CSC 225

Algorithms and Data Structures I

Rich Little

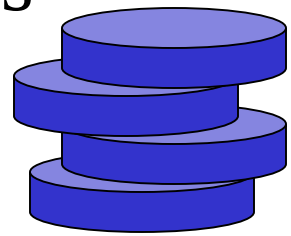
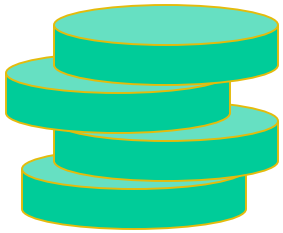
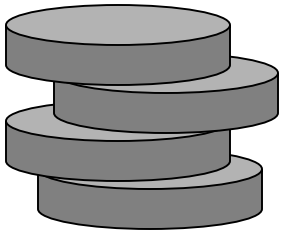
Basic Data Structures

- Arrays
- Stacks
- Queues
- Lists

X	12	3	7	24	4	1	1
A	12	7.5	7.3	11.5	10	8.5	7.4

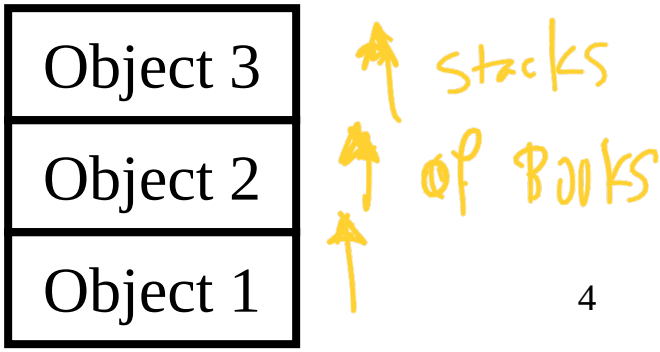
Abstract Data Type (ADT)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

Stacks

- Container of objects that are inserted and removed following the LIFO principle
  - **LIFO = last-in first-out**
- Only the most recently inserted object can be removed at *any* time.
- Earlier inserted objects can only be removed if all objects that are inserted at a later time are already removed from the stack.



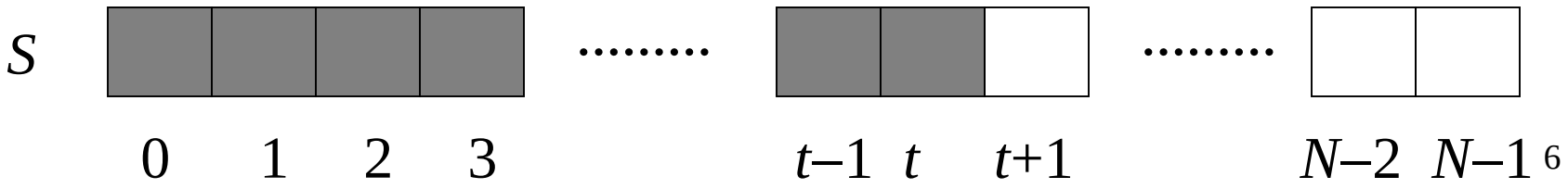
The Stack Abstract Data Type

A ***stack,*** *S*, is an abstract data type (ADT) supporting the following methods:

- **push(*o*):** Insert object *o* at the top of the stack
- **pop():** Remove from the stack and return the top object on the stack (that is, the most recently inserted element still in the stack); an error occurs if the stack is empty.

An Efficient Implementation of a Stack: The Simple Array-Based Stack

- S: N-element array, with elements stored from $S[0]$ to $S[t]$
- t: stack pointer; integer that gives the index of the top element in S
- N: specified max stack size



Main Methods of a Stack: push(*o*)

```
Algorithm push(o):
  if t + 1 = N then
     return stack is full error
  t ← t + 1
  S[t] ← o
  return
end
```

What is the worst-case run-time?

Main Methods of a Stack: pop()

```
Algorithm pop():
  if t < 0 then
     return stack is empty error
  o ← S[t]
  t ← t – 1
  return o
end
```

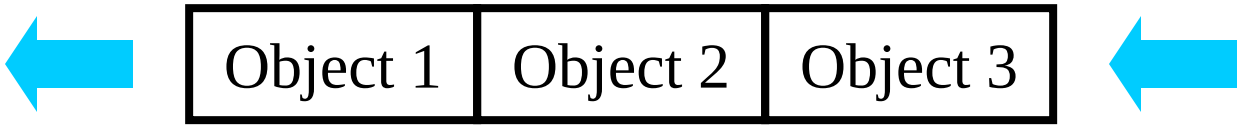
What is the worst-case run-time?

