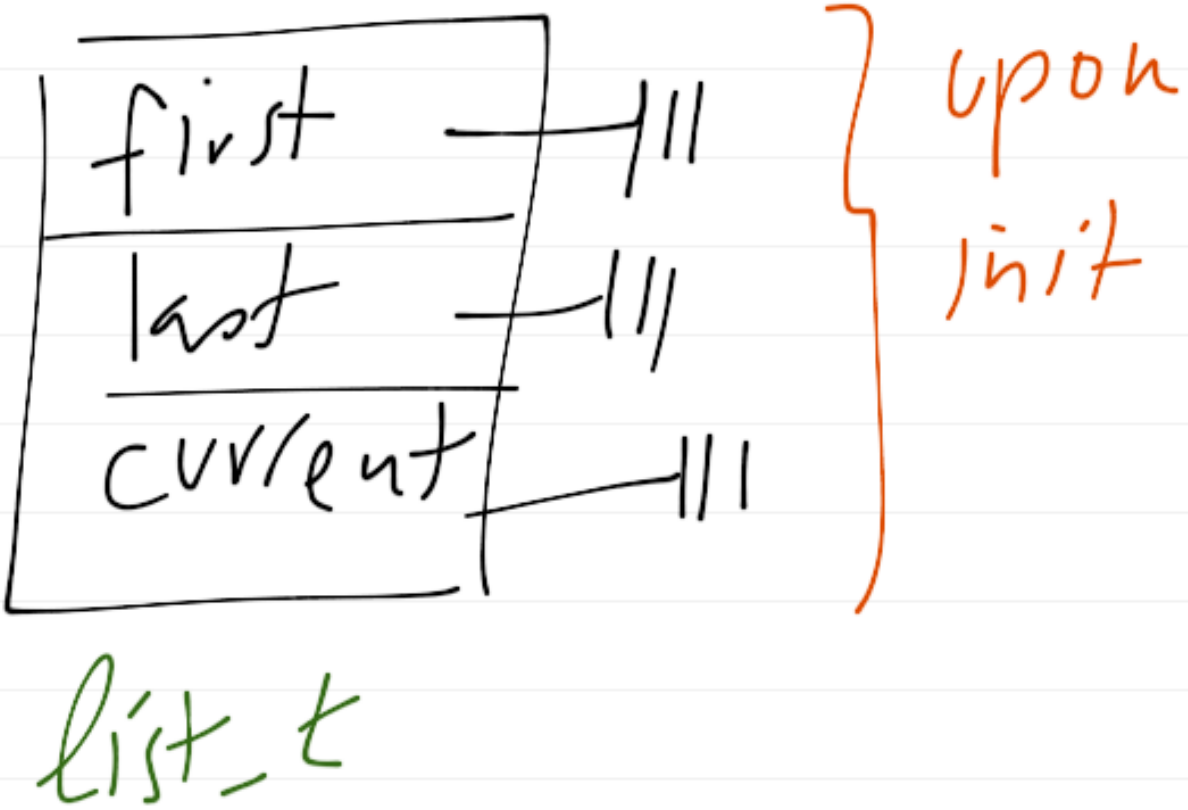
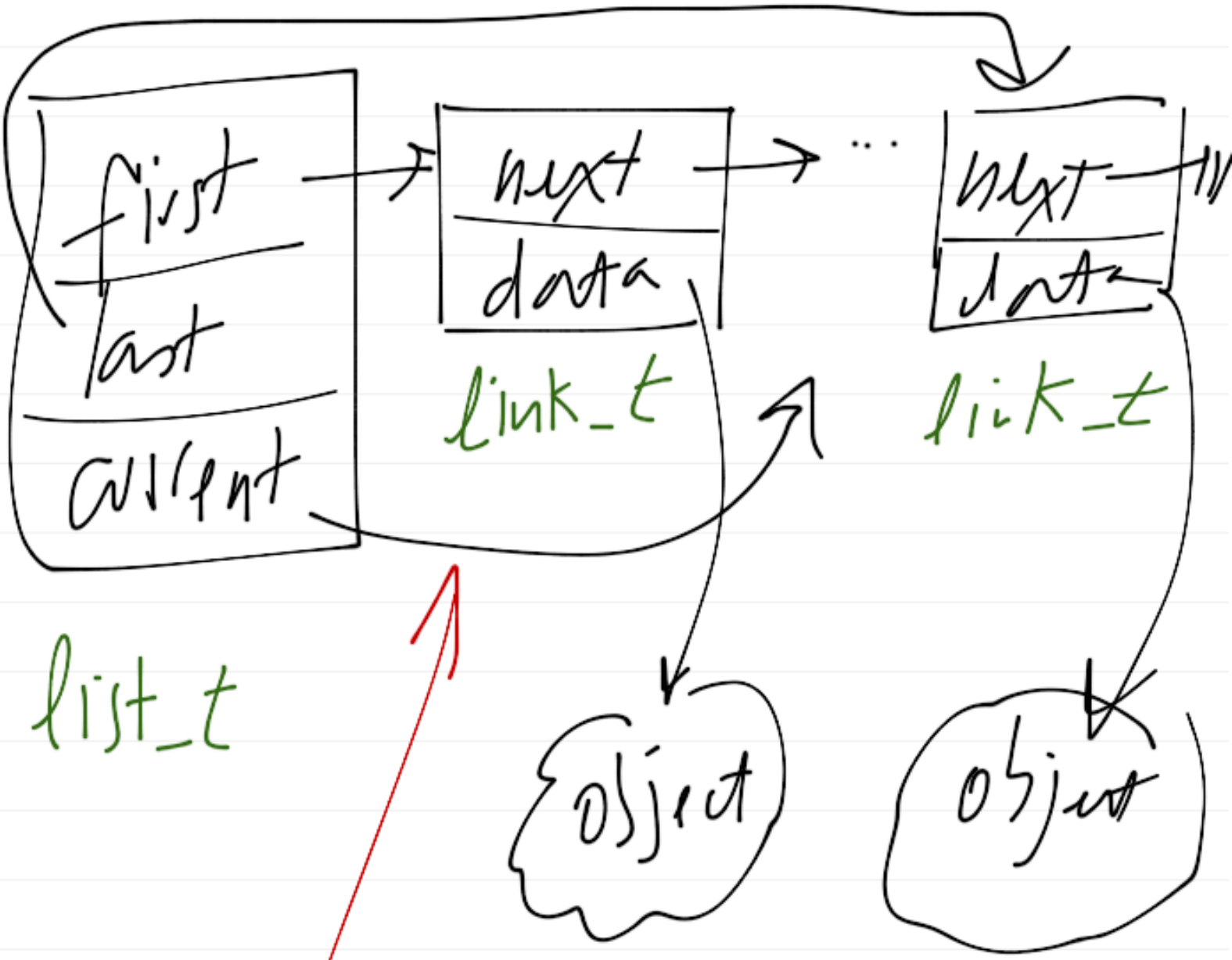
- Obj-C doubly-linked list && list iterator

