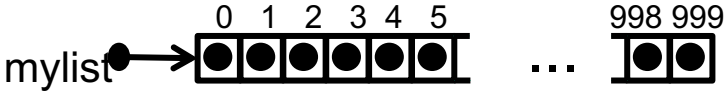
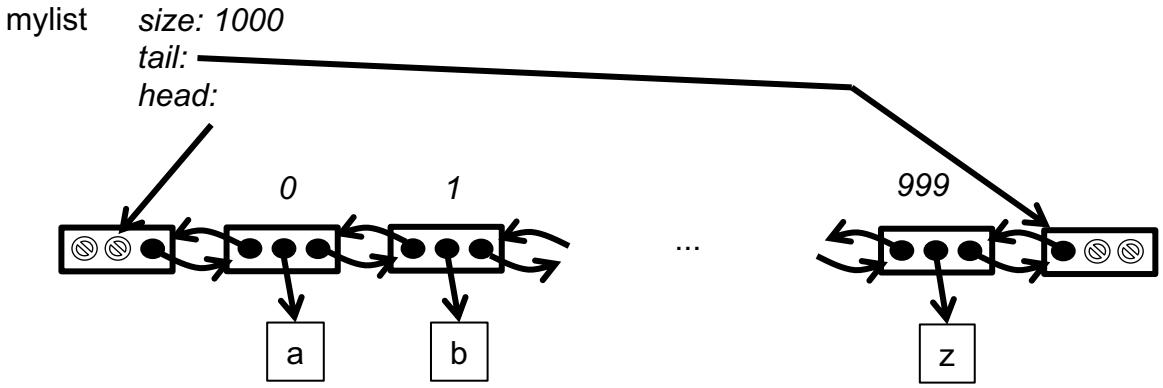
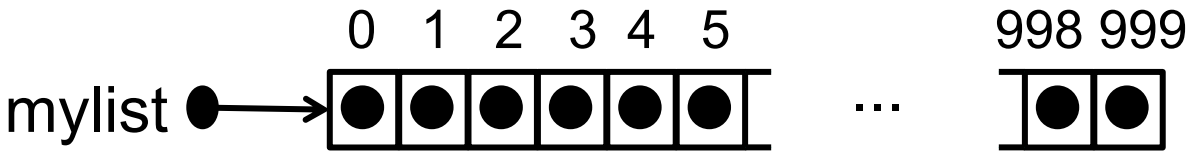
Review: List implementations

