Lecture 12: Multithreading Design Patterns and Thread-Safe Data Structures

Principles of Computer Systems

Autumn 2019

Stanford University

Computer Science Department

Lecturer: Chris Gregg

Philip Levis

PDF of this presentation

Review from Last Week

- We now have *three* distinct ways to coordinate between threads:
  - **`mutex`**: mutual exclusion (lock), used to enforce critical sections and atomicity
  - **`condition_variable`**: way for threads to coordinate and signal when a variable has changed (integrates a lock for the variable)
  - **`semaphore`**: a generalization of a lock, where there can be *n* threads operating in parallel (a lock is a semaphore with *n=1*)

Mutual Exclusion (mutex)

- A mutex is a simple lock that is shared between threads, used to protect critical regions of code or shared data structures.
  - `mutex m;`
    - `mutex.lock()`
    - `mutex.unlock()`
- A mutex is often called a lock: the terms are mostly interchangeable
- When a thread attempts to lock a mutex:
  - Currently unlocked: the thread takes the lock, and continues executing
  - Currently locked: the thread blocks until the lock is released by the current lock-holder, at which point it attempts to take the lock again (and could compete with other waiting threads).
- Only the current lock-holder is allowed to unlock a mutex
- Deadlock can occur when threads form a circular wait on mutexes (e.g. dining philosophers)
- Places we've seen an operating system use mutexes for us:
  - All file system operation (what if two programs try to write at the same time? create the same file?)
  - Process table (what if two programs call fork() at the same time?)

lock_guard<mutex>

- The **`lock_guard<mutex>`** is very simple: it obtains the lock in its constructor, and releases the lock in its destructor.
- We use a **`lock_guard<mutex>`** so we don't have to worry about unlocking a **`mutex`** when we exit a block of code

```
void function(mutex &m) {
    lock_guard<mutex> lg(m); // m is now locked
    while (true) {
      if (condition1) return; // lg automatically unlocked on return
      // ...
      if (condition2) break;
    }
    // mutex will be unlocked after this line when lg goes out of scope
}
```

- A lock guard is a good when you always want to release a lock on leaving a block.
  - If you want to unlock it later or later, don't use it (or change block structure)
- Using a lock guard is a good idea when control flow is complex (returns, breaks, multiple exit points), such that the lock would have to be unlocked in multiple places

conditional_variable

- The **`conditional_variable`** enables one thread to signal to other threads that a variable has changed (e.g., a work queue), such that a condition has changed (there's now work on the queue)
- It works *in conjunction with* a **`mutex`** to protect the shared variable.
  - A thread locks the mutex to read the variable. If the variable indicates that the thread should wait, it calls **`cv.wait(m)`**. Calling **`wait`** atomically unlocks the mutex and places the thread on a queue of threads waiting on the condition. Other threads can lock the mutex and check the condition. They'll wait too.
  - When another thread locks the mutex to update it, changing the condition, it calls **`cv.notify_all()`** then unlocks the mutex. This wakes up all threads waiting on the condition variable. They queue up to acquire the mutex and execute as before.

```
static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
  lock_guard<mutex> lg(m);
  while (permits == 0) cv.wait(m);
  permits--;
}

static void grantPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
  lock_guard<mutex> lg(m);
  permits++;
  if (permits == 1) cv.notify_all();
}
```

conditional_variable

- The pattern of waiting on a condition variable within a while loop that checks the condition is so common there's a variant of wait that supports it.

```
template <Predicate pred>
void condition_variable_any::wait(mutex& m, Pred pred) {
  while (!pred()) wait(m);
}
```

- **`Pred`** is a function that returns true or false. You can use a lambda function for it:

```
static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
  lock_guard<mutex> lg(m);
  cv.wait(m, [&permits] { return permits > 0; });
  permits--;
}
```

- Some times the operating system has used condition variables for us
  - Reading from a pipe: caller waits until someone writing to the pipe wakes it up
  - Writing to a pipe: caller waits until there's space in the pipe
  - Waiting until a child has exited

semaphore

- The **`semaphore`** class is not built in to C++, but it is a basic synchronization primitive
- You declare a semaphore with a maximum value (e.g., permits in last lecture)

```
semaphore permits(5); // this will allow five permits
```

- A thread uses a semaphore by decrementing it with a call to wait; if the semaphore value is 0, the thread blocks.
- A thread releasing a semaphore calls signal, this increments the value and triggers waiting threads to resume (and try to decrement).

```
permits.wait(); // if five other threads currently hold permits, this will block

// only five threads can be here at once

permits.signal(); // if other threads are waiting, a permit will be available
```

- A **`mutex`** is a special case of a semaphore with a value of 1. If you need a lock, use a mutex. Unlike semaphores, one error checking benefit of a mutex is that it can *only* be released by the lock-holder. But in cases when you need to allow a group of threads to be in a section of code (e.g., want to limit parallelism *2 <= n <= k*), use a semaphore.

Example: Synchronizing on a Vector

- Suppose we have a thread that puts data into a buffer, which another thread reads and processes.
- Why is this implementation ridiculously unsafe and completely broken?

```
const size_t BUF_SIZE = 8;
const size_t DATA_SIZE = 320; // 40 cycles around buffer
static char buffer[BUF_SIZE];

static void writer(char buffer[]) {
  for (size_t i = 0; i < DATA_SIZE; i++) {
    buffer[i % BUF_SIZE] = prepareData();
  }
}

static void reader(char buffer[]) {
  for (size_t i = 0; i < DATA_SIZE; i++) {
    processData(buffer[i % BUF_SIZE]);
  }
}

int main(int argc, const char *argv[]) {
  thread w(writer, buffer);
  thread r(reader, buffer);
  w.join();
  r.join();
  return 0;
}
```

