Topic 2: Arrays and Lists

Eamonn Bell

eamonn.bell@durham.ac.uk

Data Structures

- For the next three weeks, we'll study different ways to store and organize data.
- We learn about different data structures, because each has its advantages and disadvantages.

Data Structures

- For the next three weeks, we'll study different ways to store and organize data.
- We learn about different data structures, because each has its advantages and disadvantages.
- We can make this concrete with examples in Python...

Data Structures

- For the next three weeks, we'll study different ways to store and organize data.
- We learn about different data structures, because each has its advantages and disadvantages.
- We can make this concrete with examples in Python...
- ...but in general we want to develop a body of knowledge about algorithms and data structures that is sufficiently abstract to serve us well in a variety of languages and problem settings.

Our first "data structure": arrays

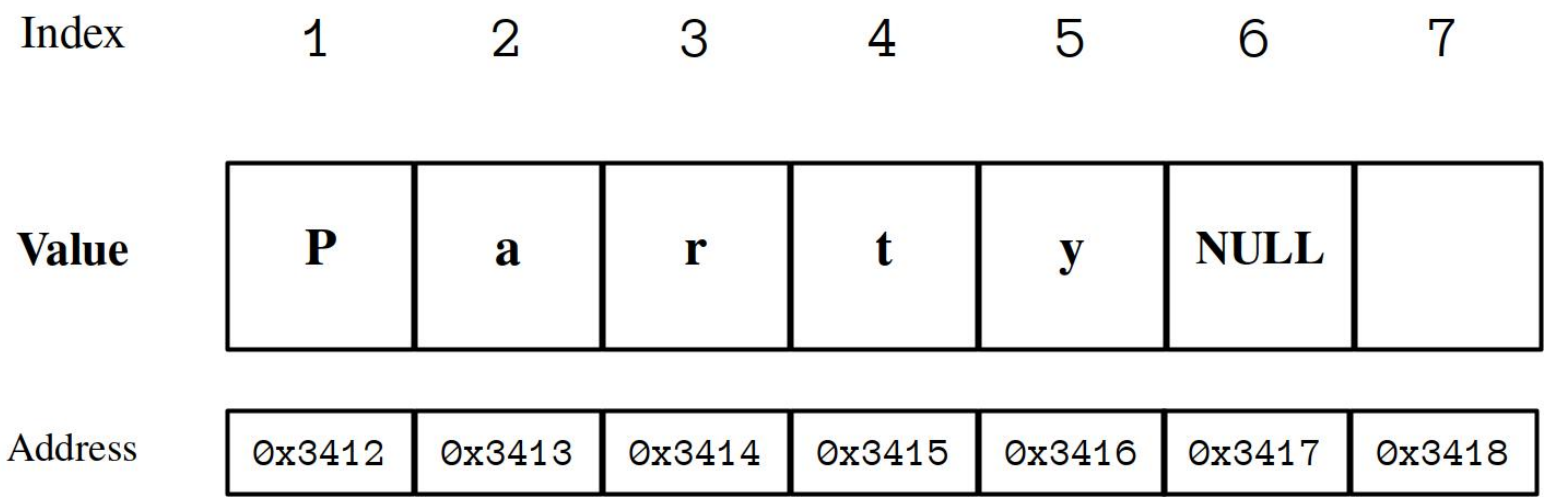
A sequence of elements $a_1, a_2, \ldots, a_n$ is called an array and usually denoted by something like `A[1], A[2], ···, A[n]` or `A[1...n]`

1. They're in consecutive memory cells, but we (usually) don't care where exactly.
2. All array elements are of the same type, e.g., integer. In pseudocode (and some languages) we can declare an array of integers as `integer A[1...n]`

The fact that arrays are contiguous (1) and homogeneous (2) makes it easy to do certain operations (and difficult to do others).

Arrays

- Here is an array of `chars` (each value represents a single character).



Arrays

- How do we find the ith element? How long does it take?

Arrays

- How do we find the ith element? How long does it take?
- What if we want to erase an element?

Arrays

- How do we find the ith element? How long does it take?
- What if we want to erase an element?
- The size of an array is fixed when we declare it. How big should we make it?