List implementations: strengths and weaknesses





Array-based lists

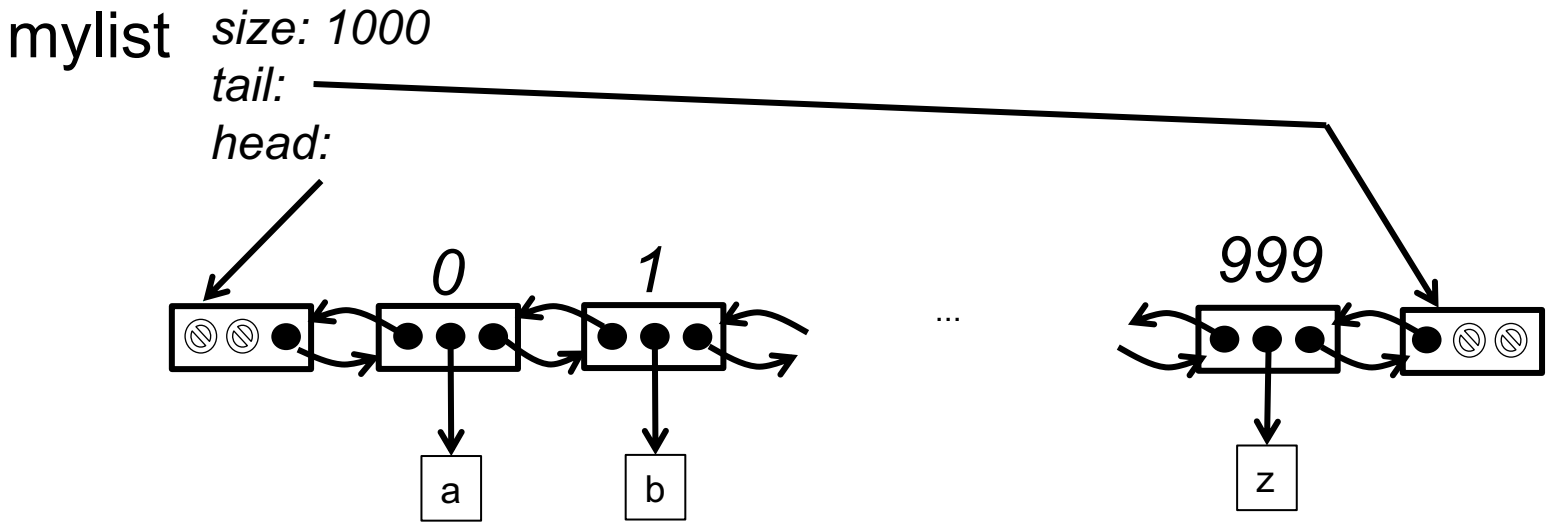


a block of memory is reserved for the references

direct access to any node using its index is O(1)

(We know the size of each reference, and we know the location of the start of the list, so simple arithmetic gives us the location of each item. In RAM we jump directly to that location.)

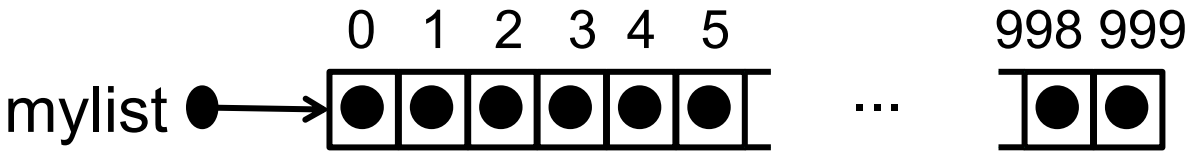
Linked lists



space is only reserved for size, head and tail
space for other elements grabbed when needed

access to individual items by following links, so O(n) worst case

Adding to array-based lists (in Python)



If enough space is reserved:

adding at end is immediate (O(1))

adding in middle requires shifting the remainder, so O(n)

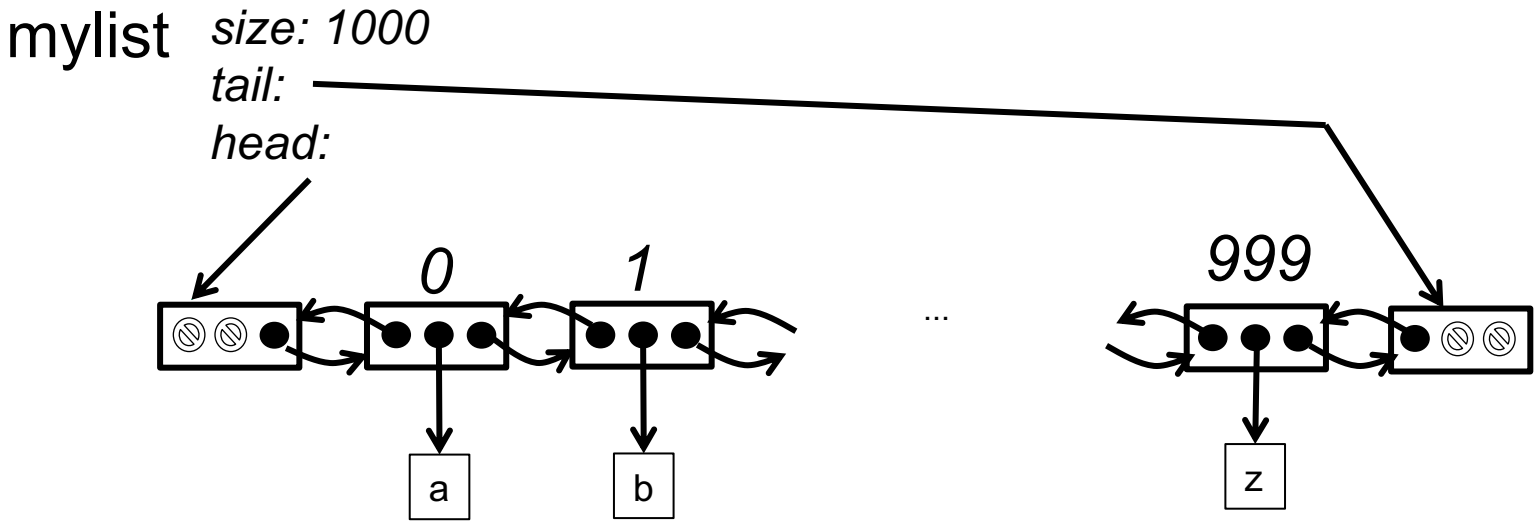
If all space occupied:

need to reserve a larger space and copy contents, so O(n)

but if managed well, we don't do this often, and so adding at end is O(1) *on average* as we build the full list

Is "O(1) on average" good enough for real-time applications?