So Many Problems

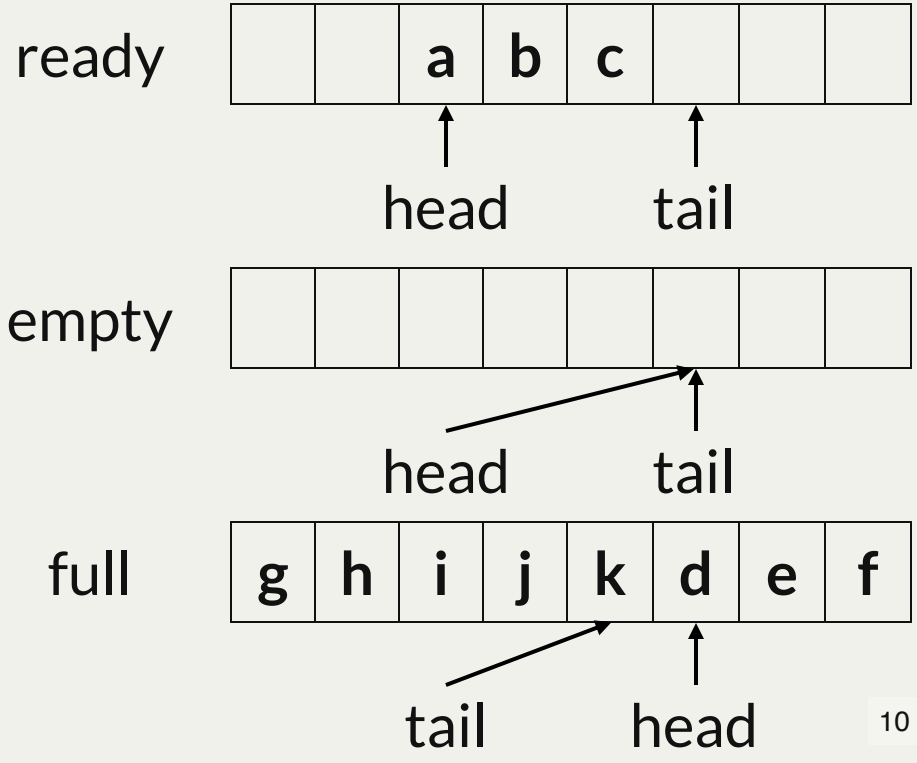
- Each thread runs independently of the other.
- The reader can read past where there's valid data.
- The writer can overwrite data before the reader has read it.

One Solution: Condition Variables

- One solution? Maintain a condition variable: vector not empty or full
  - If the vector is full, the writer thread waits on the variable. When the reader removes an element from the queue, it signals the condition variable.
  - If the vector is empty, the reader thread waits on the variable. When the writer adds an element to the queue, it signals the condition variable.
- Need to maintain a head/read index (first element) and a tail/write index (first free)
  - Empty if tail == head
  - Full if (tail + 1) % BUF_SIZE == head

```
struct safe_queue {
  char buffer[BUF_SIZE];
  size_t head;
  size_t tail;
  mutex lock;
  condition_variable_any cond;
};


int main(int argc, const char *argv[]) {
  safe_queue queue;
  thread w(writer, ref(queue));
  thread r(reader, ref(queue);
  w.join();
  r.join();
  return 0;
}
```



Safe Reading and Writing

```
static void writer(safe_queue& queue) {
    for (size_t i = 0; i < DATA_SIZE; i++) {
        queue.lock.lock();
        while (full(queue)) {
            cout << "Full" << endl;
            queue.cond.wait(queue.lock);
        }
        queue.buffer[queue.tail] = prepareData();
        queue.tail = (queue.tail + 1) % BUF_SIZE;
        queue.lock.unlock();
        queue.cond.notify_all();
    }
}

static void reader(safe_queue& queue) {
    for (size_t i = 0; i < DATA_SIZE; i++) {
        queue.lock.lock();
        while (empty(queue)) {
            cout << "Empty" << endl;
            queue.cond.wait(queue.lock);
        }
        processData(queue.buffer[queue.head]);
        queue.head = (queue.head + 1) % BUF_SIZE;
        queue.lock.unlock();
        queue.cond.notify_all();
    }
}
```

- The reader and writer rely on the condition variable to tell each other when there might be work to do.
- NB: cout use is unsafe, no oslock because not supported in C playground.

Running the Full Example

https://cplayground.com/?p=eel-gull-hippo

Multithreading Design Patterns

- Mutex protects one or more variables to provide atomicity
  - Good: allows fine-grained control of when and how long lock is held
  - Danger: requires manual locking/unlocking, bugs are easy
    - "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code", Engler at al. (SOSP 2001) uses simple inference to find cases when a variable was protected by a lock most but not all of the time (suggesting a bug)
- Scoped lock (lock_guard) protects one or more variables at granularity of basic block
  - Good: compiler promises lock will be released on leaving block
  - Danger: can't hold locks across block boundaries, so can lead to convoluted code
- Condition variables coordinate threads on shared variables locked by a mutex
  - Good: allows threads to cheaply (not spin) wait until they are needed
  - Danger: manual checking of conditions on variable (e.g. while(full(queue)))

Question period on mutexes, condition variables, and sempahores

Safe Data Structures Are Rarely Simple

- Suppose we want to use some standard data structures in our multithreaded program
  - `std::vector`
  - `std::map`
- These data structures are **not** thread-safe!
  - You have to write your own synchronization around them (like we did for the circular buffer in the reader/writer example)
  - Why can't we just have some nice simple data structures that do it all for us?
- There are some thread-safe variants, e.g. Intel's Thread Building Blocks (TBB)
  - These solve some, but not all of the problems

Example: Thread-Safe Access to a Vector

- A **std::vector** is a standard data structure, backed by an array that can grow/shrink
  - O(1) element lookup
  - O(1) append
  - O(n) random insertion or deletion
- Strawman: let's make it thread-safe by locking a mutex on each method call
  - Every method call will execute atomically
  - Only one thread can access the vector at any time, but they will interleave and we could use finer-grained locking if this is a performance problem

```
1 int vector_print_thread(std::vector<int>& ints) {
2     size_t size = ints.size();
3     for (size_t i = 0; i < size; i++) {
4         std::cout << ints[i] << std::endl;
5     }
6 }
```