Linked Lists

- A list is made up of nodes. Each node stores an element (a piece of data) plus a pointer or "link" to another node.
- The first node is called the head.
- The last node, called the tail points to null.
- The nodes may be scattered all over the memory.

Implementing a list

Assume that for list `L` we have pointers to the first as well as the last node of list:

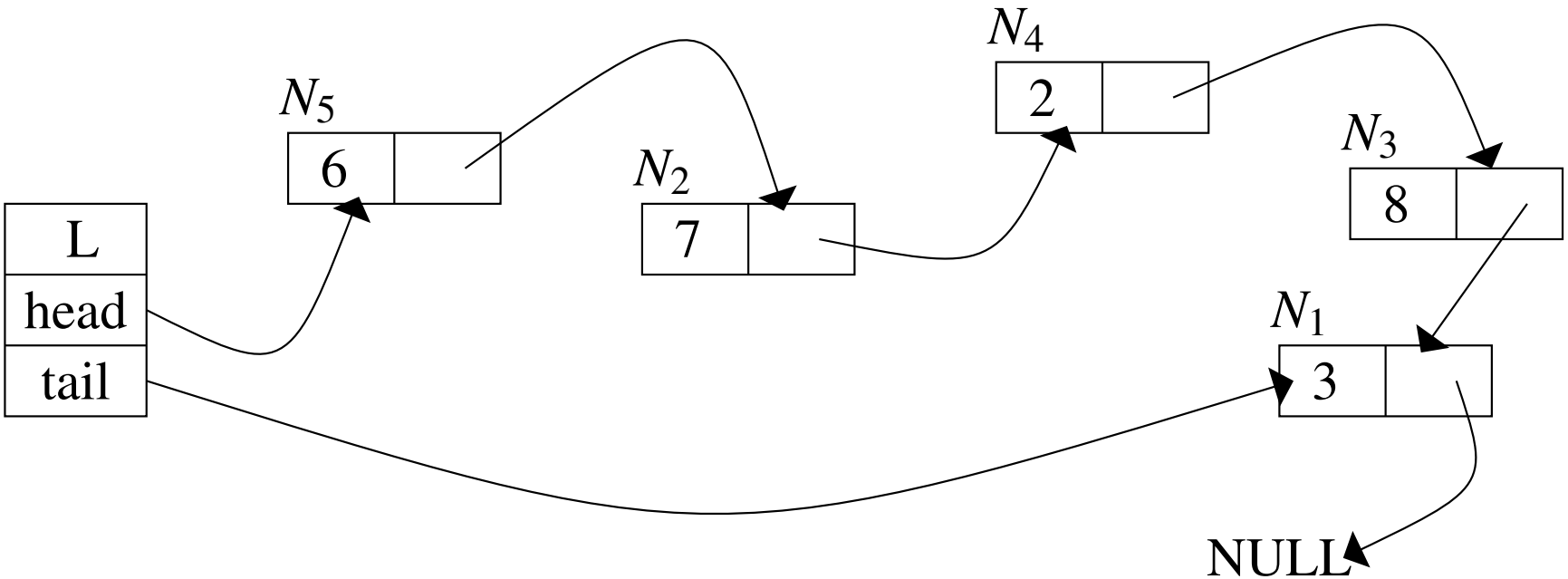
- `L.head`
- `L.tail`
- (and possibly also we have `L.size`)

May refer to node `N` using:

- `N.data`, the element
- `N.next`, the link, the next node in the list (may be `NULL`)

`NULL` means "there's nothing there", i.e., last element has no successor.

A linked list



We would like `L.find(i)` to find the *i*th piece of data in a list? How can we do this? How long will it take?

We would like `L.find(i)` to find the *i*th piece of data in a list? How can we do this? How long will it take?

Input: list L, positive integer *i*
Output: *i*th piece of data in L

We would like `L.find(i)` to find the *i*th piece of data in a list? How can we do this? How long will it take?

Input: list L, positive integer *i*
Output: *i*th piece of data in L

N = L.head

We would like `L.find(i)` to find the *i*th piece of data in a list? How can we do this? How long will it take?

Input: list L, positive integer *i*
Output: *i*th piece of data in L

```
N = L.head
if i = 1 then


        return N.data
end if
```

We would like `L.find(i)` to find the *i*th piece of data in a list? How can we do this? How long will it take?