Adding to linked lists

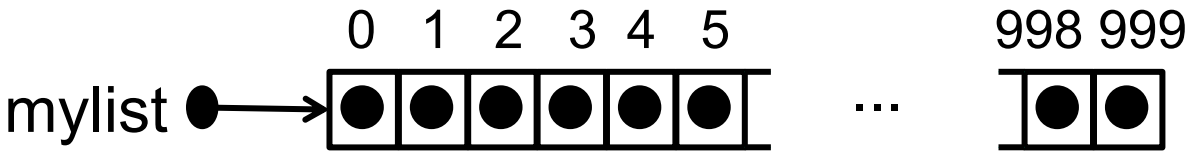


Adding an element at (either) end is O(1)

Note: guaranteed upper bound on complexity, not an average

Adding an element in the middle is O(1), if we have a reference to the position; O(n) otherwise (since we have to navigate to the correct position)

Deleting from array-based lists (in Python)

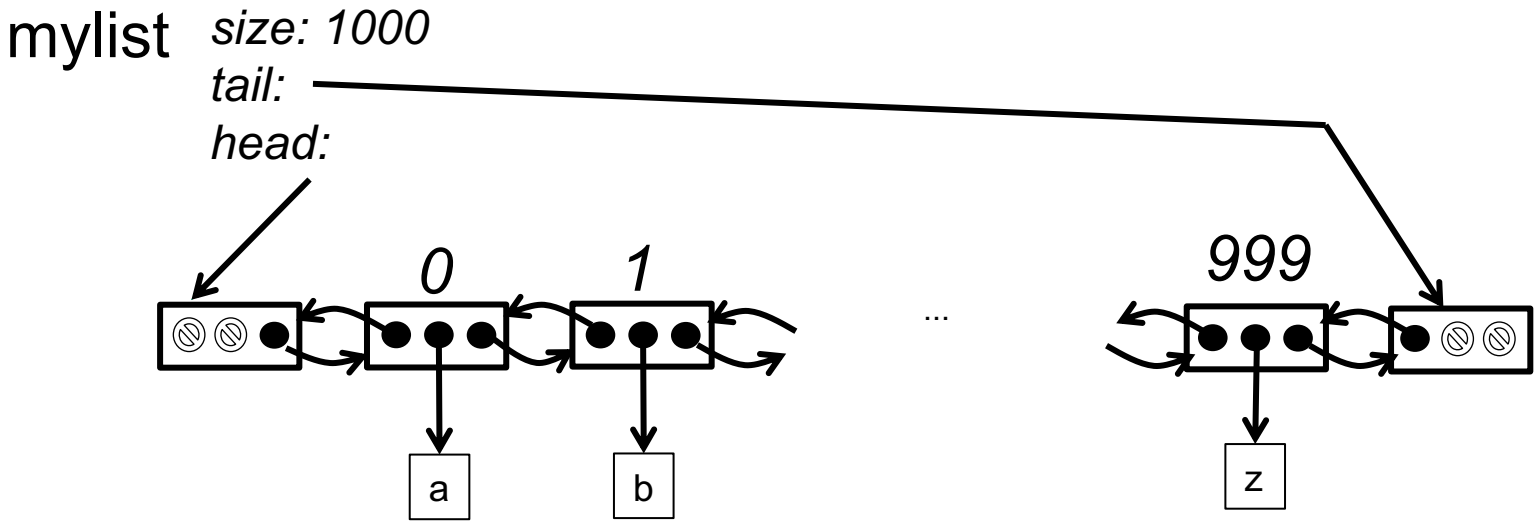


Deleting from the end is O(1) *on average*

Deleting from the middle is O(n)

Is "O(1) on average" good enough for real-time applications?

