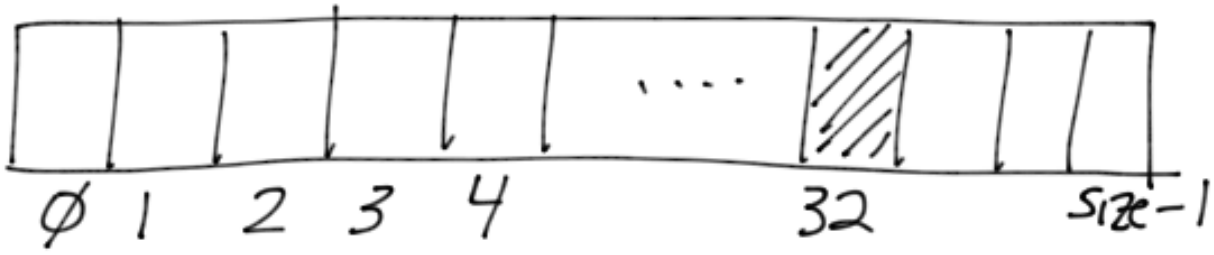
Today - Lectures 3 & 4 CS163

1) Topic #2 - Summarize ADTs
2) Topic #3 - Building Ordered Lists
   - consider efficiency trade offs between array and LLL implementations

Announcements:

* Make sure to submit the quiz each week by Mon 11:59pm. Aim to do the quizzes after watching or attending the lectures.
* Your answers & solutions will be available after the due date

Absolute Ordered Lists - Array



✓ Direct Access

array [32] = ... position 33

*(array + 32)

32 * sizeof(element)

+

temporary

X Memory - How much?

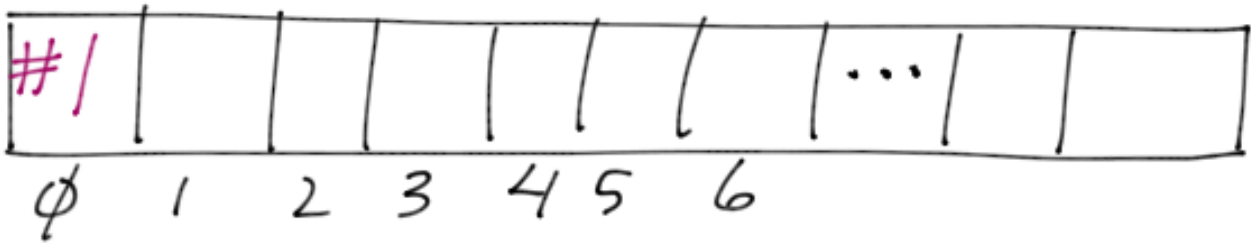
X Memory - Possible Waste

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

Relative Ordered List - Array

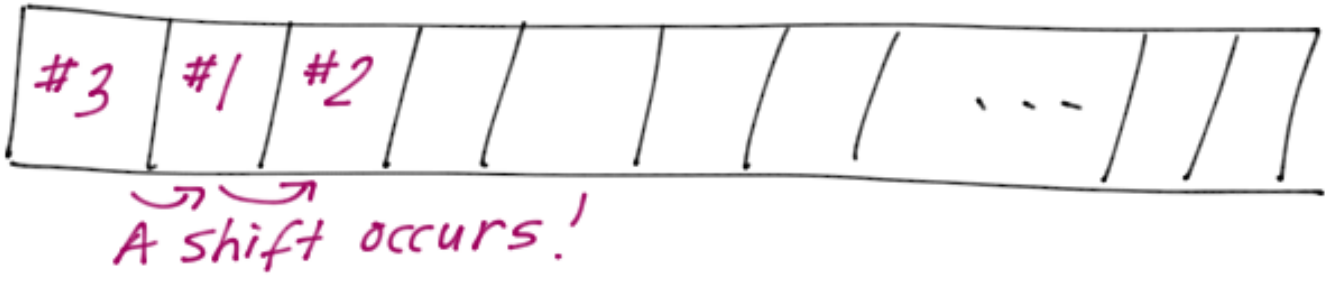
✓ Direct Access - (but not as meaningfull as for an Absolute Ordered List)

