Today - Lectures 3 & 4 CS163

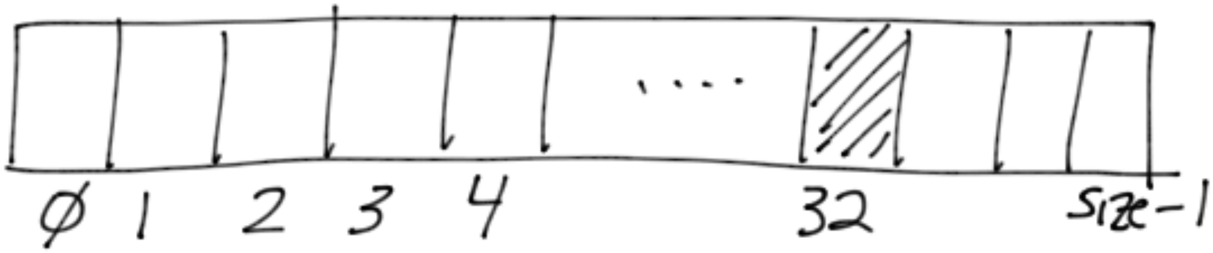
1) Topic #2 - Summarize ADTs

2) Topic #3 - Building Ordered Lists

- consider efficiency trade offs between array and LLL implementations

Announcements:

Absolute Ordered Lists - Array



✓ Direct Access

$\underbrace{\text{array}[32]}_{*(\text{array}+32)}$ = ... position 33

*(array + 32)

32 * sizeof(element)

+

temporary

X Memory - How much?

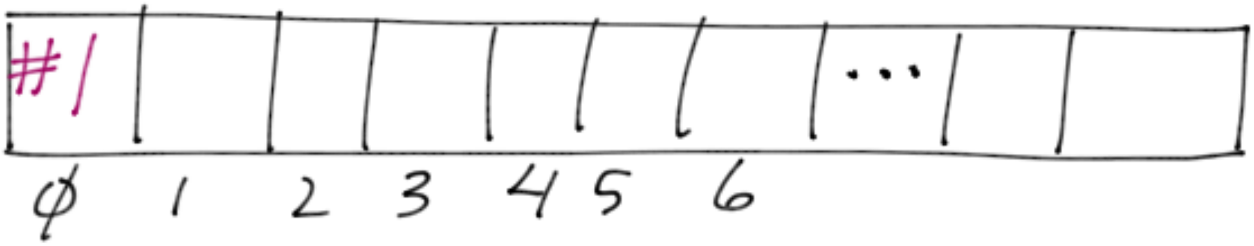
X Memory - Possible Waste

X Memory - Can we allocate all that is needed contiguously?

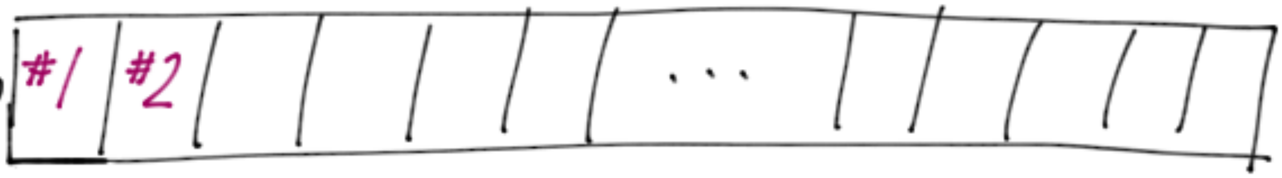
Relative Ordered List - Array

✓ Direct Access - (but not as meaningfull as for an Absolute Ordered List)