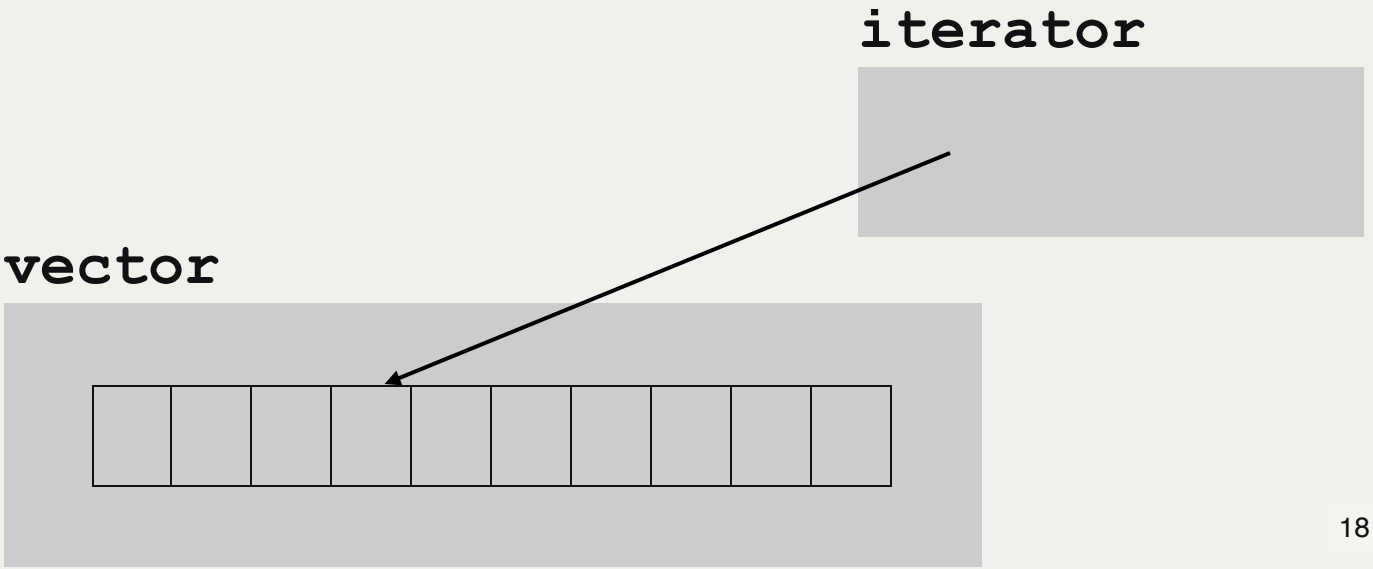
- What could go wrong here even if size() and ints[i] each run atomically?

Granularity of Atomicity

- A data structure can provide atomicity guarantees on its methods, but a caller often wants higher-level atomicity guarantees, based on its own logic and operations
  - Insert an element at the end of the vector
  - Scan across all elements to find the maximum
  - If the vector is empty, insert a value
- Providing each and every use case as an atomic method will give you a terrible abstraction: large, complex, huge code size, etc.
  - Java tried making every object have an integrated lock
    - All methods marked **synchronized** are atomic
    - You can also take an object's lock with **synchronized(obj)**
    - Encourages coarse-grained locking, no flexibility on type of locks
    - Doesn't support semaphores/pools
    - General Java failure of being first major language to try a good idea and getting it wrong, then later languages learned from the mistakes and did it much better
- Within a particular system/library, there are narrower and well understood patterns
  - System/library composes synchronization and data structures at right levels of abstraction

Interior Mutability and Visibility

- A std::vector is a standard data structure, backed by an array that can grow/shrink
  - O(1) element lookup
  - O(1) append
  - O(n) random insertion or deletion
- Iterators are a common design pattern for standard data structures
  - Allow you to pass around an object that allows you to traverse a structure independently of what that structure is
  - If your code uses an iterator, it could be given an iterator for a list, or for a vector, or for a map, and work with any of them
  - Iterator has pointers to internal data structures

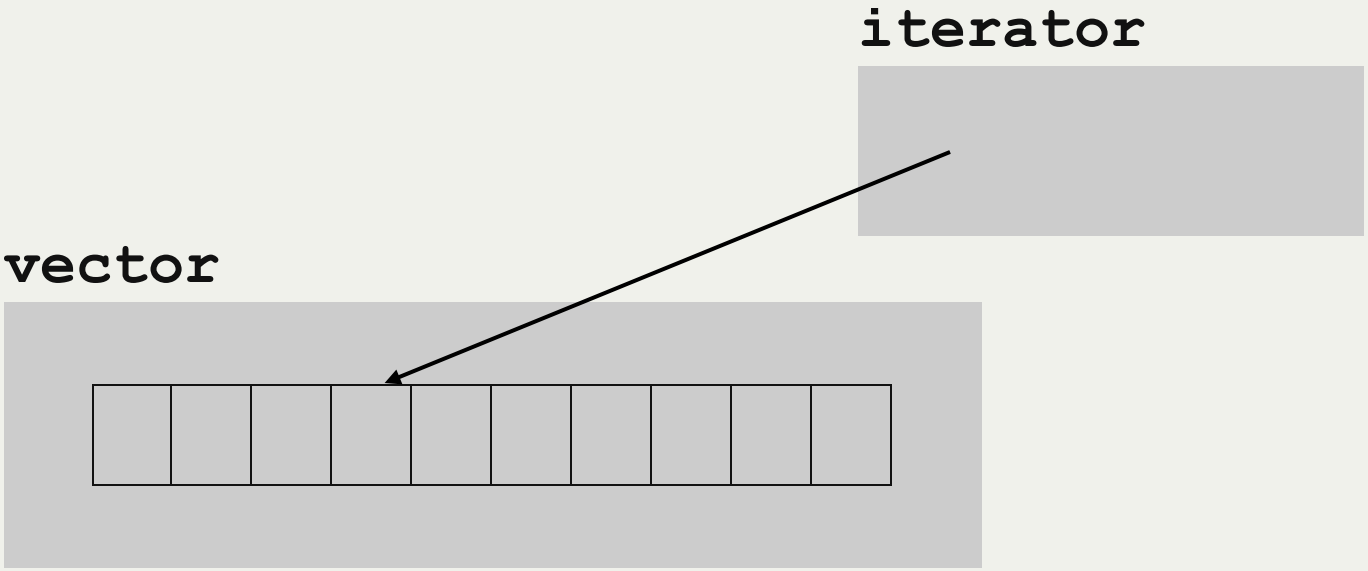


Example Iterator Code

https://cplayground.com/?p=marmoset-quail-cat

Thread-Safe Iterator Access

- What happens to an iterator if another thread removes the end of the vector?
- What happens if an insertion is past the end of the array?
  - The vector allocates a bigger array and copies the data over
  - What happens to the iterator pointer?



