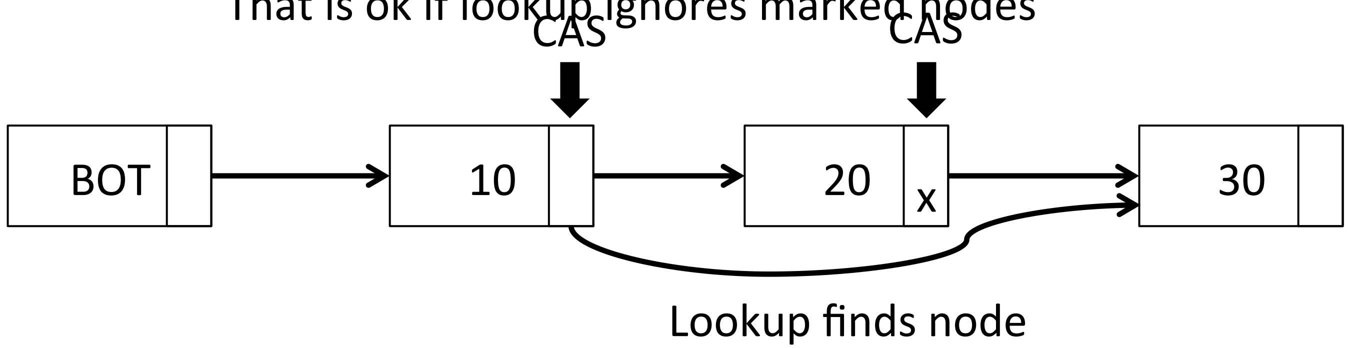
- We need to ensure two locations remain the same while we update one of them
- CAS operates on a single location
- We need double-compare-single-swap operation to (simply) solve this problem
  - This is not available in hardware

How to solve the problem?

- Mark the removed nodes using a spare bit from the next field
  - There is a spare bit if we assume word-aligned data structures
- Marking of the node is the linearization point of the operation
- After the node is marked, try to unlink it too
- Other operations can help with the unlinking

Remove (version 2)

What if unlink fails?
First, mark the node
Then, try to unlink 10's next pointer
That is ok if lookup ignores marked nodes



Remove(20)

Lookup (version 2)

- Similar to version 1
- Search ignores marked nodes
- It tries to unlink marked nodes

Insert (version 2)

- Similar to version 1
- Search ignores marked nodes
- It tries to unlink marked nodes
- The rest is the same
  - No need to change the insert of the node
  - CAS fails if the node became marked since it's been read

Pseudo code

```
Node *search(int k, Node **left) {
      Node *left_next, *right;
search_again:
      do {
            Node *t = head; Node *t_next = head.next;
            /* 1: Find left_node and right_node */
            do {
                  if(!is_marked(t_next)) {
                        (*left) = t;
                        left_next = t_next;
                  }
                  t = unmark(t_next);
                  if(t == NULL) break;
                  t_next = t.next;
            } while (is_marked(t_next) || (t.key<k));
            right = t;
            /* 2: Check nodes are adjacent */
            if(left_next == right)
                  if((right != NULL) && is_marked(right.next)) goto search_again;
                  else return right_node;
            /* 3: Remove one or more marked nodes */
            if(CAS(&(left.next), left_next, right))
                  if((right != NULL) && is_marked(right.next)) goto search_again;
                  else return right_node;
      } while (true);
}
```

Pseudo code (cont)

```
Node *Lookup(int key) {
   Node *right, *left;
   right = search(key, &left);
   if(right == NULL || right.key != key)
      return NULL;
   else
      return right;
}
```

Pseudo code (cont)

```
bool Insert(Node *node) {
   Node *right, *left;
   do {
      right = search(node.key, &left);
      if((right != NULL) && (right.key == key))
         return false;
      node.next = right;
      if(CAS(&(left.next), right, node))
         return true;
   } while (true);
}
```

Pseudo code (cont)

```
Node *Remove(int key) {
    Node *right, *right_next, *left;
    do {
        right = search(key, &left);
        if((right == NULL) || (right.key != key))
            return NULL;
        right_next = right.next;
        if(!is_marked(right_next))
            if(CAS(&(right.next), right_next, mark(right_next)))
                break;
    } while(true);
    if(!CAS(&(left.next), right, right_next))
        right = search(right.key, &left);
    return right;
}
```

Important assumption

- Removed nodes are not reused until all operations that are in progress are finished
- How to ensure this?
  - Garbage collector
  - Concurrent memory managers

References

- T. Harris “A Pragmatic Implementation of Non-Blocking Linked-Lists.” DISC 2001.
- M. M. Michael “Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects.” IEEE Transactions on Parallel and Distributed Systems, June 2004.
- M. Herlihy, V. Luchangco, P. Martin, and M. Moir. “Non- blocking memory management support for dynamic-sized data structures.” ACM Transactions on Computer Systems, May 2005.