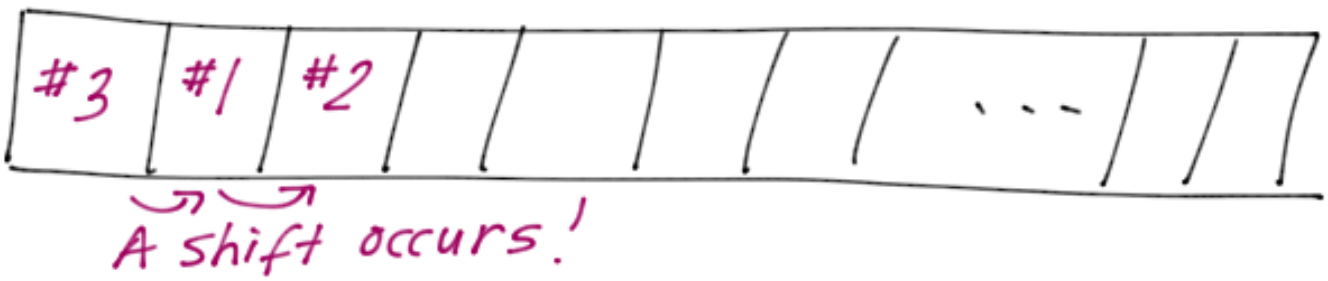
Add 1st # at any position



Add again at any position greater than 1



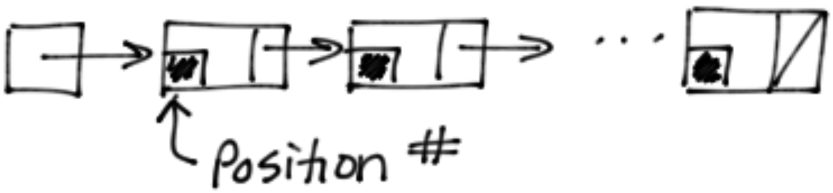
Add now at position 1



x Shifting - when data is inserted anywhere other than an "append", the data must move down (hopefully using an array of <u>pointers</u> to <u>the</u> data)

x Memory Issues (as before)

$\underbrace{\text{Absolute Ordered List}}_{\text{ADT}}$ – $\underbrace{\text{Linear Linked List}}_{\text{Data Structure}}$



✓ Memory – Not Required to be contiguous
- No need to know up front how much memory is needed
- "NO" waste for "over" estimating

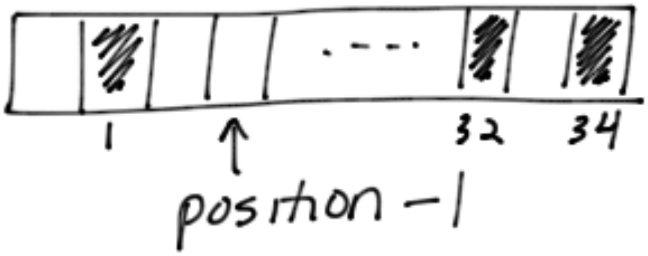
X Memory – 1 extra address per node
(10,000 nodes * sizeof an address)

X Sequential Search – No Direct Access

X Requires a "position" data member

Absolute Ordered List – Performance

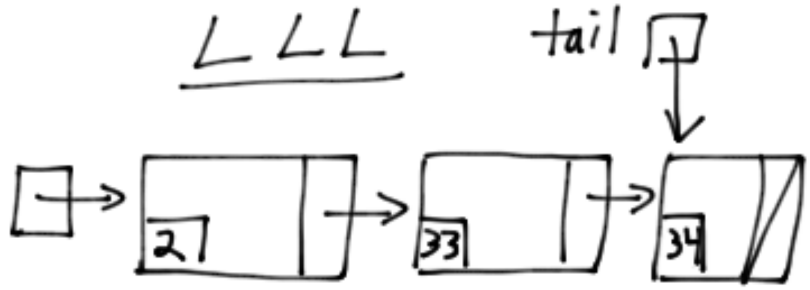
Array



array[position -1]

* Replaces what is there
* Requires initializing $\boxed{\text{all}}$ elements
* 10,000 elements + 1 ptr