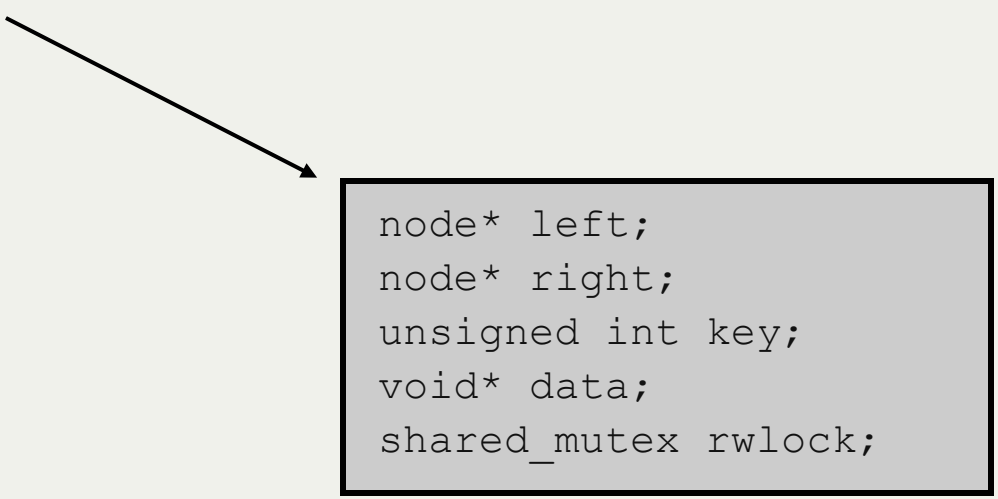
Thread-Safe Trees

- Suppose we want a thread-safe tree that allows concurrent readers and writers
  - Fine-grained locks: not just one lock around the whole tree
  - Per-node read/write locks: many concurrent readers or one writer
- Let's consider what locks we need for three operations
  - lookup: return the value associated with a key
  - insert: insert a value with a key
  - remove: remove a key and its value

Tree Node Structure

- Data access rule: many readers, but at most one writer (a *read-write lock*)
- For example, we can used a **shared_mutex**
  - **shared_mutex.lock()**: acquire exclusive (write) lock
  - **shared_mutex.shared_lock()**: acquire shared (read) lock
- Each tree node has a shared_mutex; also, the root pointer does
- Insertion requires having exclusive access to the parent (changing its null pointer)

```
shared_mutex root_lock;
node* root;
```



Inserting Into the Tree

- `insert(unsigned int key, void* data)`
- Use a helper function
  - `insert_at(node* node, unsigned int key, void* data)`
- Insert pseudocode

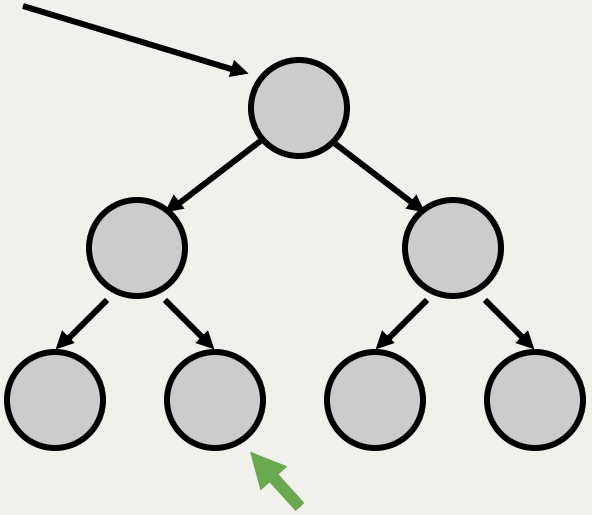
```
lock root pointer lock
if root pointer is null:
    insert at root
    unlock root pointer lock
    return
else:
    lock root node
    unlock root pointer
    insert_at(root_node)
```

Insert_at pseudocode

```
if (key < node.key):
    if node.left == NULL:
        insert at node.left
        unlock node
    else:
        lock node.left
        unlock node
        insert_at(node.left)
else:
    if node.right == NULL:
        insert at node.right
        unlock node
    else:
        lock node.right
        unlock node
        insert_at(node.right)
```