Input: list L, positive integer *i*
Output: *i*th piece of data in L

```
N = L.head
if i = 1 then


        return  N.data
end if
for j = 2 to i do
        N = N.next


end for
```

We would like `L.find(i)` to find the *i*th piece of data in a list? How can we do this? How long will it take?

Input: list L, positive integer *i*
Output: *i*th piece of data in L

```
N = L.head
if i = 1 then

        return N.data
end if
for j = 2 to i do
        N = N.next


end for
return N.data
```

We would like `L.find(i)` to find the *i*th piece of data in a list? How can we do this? How long will it take?

Input: list L, positive integer *i*
Output: *i*th piece of data in L

```
N = L.head
if i = 1 then
        return N.data
end if
for j = 2 to i do
        N = N.next
        if N = NULL then
                return out of range
        end if
end for
return N.data
```

We would like `L.find(i)` to find the *i*th piece of data in a list? How can we do this? How long will it take?

Input: list L, positive integer *i*
Output: *i*th piece of data in L

```
N = L.head
if i = 1 then
        if N = NULL then
                return  out of range
        end if
        return  N.data
end if
for j = 2 to i do
        N = N.next
        if N = NULL then
                return  out of range
        end if
end for
return  N.data
```

We would like `L.find(i)` to find the *i*th piece of data in a list? How can we do this? How long will it take?

```
Input:  list L, positive integer i
Output:  ith piece of data in L
   if i > L.size then
         return  out of range
   end if
   N = L.head
   if i = 1 then




         return  N.data
   end if
   for j = 2 to i do
         N = N.next




   end for
   return  N.data
```

Welcome to Algorithms and Data Structures

The lecture will begin at 5 past the hour. While you wait consider this pseudocode and work out what it does (this is the final slide below).

```
Input:  n numbers in array A[0], . . ., A[n-1]
Output:  ?
   for i = 0 to n-2 do
         e = A[i]
         p = i
         for j = i+1 to n-1 do
               if A[j] < e then
                     e = A[j]
                     p = j
               end if
         end for
         swap A[i] and A[p]
   end for
   return  A
```

Try the questions at `https://pollev.com/eamonn` or `eamonn` on the PollEverywhere app

How can we alter the list? Anything that inserts or removes must fix all references to (predecessors and) successors.

How can we alter the list? Anything that inserts or removes must fix all references to (predecessors and) successors.

Deletion of the head

Input: list L

Output: L with head deleted

How can we alter the list? Anything that inserts or removes must fix all references to (predecessors and) successors.

Deletion of the head

Input: list L

Output: L with head deleted

L.head = L.head.next

How can we alter the list? Anything that inserts or removes must fix all references to (predecessors and) successors.

Deletion of the head

Input: list L

Output: L with head deleted

```
L.head = L.head.next
L.size = L.size - 1
```

How can we alter the list? Anything that inserts or removes must fix all references to (predecessors and) successors.

Deletion of the head

Input: list L
Output: L with head deleted

```
if L.head = NULL then

end if

L.head = L.head.next
L.size = L.size - 1
```

How can we alter the list? Anything that inserts or removes must fix all references to (predecessors and) successors.

Deletion of the head

Input: list L
Output: L with head deleted

```
    if L.head = NULL then
        return  no head to delete
    end if




    L.head = L.head.next
    L.size = L.size - 1
```

How can we alter the list? Anything that inserts or removes must fix all references to (predecessors and) successors.

Deletion of the head

Input: list L
Output: L with head deleted

```
if L.head = NULL then
        return no head to delete
end if
if L.head = L.tail then




end if
L.head = L.head.next
L.size = L.size - 1
```

How can we alter the list? Anything that inserts or removes must fix all references to (predecessors and) successors.

Deletion of the head

Input: list L
Output: L with head deleted

```
if L.head = NULL then
        return  no head to delete
end if
if L.head = L.tail then
        L.head = NULL




end if
L.head = L.head.next
L.size = L.size - 1
```

How can we alter the list? Anything that inserts or removes must fix all references to (predecessors and) successors.

