Data Structures I
Stack, Queue, Deque and Linked Lists

Gianmaria Silvello

Department of Information Engineering
University of Padua
gianmaria.silvello@unipd.it
http://www.dei.unipd.it/~silvello/

Fundamentals of Information Systems
Master Degree in Data Science

Outline

- Lists
- Stacks
- Queues
- Deques
- Linked lists
  - Singly Linked Lists
  - Circular Linked Lists
  - Doubly Linked Lists

Reference: Chapters 6 and 7
of Goodrich, Tamassia and Goldwasser

Lists

- A list is a sequential collection of values, it is a data structure

- Each value has a location (an index)

- Indexes range from 0 to n-1 (where n is the length of the list) or from 1 to n

- Lists are heterogeneous = values can be of any type (strings are homogeneous because their elements are characters)

Python methods

- `list.append(x)` -> Add an item to the end of the list;
- `list.extend(L)` -> Extend the list by appending all the items in the given list;
- `list.insert(i, x)` -> Insert an item x at a given position i. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list;
- `list.remove(x)` -> Remove the first item from the list whose value is x. It is an error if there is no such item;
- `list.pop(i)` -> Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list;
- `list.index(x)` -> Return the index in the list of the first item whose value is x. It is an error if there is no such item;
- `list.count(x)` -> Return the number of times x appears in the list.

Stack

Stack

- A stack is a collection of objects (list) following the **LIFO** principle
- LIFO = Last In First Out

Stack

- A stack is a collection of objects (list) following the **LIFO** principle
- LIFO = Last In First Out

Stack

- A stack is a collection of objects (list) following the **LIFO** principle
- LIFO = Last In First Out

Top of the stack

Stack

- A stack is a collection of objects (list) following the **LIFO** principle
- LIFO = Last In First Out

Stack

- A stack is a collection of objects (list) following the **LIFO** principle
- LIFO = Last In First Out

Elements are pushed in the stack and popped out of the stack

Stack

- Web browser back button

The browser saves the visited pages in a stack and when the back button is clicked it pops out the last page from the stack

- Editor “undo” function

The editor saves the updates in a stack and when the undo button is clicked it pops out the modification and reverts back to the previous stage page

Stack ADT

ADT = Abstract Data Type
It's a mathematical model of a data structure
that it specifies the type of the data stored,
the supported operations, and the parameters type

- `S.push(e)`
  - add an element on top
- `S.pop()`
  - Remove and return the top element
- `S.top()`
  - Return a reference to the top element (index)
- `S.is_empty()`
  - Return true if the stack is empty, false otherwise
- `len(S)`
  - Return the length of the stack

Queue

Queue

- A queue is a collection of objects following the FIFO (First In-First Out) principle

Queue ADT

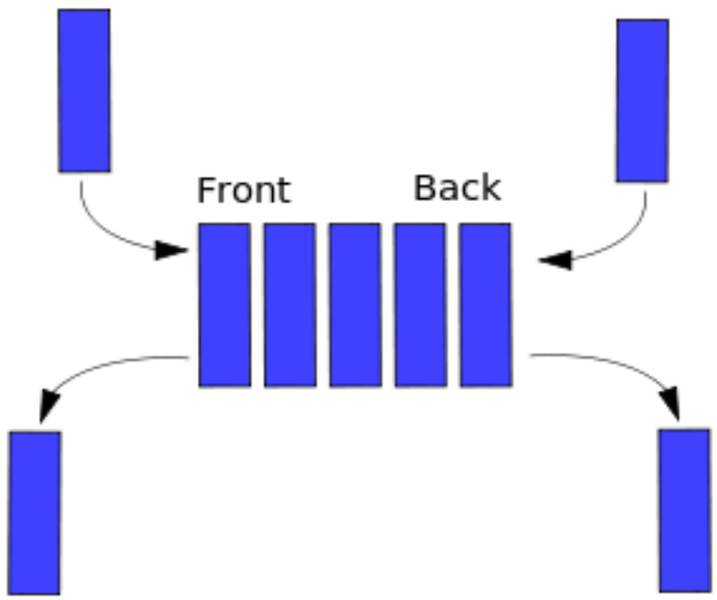
- `Q.enqueue(e)`
  - add an element e to the tail of the queue
- `Q.dequeue()`
  - Remove and return the element in front of the queue
- `Q.first()`
  - Return a reference to the first element (index)
- `Q.is_empty()`
  - Return true if the queue is empty, false otherwise
- `len(Q)`
  - Return the length of the queue

Check the Jupiter notebook

Deque

Deque (Deck)

- A deque is a double ended queue
- It is a queue where you can get the first() or the last() element
- You can also add an element to the front or to the back of a deque



Deque ADT

- `Q.add_first(e)`
  - add an element e to the front of the deque
- `Q.add_last(e)`
  - add an element e to the back of the deque
- `Q.delete_first()`
  - Return and remove the head (first element) of the deque
- `Q.delete_last()`
  - Return and remove the tail (last element) of the deque
- `Q.first(), Q.last(), Q.is_empty(), len(Q)`
  - Same as before

Implement the Deque ADT in Python

- It can be done by using the Collections module
  - See the Jupiter notebook

- Remember that in a deque `append()` adds to the back and that `popleft()` removes from the head

- Given a deque `D` in Python, you can get an element at index `i` as `D[i]`
  - `Deque:` Indexed access is O(1) at both ends but slows to O(n) in the middle. For fast random access, use lists instead. [Python doc]

Linked Lists

Linked List

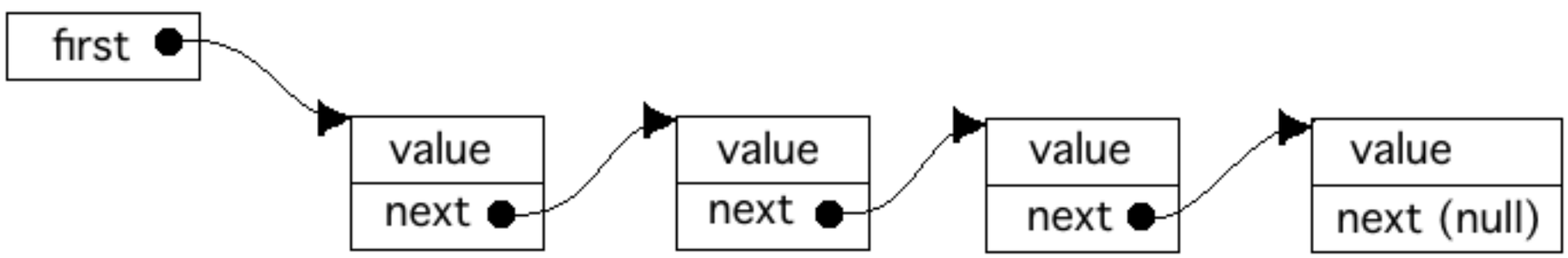
- A linked list is a data structure where elements are arranged in linear order