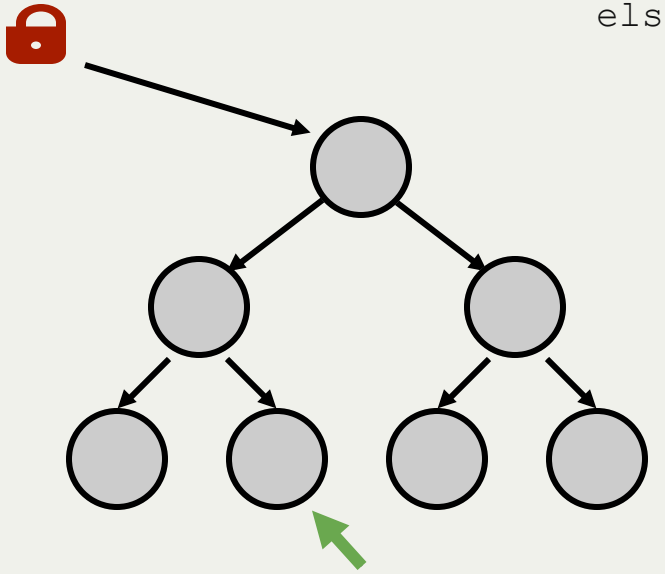
Example

- Suppose we want to insert at the green point



Example

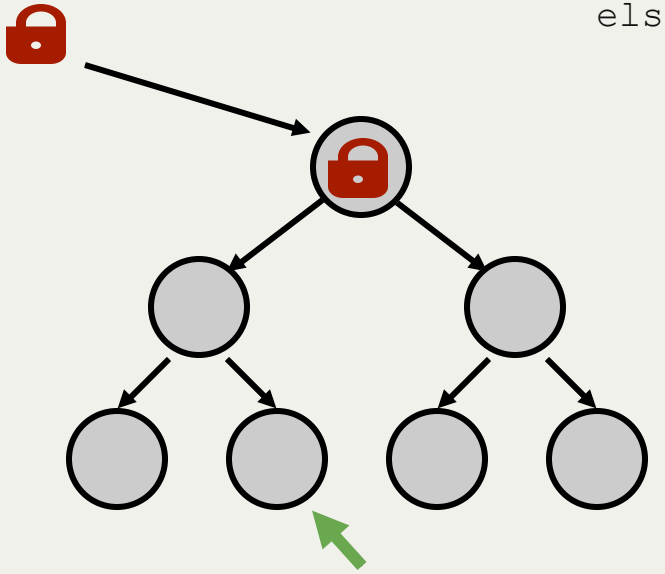
- Suppose we want to insert at the green point



```
lock root pointer lock
if root pointer is null:
    insert at root
    unlock root pointer lock
    return
else:
    lock root node
    unlock root pointer
    insert_at(root_node)
```

Example

- Suppose we want to insert at the green point

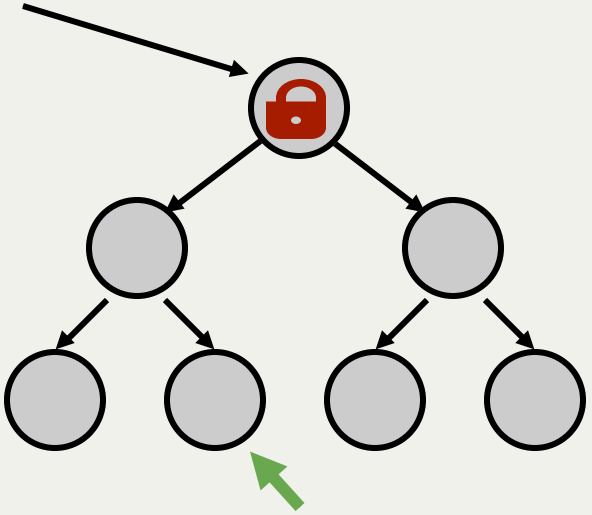


```
lock root pointer lock
if root pointer is null:
    insert at root
    unlock root pointer lock
    return
else:
    lock root node
    unlock root pointer
    insert_at(root_node)
```

Example

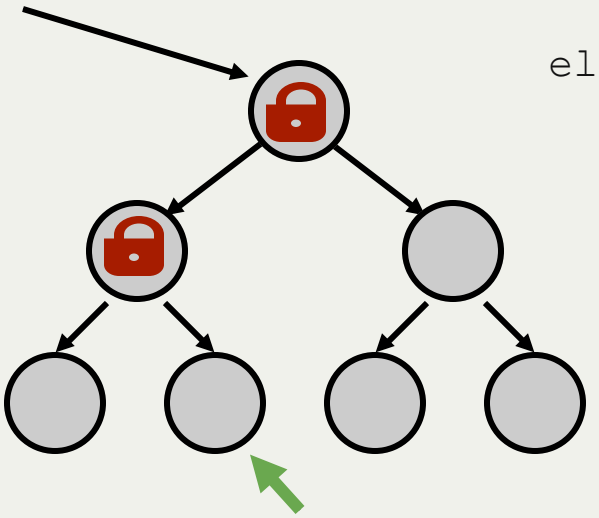
- Suppose we want to insert at the green point

```
lock root pointer lock
if root pointer is null:
    insert at root
    unlock root pointer lock
    return
else:
    lock root node
    unlock root pointer
    insert_at(root_node)
```



Example

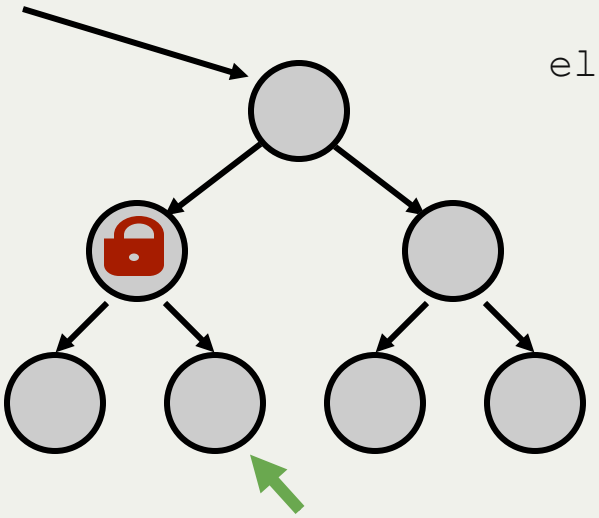
- Suppose we want to insert at the green point



```
if (key < node.key):
    if node.left == NULL:
        insert at node.left
        unlock node
    else:
        lock node.left
        unlock node
        insert_at(node.left)
else:
    if node.right == NULL:
        insert at node.right
        unlock node
    else:
        lock node.right
        unlock node
        insert_at(node.right)
```

Example

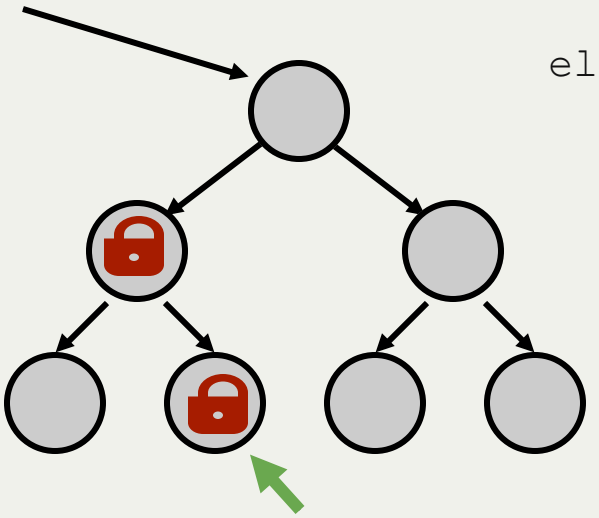
- Suppose we want to insert at the green point



```
if (key < node.key):
    if node.left == NULL:
        insert at node.left
        unlock node
    else:
        lock node.left
        unlock node
        insert_at(node.left)
else:
    if node.right == NULL:
        insert at node.right
        unlock node
    else:
        lock node.right
        unlock node
        insert_at(node.right)
```