Deleting from linked lists



Deleting from start or end is O(1)

Deleting from the middle is O(1), if we have a reference to the position; O(n) otherwise (since we have to navigate the list to find the item or the position)

Note

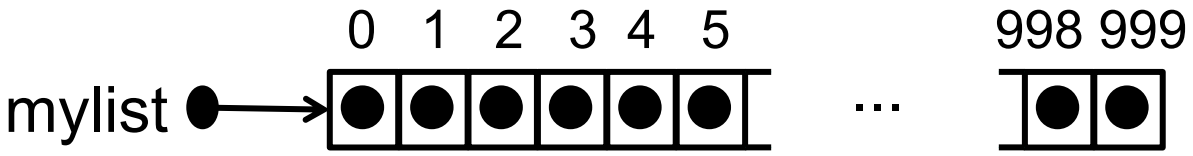
When operations on the two types of list have the same complexity, array-based lists are usually faster

- reserving space in a linked list for each new internal node takes time
- updating linked list nodes takes 2 or 4 assignments

Array-based lists may use less space (despite the extra space needed for growing the lists)

- doubling reserved space for an (array-based) list takes twice as much space (*so at most 2*length*(space per reference)*)
- maintaining the (linked) list nodes requires 2 or 3 references per node (*so at least 2*length*(space per reference)*)

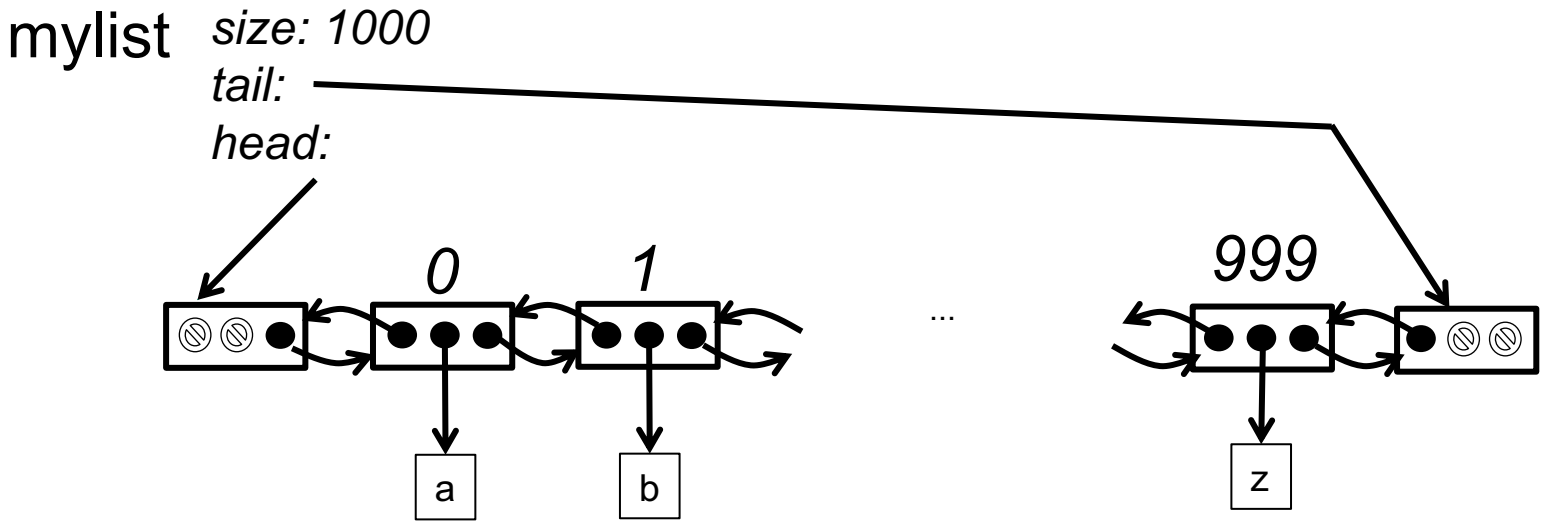
Sorting an array-based list



You should have already seen some sorting algorithms in CS1113 and CS1117

Further investigation of sorting algorithms will be covered in CS2516

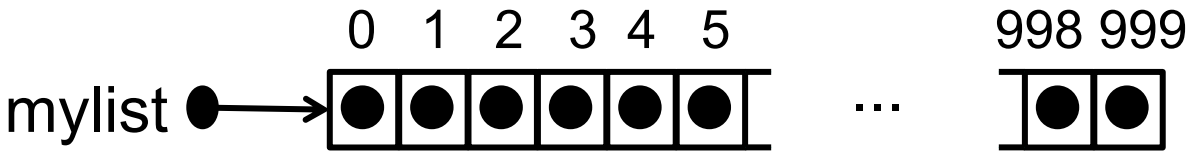
Sorting a linked list



Sorting a linked list is different to sorting an array-based list
- linked lists do not have direct access to arbitrary positions

