Topic 2: Arrays and Lists

Matthew Johnson

matthew.johnson2@dur.ac.uk

Data Structures

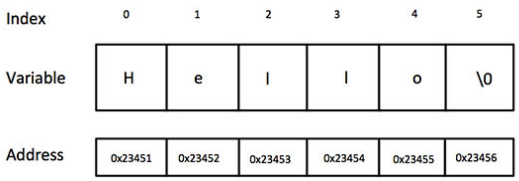
- For the next three weeks, we'll study different ways to store and organize data.
- We learn about different data structures, because each has its advantages and disadvantages.

Our first "data structure": arrays

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]`
They're in consecutive memory cells, but we (usually) don't care where exactly.

All array elements are of the same type, e.g., integer. Can declare as `integer A[1...n]`

Arrays



Arrays

- How do we find the *i*th 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:

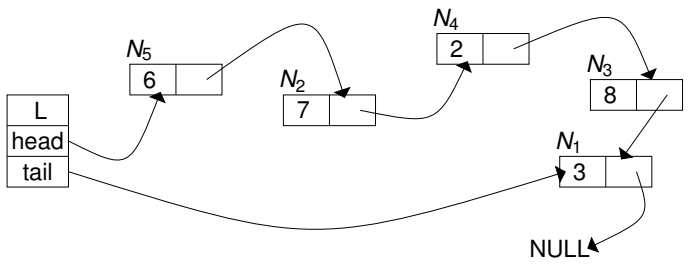
- `L.head`
- `L.tail`
- (and possibly also we have `L.size`)

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?

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

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

Insertion of v after N

Input: list L, data v, node N

Output: L with v inserted after N

Arrays versus lists

	Array	Linked List
Data Access		
Insertion, Deletion		

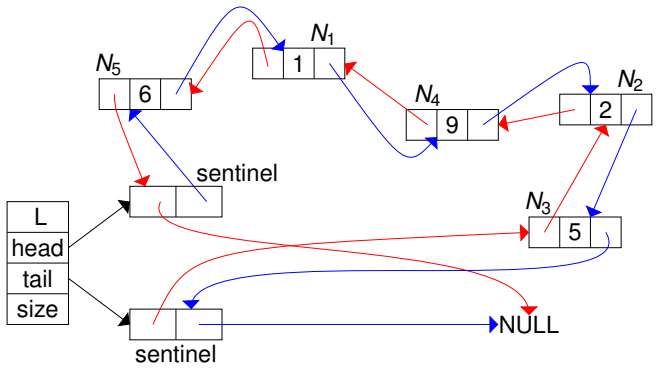
Doubly Linked Lists

- 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.

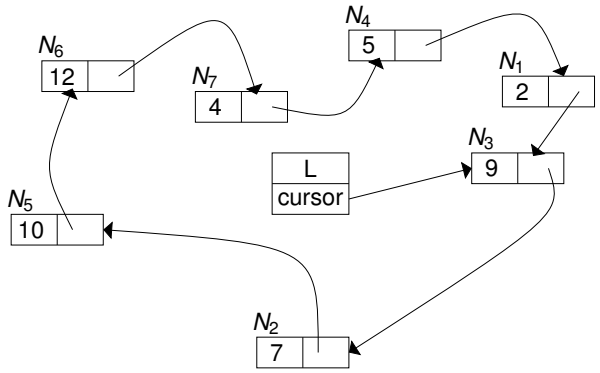
A doubly linked list



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


```
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)?