- Unlike Arrays where the order is maintained by indexes, in linked lists the order is maintained by means of a pointer in each object

- You cannot use indexes to access the elements of a linked list

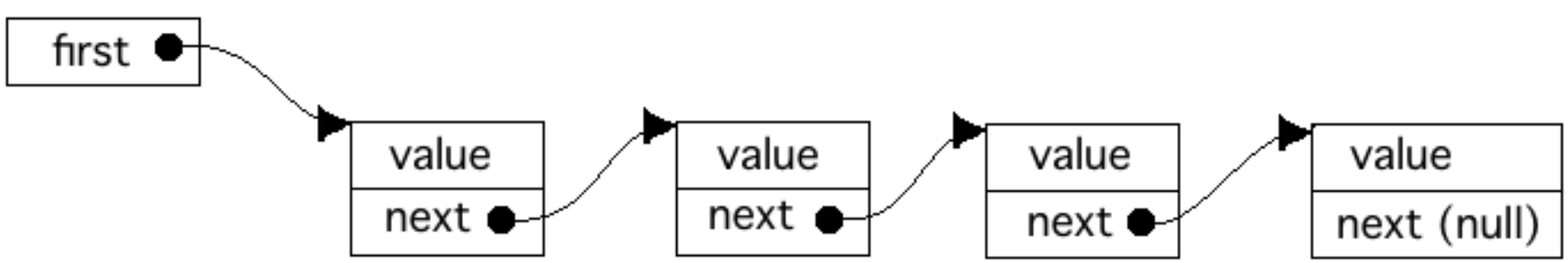
Singly Linked List

- Every node contains a value and a pointer to the next element
- The first element is the head of the list and the last element is the tail
- *Traversing* a list means moving from the head to the tail

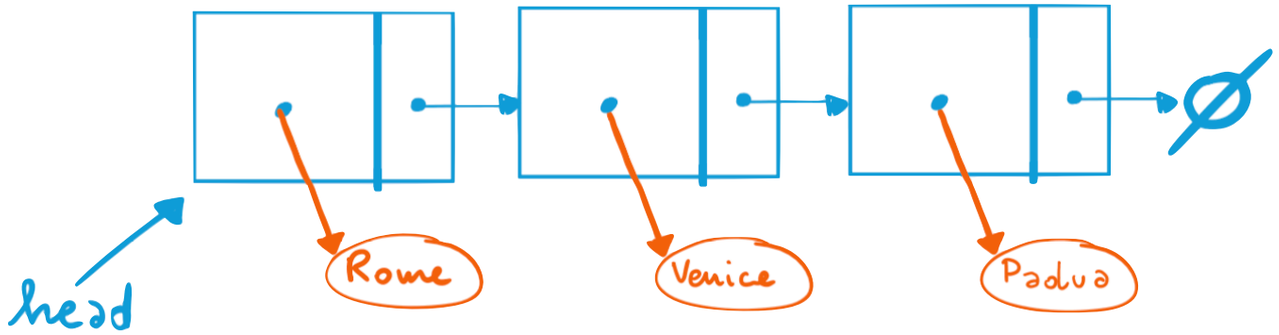


Singly Linked List

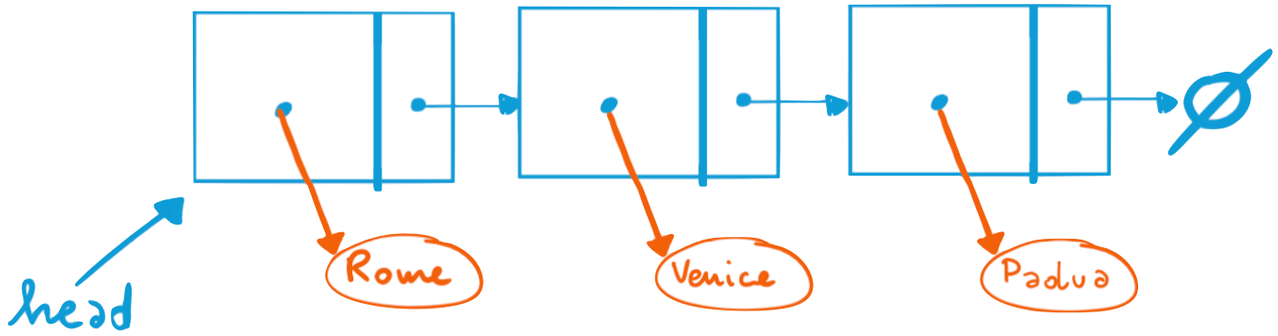
- A Linked List must maintain a pointer to the head, otherwise there is no way to locate that node
- Sometimes also a pointed to the tail is stored (to avoid traversal)
- List “nodes” are “objects”