LLL



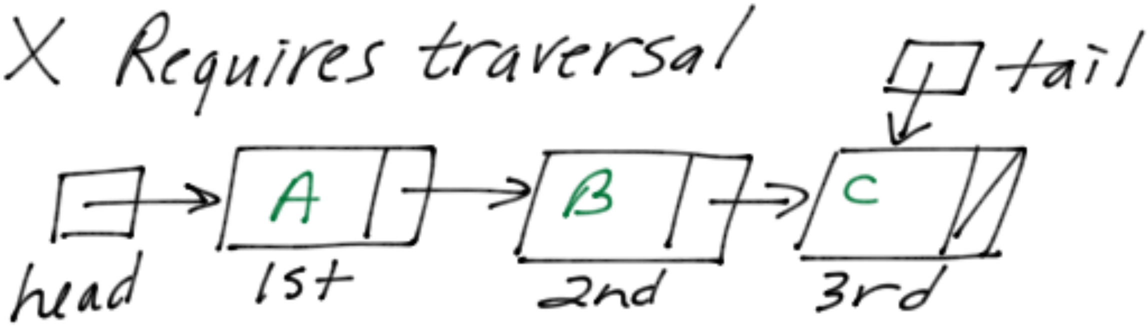

```
previous = NULL;
current = head;
while (current &&
    current->pos < toadd-pos)
{
    previous = current;
    current = current->
                next;
}
if (previous)
{   previous->next =
        new node;
    previous = previous->next;
    previous->data = ____;
    previous->pos = toadd
                    position;
    previous->next = current;
} else
    // add at head
```

Relative Ordered List – Linear Linked List

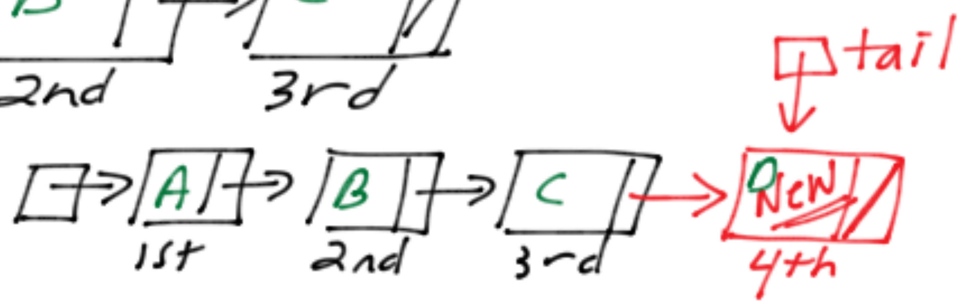
✓ Memory – No need to be aware of # items
- Not required to be contiguous
- No need to store position #'s (in fact, position #'s change — so it is important $\boxed{\text{NOT}}$ to store position numbers.

✓ Avoids Shifting!

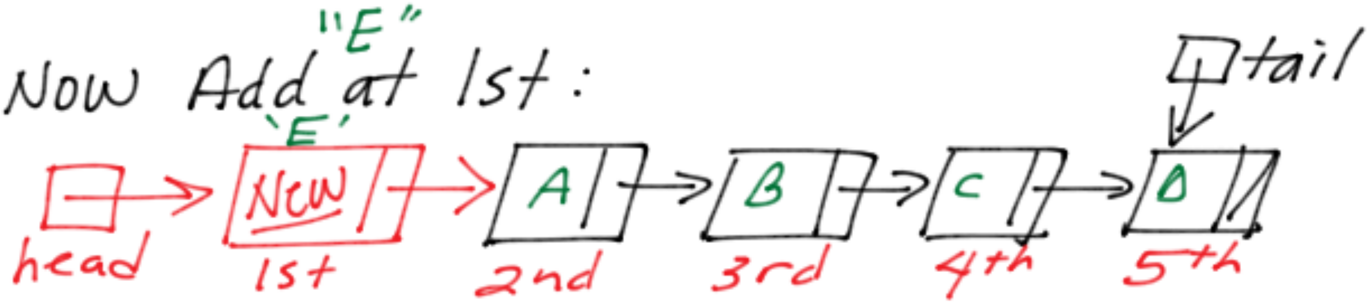
X Requires traversal



Add at 55th:
"D"

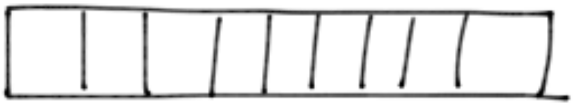


Now Add "E" at 1st:



Relative Ordered List – Performance

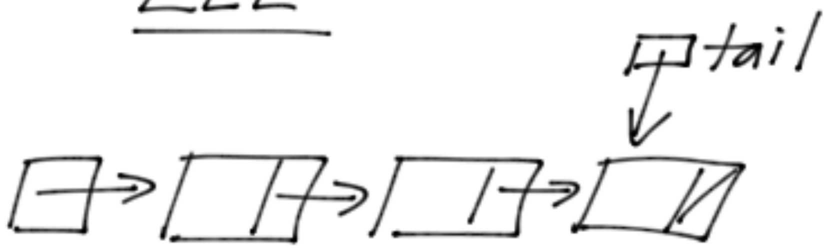
Array



- Append has direct Access:

array[lastused] = ...

LLL



- Append is quick with a tail pointer

```
tail->next = new node;
tail = tail->next;
tail->data = ...
tail->next = NULL;
```

Performance Results:

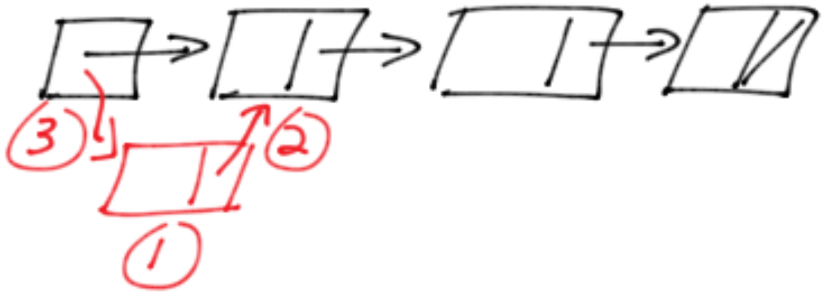
~8 op/fetch vs ~18

Relative Ordered List — Performance

Array

Add at beginning
requires shifting!

LLL




```
temp = new node;
temp->data = ....
temp->next = head;
head = temp;
```

Performance Results:

8+ 10,000*shift vs ~12

Doubly Linked List

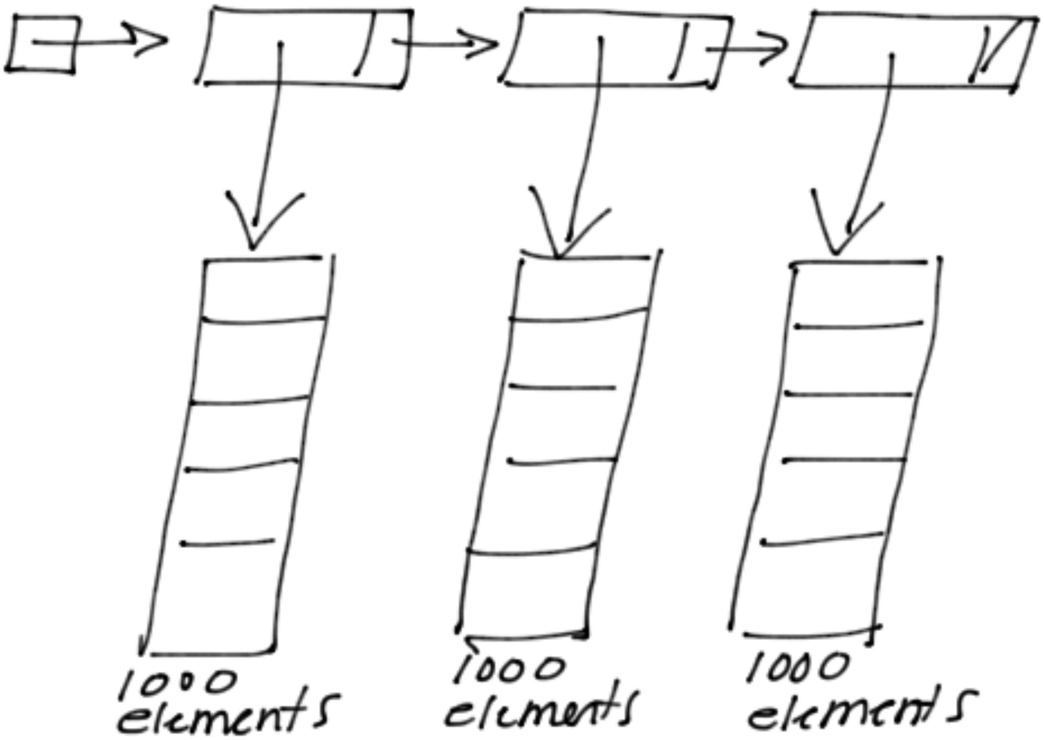

```
struct node
{
    data some_data;
    node *previous;
    node * next;
};
```