(lab 12)

- old list:





current pointer pointing to some list element

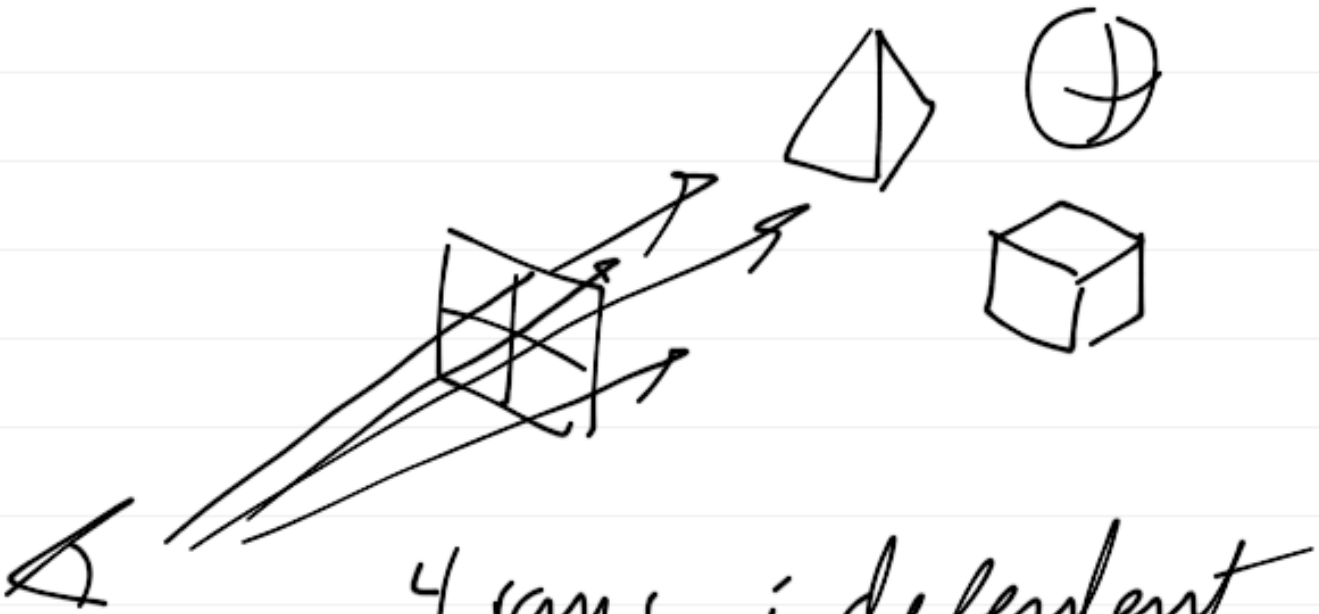
- problem with this implementation
  (problem with this design)

point of data structure
and alg. analysis class
(CPSC 212)

→ only one thread can
iterate thru list at a
time

- what if we want two (or more) concurrent threads to iterate thru list?

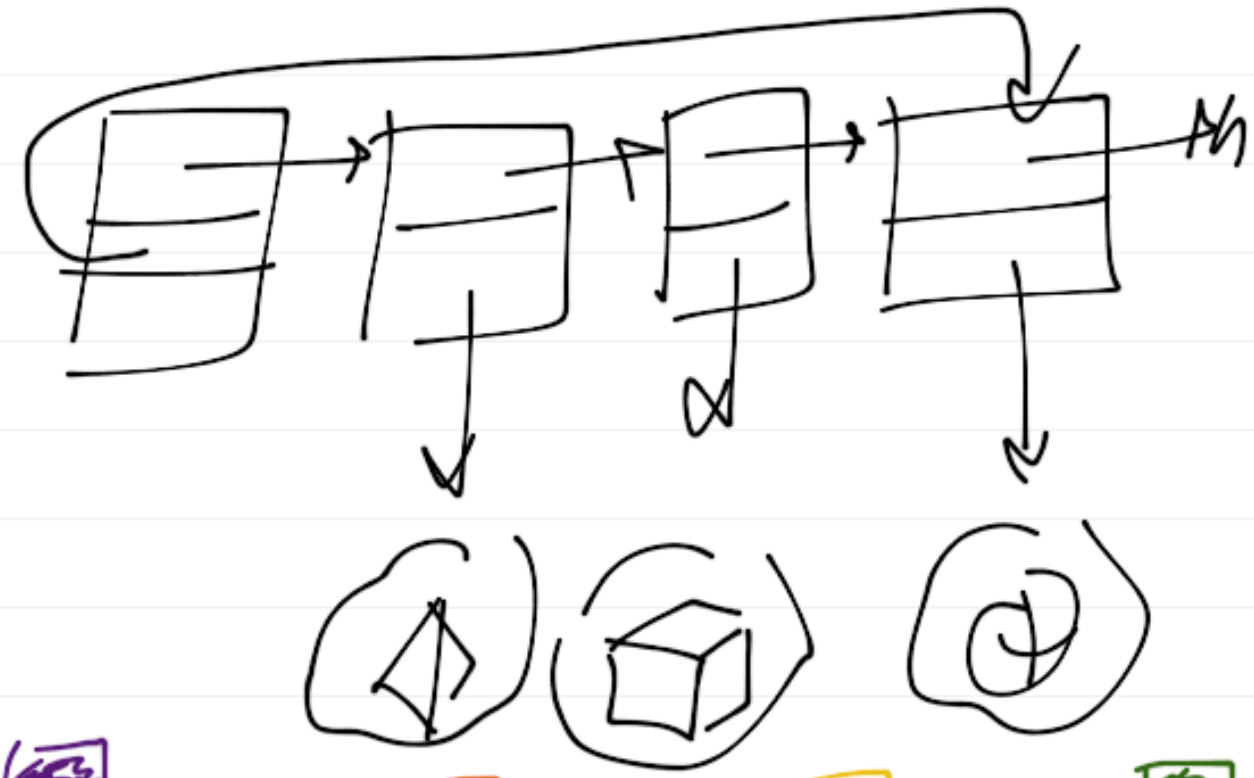
- Consider parallel implementation of ray tracer:



4 rays, independent

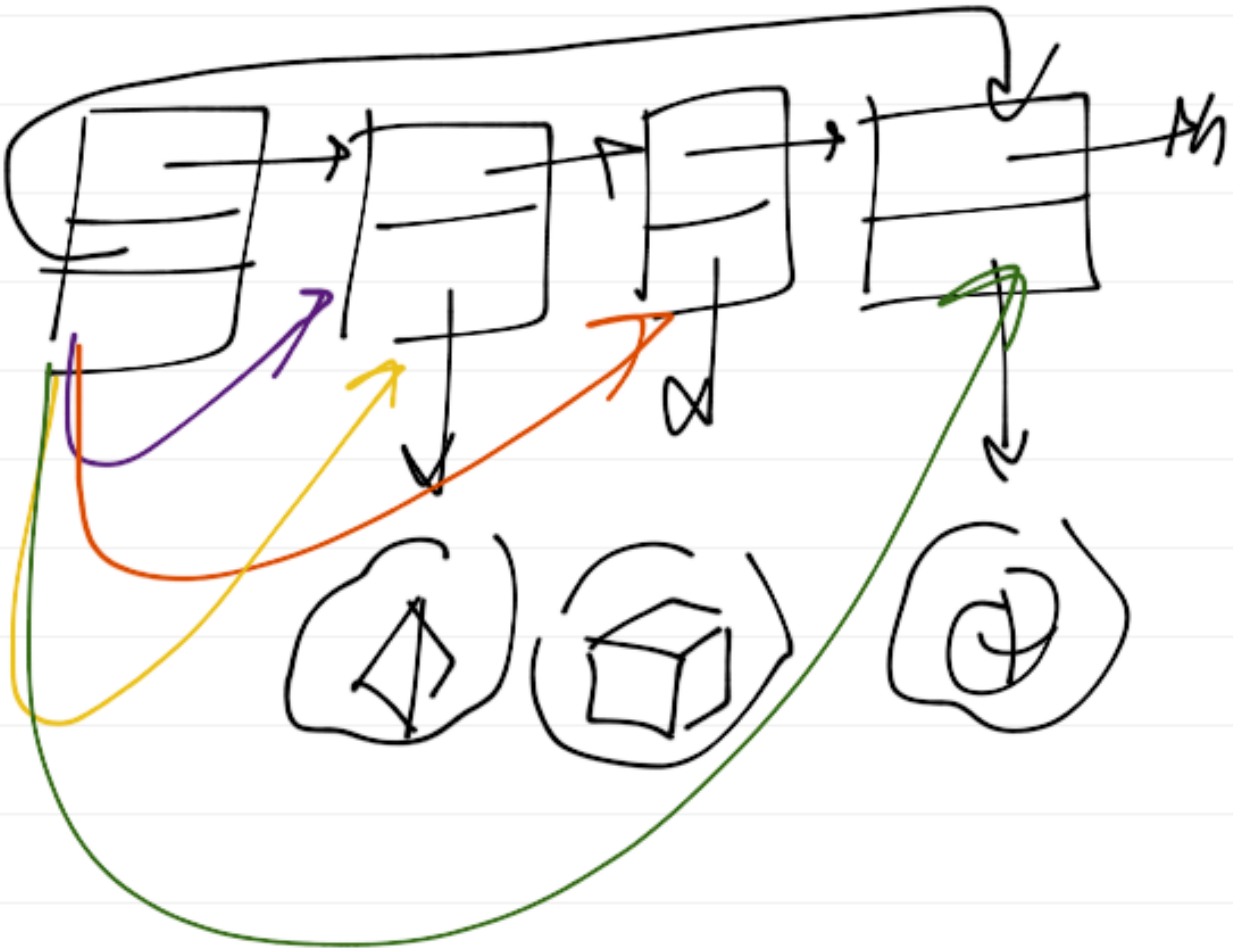
⇒ 4 threads

- list used to store objects:



RACE CONDITIONS

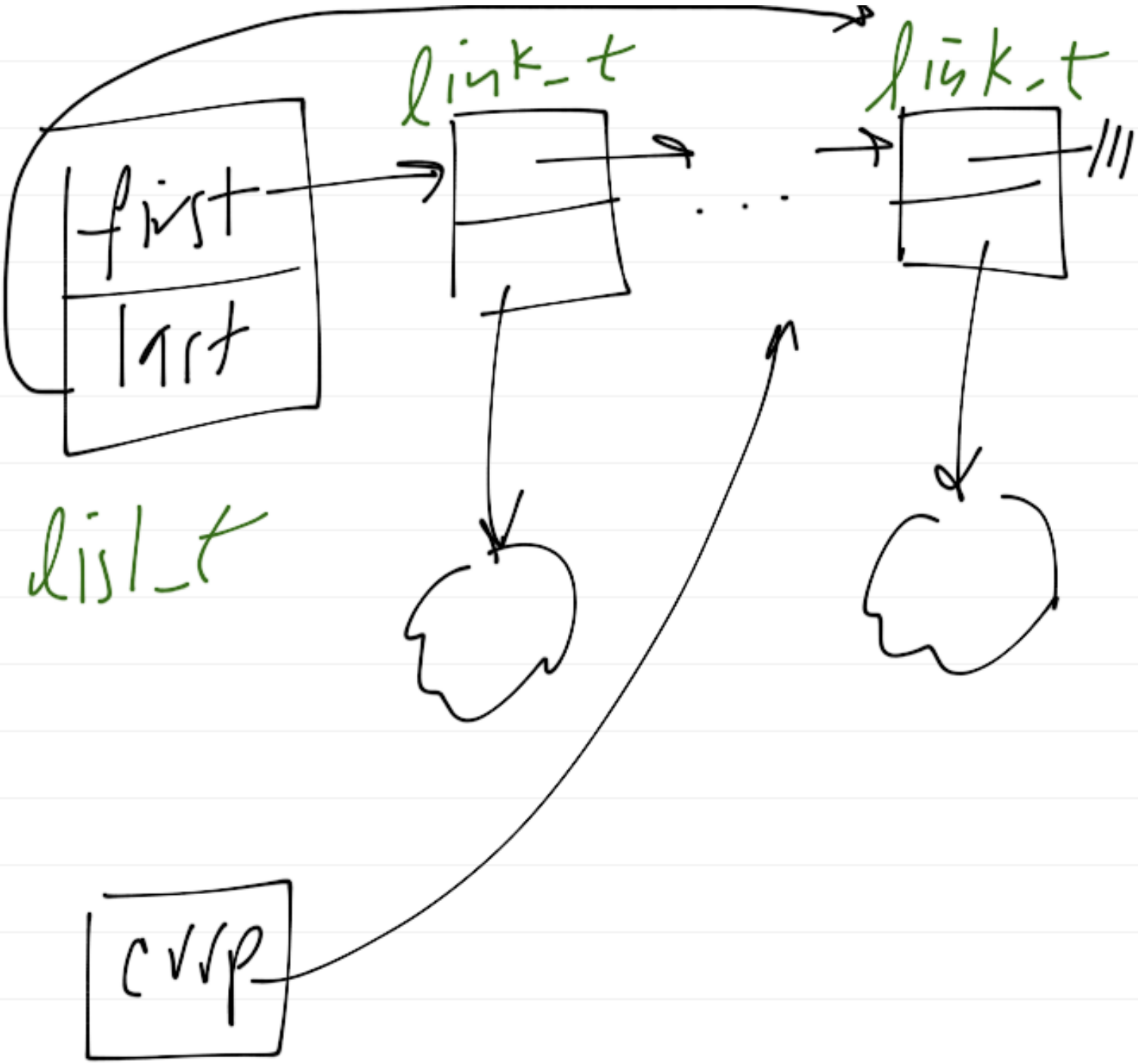
ray 1	ray 2	ray 3	ray 4
current =	current =	current =	current =
first;	first;	first;	first;
current =	current =	current =	current =
next;	next;	next;	next;