Deletion of the head

Input: list L
Output: L with head deleted

```
if L.head = NULL then
        return  no head to delete
end if
if L.head = L.tail then
        L.head = NULL
        L.tail = NULL



end if
L.head = L.head.next
L.size = L.size - 1
```

How can we alter the list? Anything that inserts or removes must fix all references to (predecessors and) successors.

Deletion of the head

```
Input:  list L
Output:  L with head deleted
    if L.head = NULL then
            return  no head to delete
    end if
    if L.head = L.tail then
            L.head = NULL
            L.tail = NULL
            L.size = L.size - 1

    end if
    L.head = L.head.next
    L.size = L.size - 1
```

How can we alter the list? Anything that inserts or removes must fix all references to (predecessors and) successors.

Deletion of the head

```
Input:  list L
Output:  L with head deleted
   if L.head = NULL then
         return  no head to delete
   end if
   if L.head = L.tail then
         L.head = NULL
         L.tail = NULL
         L.size = L.size - 1
         return
   end if
   L.head = L.head.next
   L.size = L.size - 1
```

Insertion of v after N

Input: list L, data v, node N

Output: L with v inserted after N

Insertion of v after N

Input: list L, data v, node N

Output: L with v inserted after N

node M

Insertion of v after N

Input: list L, data v, node N

Output: L with v inserted after N

node M

M.data = v

Insertion of v after N

Input: list L, data v, node N

Output: L with v inserted after N

```
node M
M.data = v
N.next = M
M.next = N.next
```

Insertion of v after N

Input: list L, data v, node N

Output: L with v inserted after N

```
node M
M.data = v
M.next = N.next
N.next = M
```

Insertion of v after N

Input: list L, data v, node N

Output: L with v inserted after N

```
node M
M.data = v
M.next = N.next
N.next = M
if L.tail = N then


end if
```

Insertion of v after N

Input: list L, data v, node N

Output: L with v inserted after N

```
node M
M.data = v
M.next = N.next
N.next = M
if L.tail = N then
        L.tail = M
end if
```

Arrays versus lists

	Array	Linked List
Data Access		
Insertion, Deletion		

Arrays versus lists

	Array	Linked List
Data Access	fast	
Insertion, Deletion		

Arrays versus lists

	Array	Linked List
Data Access	fast	slow
Insertion, Deletion		

Arrays versus lists

	Array	Linked List
Data Access	fast	slow
Insertion, Deletion	slow	

Arrays versus lists

	Array	Linked List
Data Access	fast	slow
Insertion, Deletion	slow	fast

Doubly Linked Lists

- A node in a doubly linked list stores two references:
  - a next link, which points to the next node in the list
  - a prev link, which points to the previous node in the list
- To simplify, we add two dummy or sentinel nodes at the ends of the doubly linked list:
  - the header has a valid next reference but a null prev reference
  - the trailer has a valid prev reference but a null next reference