Further investigation of sorting algorithms will be covered later in CS2516

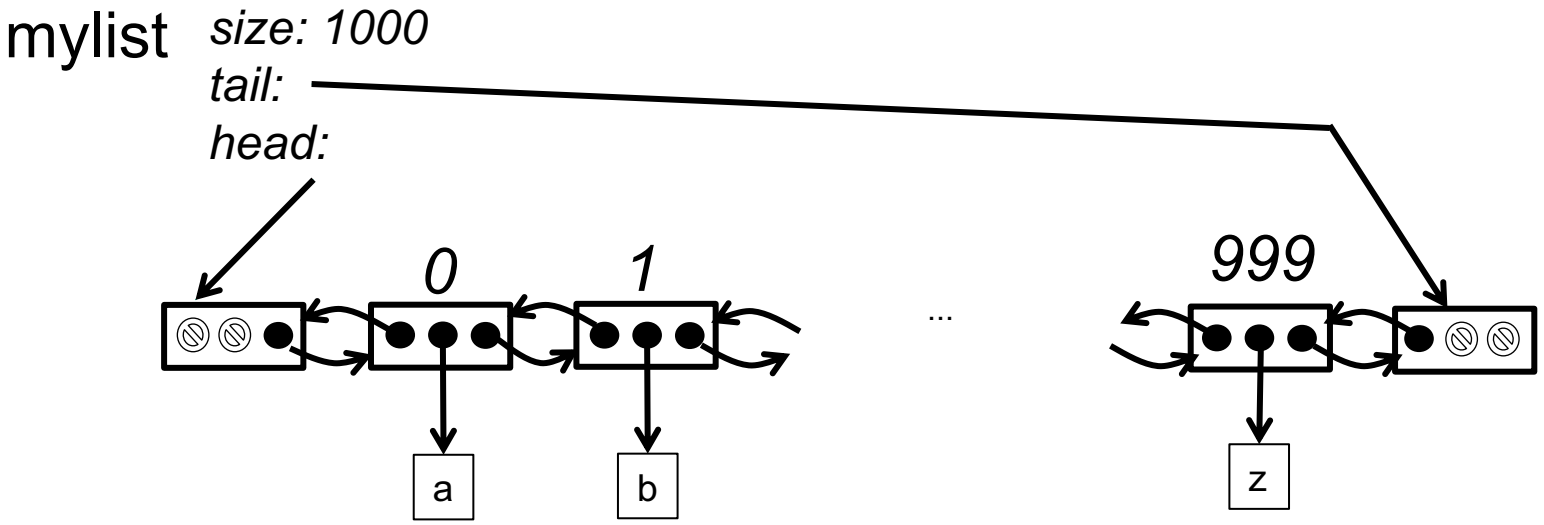
Searching an array-based list



(unordered)
Linear search: O(n) worst case

(ordered)
Binary search: O(log n)

Searching a linked list



no direct access to any node except for head and tail

must step through the list to access intermediate nodes

Linear search:
O(n) worst case

Summary

	Array-based	Linked list
Arbitrary access	O(1)	O(n)
Add at end	O(n) worst case O(1) on average	O(1)
Add in middle	O(n)	O(1) if given reference O(n) if given position
Delete from end	O(n) worst case O(1) on average	O(1) if given reference O(n) if given position
Delete from middle	O(n)	O(1) if given reference O(n) if given position
sorting	(see semester 2)	(see semester 2)
searching	Unordered: O(n) Ordered: O(log n)	Unordered: O(n) Ordered: O(n)

In practice, when the complexity class is the same for an operation, array-based lists are almost always faster – DLLs need more memory for all the *next* and *prev* references, and require the *next* and *prev* references to be updated after every change. Python's implementation of array-based lists is efficient.