Add a node to a linked list

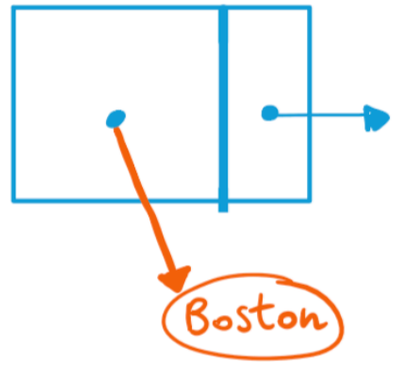


Add a node to a linked list

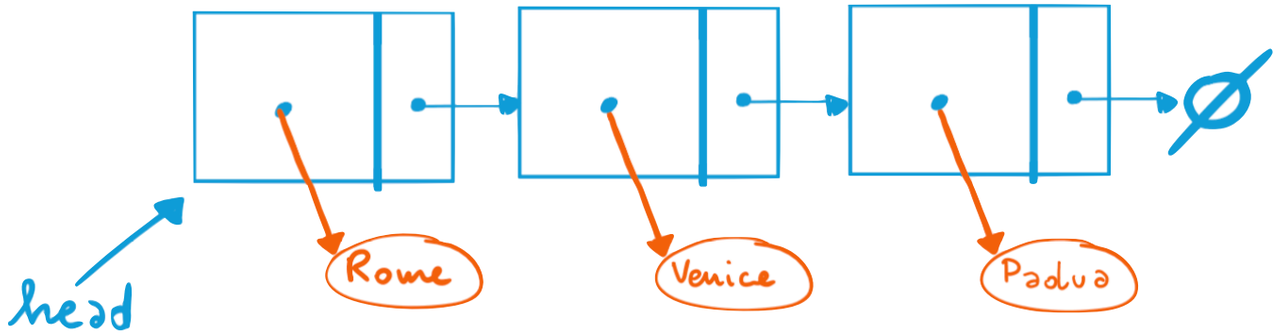


① Create a new node

```
newest = Node(e)
```

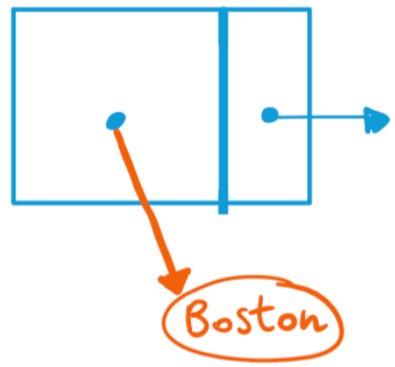


Add a node to a linked list



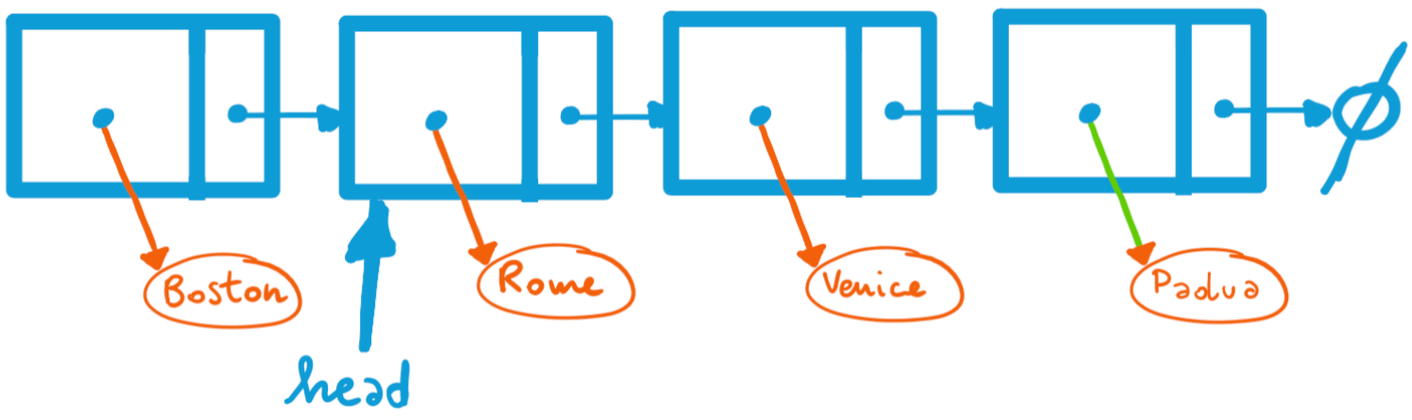
① Create a new node

```
newest = Node(e)
```

