COMP20230: Data Structures & Algorithms
Lecture 9: Linked Lists

Dr Andrew Hines

Office: E3.13 Science East
School of Computer Science
University College Dublin

andrew.hines@ucd.ie

Last Day

Abstract Data Types

Talked about sequences – a simple ADT.
How it could be implemented using an array or linked list.
Looked at arrays.

Other ADTs can extend/adapt a sequence

e.g. Queues, Stacks, etc.

Complexity Analysis

Sequence using an array.

Operations	Array
`size, is_empty`	$\mathcal{O}(1)$
`get_elem_at_rank`	$\mathcal{O}(1)$
`set_elem_at_rank`	$\mathcal{O}(1)$
`insert_element_at_rank`	$\mathcal{O}(n)$
`remove_element_at_rank`	$\mathcal{O}(n)$
`insert_first, insert_last`	$\mathcal{O}(1)$
`insert_after, insert_before`	$\mathcal{O}(n)$

Today we will compare to Linked list complexity.

Linked Lists

Linked Lists:

- Why and what
- Basic operations
- Traversing
- Inserting
- Removing

Take home messsage

Linked Lists and arrays are the underlying implementations to turn ADTs into data structures

Linked Lists

Why not just arrays?

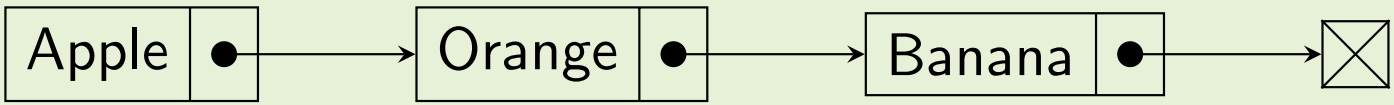
Linked lists are alternative option with different attribute (speed of search vs edit).
Match the implementation to the activity (horses for courses)

Linked Lists

Why not just arrays?

Linked lists are alternative option with different attribute (speed of search vs edit).
Match the implementation to the activity (horses for courses)

Linked lists are a set of element bearing nodes threaded together



Linked lists are

a set of *element bearing nodes* **threaded together**

Nodes

Nodes contain:

Element

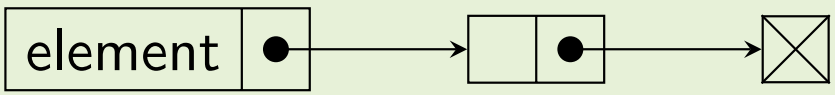
an object or primitive data type (e.g. String, integer, instance of a `Person` class etc.

next • →

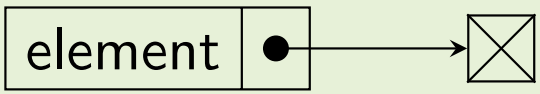
a pointer/reference to:

- the node's successor
- `None`/null

Linked list node, successor node and null terminator



Linked list node and a null terminator



Advantages and Applications

Advantages

- No predetermined size
- Space usage proportional to size
- Some manipulations more efficient than arrays

Disadvantages

- Size: 4/8 bytes (octets) for each 32/64 bit address pointer
- No direct access via index to individual elements in list

Advantages and Applications

Advantages

- No predetermined size
- Space usage proportional to size
- Some manipulations more efficient than arrays

Disadvantages

- Size: 4/8 bytes (octets) for each 32/64 bit address pointer
- No direct access via index to individual elements in list

Applications

- Linked lists are found in many applications
- File systems of most operating systems
- Whenever an application needs to deal with unknown and potentially changing number of elements

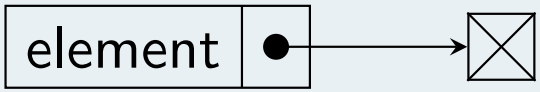
Linked List Operations

- Creating
- Access
- Traverse
- Insert node
- Remove node

Linked List Operations: Create

A node *is* a list

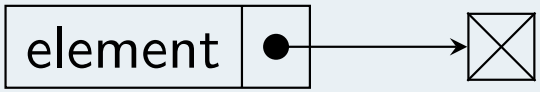
- element/object
- next pointer = None/null



Linked List Operations: Create

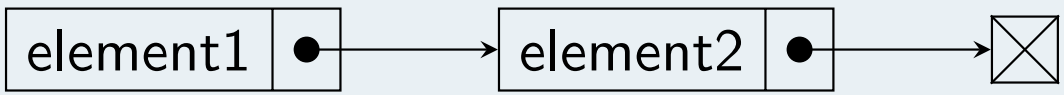
A node *is* a list

- element/object
- next pointer = None/null



Add elements by concatination

- each addition is just modifying the "next" pointer target address
- Structure built "organically" node by node, not created all at once like an array

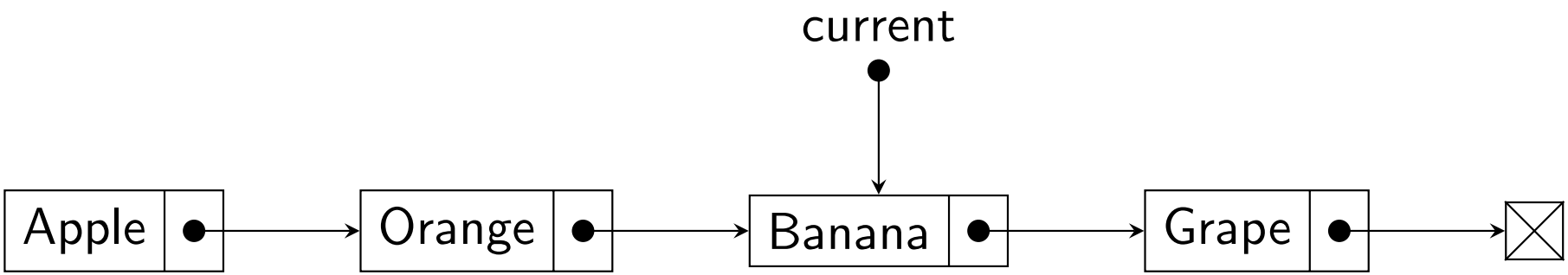


Linked List Operations: Basic Operations (Get)

Lists have a current element index

Two methods to access two fields:

- `current.get_element()` returns "Banana"
- `current.get_next()` returns "Grape"



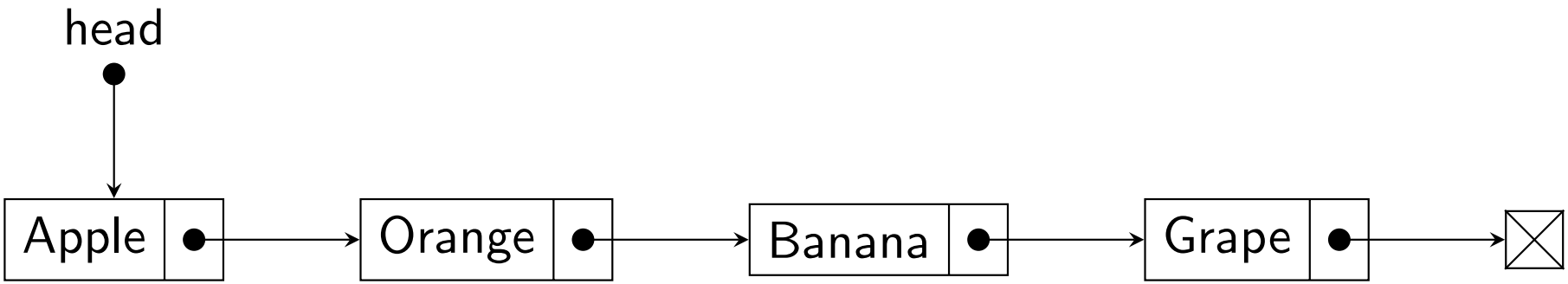
Linked List Operations: Traversing a List