Add 1st # at any position



Add again at any position greater than 1

Add now at position 1

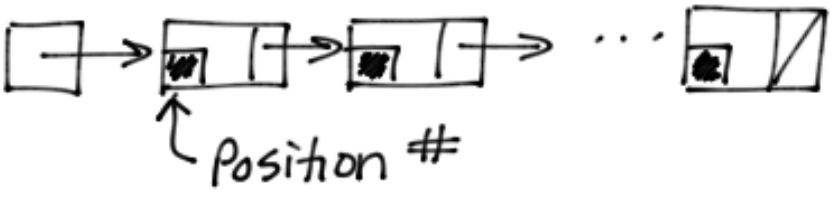


X Shifting - when data is inserted anywhere other than an "append", the data must move down (hopefully using an array of pointers to the data)

x Memory Issues (as before)

Absolute Ordered List - Linear Linked List

ADT — Data structure



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

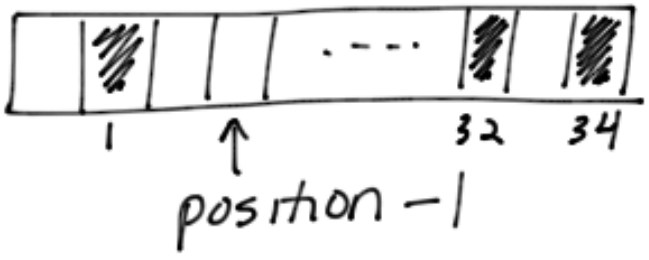
X Memory - 1 extra address per node
(10,000 nodes * sizeof an address)

X Sequential Search - No Direct Access

X Requires a "position" data member

Absolute Ordered List – Performance

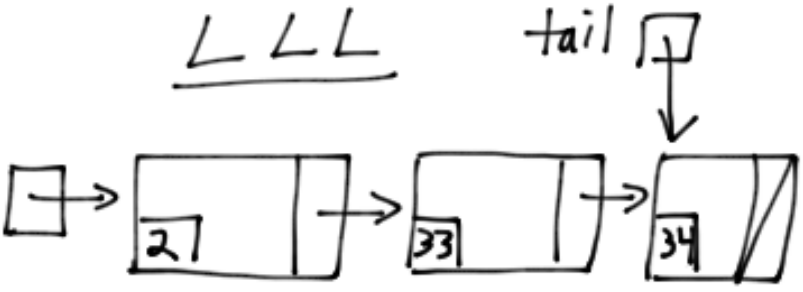
Array



array[position - 1]

* Replaces what is there
* Requires initializing all elements
* 10,000 elements + 1 ptr