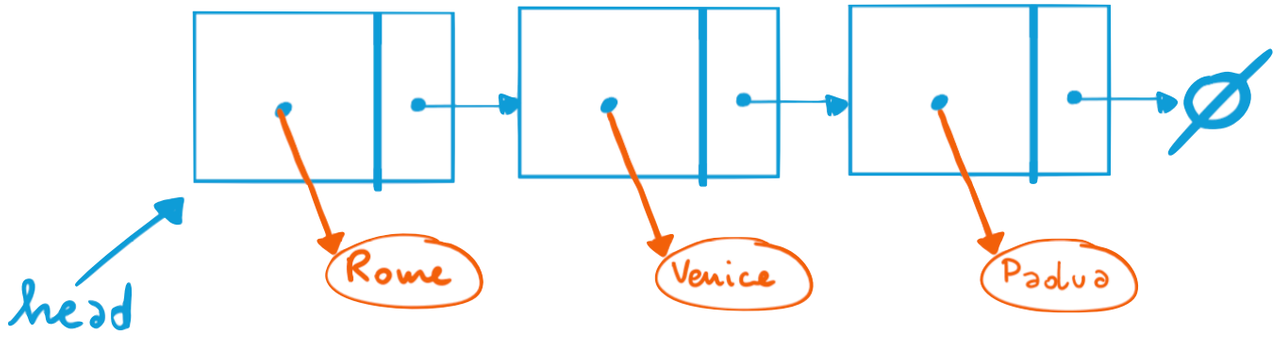


② New node points to old head

```
newest.next = L.head
```

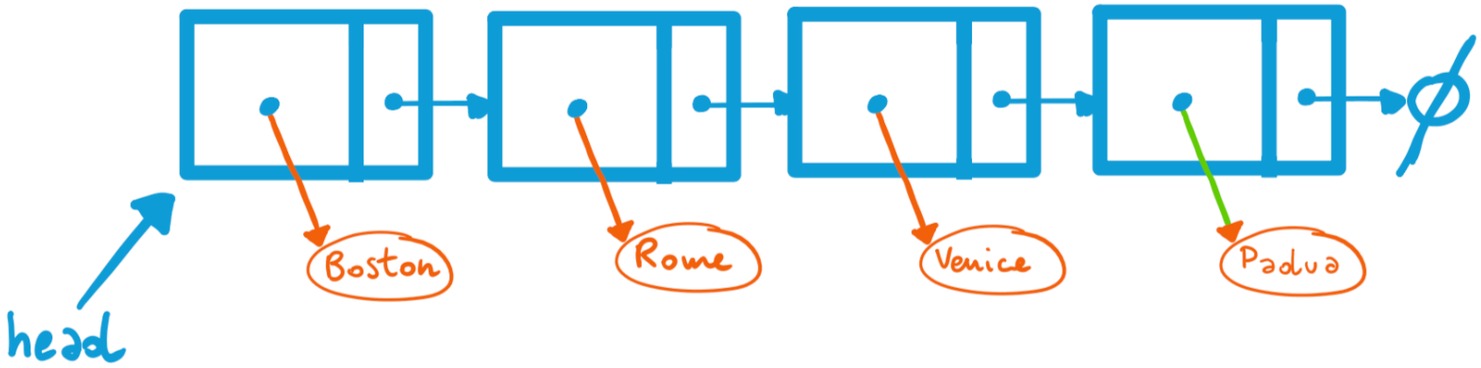


Add a node to a linked list

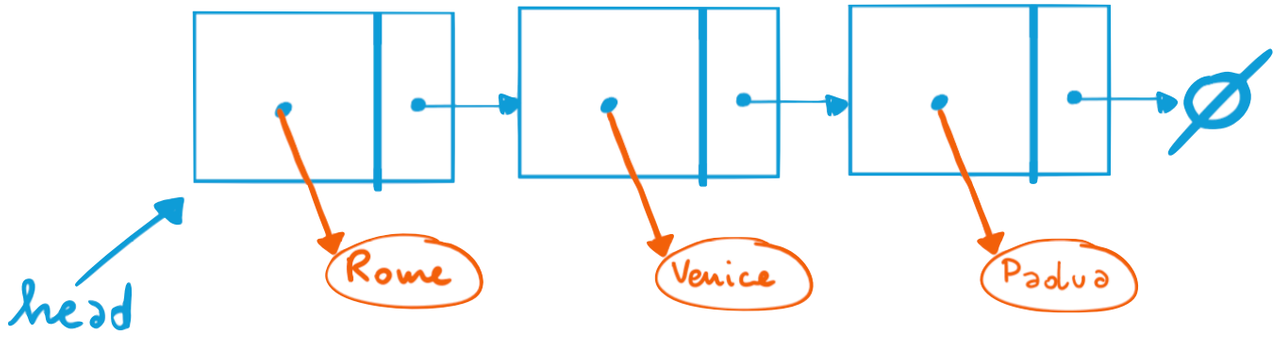


(3) Head points to the new node

L.head = newest

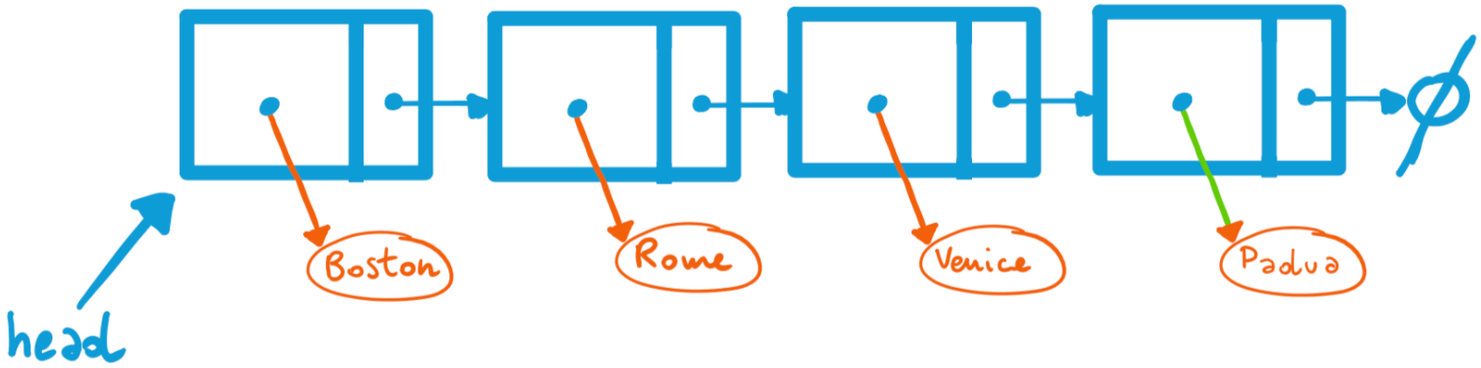


Add a node to a linked list



③ Head points to the new node

L.head = newest

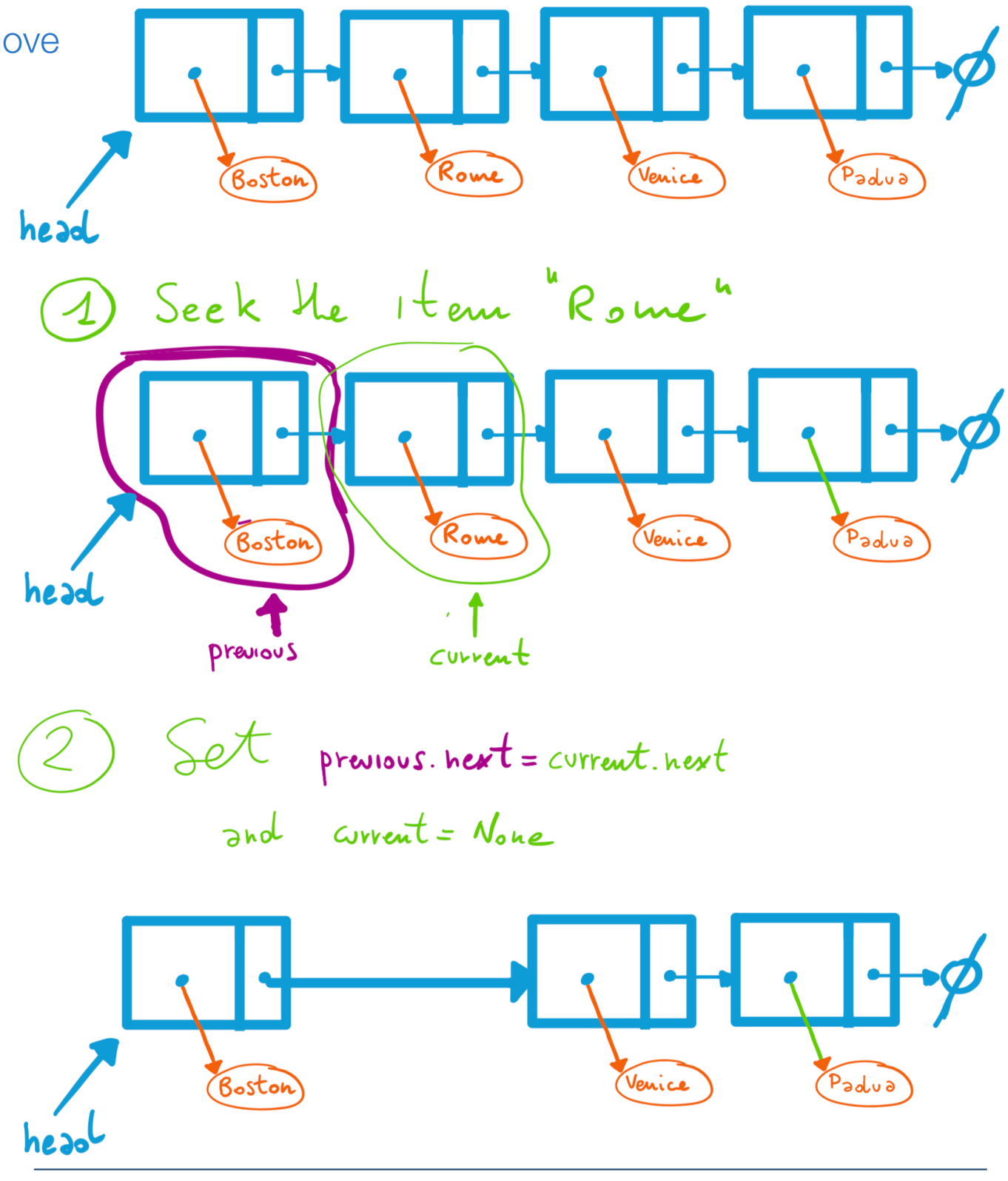


④ Update list size

L.size = L.size + 1

Remove a node

Given a linked list, remove the item “Rome”



Singly Linked List: Exercise

- Given a singly linked list of N nodes. The task is to find middle of the linked list.