Algorithm list_traversal

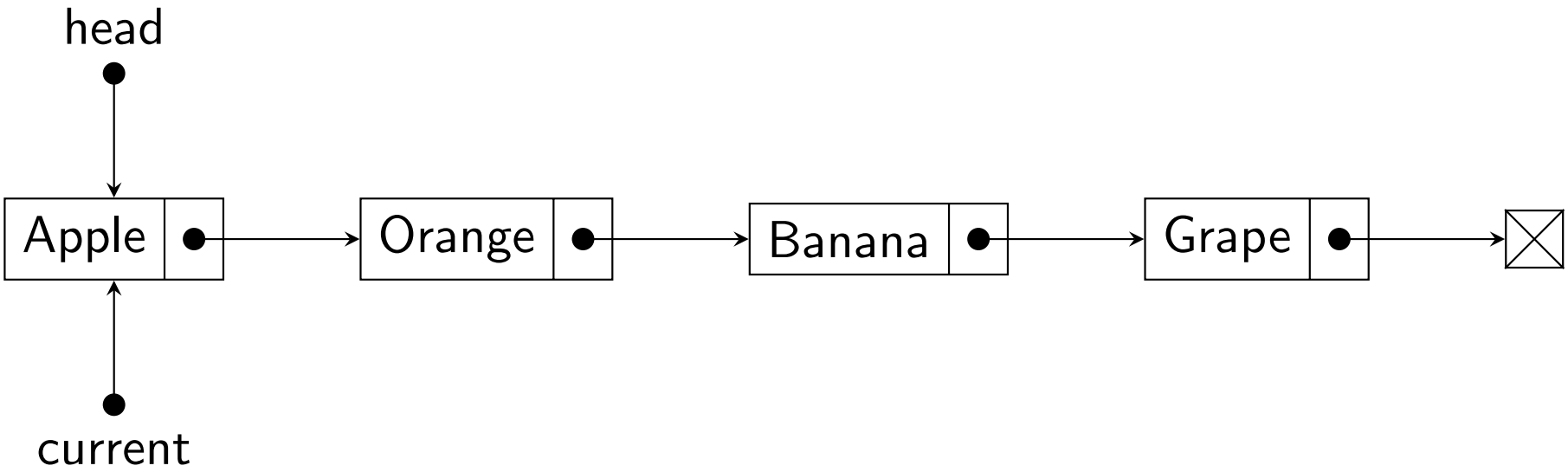
Input: L a Linked-list, *head* a pointer to the first node
Output: every node in the list has been seen

```
current ← head
while current! = None/null do
    print current.get_element()
    current ← current.get_next()
end while
print "Finished!"
```

Linked List Operations: Traversing a List

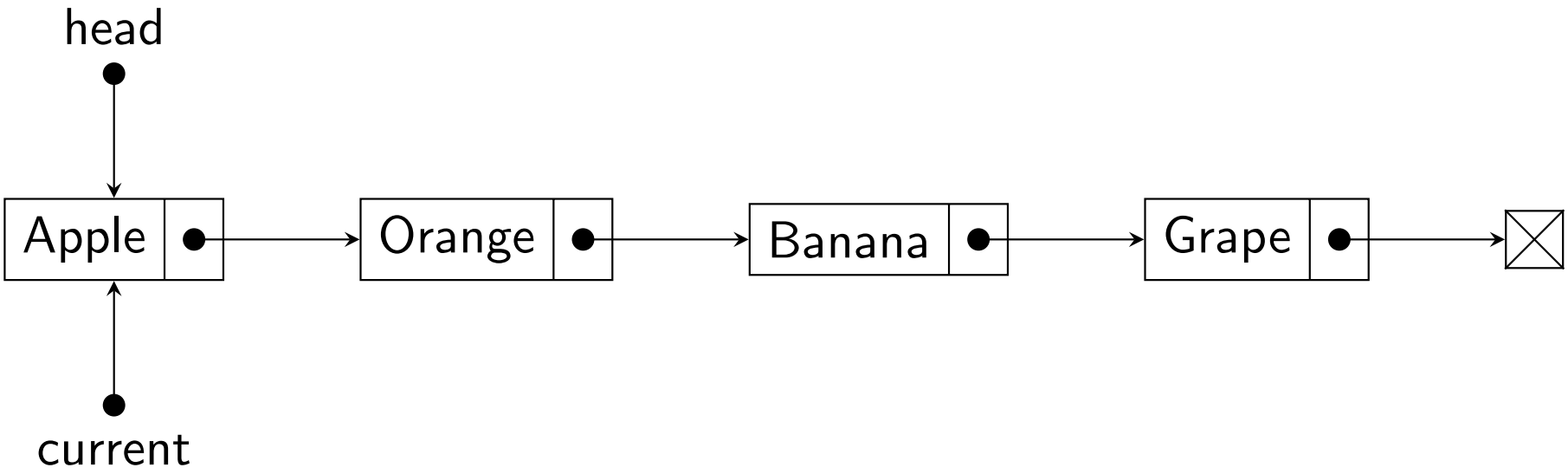


Linked List Operations: Traversing a List



List traversal output

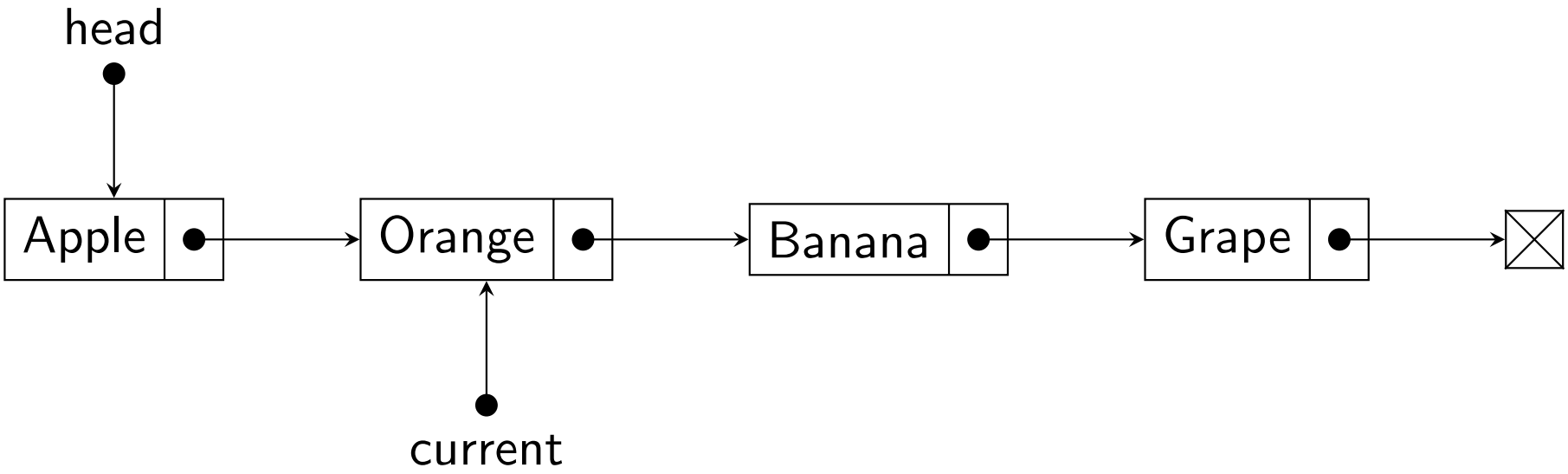
Linked List Operations: Traversing a List