LLL



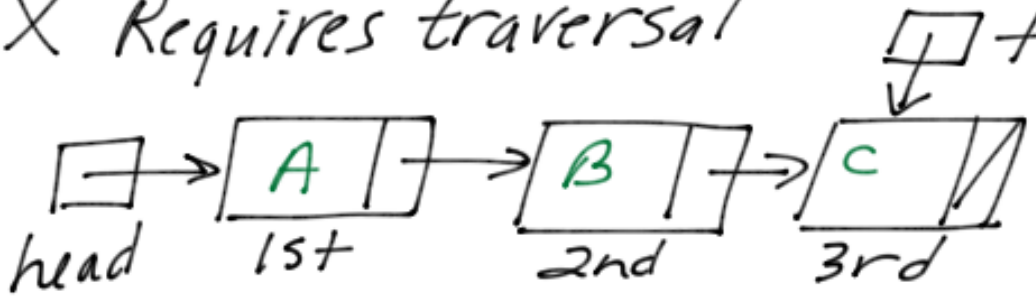

```
previous = NULL;
current = head;
while (current &&
    current->pos < toadd_pos)
{
        previous = current;
        current = current->
                  next;
}
if (previous)
{       previous->next =
            new node;
        previous = previous->next;
        previous->data = ____;
    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

Relative Ordered List — Linear Linked List

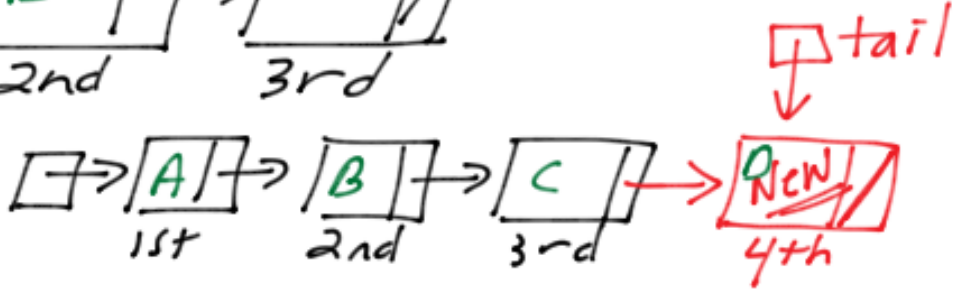
✓ Memory - No need to be aware of # items
- Not required to be contiguous
- No need to store position #'s
(in fact, position #'s change — so it is important NOT to store position numbers.

✓ Avoids Shifting!

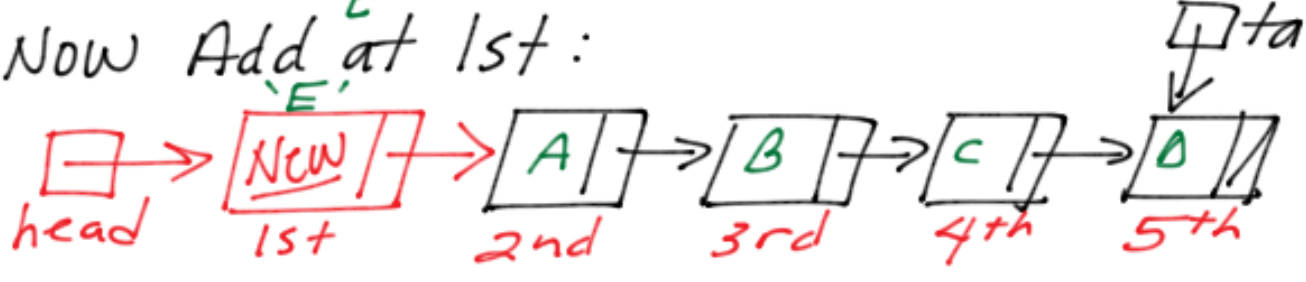
X Requires traversal



Add at 55th: "D"

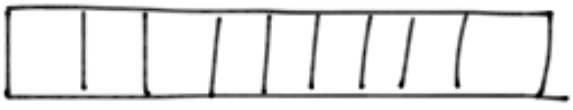


Now Add "E" at 1st:



Relative Ordered List - Performance

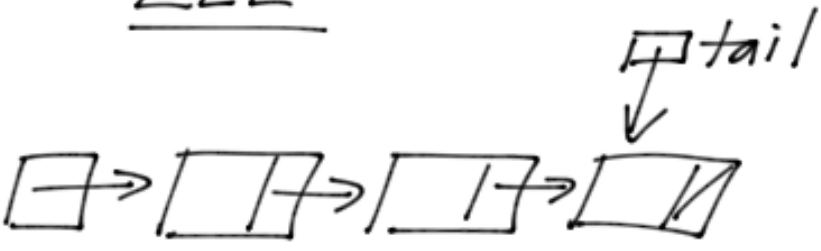
Array



- 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:

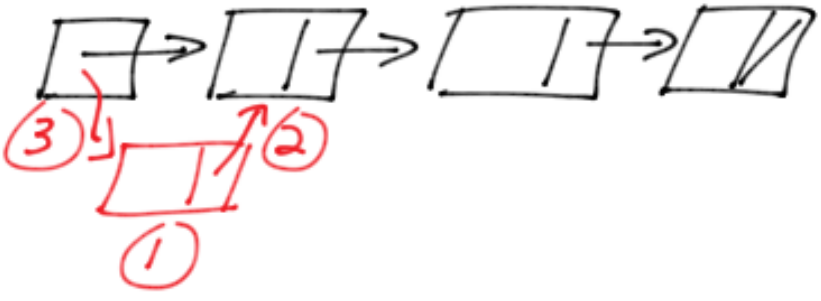
~8 op/fetch vs ~18

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

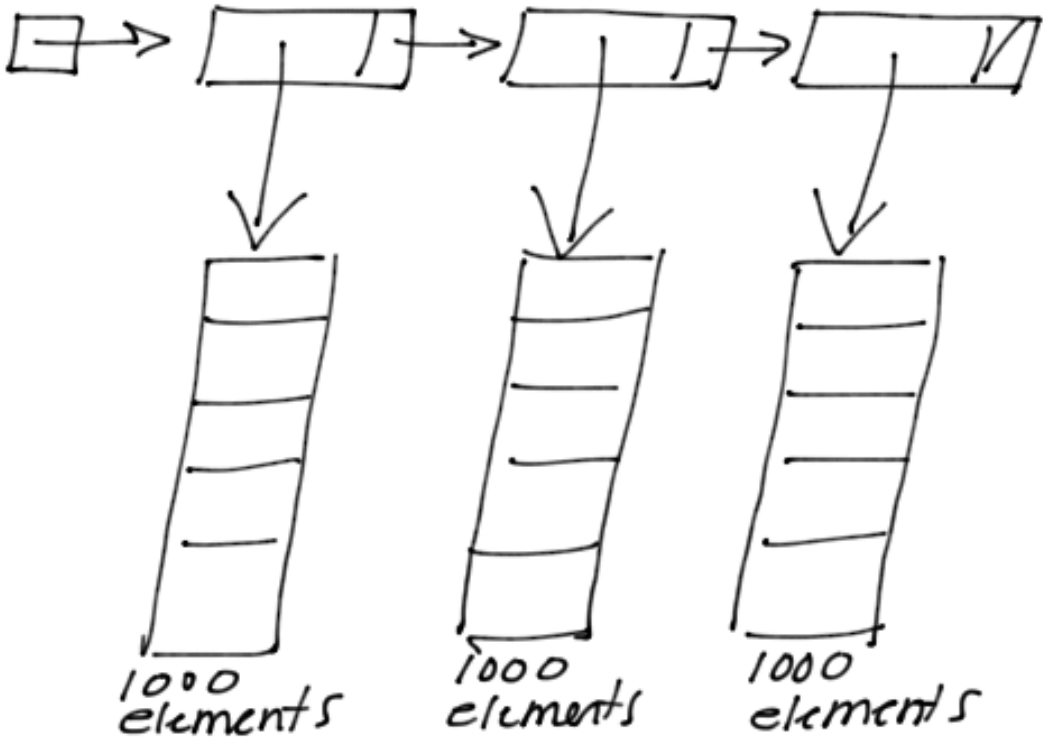