✗ Memory Overhead (2 addresses per node)

✓ Flexibility of Progressing forward or backwards as needed

Absolute Ordered List — Alternative

"Flexible Array"




```
struct node
{
    data *array;
    node *next;
};
```

* Positions 1–ArraySize uses 1st array

* (Position# − 1) / Array Size } Tells you how far to traverse

  ↑ quotient for integer division

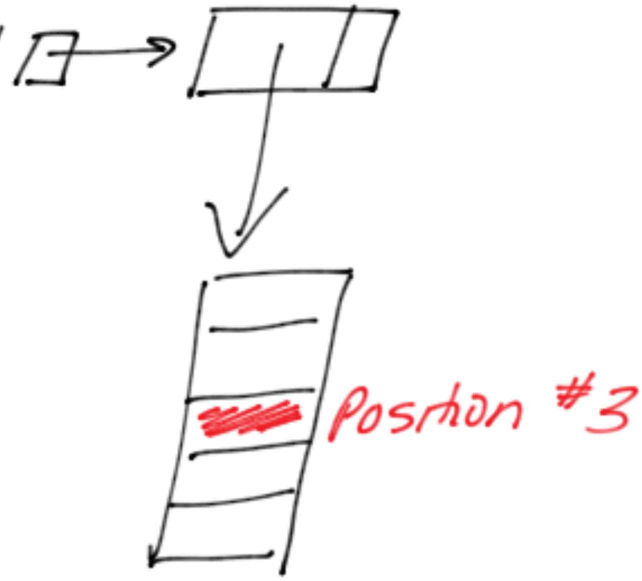
* (position# − 1) % Array Size } Tells you which index in the array to use

  ↑ remainder

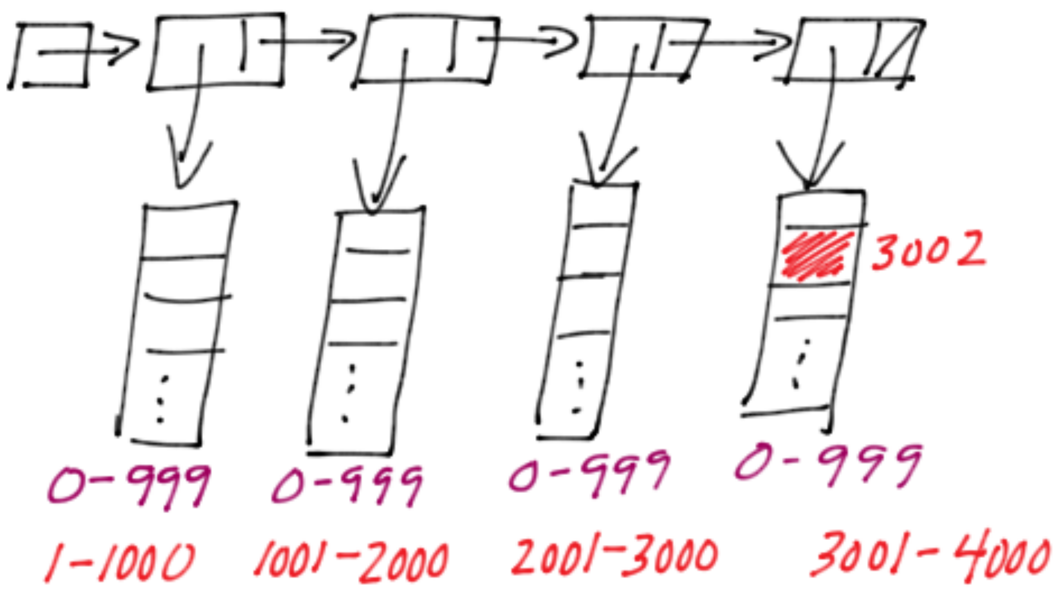
Absolute Ordered List — "Flexible" Array