List traversal output

Apple

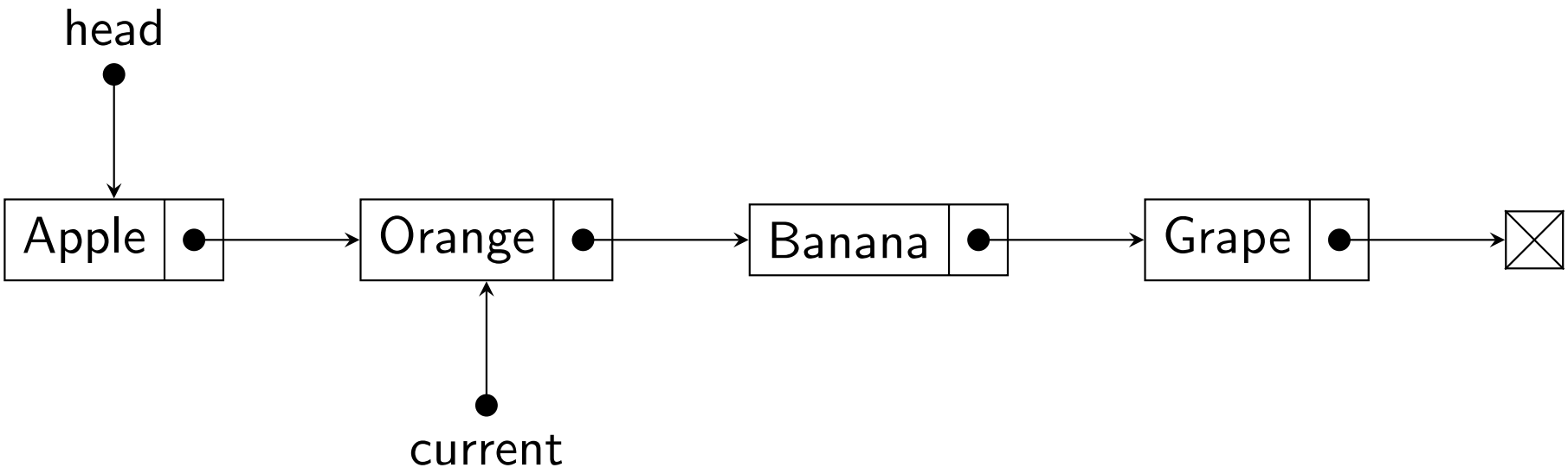
Linked List Operations: Traversing a List



List traversal output

Apple

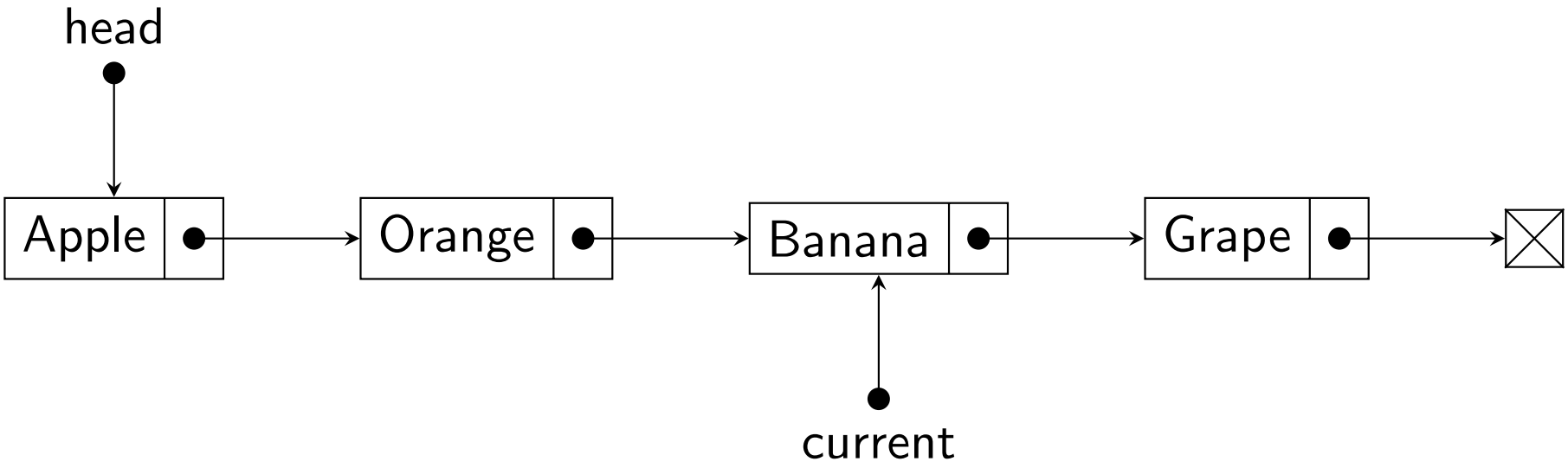
Linked List Operations: Traversing a List



List traversal output

Apple
Orange

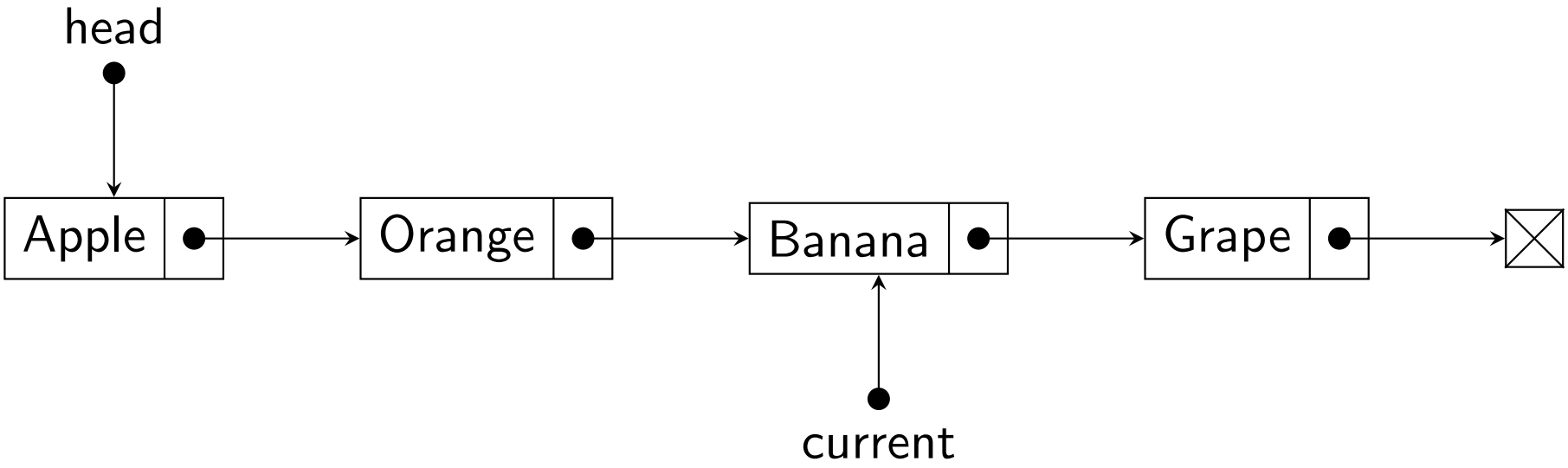
Linked List Operations: Traversing a List