```
struct node
{
    data some_data;
    node *previous;
    node *next;
};
```

✗ Memory Overhead (2 addresses per node)

✓ Flexibility of Progressing
forward or backwards as needed

Absolute Ordered List — Alternative

"Flexible Array"



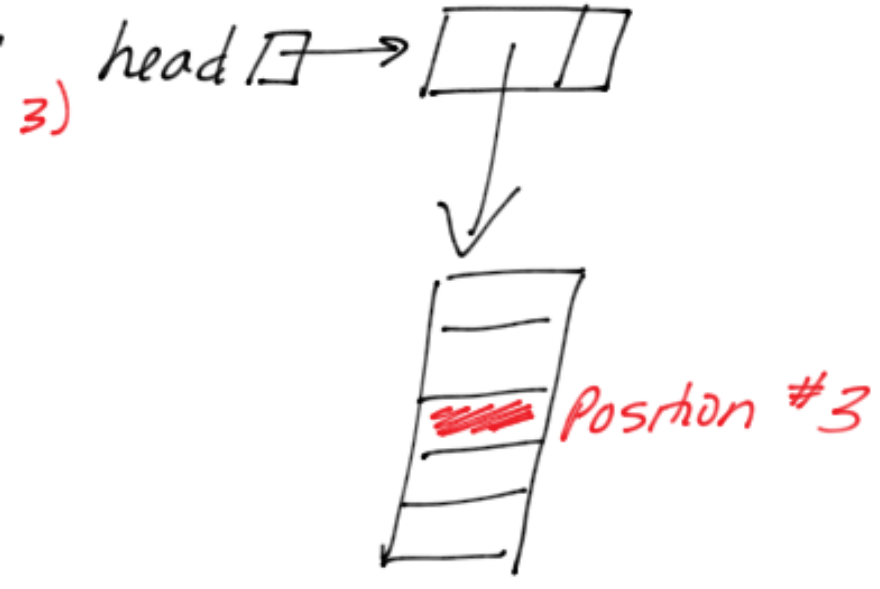