- For example, if given linked list is 1->2->3->4->5 then output should be 3.

- If there are even nodes, then there would be two middle nodes, we need to print second middle element. For example, if given linked list is 1->2->3->4->5->6 then output should be 4.

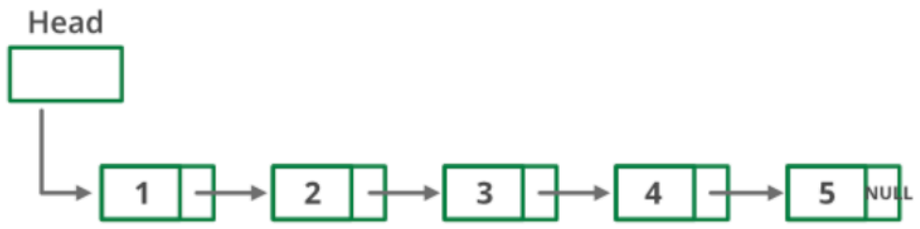
Singly Linked List: Solution 1

- Traverse the whole linked list and count the number of nodes. Now traverse the list again till count/2 and return the node at count/2.

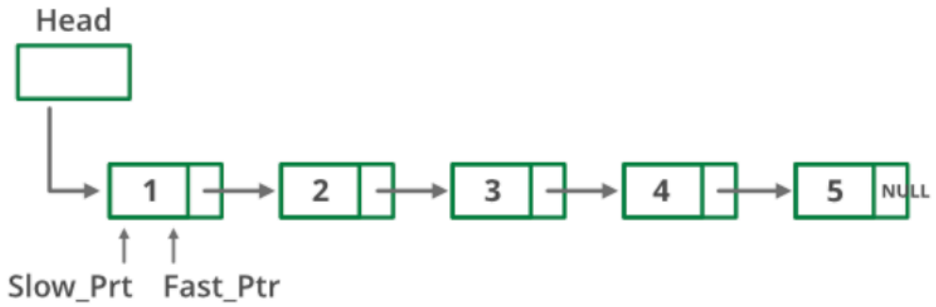
- Liner running time.

Singly Linked List: Solution 2

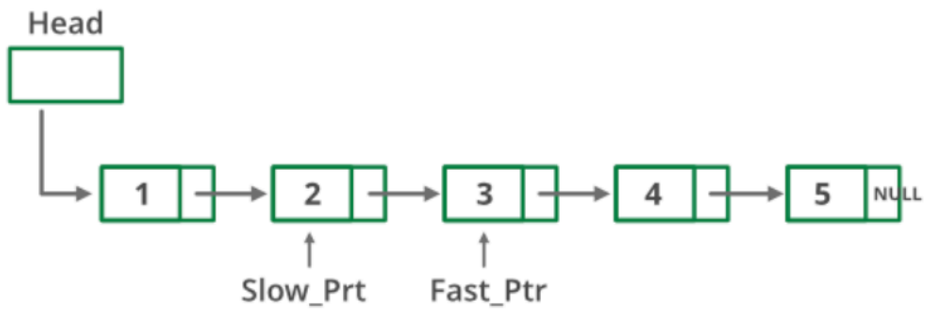
Initially :



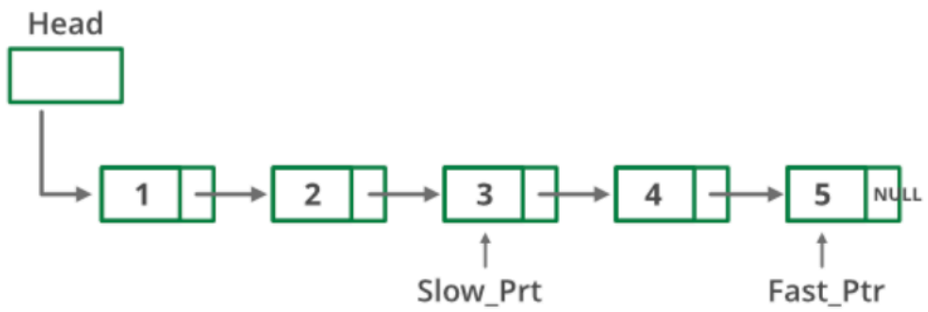
Step 1:



Step 2:



Step 3:



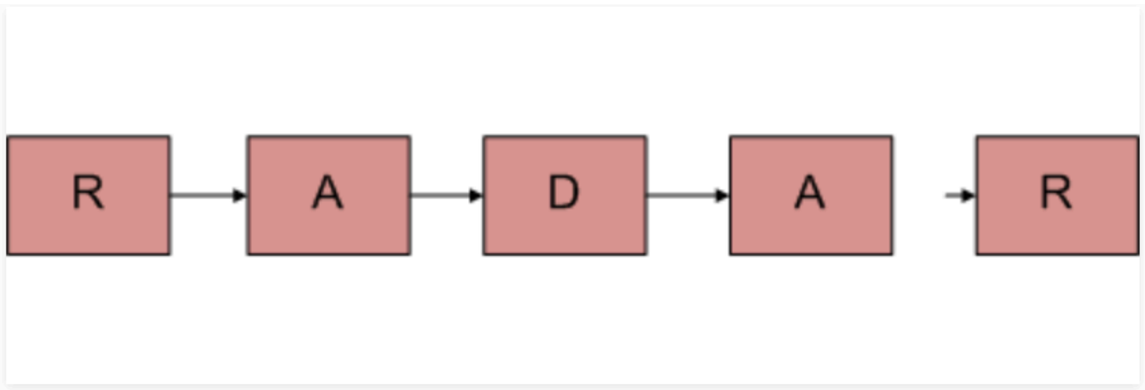
Singly Linked List: Solution 2

- Traverse linked list using two pointers. Move one pointer by one and other pointer by two. When the fast pointer reaches end slow pointer will reach middle of the linked list.

- Liner running time, but better than before, why?

Singly Linked List: Exercise

- Given a singly linked list of characters, write a function that returns true if the given list is a *palindrome*, else false.



Singly Linked List: Exercise

- Use Stacks
  - Traverse the given list from head to tail and push every visited node to stack.
  - Traverse the list again. For every visited node, pop a node from stack and compare data of popped node with currently visited node.
    - If all nodes matched, then return true, else false.