List traversal output

Apple
Orange

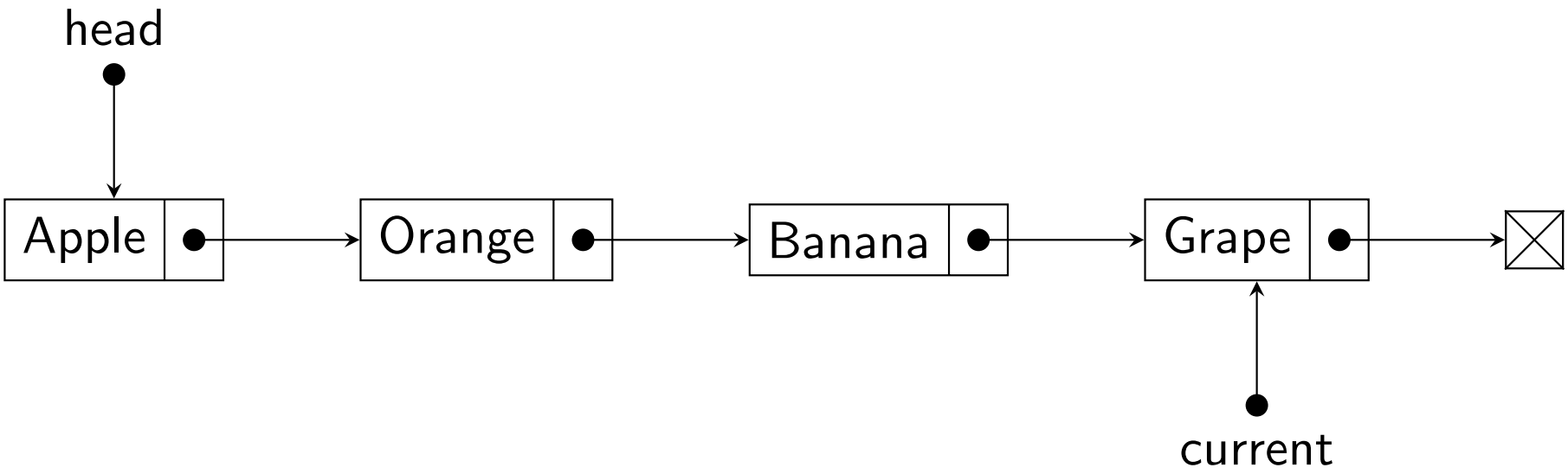
Linked List Operations: Traversing a List



List traversal output

Apple
Orange
Banana

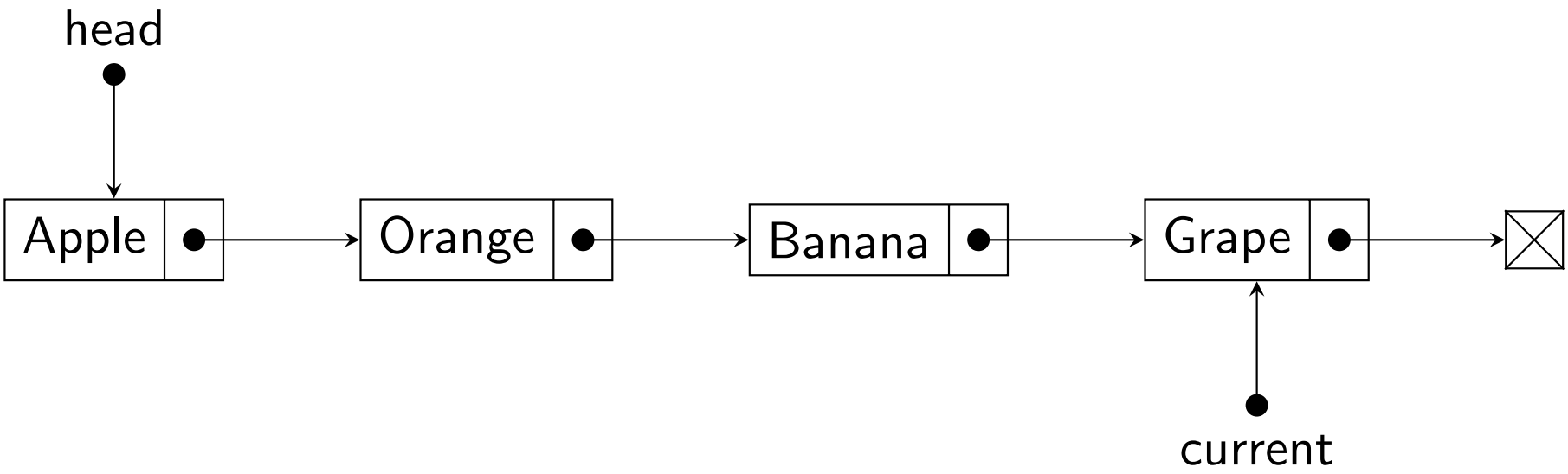
Linked List Operations: Traversing a List



List traversal output

Apple
Orange
Banana

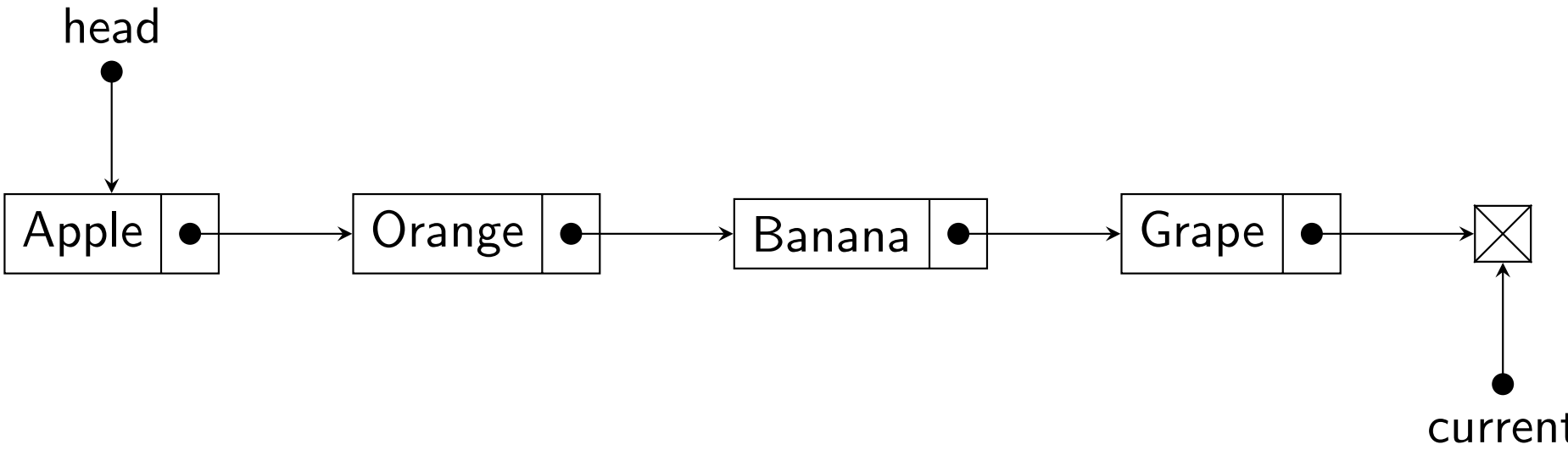
Linked List Operations: Traversing a List