```
struct node
{
    data *array;
    node *next;
};
```

* 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

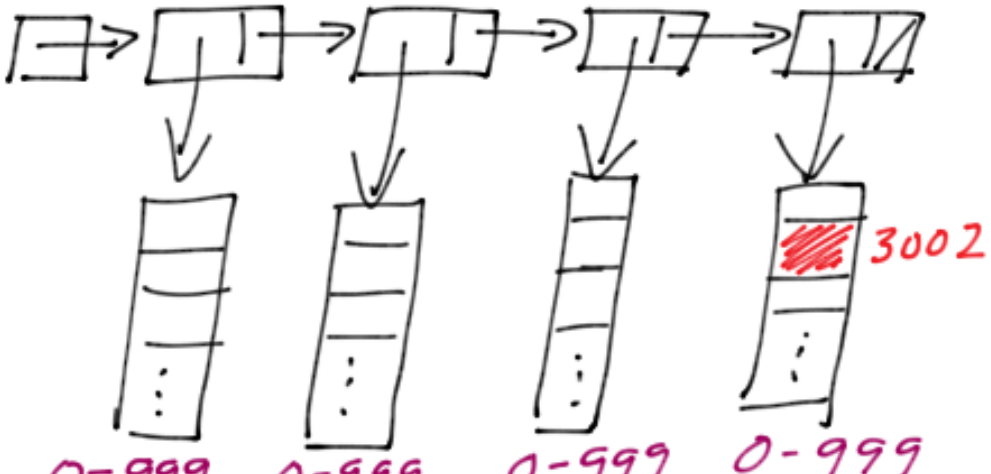
Absolute Ordered List — "Flexible" Array

① NO ITEMS: head

② 1st item: (position 3)



③ Add to position 3002



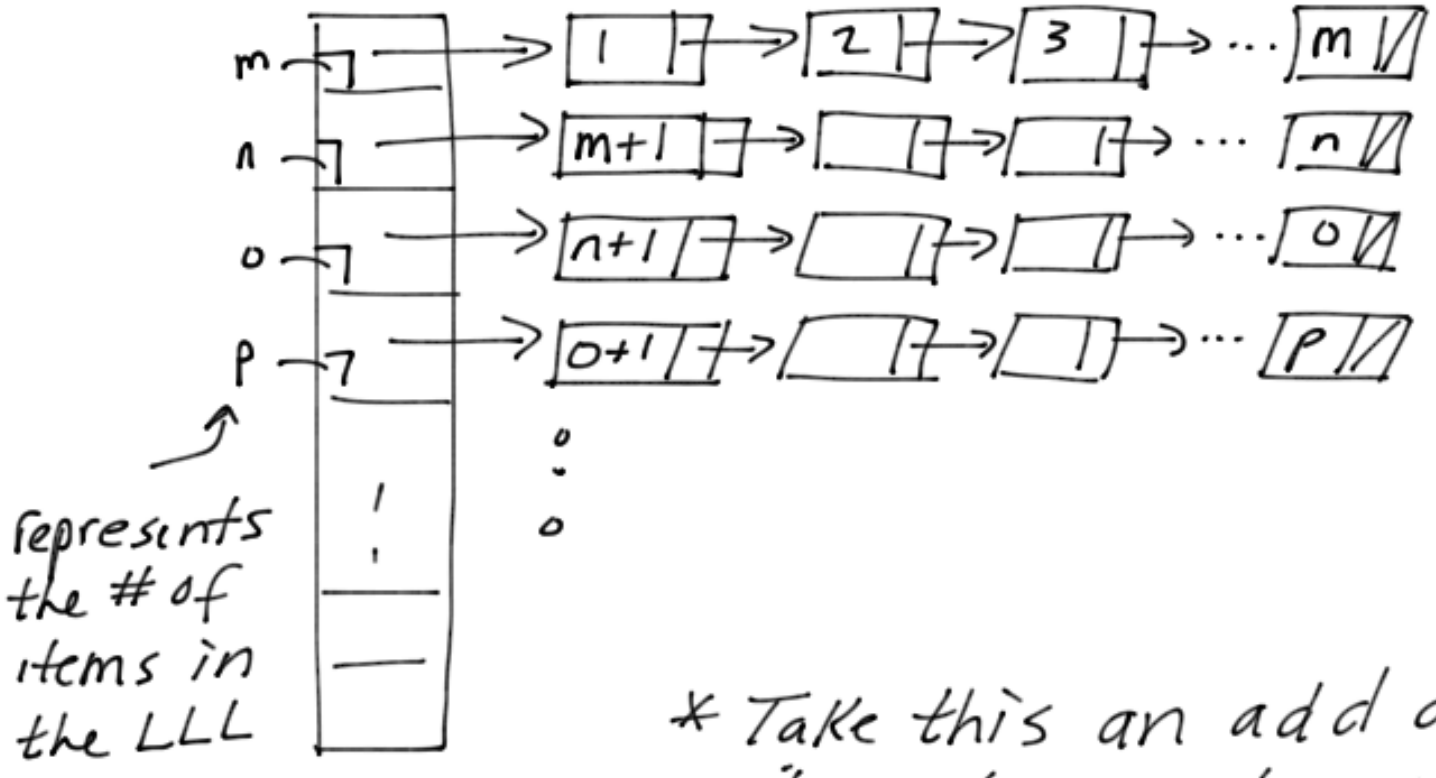
index	0-999	0-999	0-999	0-999