- The time complexity of the above method is O(n)

Exercise: Reverse a linked list

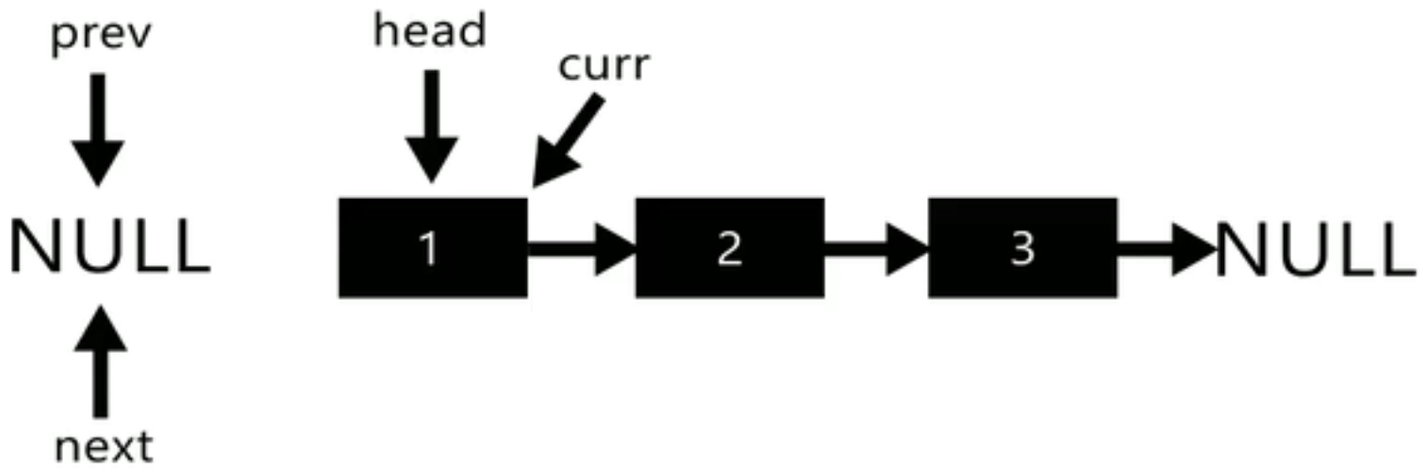
- Given pointer to the head node of a linked list, the task is to reverse the linked list. We need to reverse the list by changing links between nodes.

- Input: Head of following linked list
  1->2->3->4->NULL
  Output: Linked list should be changed to,
  4->3->2->1->NULL

- Input: Head of following linked list
  1->2->3->4->5->NULL
  Output: Linked list should be changed to,
  5->4->3->2->1->NULL

Exercise: Reverse a linked list

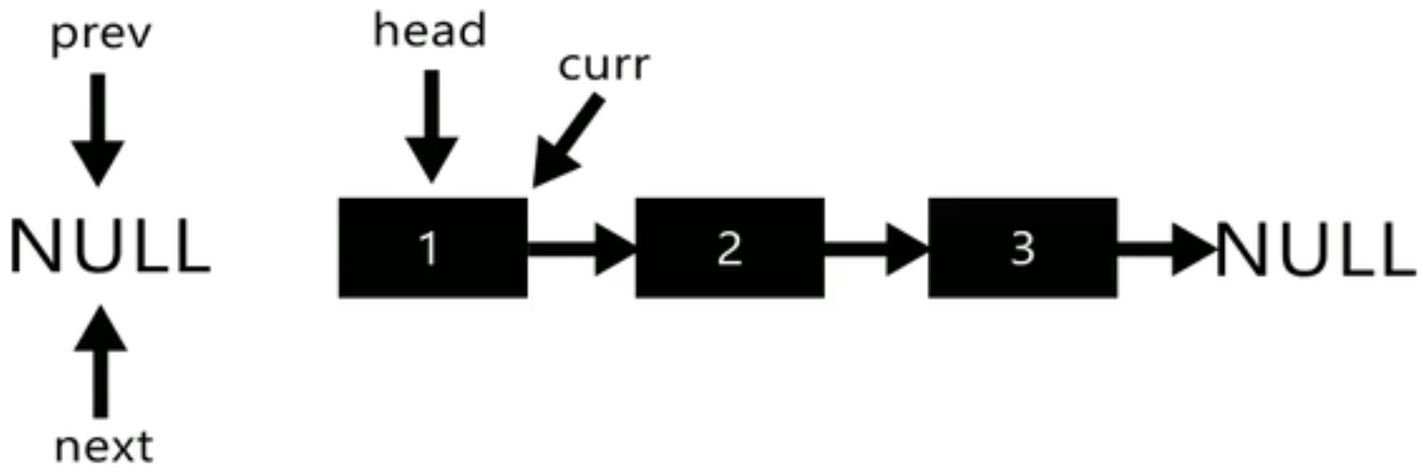
Exercise: Reverse a linked list




```
while (current != NULL)
     {
          next  = current->next;
          current->next = prev;
          prev = current;
          current = next;
     }
     *head_ref = prev;
```

https://www.geeksforgeeks.org/reverse-a-linked-list/

Exercise: Reverse a linked list

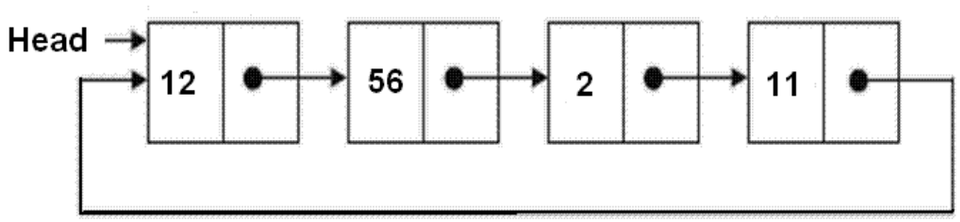



```
while (current != NULL)
     {
          next  = current->next;
          current->next = prev;
          prev = current;
          current = next;
     }
     *head_ref = prev;
```

https://www.geeksforgeeks.org/reverse-a-linked-list/

Circularly Linked Lists

- The next pointer of the last element points to the head of the list
- The traversal is a loop
- Be careful when you use it!