Example

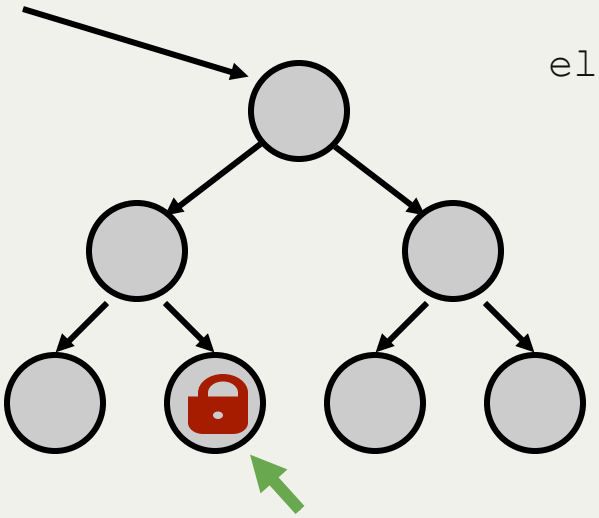
- Suppose we want to insert at the green point



```
if (key < node.key):
    if node.left == NULL:
        insert at node.left
        unlock node
    else:
        lock node.left
        unlock node
        insert_at(node.left)
else:
    if node.right == NULL:
        insert at node.right
        unlock node
    else:
        lock node.right
        unlock node
        insert_at(node.right)
```

Example

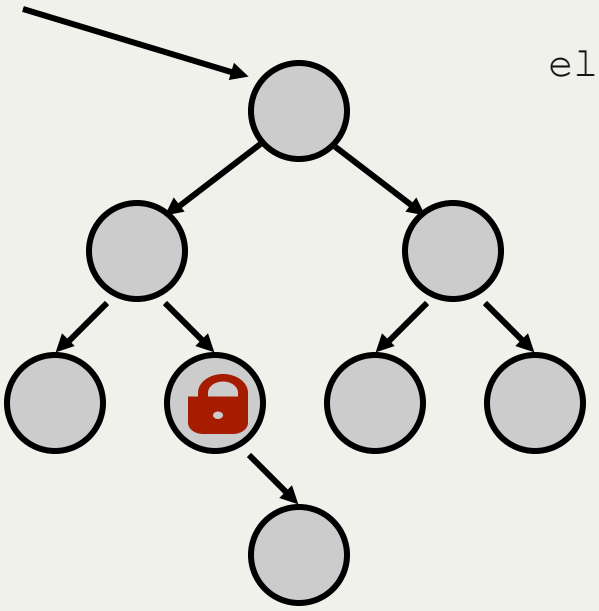
- Suppose we want to insert at the green point



```
if (key < node.key):
    if node.left == NULL:
        insert at node.left
        unlock node
    else:
        lock node.left
        unlock node
        insert_at(node.left)
else:
    if node.right == NULL:
        insert at node.right
        unlock node
    else:
        lock node.right
        unlock node
        insert_at(node.right)
```

Example

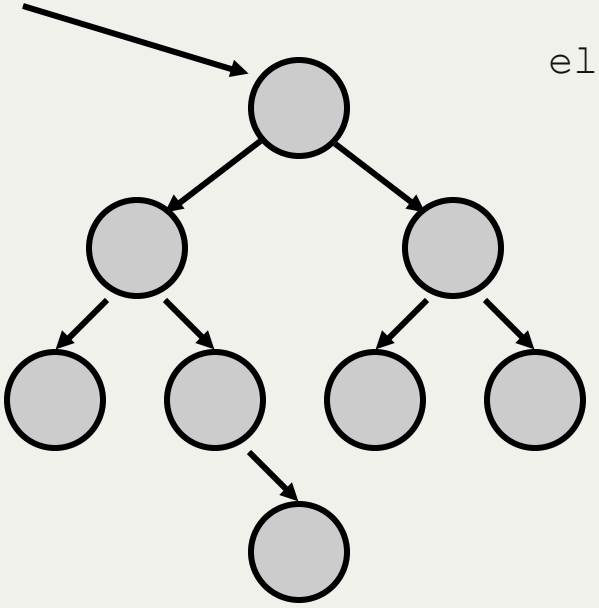
- Suppose we want to insert at the green point



```
if (key < node.key):
    if node.left == NULL:
        insert at node.left
        unlock node
    else:
        lock node.left
        unlock node
        insert_at(node.left)
else:
    if node.right == NULL:
        insert at node.right
        unlock node
    else:
        lock node.right
        unlock node
        insert_at(node.right)
```