Let's look at searching again …

	Array-based	Linked list
Arbitrary access	O(1)	O(n)
Add at end	O(n) worst case O(1) on average	O(1)
Add in middle	O(n)	O(1) if given reference O(n) if given position
Delete from end	O(n) worst case O(1) on average	O(1) if given reference O(n) if given position
Delete from middle	O(n)	O(1) if given reference O(n) if given position
sorting	(see semester 2)	(see semester 2)
searching	Unordered: O(n) Ordered: O(log n)	Unordered: O(n) Ordered: O(n)

Bad!

Good!

Good!

Bad!

But: how do we build and maintain an ordered list?

We have to add / delete at specific positions in the middle …

Can we do something else with ‘links’?

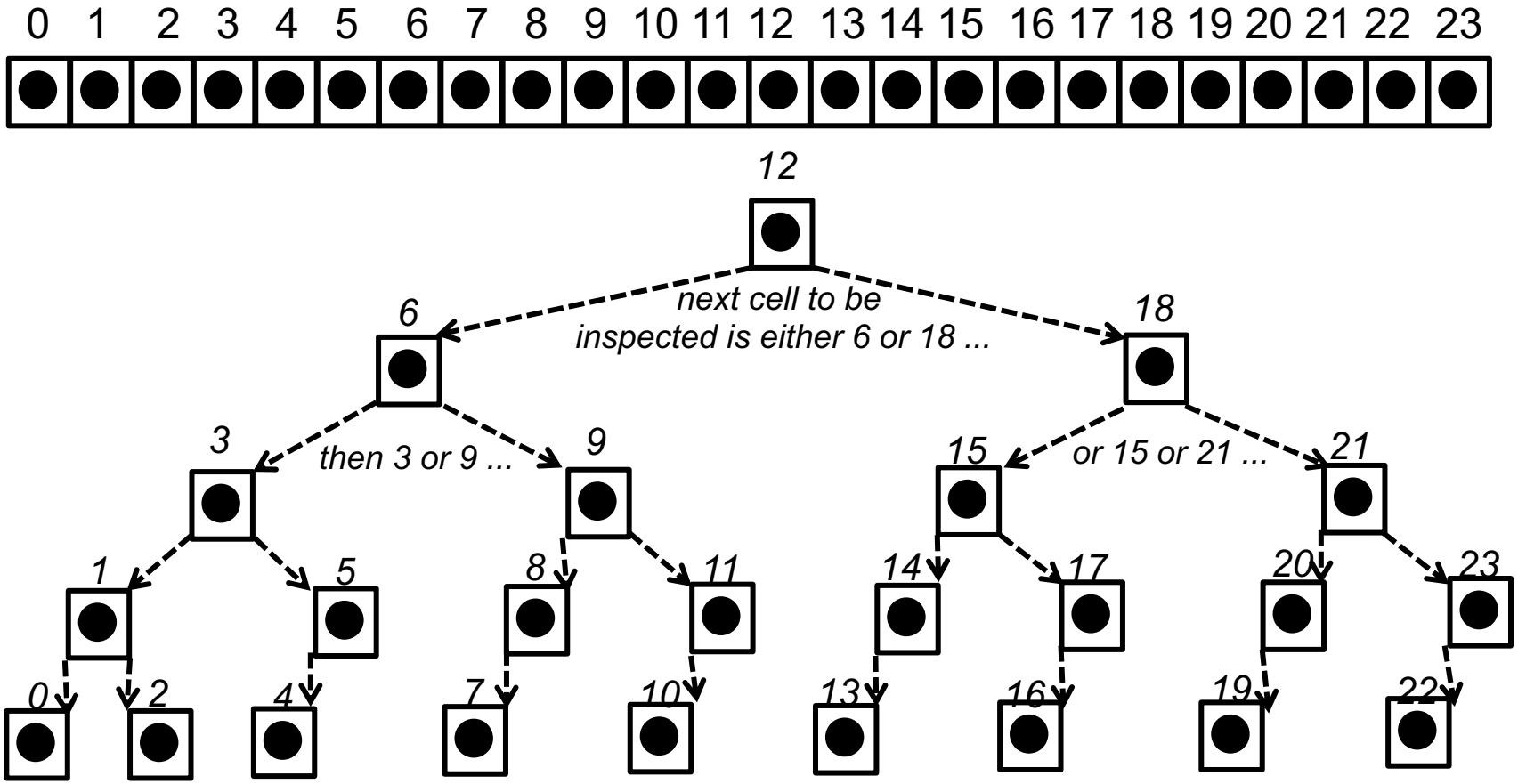
For now, just focus on searching.