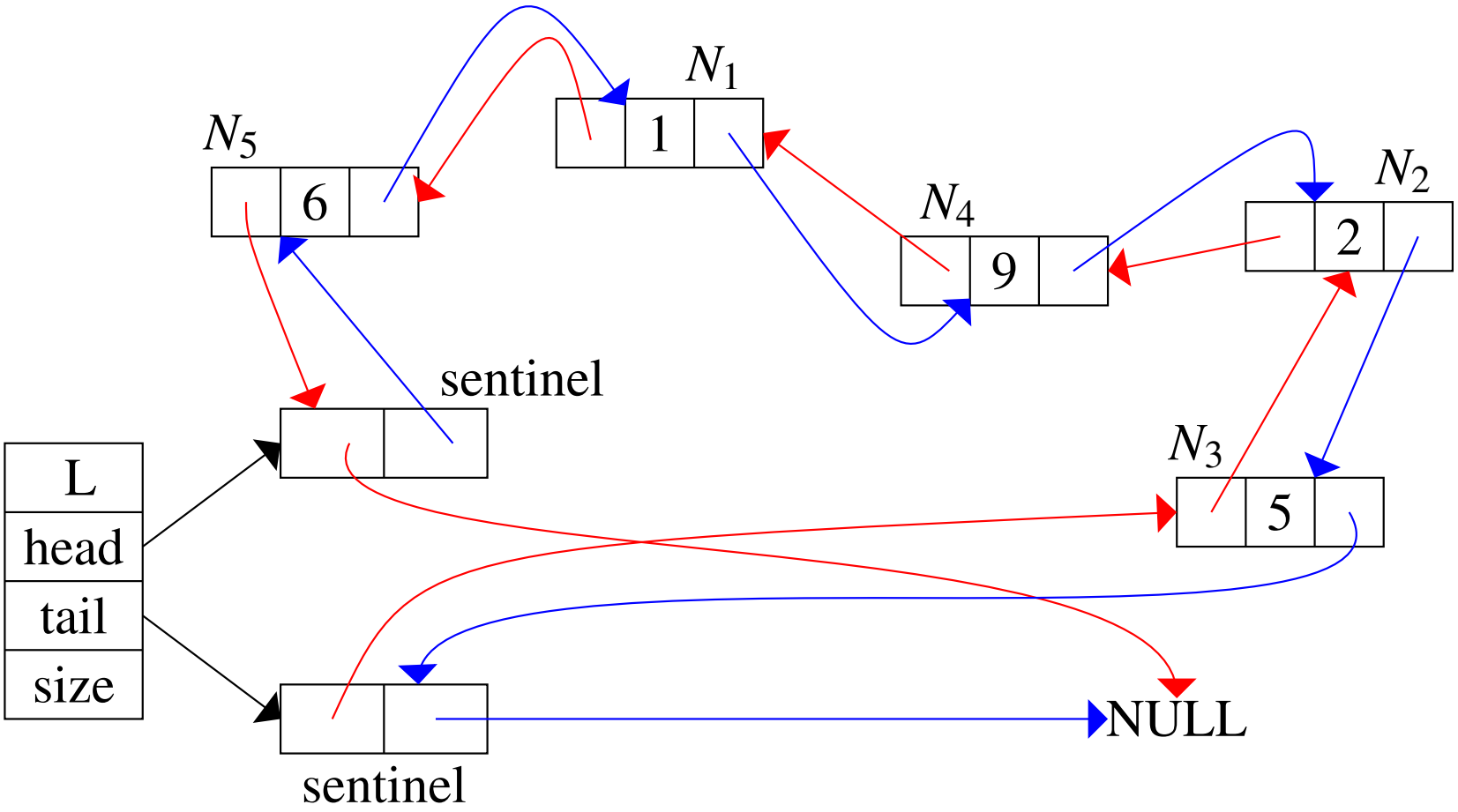
Doubly Linked Lists

- A node in a doubly linked list stores two references:
  - a next link, which points to the next node in the list
  - a prev link, which points to the previous node in the list
- To simplify, we add two dummy or sentinel nodes at the ends of the doubly linked list:
  - the header has a valid next reference but a null prev reference
  - the trailer has a valid prev reference but a null next reference

A doubly linked list needs to store references to the two sentinel nodes and a size counter keeping track of the number of nodes in the list (not counting sentinels).

An empty list would have the two sentinel nodes pointing to each other.