Array-Based Implementations of a Stack: Advantages and Disadvantages

- Simple
- Efficient: $O(1)$ per operation
- The stack *must* assume a fixed upper bound N
- Memory might be wasted or a stack-full error can occur!
- If good estimate for stack size is known: Array is the best choice!!

Queues

- Container of items that are inserted and removed following the FIFO principle
  - **FIFO = first-in first-out**
- Next up is always the item that has been in the queue the longest
- Insertion is possible at any time
- Later inserted objects can only be removed if all objects that are inserted at an earlier time are already removed from the queue.



The Queue Abstract Data Type

A ***queue***, *Q*, is an abstract data type (ADT) supporting the following methods:

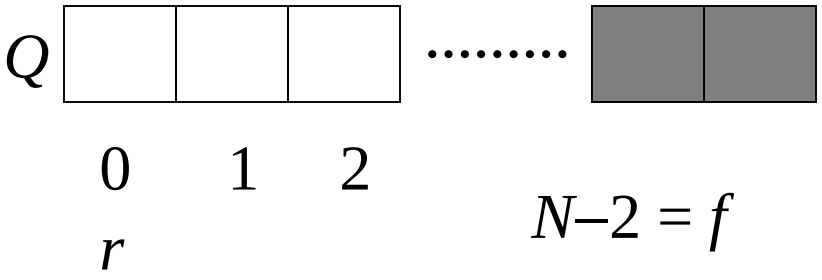
- **enqueue(*o*):** Insert object *o* at the rear of the queue
- **dequeue():** Remove and return from the queue the object at the front; an error occurs if the queue is empty

An Efficient Implementation of a Queue: The Simple Array-Based Queue

- Q: N-element array
- f: index to the cell of Q storing the first element of Q (init is $f = 0$), unless the queue is empty ($f = r$)
- r: index to the next available array cell in Q (init is $r = 0$)

The Simple Array-Based Queue: "wrap around"

- What if for example $f = N-2$? How many elements can be stored in Q?



Count modulo N!

$$x \bmod y = x - \lfloor x / y \rfloor y, y \neq 0$$

Another problem

- What happens if we enqueue N objects without any dequeuing?

We obtain $f = r$! (Which implies that the queue is empty)

Main Methods of a Queue: enqueue(*o*)

```
Algorithm enqueue(o):
  if (N−f+r) mod N = N−1 then
     return queue is full error
  Q[r] ← o
  r ← (r + 1) mod N
  return
end
```

What is the worst-case run-time?

Main Methods of a Queue: dequeue()

```
Algorithm dequeue():
  if f = r then
     return queue is empty error
  temp ← Q[f]
  f ← (f + 1) mod N
  return temp
end
```

What is the worst-case run-time?

Array-Based Implementations of a Queue: Advantages and Disadvantages

- Simple
- Efficient: $O(1)$ per operation
- The queue has a fixed upper bound N (for N–1 elements in a full queue)
- If a good estimate for the size of the queue is known: an array is the best choice!

The List ADT

- The List ADT models a sequence of positions storing arbitrary objects linearly
- Allow access to all elements in the list
  - Not just the endpoints
- Relative referencing or absolute referencing like using an index in an array (or rank in a vector)
  - Linked-lists vs. index-based lists

Index-Based Lists

- Suppose we are given a linear sequence, S, containing n elements.
- We can uniquely refer to each element e using a rank (index) in the range [0, n-1].
  - Equals the number of elements before e.
- Supports the following methods:
  - **get(r):** Return the element of S with index r.
  - **set(r, e):** Replace with e the element at r. Returns previous element.
  - **add(r, e):** Insert a new element e into S at index r.
  - **remove(r):** Remove from S the element at index r.

The List ADT as an Array

- If we use an array, *A*, to implement a list, we use the index as the position.
  - That is, keep track of current index
- Thus, references to elements are done through the current index position
- Let *N* be the maximum size of the array
- To implement **get(*r*)** we simply return *A*[*r*].

Algorithm add(*r*, *e*) - Array

```
Algorithm add(r,e):
  if n = N then
     return "Array is full"
  if r < n then
     for i ← n-1 to r do
          A[i+1] ← A[i]
  A[r] ← e
  n ← n+1
end
```

What is the worst-case run-time?