List traversal output

Apple
Orange
Banana
Grape

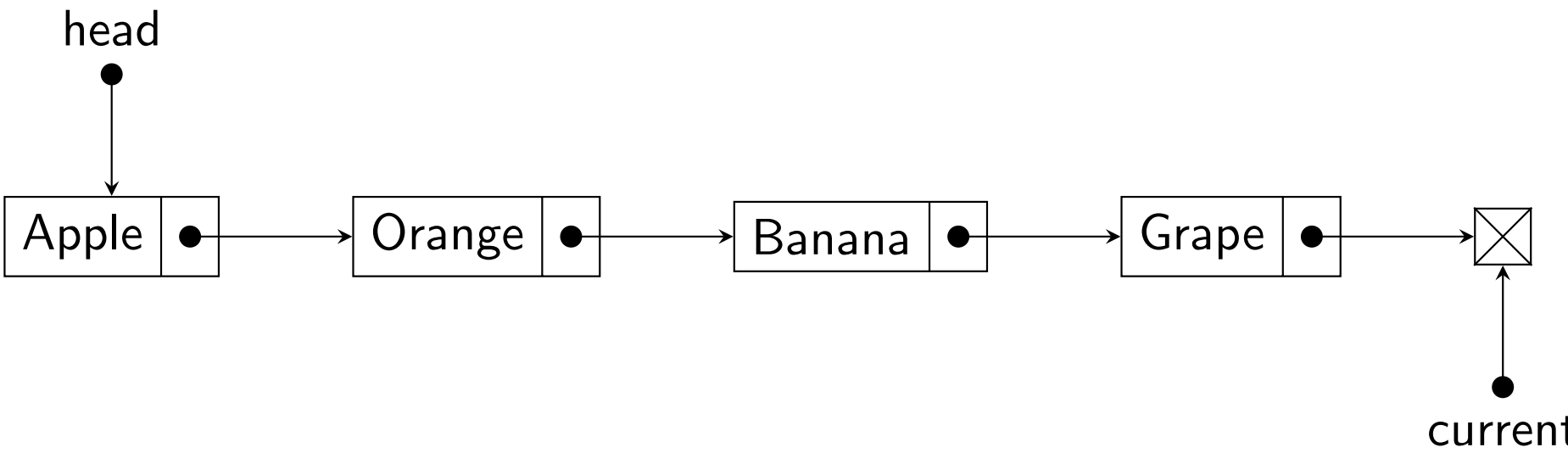
Linked List Operations: Traversing a List



List traversal output

Apple
Orange
Banana
Grape

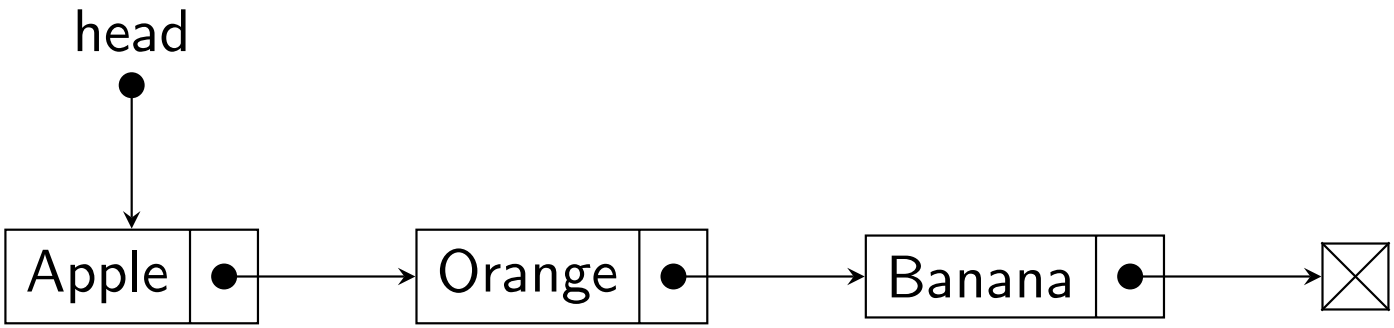
Linked List Operations: Traversing a List



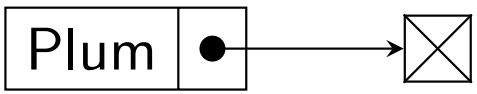
List traversal output

Apple
Orange
Banana
Grape
Finished!

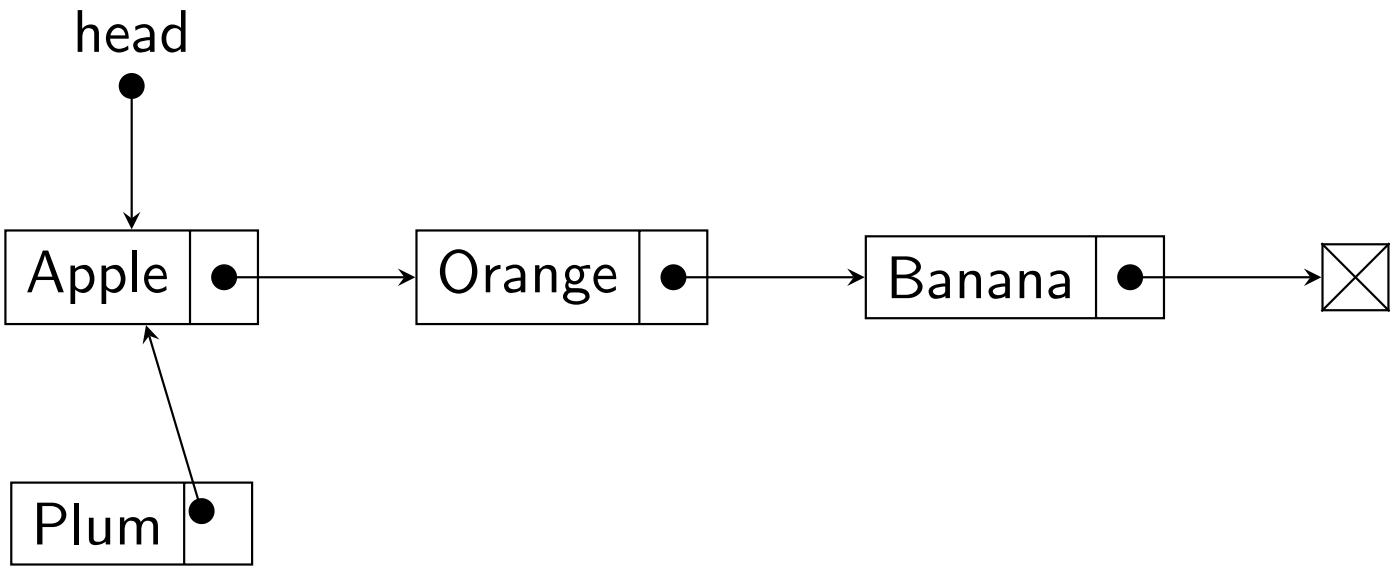
Linked List Operations: Inserting a node (at the head)



A new node to be added to the beginning (head) of the list:

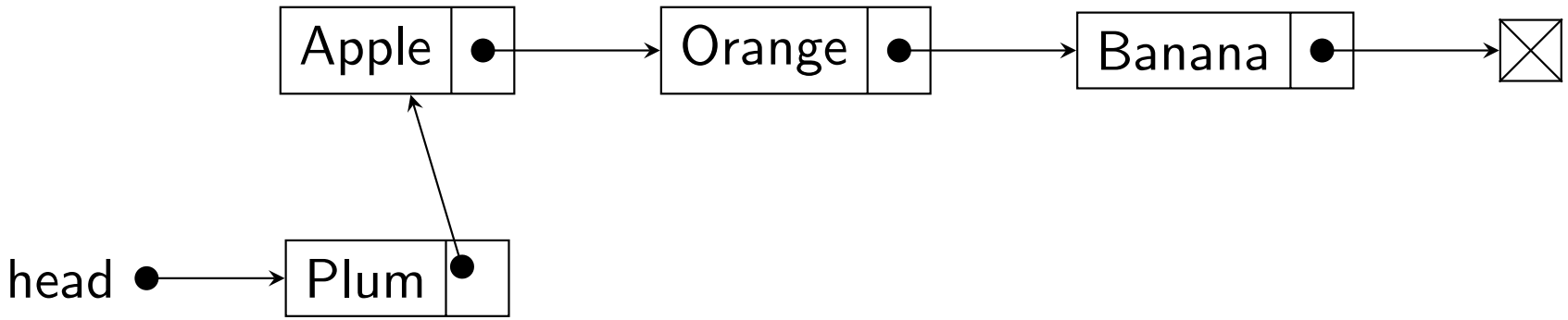


Linked List Operations: Inserting a node (at the head)