Can we reorganize the linked list so that it allows something like binary search?

Perhaps find the sequence of cells that would be visited in the array-based list during a search, and use that to re-structure our lists?

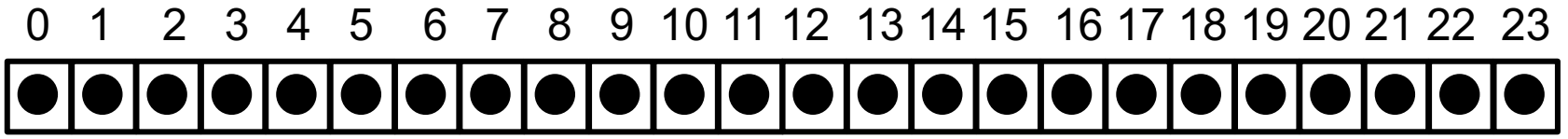
Sequence of cell lookups in binary search

A sorted list:

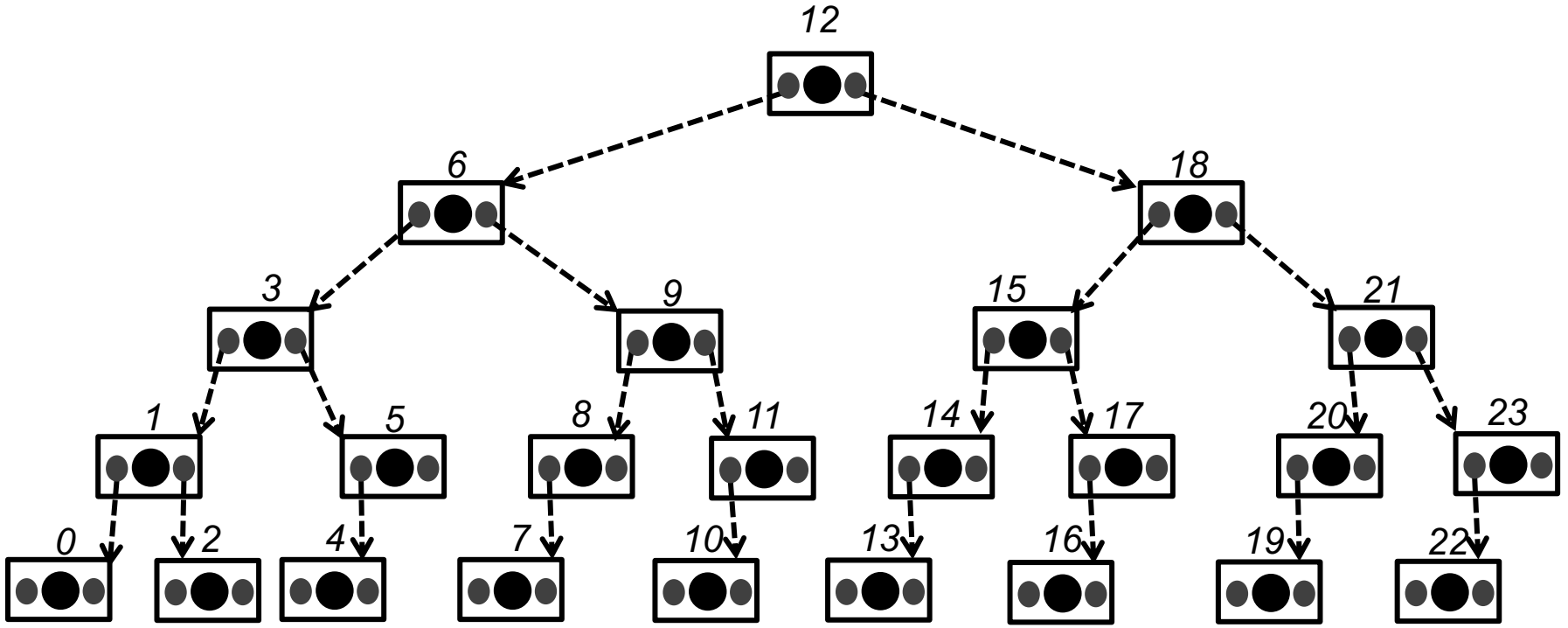


Linked structures for efficient searching?

A sorted list:



can be searched in no more than 5 steps ...



Next Lecture

Trees