- each thread overwrites shared resource (current pointer)
- need this to be private

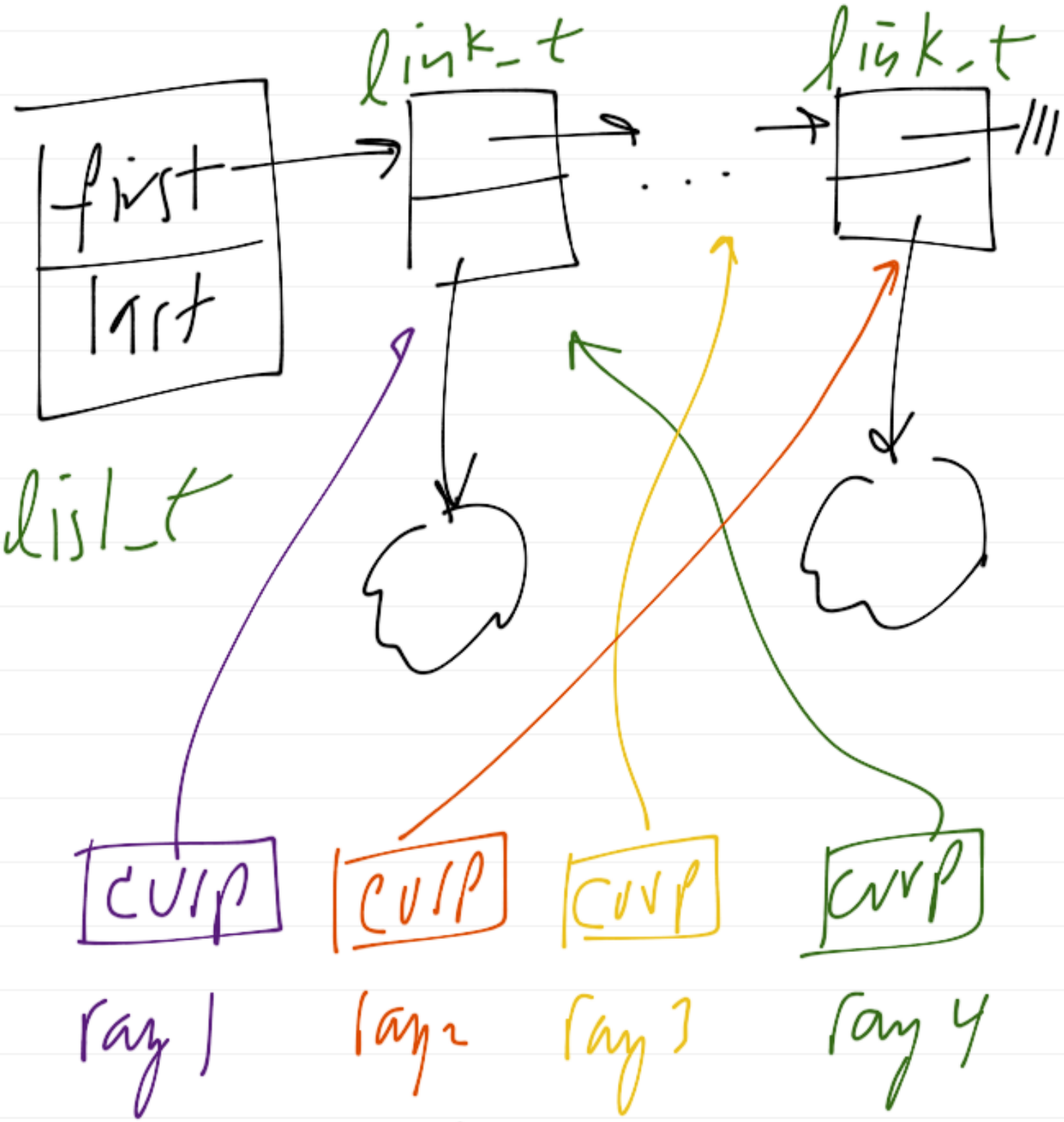
- each thread needs its own private copy of current ptr.

- need to remove current pointer (curp) from list_t into its own type: litr_t for "list iterator type"



litr_t

- now each thread has own litr_t copy



- each thread iterates thru a shared list via private iterators

- intended usage:

```
list_t *list =
[(list_t *)[list_ alloc] init];

litr_t *litr = nil;

for(litr = [list begin];
    ![litr equiv: [list end]];
                [litr next])

[[litr data] print: stderr];
```

[list begin] — returns an iterator pointing at first element (conceptually)

[list end] — returns iterator pointing at last element (conceptually)

- if [list begin] and [list end] return iterators pointing at same list node, list is empty (by design)

- need to test for list_t equivalence, but can't just use "=="