1. NO ITEMS: head

2. 1st item: (position 3) head



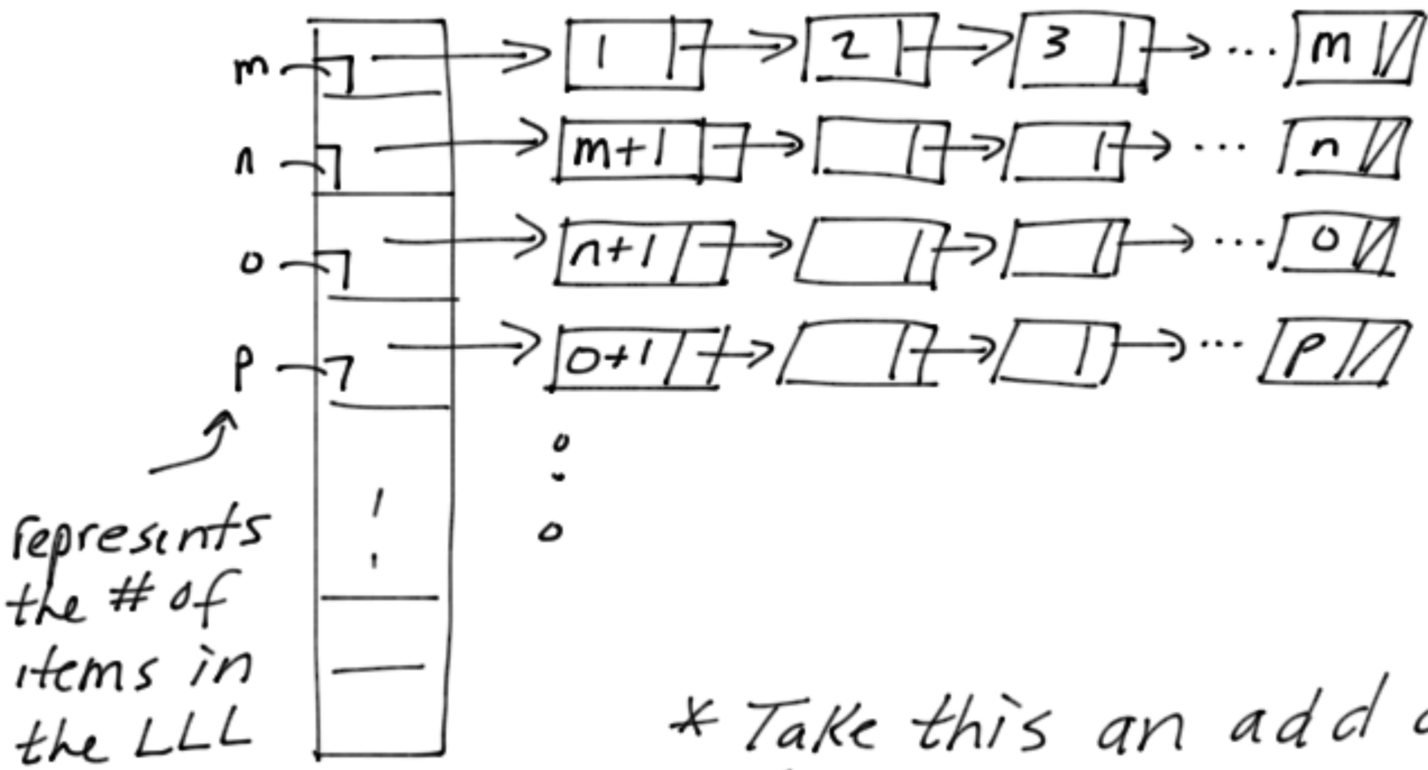
3. Add to position 3002



3002/1000 } 3 nodes to traverse 3002 % 1000 } 2nd element

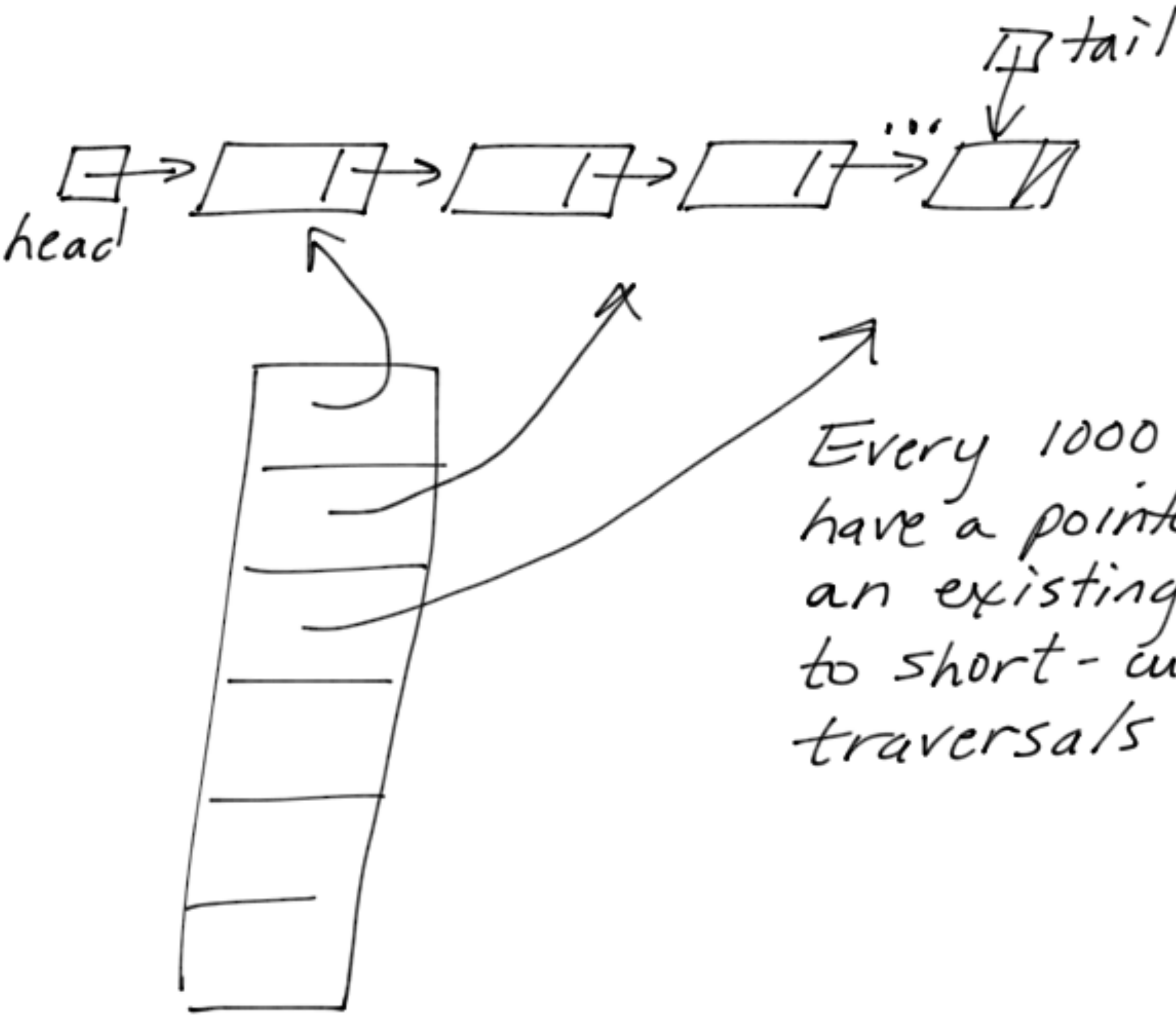
Relative Ordered List - Alternative

"Array of LLL"



* Take this an add an item to see how we can reduce the traversal overhead and still avoid shifting!!

Or....



Every 1000 or so
have a pointer into
an existing LLL
to short-cut
traversals