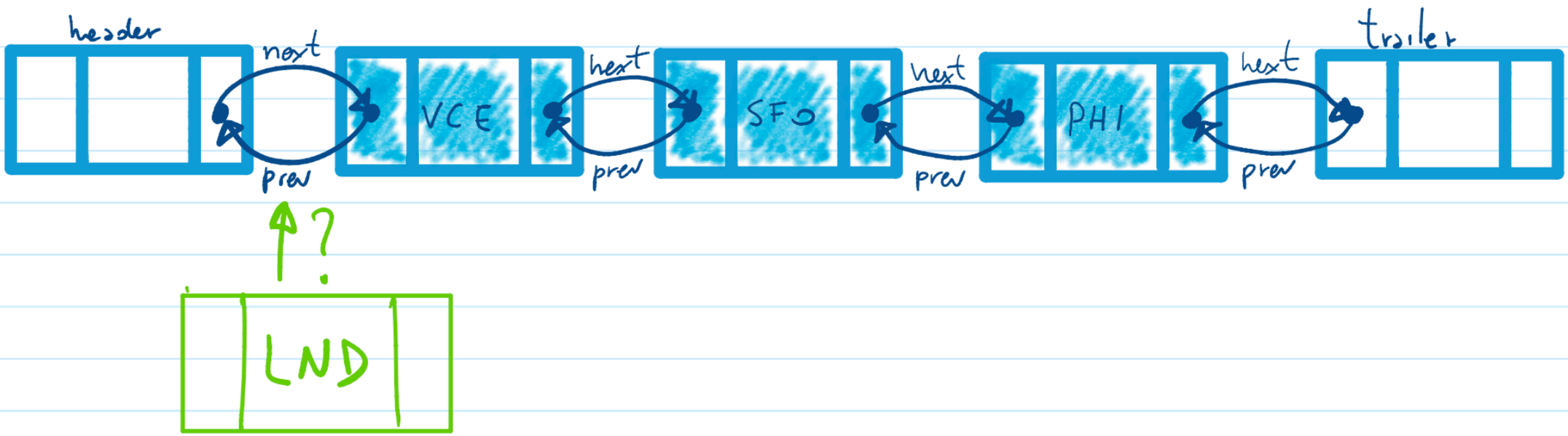
Doubly Linked Lists

- It provides greater symmetry than the singly linked list
- It efficient when adding or deleting nodes on both sides
- It uses header and trailer sentinels

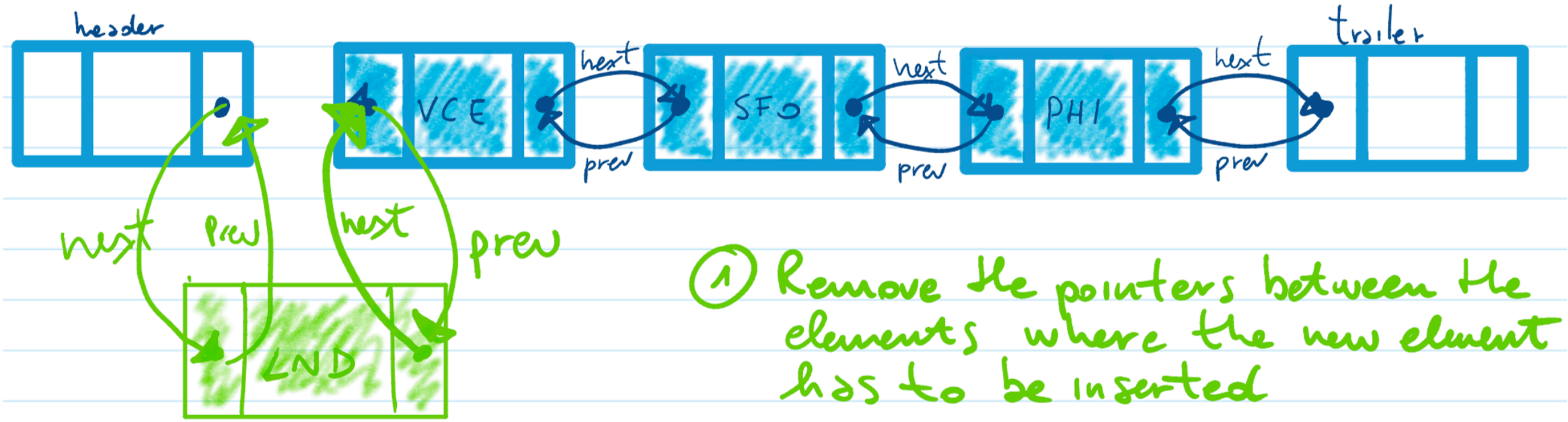
Doubly Linked Lists

① Header and trailer require space, but they add some advantages

② You can treat all the insertions in a unified way and there will always be an empty node before a new node.



Doubly Linked Lists



1. Remove the pointers between the elements where the new element has to be inserted
2. Create the new pointers.

Lists or Arrays?

Array vs List-based data structures

- Access complexity
  - Arrays: O(1)-time access to an element based on an index
  - Lists: O(n) in a linked list to do the traversal

- Arrays: be careful with the resizing issue

- Memory consumption for *n* elements
  - Arrays: we may need to store *2n* elements (dynamic resizing)
  - Lists: we store n elements and n references (singly linked lists) and 2n references (doubly linked lists)