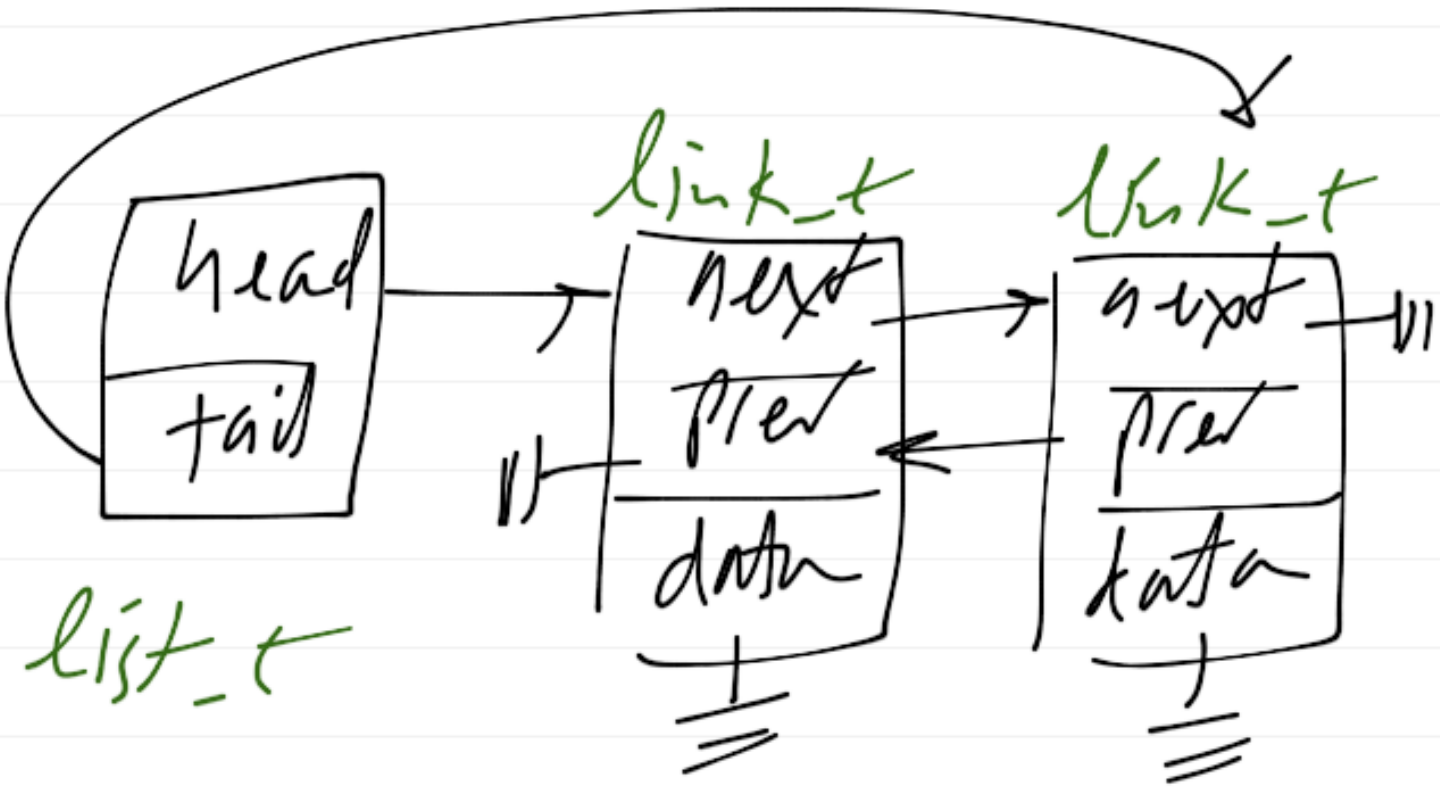
[list equiv: [list end]] -

tests to see if this list points to same list link as [list end]

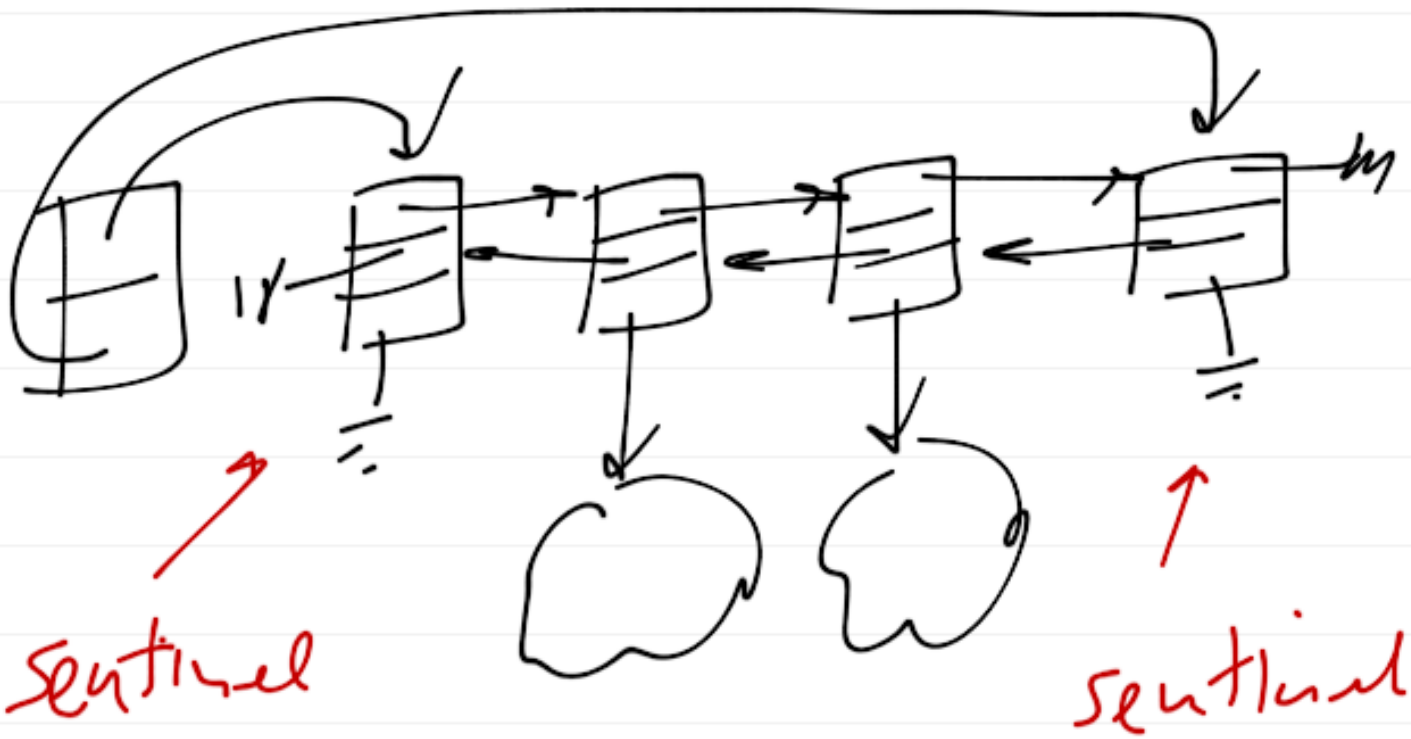
- BUT... our list design now changes to a doubly-linked list

- We now construct a list_t so that it contains two sentinel nodes (nodes with no data)

- empty list (upon init):



- list with data:



[list begin] returns

[head next]

[list end] returns

[tail]

[list begin] returns iterator pointing to first list element — called be sentinel node tail

- implementation:

```
- (list_t *) begin
{
  return [[(list_t *)
    [list_t alloc] init:
      [head next]] autorelease];
}
```

[list end] returns iterator pointing to tail sentinel node

- implementation:

```
- (list_t *) end
{
    return [[ (list_t *)
        [list_t alloc] init: tail
            autorelease],
}
```

- What does list_t object look like?