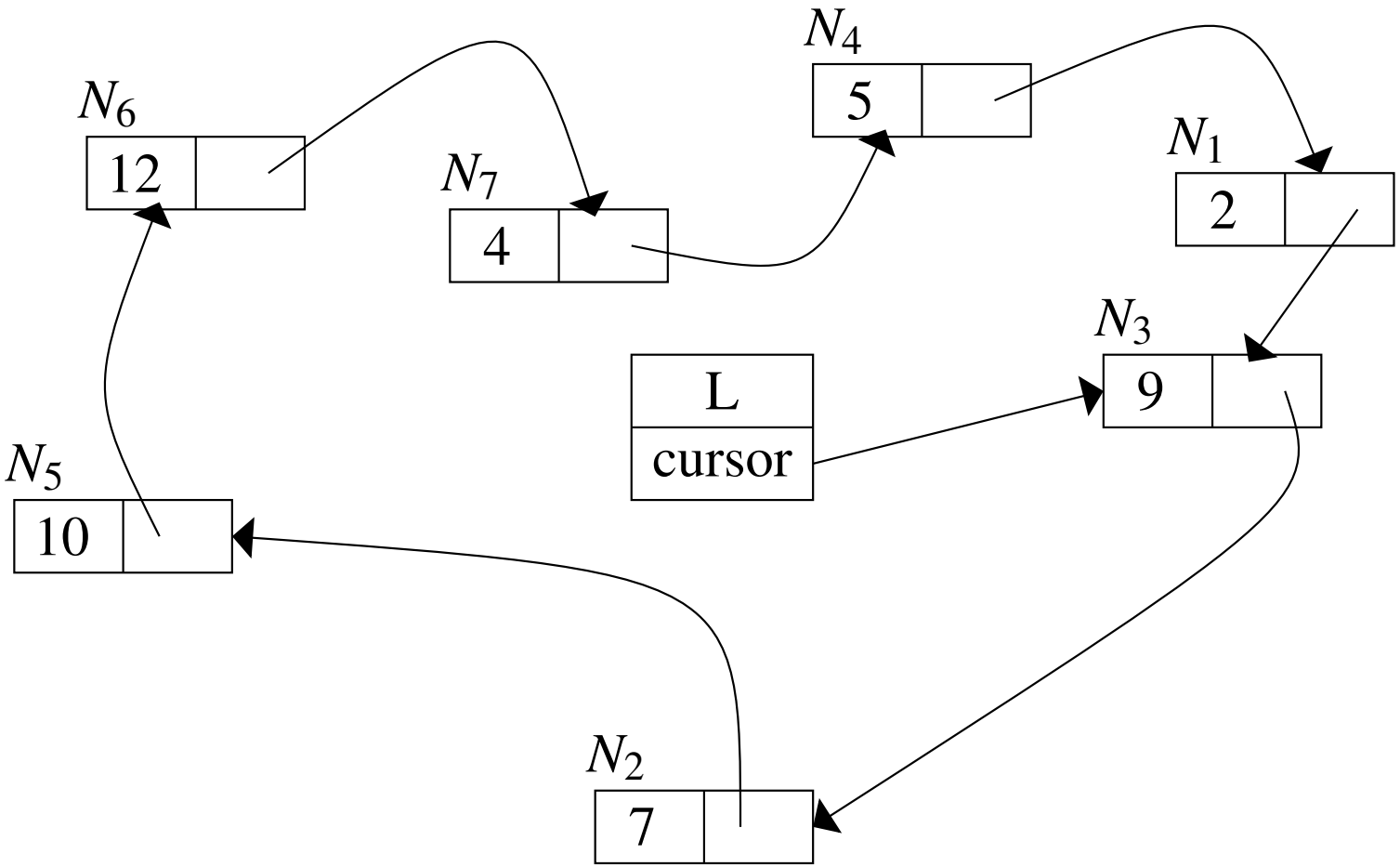
A doubly linked list



Circularly Linked Lists

- A circularly linked list has the same kind of nodes as a singly linked list. That is, each node has a next pointer and a reference to an element.
- The circularly linked list has no beginning or end. You can think of it as a singly linked list, where the last nodes next pointer, instead of being `NULL`, points back to the first node.
- Instead of references to the head and the tail, we mark a node of the circularly linked list as the cursor. The cursor is the starting node when we traverse the list.

A circularly linked list



Deletion from doubly linked list

Input: list L, node N

Output: L with N removed

Deletion from doubly linked list

Input: list L, node N

Output: L with N removed

M = N.prev

Deletion from doubly linked list

Input: list L, node N

Output: L with N removed

M = N.prev

P = N.next

Deletion from doubly linked list

Input: list L, node N

Output: L with N removed

```
M = N.prev
P = N.next
M.next = P
```

Deletion from doubly linked list

Input: list L, node N

Output: L with N removed

```
M = N.prev
P = N.next
M.next = P
P.prev = M
```

Deletion from doubly linked list

Input: list L, node N

Output: L with N removed

```
M = N.prev
P = N.next
M.next = P
P.prev = M
L.size = L.size - 1
```

```
Input:  n numbers in array A[0], . . ., A[n-1]
Output:  ?
   for i = 0 to n-2 do
         e = A[i]
         p = i
         for j = i+1 to n-1 do
               if A[j] < e then
                     e = A[j]
                     p = j
               end if
         end for
         swap A[i] and A[p]
   end for
   return  A
```

What is the output and how it is obtained. How long does this procedure take (that is, say, how many times do we make the comparison of $A[j]$ and e in the condition of the **if** statement)?