New node pointer changed from None/null to address of head element

Linked List Operations: Inserting a node (at the head)



Head moved to point at new node

Linked List Operations: Inserting a node (at the head)

Algorithm add_first

Input: L a Linked-list, e an element, *head* a pointer to the first node

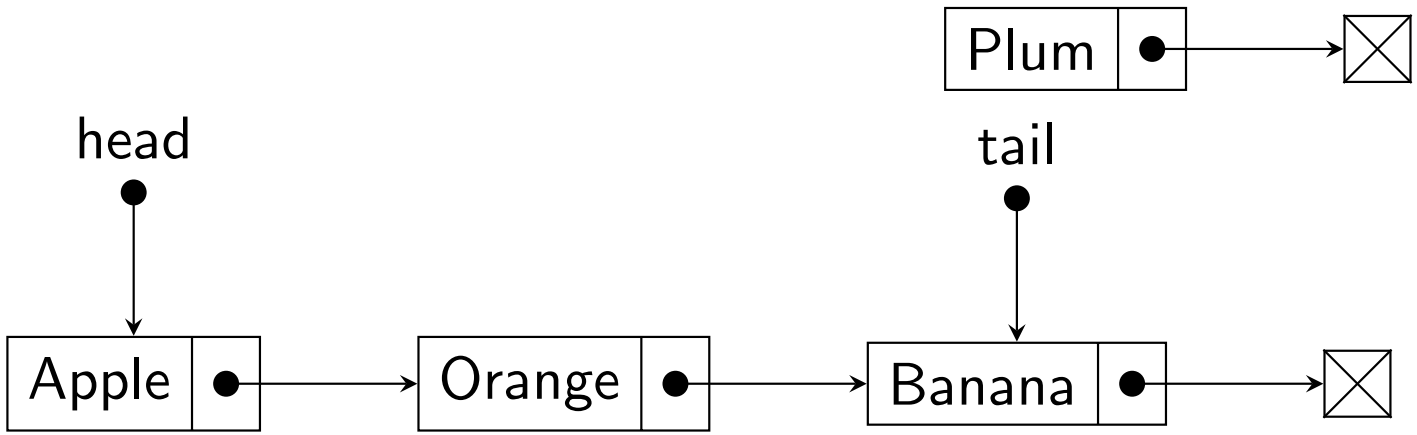
Output: e is added at the head

$newest \leftarrow e$

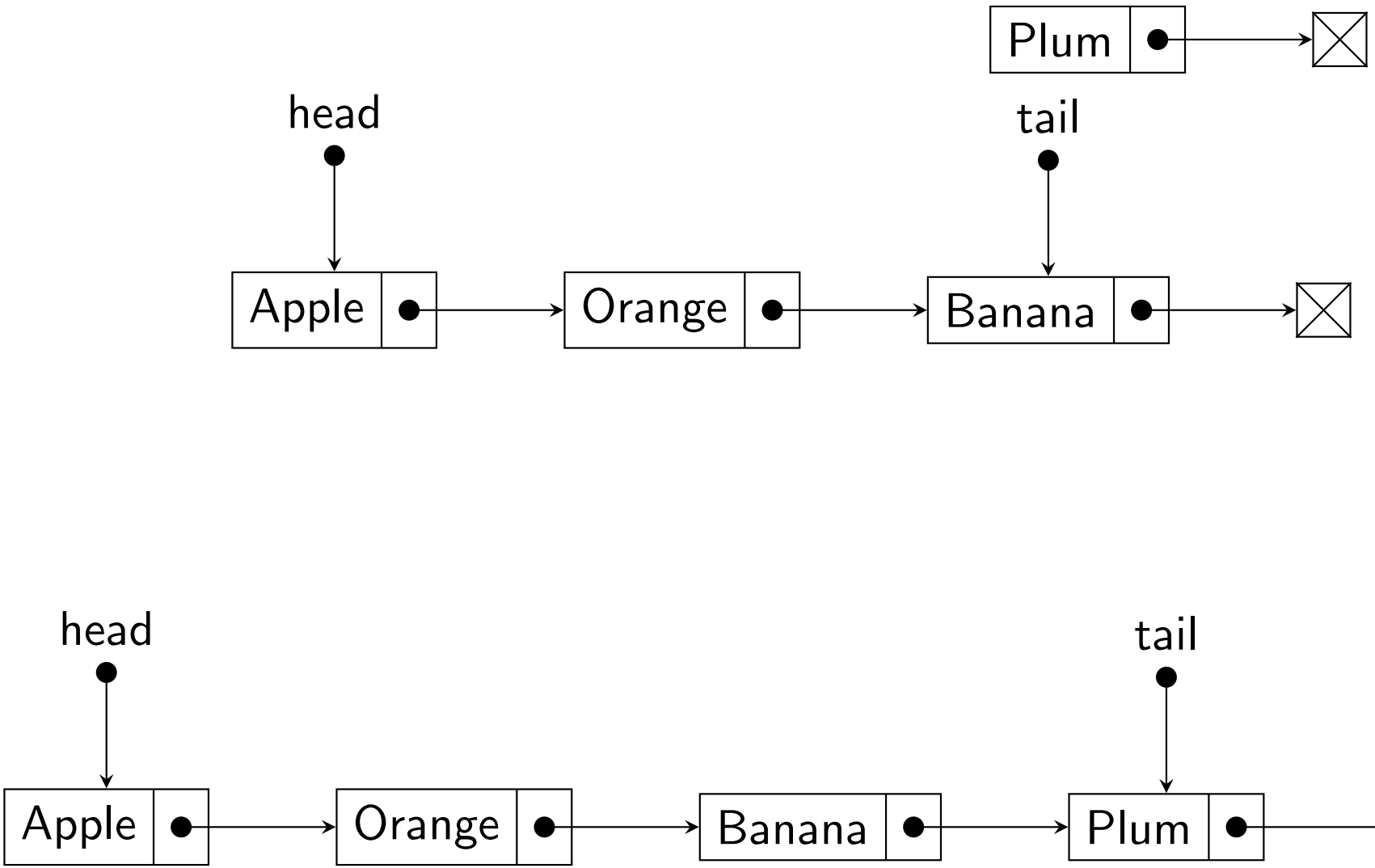
$newest.next \leftarrow L.head$

$L.head \leftarrow newest$

Linked List Operations: Inserting a node (at the tail)



Linked List Operations: Inserting a node (at the tail)