- just a pointer to link_t

```
@interface list_t: NSObject
{
  link_t * curp;
}
  :
@end
```

- member functions (litr_t):

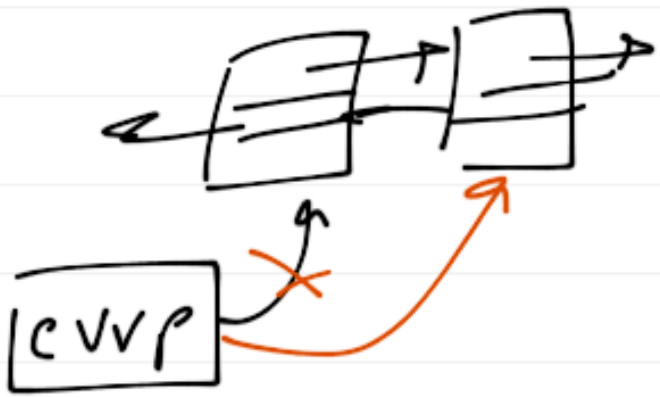
```
// constructors
-(id) init;
-(id) init: (link_t *) _link;

// accessors
-(link_t *) curp;
-(id) data;

// operators
-(id) next;
-(id) prev;
-(bool) equiv: (litr_t *) rhs;
```

// right-hand side

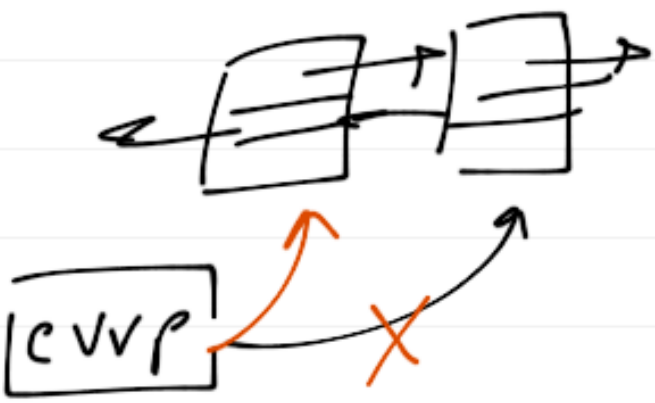
- implementation:

```
-(id) init {return [self init: nil];}
-(id) init: (link_t *) link
{
    if (!(self = [super init]))
        return nil;
    curp = _link;
    return self;
}
```

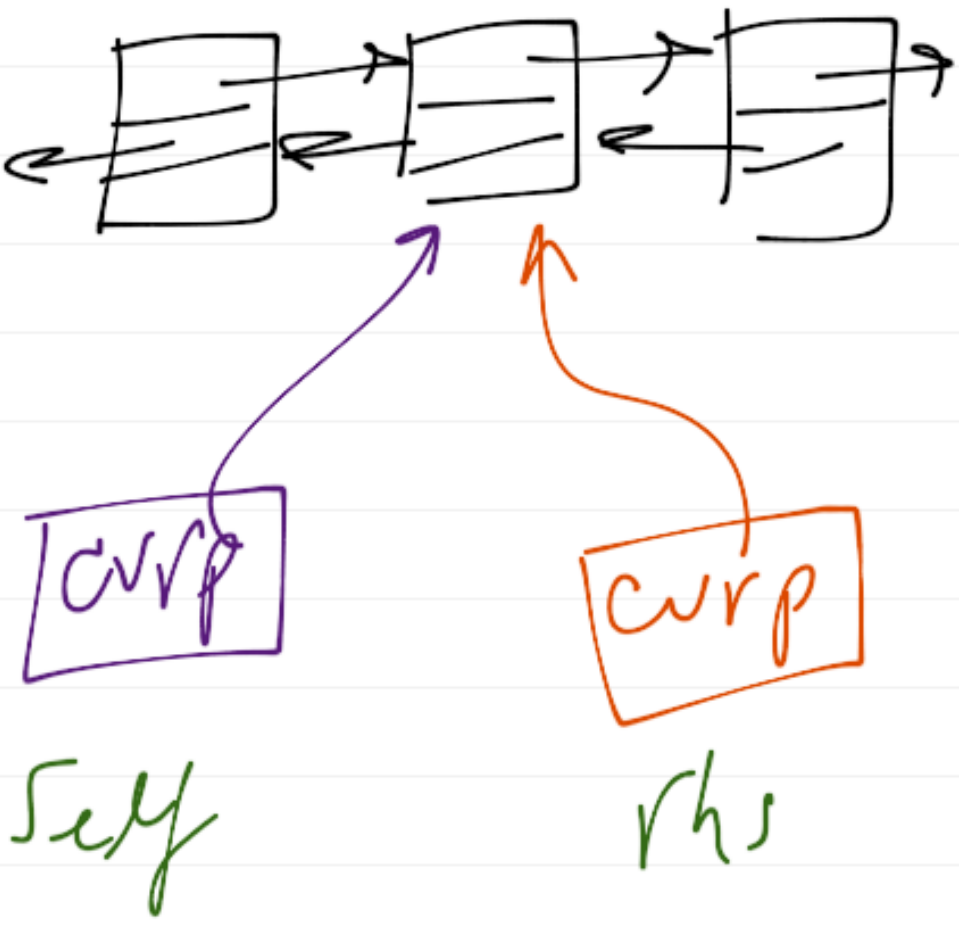
```
- (id) next  // advance iterator
{
    curp = [curp next];
    return self;
}
```




```
- (id) prev  // retract iterator
{
    curp = [curp prev];
    return self;
}
```



```
-(bool) eqv: (lit_t *) rhs
{
  return curp == [rhs curp];
}
```



- what dont list_t ?

```
@interface list_t : NSObject
{
  link_t *head;
  link_t *tail;
}
  :
@end
```

- other member functions (list_t)
  (you've seen begin, end)

- (id) init; // constructor

- (litr_t *) push_back : (id) _data;
  // add to end

- (litr_t *) insert : (litr_t *)
  _litr : (id) _data;
  // insert data before _litr

```
-(void) pop_front;
    // remove front node
        (& release memory)

-(list_t *) erase:
        (list_t *) _litr;
    //erase node at iterator
```

```
- (id) front;
    // return pt. to data
       of first element

- (id) back;
    // return data of last
       element

- (bool) empty;
    // true if list is empty
```