Example

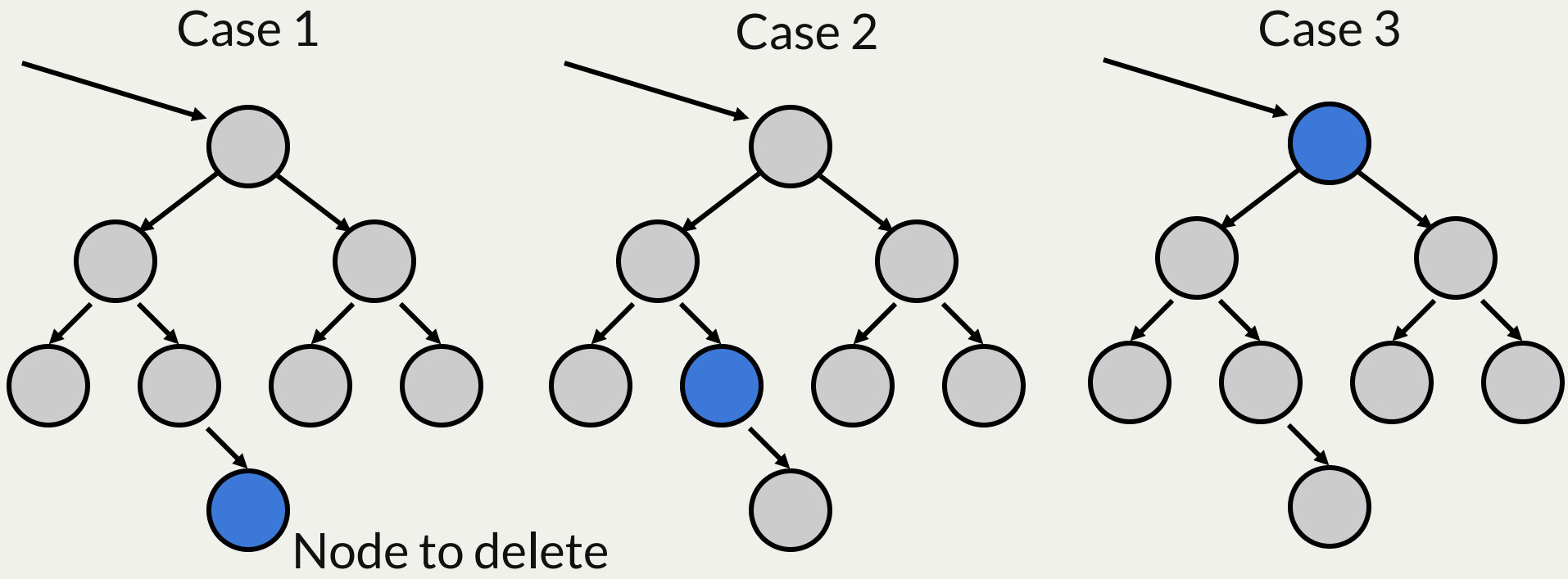
- Suppose we want to insert at the green point



```
if (key < node.key):
    if node.left == NULL:
        insert at node.left
        unlock node
    else:
        lock node.left
        unlock node
        insert_at(node.left)
else:
    if node.right == NULL:
        insert at node.right
        unlock node
    else:
        lock node.right
        unlock node
        insert_at(node.right)
```

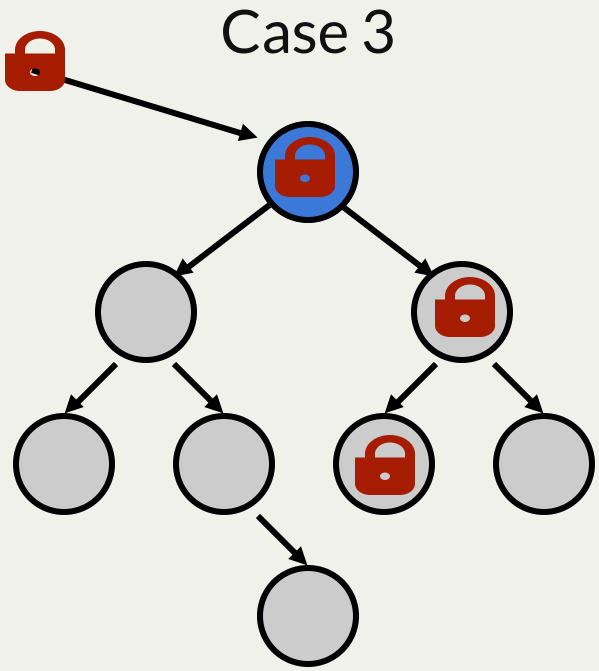
Deletion

- Deletion is much harder...
- Three cases (two easy)
  1. Node to delete is leaf: delete it, set parent pointer to null
  2. Node to delete has one child: delete it, set parent pointer to its child
  3. Node to delete has two children: find successor leaf, replace with this node



Hard Deletion Case

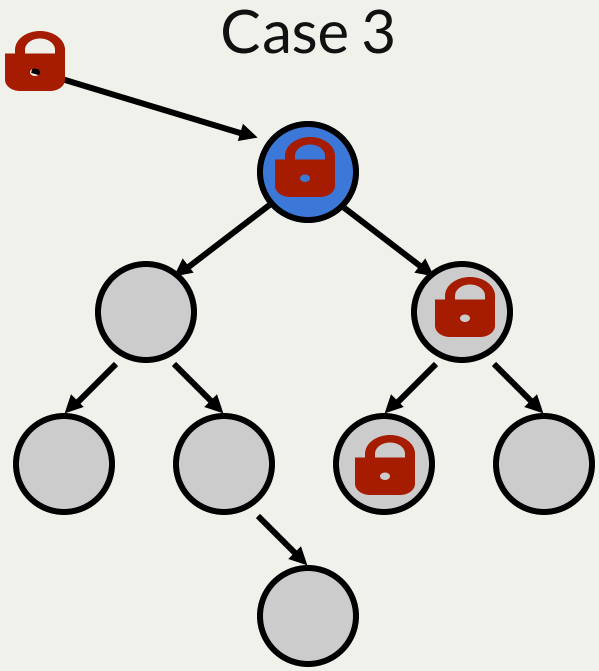
- Need to hold 4 locks!
  1. Parent of node to be deleted
  2. Node to be deleted
  3. Parent of node to be moved
  4. Node to be moved