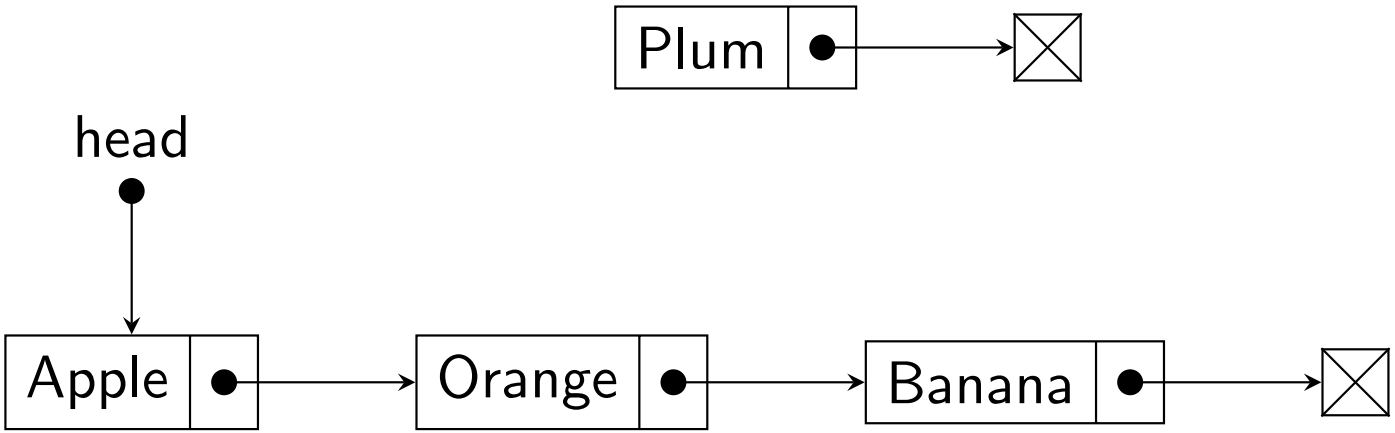
Linked List Operations: Inserting a node (at the tail)

Algorithm add_last

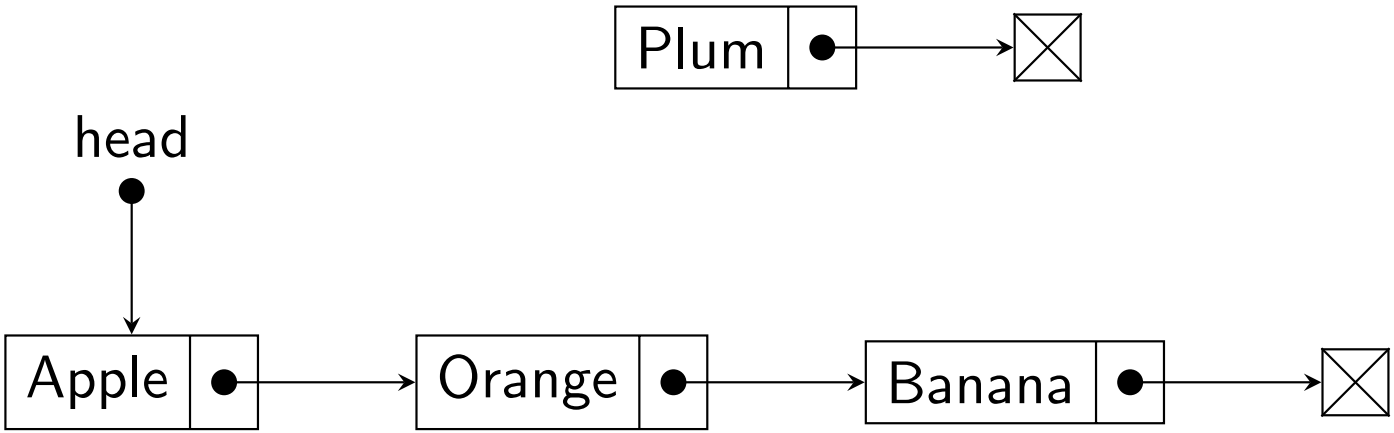
Input: L a Linked-list, e an element, *tail* a pointer to the last node
Output: e is added at the tail

$newest \leftarrow e$
$newest.next \leftarrow None$
$L.tail.next \leftarrow newest$
$L.tail \leftarrow newest$

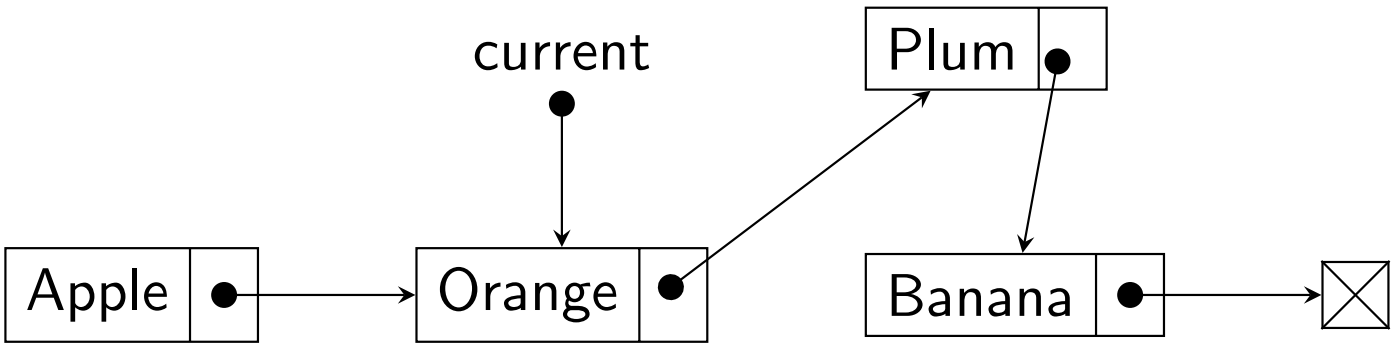
Linked List Operations: Inserting a node (in the middle)



Linked List Operations: Inserting a node (in the middle)



Move `current` from `head` to rank; set new `next` to current `next`; and current `next` to new node address



Linked List Operations: Inserting a node (in the middle)

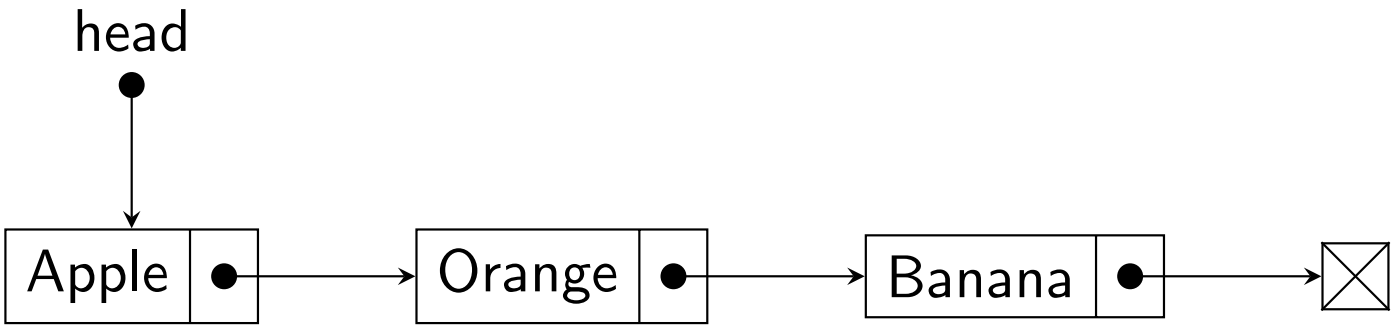
Algorithm add

Input: L a Linked-list, e an element, *head* a pointer to the first node, r the rank where the node is to be inserted