- implementation:

```
- (bool) empty
{ return [head next] == tail;
}
```

- empty list:

head tail

[head next] == tail ✓

```
-(id) front
{
  return [[self begin] data];
}
```

iterator

iterator's 'data' member fun.

```
-(id) back
{
  return [[[self end] prev] data];
}
```

tail

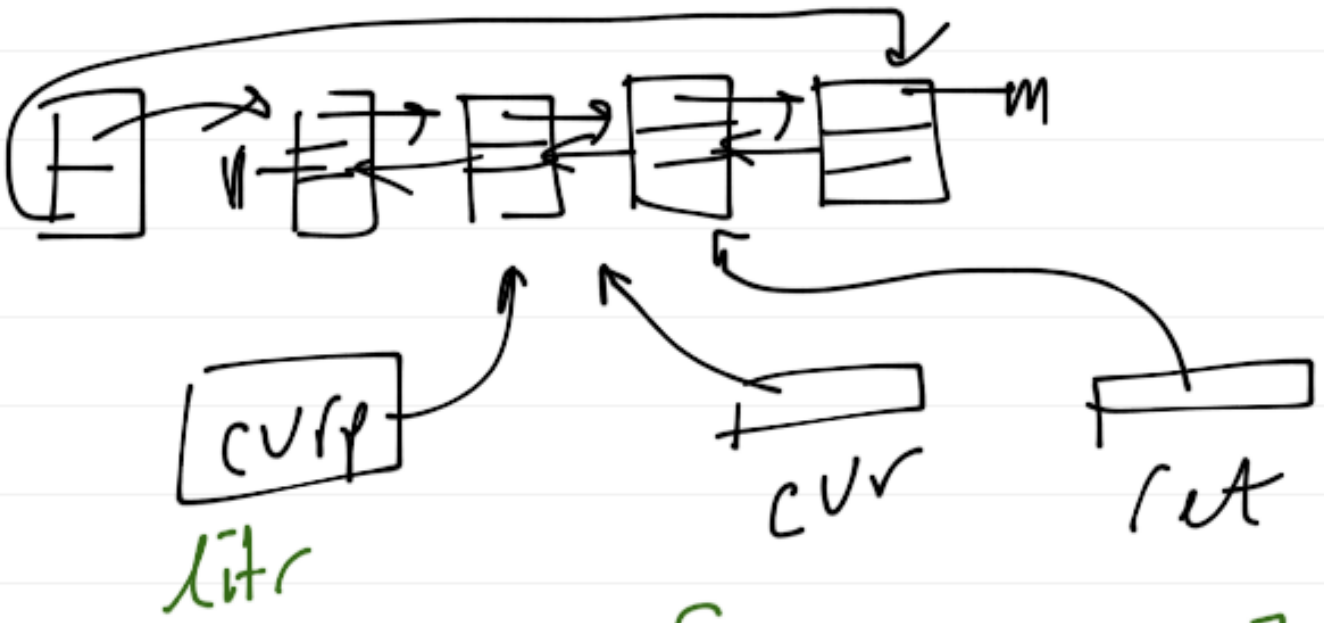
last element

data

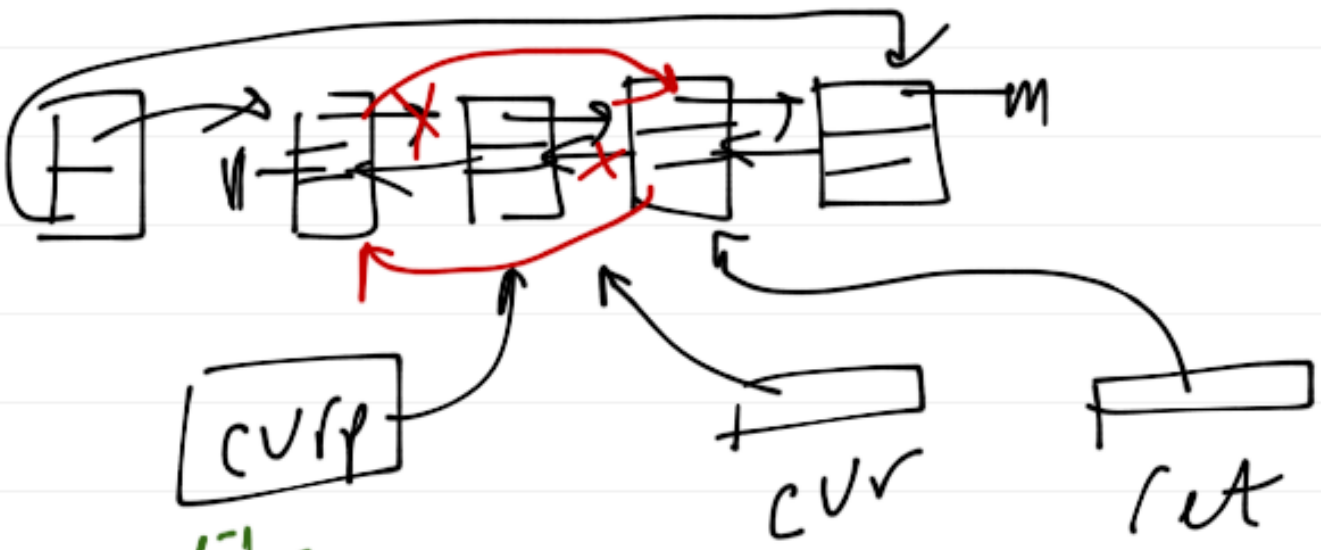
```
-(void) pop_front
{ [self erase: [self begin]];
}
```

iterator

```
-(list_t *) erase: (list_t *) _list
{ link_t *cur = [_list curp];
  link_t *ret = [(list_t *)
     [list_t alloc] init: [cur next]];
  [[cur prev] setnext: [cur next]];
  [[cur next] setprev: [cur prev]];
  [cur release];
  return [ret autorelease];
}
```

