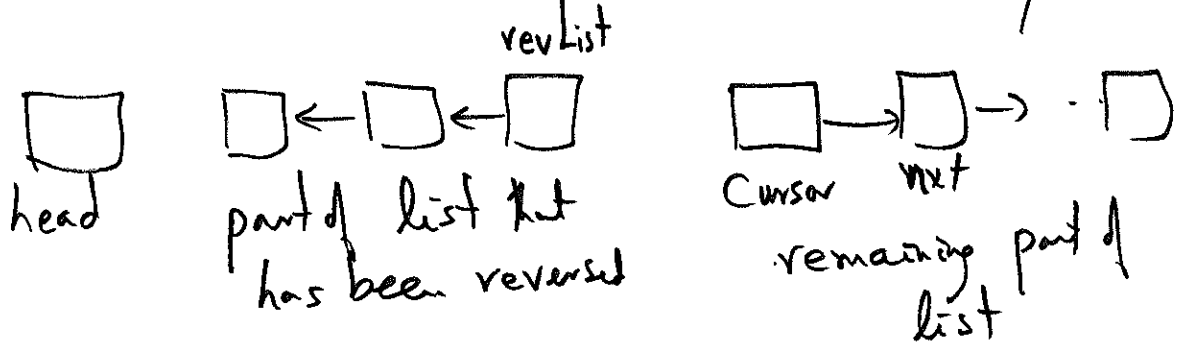
reverse( ) // reverse elements of a linked list.

// SLL, dummy header, Entry class with element, next pointer.

// LI:



// Initialization

revList ← null cursor ← head.next

While cursor ≠ null do

// Move cursor to revList

nxt ← cursor.next

cursor.next ← revList

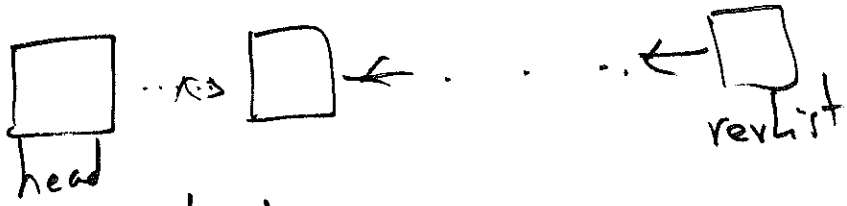
revList ← cursor

cursor ← nxt

RT = O(n)

Extra space = O(1).

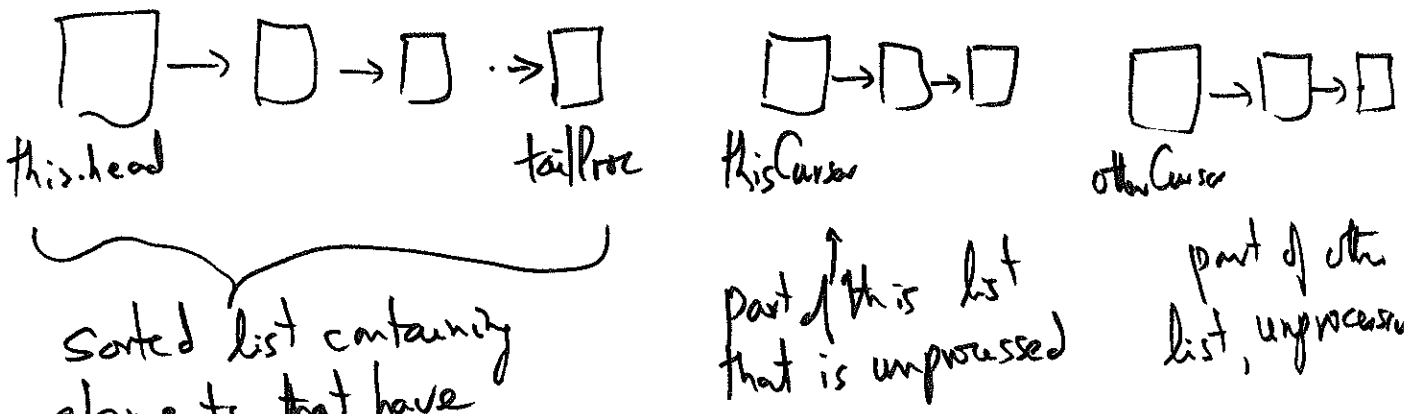
// Termination:



head.next ← revList

merge (SLL<T> other) : // merge this list (sorted) with other list (sorted) into one sorted list

//LI:



tailProc : tail of processed list, thisCursor = head of unprocessed this list
otherCursor = ... other ..

// Initialization:

tailProc ← this.head thisCursor ← this.head.next
otherCursor ← other.head.next

while thisCursor ≠ null and otherCursor ≠ null do
  if thisCursor.element ≤ otherCursor.element then
    tailProc.next ← thisCursor ; tailProc ← thisCursor
    thisCursor ← thisCursor.next
  else tailProc.next ← otherCursor
    tailProc ← otherCursor
    otherCursor ← otherCursor.next
if thisCursor = null then tailProc.next ← otherCursor
  this.tail ← other.tail
else tailProc.next ← thisCursor

$R_T = O(n)$
Extra space $= O(1)$

```
intersection(List a, List b, List res) :// a, b = Lists, sorted sets.
    it1 ← a.iterator()                    res = empty list (output)
    it2 ← b.iterator()
    x1 ← next(it1)        x2 ← next(it2)
    while x1 ≠ null and x2 ≠ null do
        if x1 < x2 then x1 ← next(it1)
        else if x1 > x2 then x2 ← next(it2)
        else
            res.add(x1)
            x1 ← next(it1)
            x2 ← next(it2)
```

Note: In pseudocode: if $x_1 < x_2$

⇓

Code: if(x1.compareTo(x2) < 0)

$RT = O(n)$ $\quad n = a.size() + b.size()$.

Fail safe next:

```
return it.hasNext()
    ? it.next() : null;
```

```
next(Iterator<T> it):
    if it.hasNext() then
        return it.next()
    else return null
```

Queues: List in FIFO (First-in First-out) order

Enqueue(x) = add(x) – add a new element at the rear of queue.

Dequeue() = remove() – remove and return element at the front of the queue.

isEmpty(), size(), clear()

Implementations:

① LinkedList: add(x) ⟶ list.addLast(x) -O(1)
remove() → list.removeFirst() -O(1)

↳ used in single threaded applications.
e.g: Breadth-first search (BFS)

```
Queue<Vertex> q = new LinkedList<>();
```

② ArrayDeque: array-based double ended queue.
add/remove elements from both ends.
Active elements occupy some part of array (not necessarily from index 0).

③ Bounded sized queues. – used in multithreaded/multiprocessor apps
Producer/Consumer applications.
add(x) → offer(x)
remove() → poll()