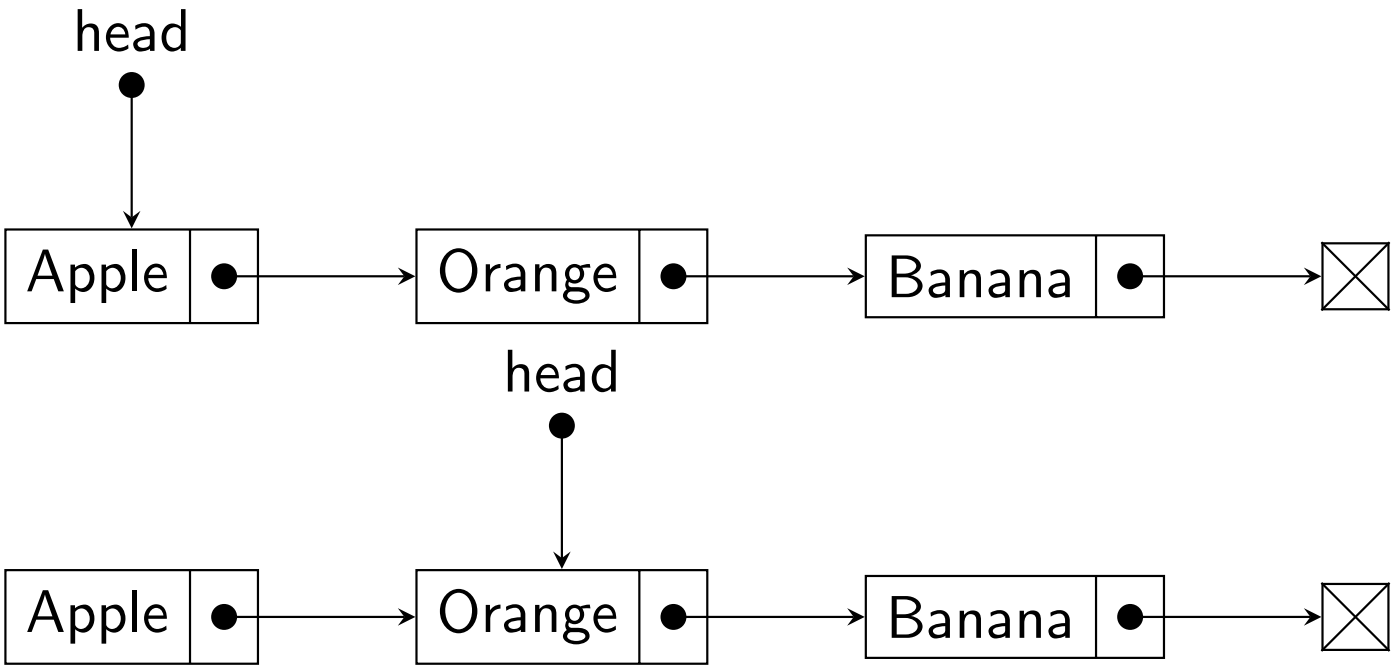
Output: e is added at rank r

```
if r > 0 then
    newest ← e
    current ← head
    while current.next ! = None/null and r > 1 do
        current ← current.next
        r ← r − 1
    end while
    newest.next ← current.next
    current.next ← newest
else
    add_head(L, e)
end if
```

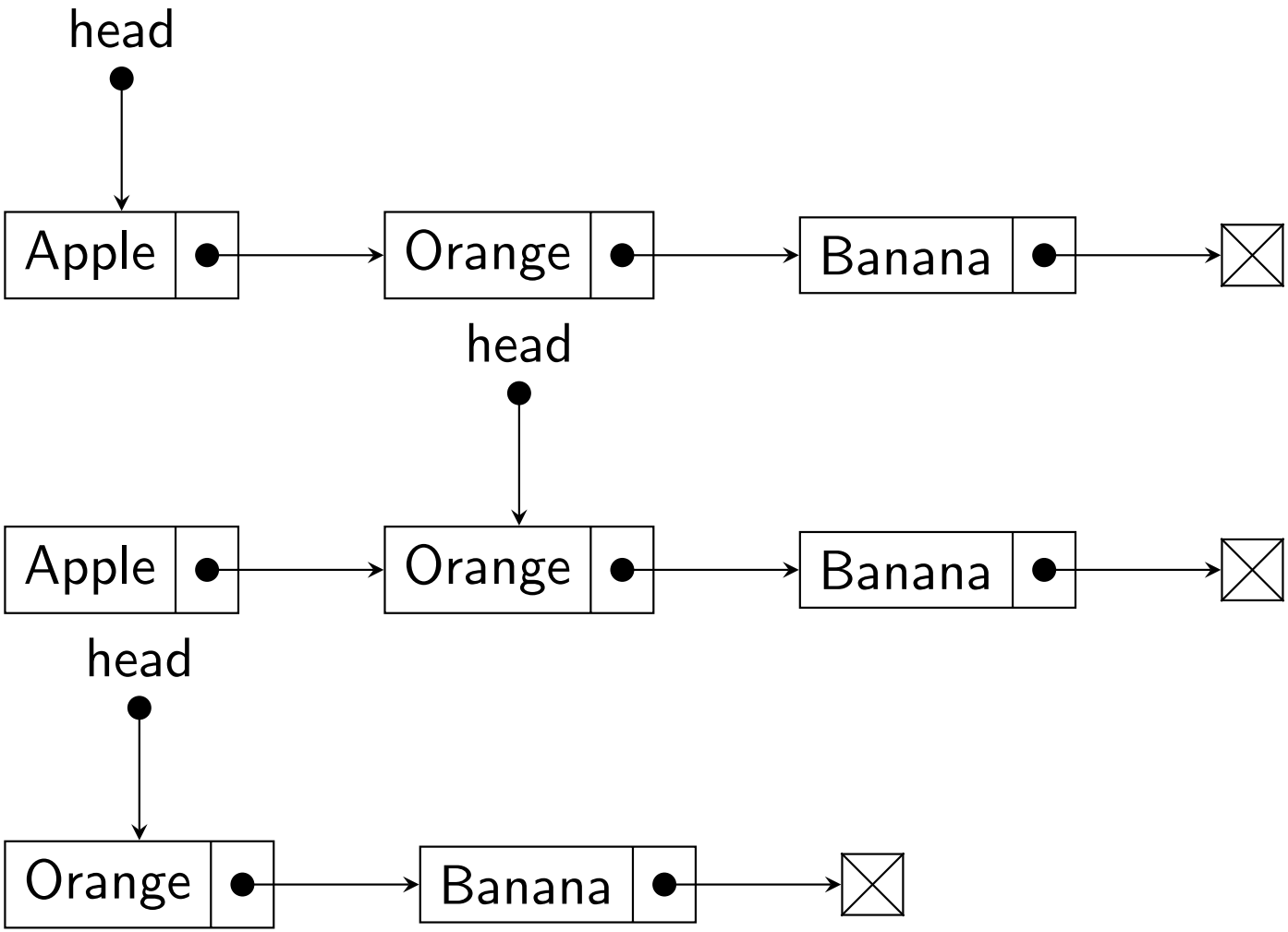
Linked List Operations: Removing a node (at the head)



Linked List Operations: Removing a node (at the head)



Linked List Operations: Removing a node (at the head)



Linked List Operations: Removing a node (at the head)

Algorithm remove_first

Input: *L* a Linked-list, *head* a pointer to the first node
Output:

```
if L.head = None then
    Error :- L is empty
end if
L.head ← L.head.next
```

Linked List Operations: Removing a node (at the tail)

A messy and expensive process!

- Unchain the last node (make it garbage) by pointing the second last node to None/null and making it last
- You cannot go backwards in a linked list (only forwards, i.e. next), so you need to work through the list and keep track of where you are with a pointer.
- Often not implemented

Complexity Analysis: Array vs Linked List

Operations	Array	Linked List
`size, is_empty`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`get_elem_at_rank`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`set_elem_at_rank`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`insert_element_at_rank`	$\mathcal{O}(n)$	$\mathcal{O}(1)^*$
`remove_element_at_rank`	$\mathcal{O}(n)$	$\mathcal{O}(1)^*$
`insert_first, insert_last`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`insert_after, insert_before`	$\mathcal{O}(n)$	$\mathcal{O}(1)$

* "search +": Don't count traversal twice. i.e. If we are already at the rant then insert is $\mathcal{O}(1)$ otherwise it is `get_elem_at_rank`+`insert_element_at_rank`.

Take home?

Linked lists for flexible data

Arrays for fast access