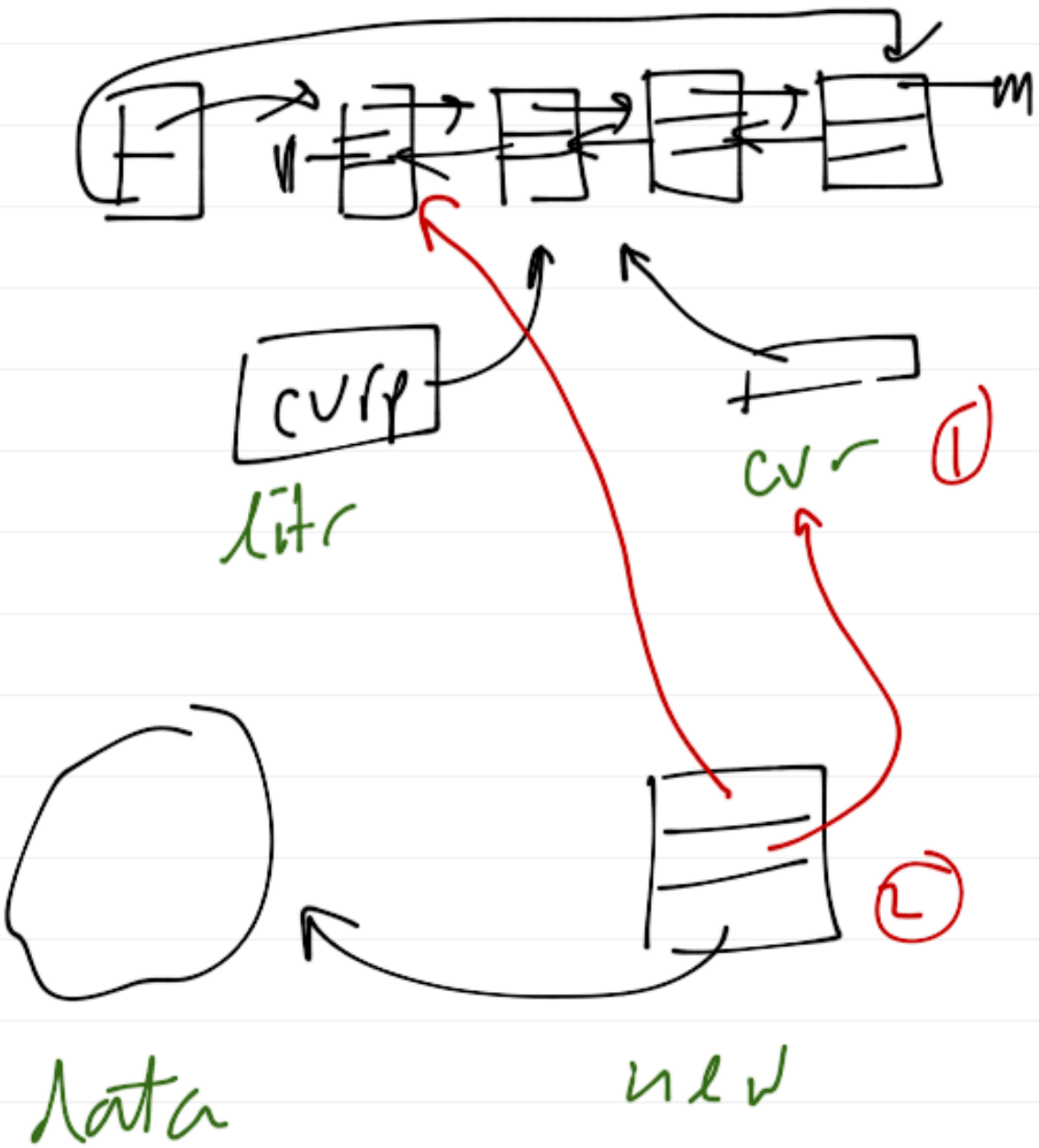


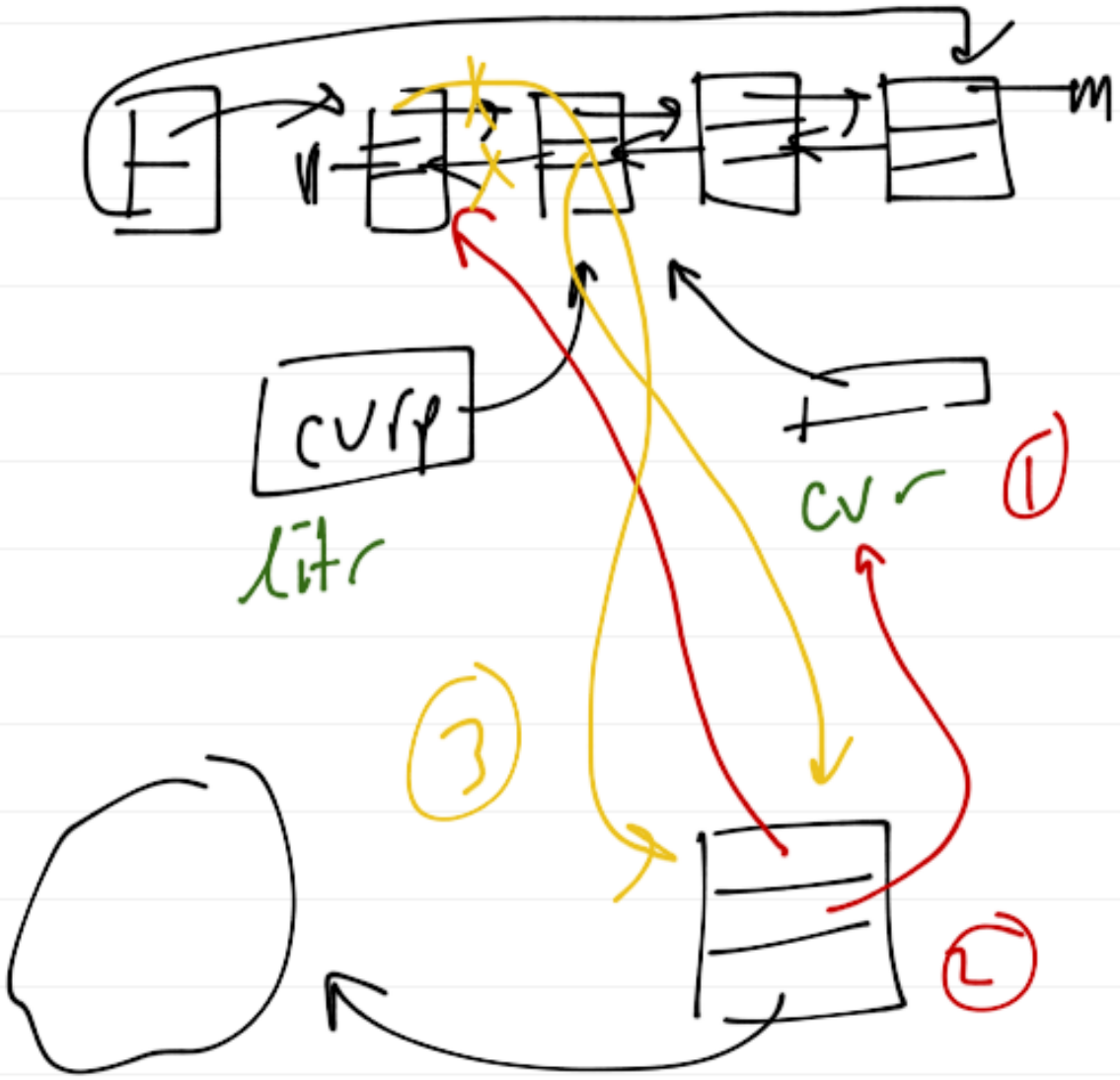

```
link_t *cur = [_litr curp];
link_t *ret = [(litr_t *)
  [litr_t alloc] init: [cur next]];
```




```
[[cur prev] setnext: [cur next]];
[[cur next] setprev: [cur prev]];
```

```
-(id) init
{
    if (!(self = [super init]))
        return nil;
    head = [(link_t *)
        [link_t alloc] init];
    tail = [(link_t *)
        [link_t alloc] init];
    [head setNext: tail];
    [tail setPrev: head];
    return self;
}
```


curr
curr
1
2
data
new



- new gets inserted ahead of curp
- push_back calls insert