position:	1-1000	1001-2000	2001-3000	3001-4000

3002/1000 } 3 nodes to traverse

3002 % 1000 } 2nd element

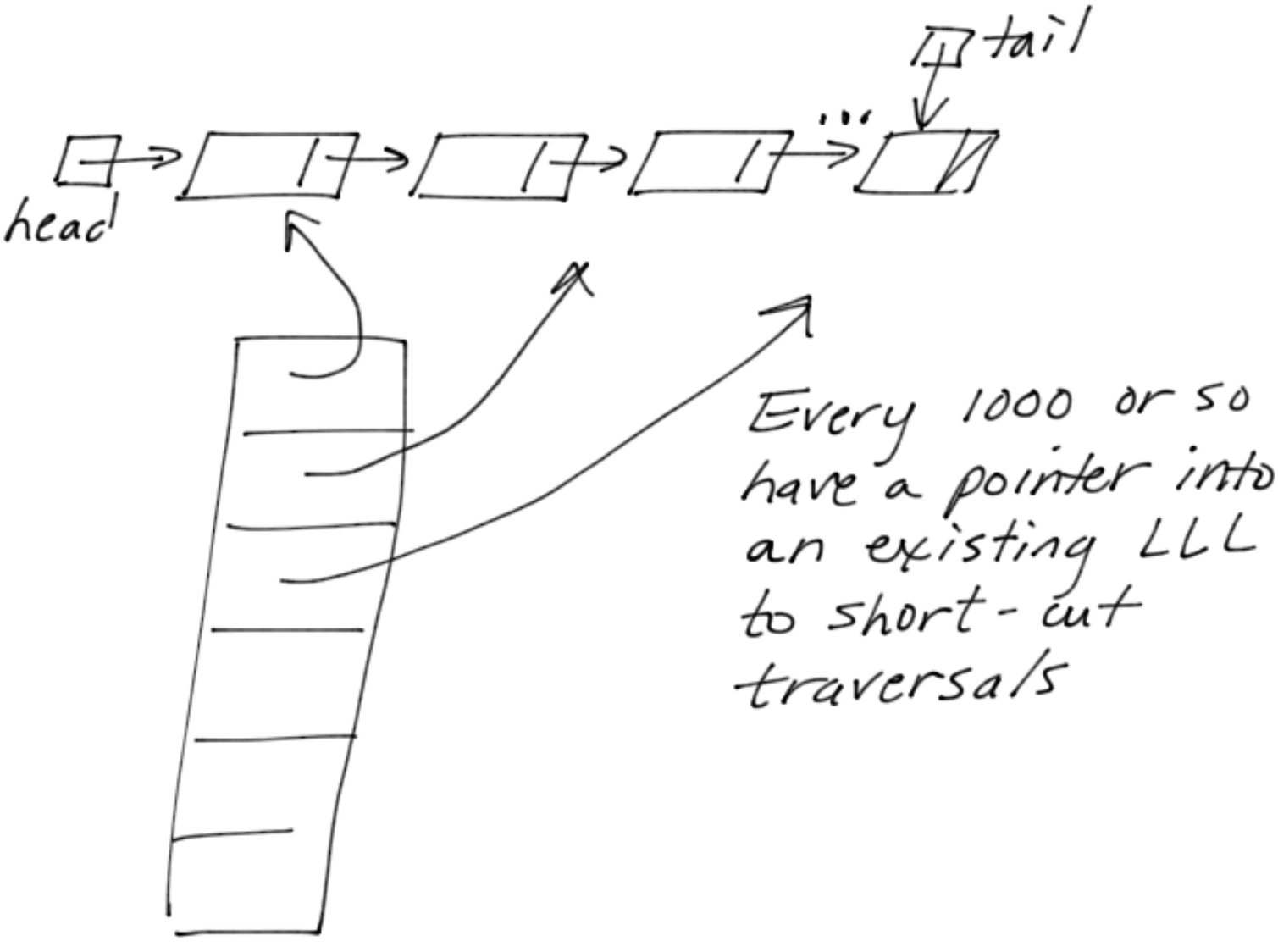
Relative Ordered List - Alternative

"Array of LLL"



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

Or....


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