SEMINAR 2

List

```
begin()
end()
insert(it, v)
       (before)
erase(it)
```

Iterator

```
++
--
==
*
```

```
struct Node {
    value v;
    Node* prev;
    Node* next;
    mutex m;
}
```

if unlocked, next and prev are const
prev -> next == this
next -> prev == this

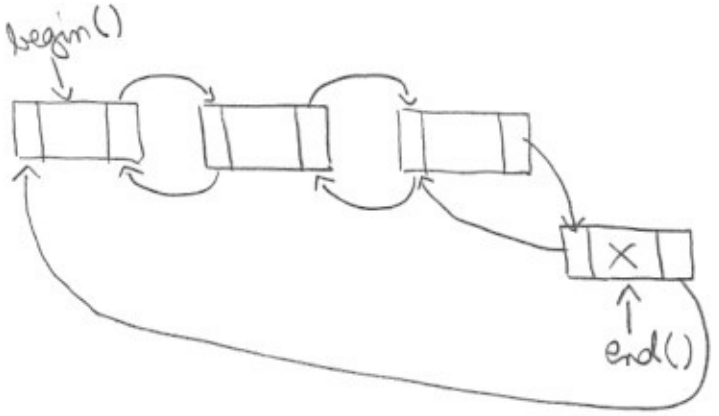
for changing prev; we need to lock prev -> m
for changing next; we need to lock next -> m

OR

```
class Iterator {
    Node* c;
    List* p;
}

class List {
    Node placeholder;
    Mutex m;
}
```

for thread-safe implementation (version I)



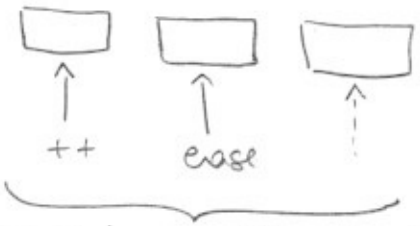

```
it
++it
--it
operator++(){
    c = c -> next;
}
operator--(){
    c = c -> prev;
}
```

```
for(it = l.begin(); it != l.end(); it++){
}
```

(thread) (iterator) (value)
T1 insert(it1, 1)

T2 insert(it2, 2)

→ we have 2 iterators, one performs ++ operation and the other perform Erase operation:



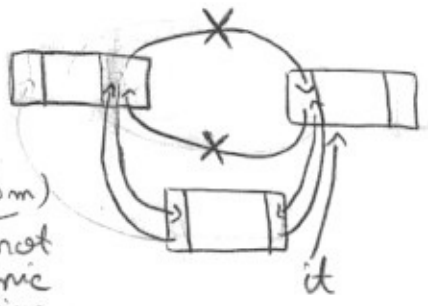
we end up with an invalid iterator

Non-thread safe implementation ⇒ thread safe implementation : Version I + Version II

iterator

```
insert(it, v){
    Node* newNode = new Node;
    newNode -> v = v;                    Version II
→ m.lock()  (Version I)  | unique.lock<mutex> l1(it->m)
                         | unique.lock<mutex> l2(it->prev->m)
    it.c -> prev -> next = newNode;
    newNode -> prev = it.c -> prev;
    it.c -> prev = newNode;
    newNode -> next = it.c;
→ m.unlock()
}
```

this is not an atomic operation

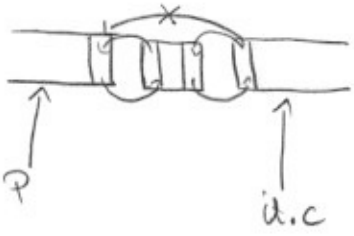


Version I

```
operator ++(){
    p -> m.lock();
    c = c -> next;
    p -> m.unlock();
}
```

```
operation --(){
    p -> m.lock();
    c = c -> prev;
    p -> m.unlock();
}
```

```
{ Node* p = it.c -> prev
→ p -> m.lock()
```

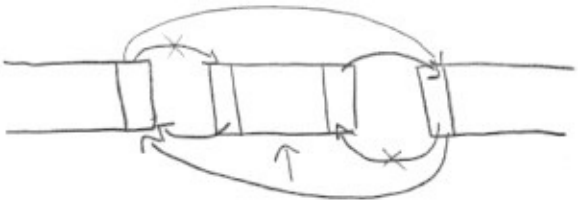


When the struct Node contains mutex m ⇒ Version II

```
operator ++(){
    unique.lock<mutex> l(it.c -> m)
    c = c -> next;
}
```

```
operator --(){
    unique.lock<mutex> l(it.c -> m)
    c = c -> prev;
}
```

Version II



this op is not atomic

```
erase(it){
deadlock { unique_lock<mutex> l0(it.c->m);
           unique_lock<mutex> l1(it.c->prev->m),
           unique_lock<mutex> l2(it.c->next->m);

    it.c->prev->next = it.c->next;
    it.c->next->prev = it.c->prev;

    delete it.c;
}
```

->to avoid these issues, we can implement a solution with try_lock:

```
it.c->m.lock()
while(!it.c->next->m.try_lock(){
        it.c->m.unlock(),
        sleep();
        it.c->m.lock();
}
it.c->prev->m.lock();
```

Producer - Consumer problem

Let's assume we have a producer that somehow will have to signal a consumer:

producer:

```
result = ...
e.signal();
```

consumer:

```
e.wait();
use(result)
```

consumer2:

```
use(result)
```

=> 2 possible solutions ← future event

write one & read many

One Shot Event e

bool signaled = false.