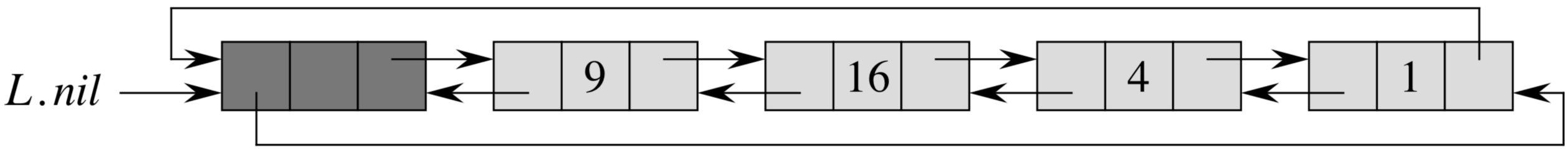
Arrays vs Linked Lists

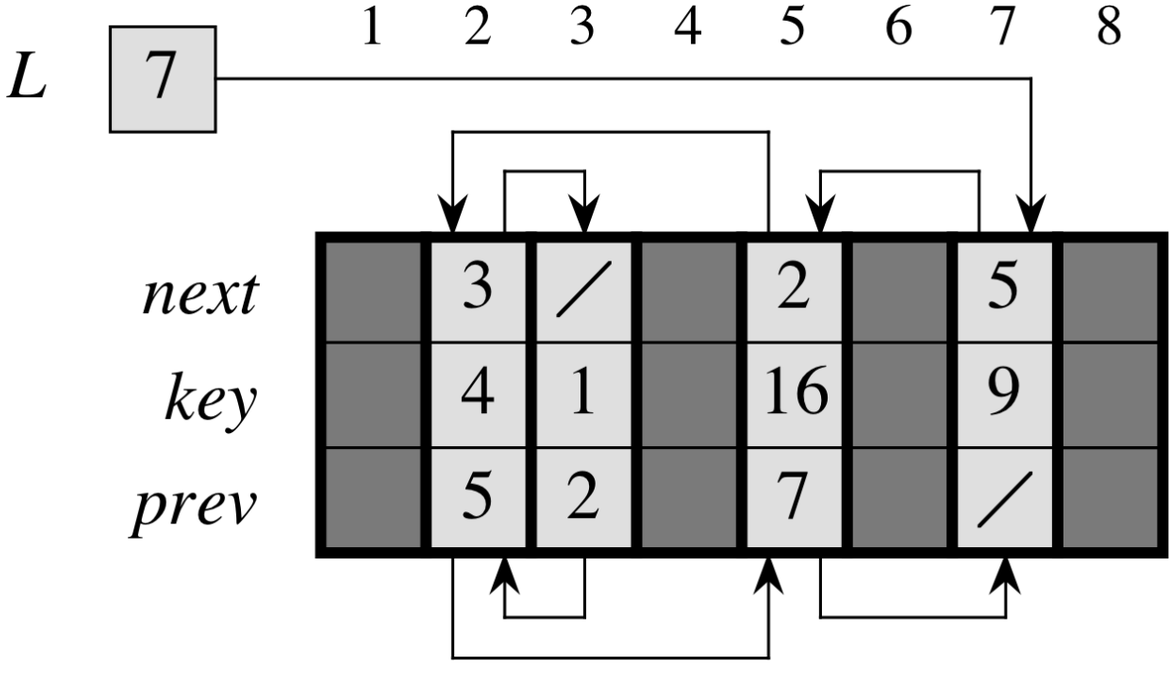
Arrays	Linked Lists
An array is a collection of elements of a similar data type.	Linked List is an ordered collection of elements of the same type in which each element is connected to the next using pointers.
Array elements can be accessed randomly using the array index.	Random accessing is not possible in linked lists. The elements will have to be accessed sequentially.
Data elements are stored in contiguous locations in memory.	New elements can be stored anywhere and a reference is created for the new element using pointers.
Insertion and Deletion operations are costlier since the memory locations are consecutive and fixed.	Insertion and Deletion operations are fast and easy in a linked list.
Memory is allocated during the compile time (Static memory allocation).	Memory is allocated during the run-time (Dynamic memory allocation).
Size of the array must be specified at the time of array declaration/initialization.	Size of a Linked list grows/shrinks as and when new elements are inserted/deleted.

https://www.faceprep.in/data-structures/linked-list-vs-array/

Implementing linked lists with arrays

Doubly linked list L:





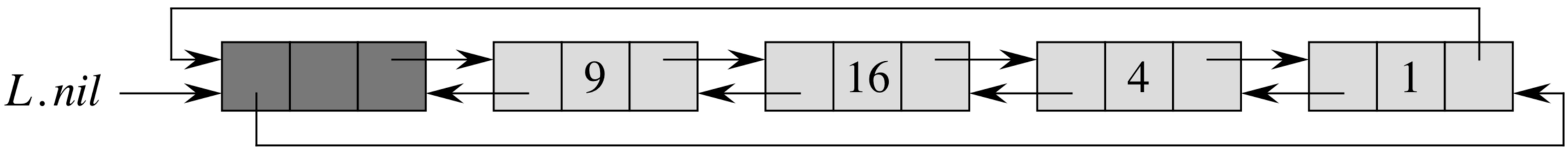
Implementation with 3 arrays:

next keeps the pointers to the next elements
key contains the actual elements
prev keeps the pointers to the previous elements

The variable **L** keeps the index of the head

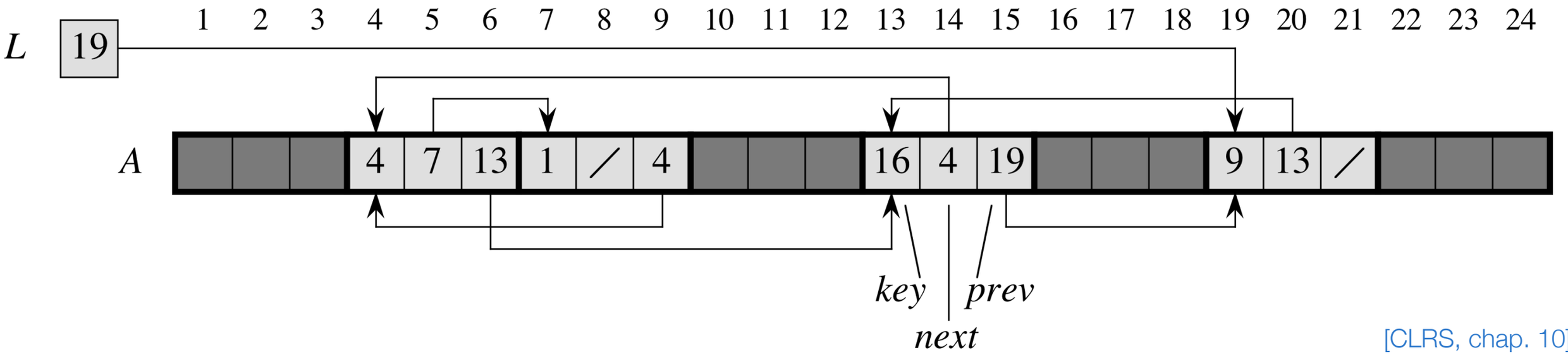
[CLRS, chap. 10]

Implementing linked lists with a single array



An object occupies a contiguous sub-array *[j … k]* where each attribute of an object corresponds to an offset.

In the range from *0* to *k-j*, and a pointer to the object is the index *j*. The offsets corresponding to *key*, *next* and *prev* are 0,1,2



[CLRS, chap. 10]

Positional List

Positional list

- It is a data structure that allows us to perform arbitrary insertions and deletions or to refer to elements anywhere in a list
- Numerical indexes are good, but they require to scan the entire list to find a specific element and to change dynamically when we update a list
  - an index does not always refer to the same element within a list
- A positional list is an abstraction that provides the ability to identify the position of an element in a list
- `p.element()`

Positional list ADT

Position	p	q	w
Element	e1	e2	e3

- `L.first()`
  - Return the position of the first element in the list
- `L.last()`
  - Return the position of the last element in the list
- `L.before(p)(L.after(p))`
  - Return the position of the list immediately before p
- `L.is_empty(), len(L)`
- `iter (L)`
  - Return a forward iterator for the elements of the list

Positional list ADT

Position	p	q	w
Element	e1	e2	e3

- `L.add_first(e)` (specular: `L.add_last(e)`)
  - Insert a new element `e` at the front (at the end) of the list
- `L.add_before(p,e)` (specular: `L.add_after(p,e)`)
  - Insert a new element `e` before (after) position `p`
- `L.replace(p,e)`
  - Replace the element at position `p` with the element `e`
- `L.delete(p)`
  - Remove and return the element at position p