Hard Deletion Case

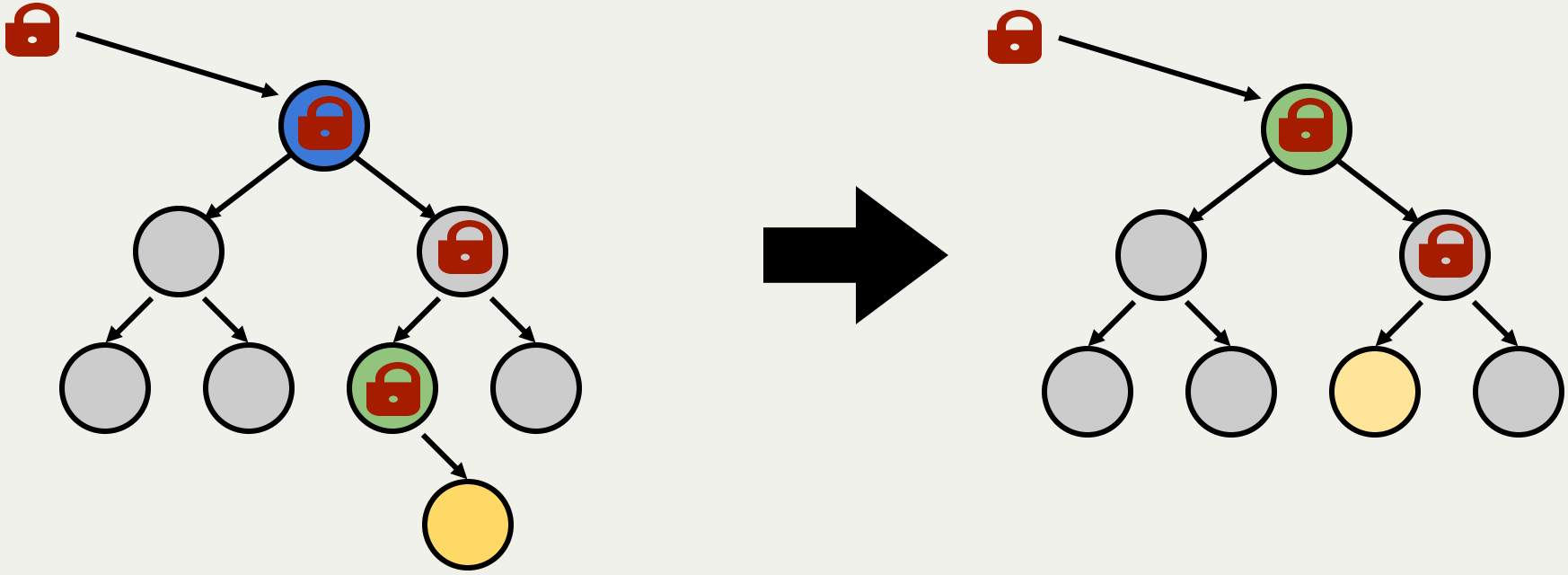
- Need to hold 4 locks!
  1. Parent of node to be deleted
  2. Node to be deleted
  3. Parent of node to be moved
  4. Node to be moved
- Basic pseudocode

```
find node to delete, hold lock and parent lock
find successor of node, hold lock and parent lock
change node's parent to point to successor
change successor to point to node's children
release node parent lock
release node lock
release successor parent lock
release successor lock
```



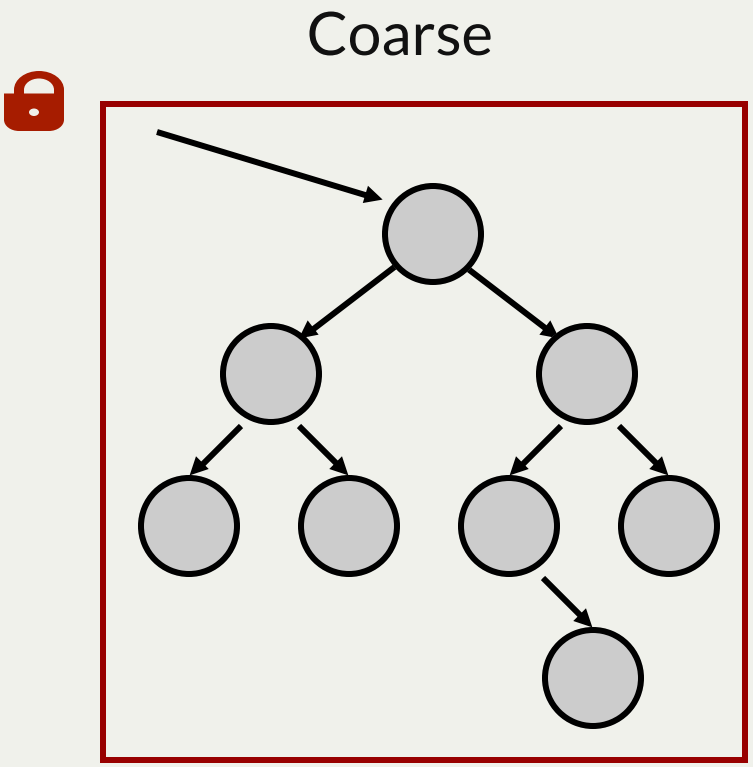
Another Edge Case

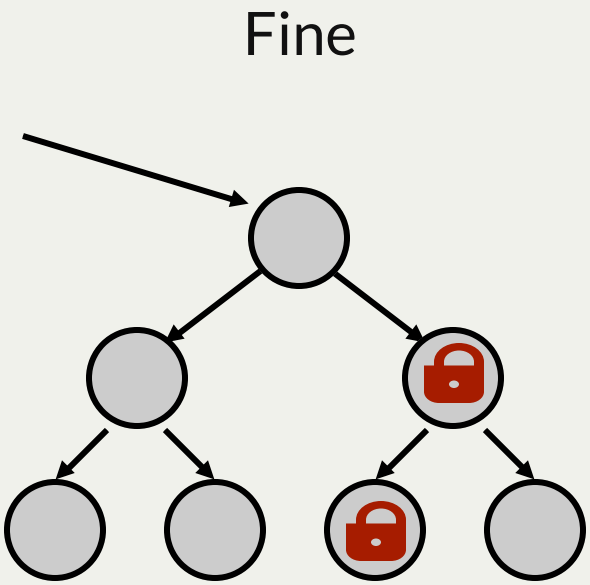
- What if the successor isn't a leaf?
- It can't have both children (or successor would be left child)
  - swap node with successor
  - swap successor with its child



Fine-Grained Versus Coarse-Grained Locks

- What are the advantages of coarse grained locks?
- What are the advantages of fine grained locks?





Thread-Safe Data Structures

- Common data structure libraries are rarely thread-safe
  - Each user of the library has its own atomicity requirements
  - Trying to cover all of them would be a terrible library
  - Applications hand-design thread-safe data structures
- Application-level control of concurrency allows the application do decide on tradeoff between concurrency and overhead
  - Fine-grained locking is good for high concurrency, but costs a lot if only one thread uses the data structure
  - Coarse-grained locking keeps overhead low, but can be a bottleneck under high concurrency (large atomic sections)

Question Period