Algorithm remove(*r*) - Array

```
Algorithm remove(r):
  e ← A[r]
  if r < n-1 then
     for i ← r to n-2 do
          A[i] ← A[i+1]
  n ← n-1
  return e
end
```

What is the worst-case run-time?

Analysis of Index-Based Lists

Running times of the index-based methods:

Method	**Time**
get(r)	$O(1)$
set(r,e)	$O(1)$
add(r,e)	$O(n)$
remove(r)	$O(n)$

e = new element

r = position of element in array.

Linked Lists

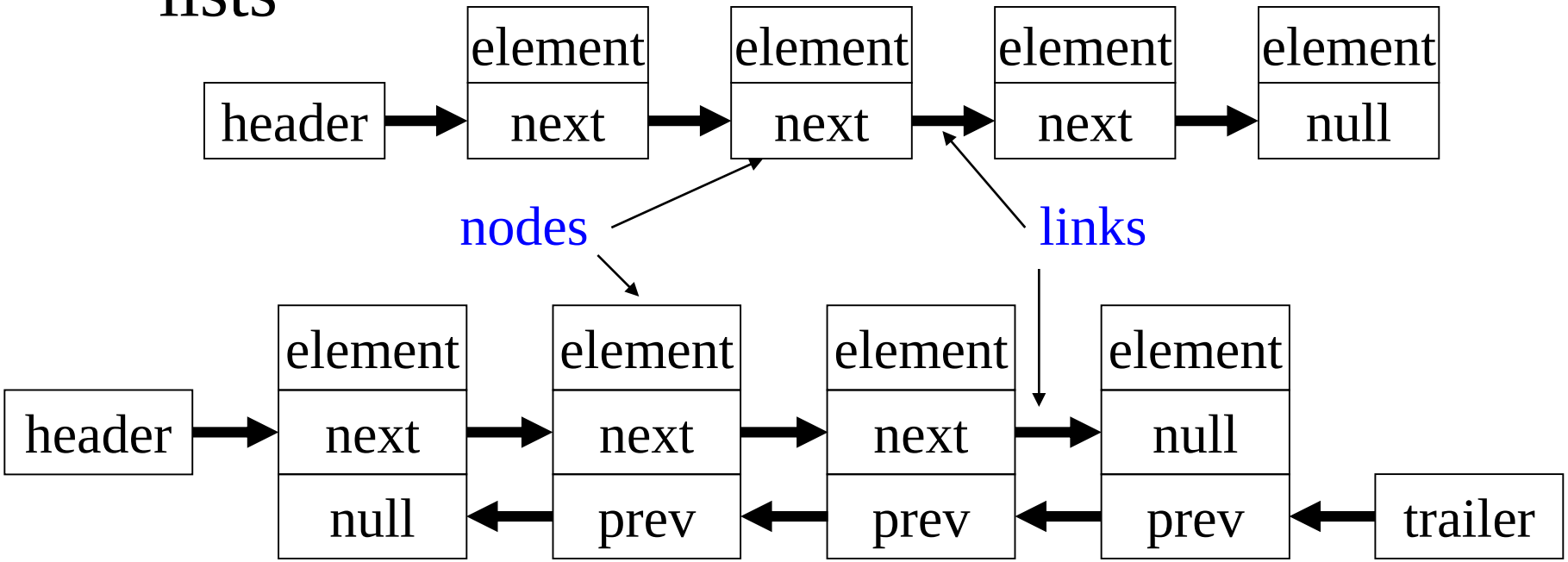
- Each element is stored in a special node object, with pointers to nodes before and after it.
- The nodes themselves represent the place in the list.
- A position is itself an object that supports:
  - **element():** Return the element at current position.
- Other list methods supported in a linked list:
  - **first():** Return position of 1st element of *S*
    last(): Return position of last element of *S*
    before(*p*): Return position of the element of *S* preceding the one at position *p* (error occurs if *p* is 1st element)
  - **after(*p*):** Return position of the element of *S* following the one at position *p* (error occurs if *p* is last element)

The List ADT Linked List

- We can also include the following update methods
  - **insertBefore(*p*, *e*):** Insert a new element *e* into *S* before position *p*
  - **insertAfter(*p*, *e*):** Insert a new element *e* into *S* after position *p*
  - **remove(*p*):** Remove from *S* the element at position *p*

Singly and Doubly Linked Lists

- A ***position*** of an element is defined ***relatively*** (i.e., in terms of its neighbors)
- Special ***header*** and ***trailer*** nodes are added to allows easier use and analysis of linked-lists



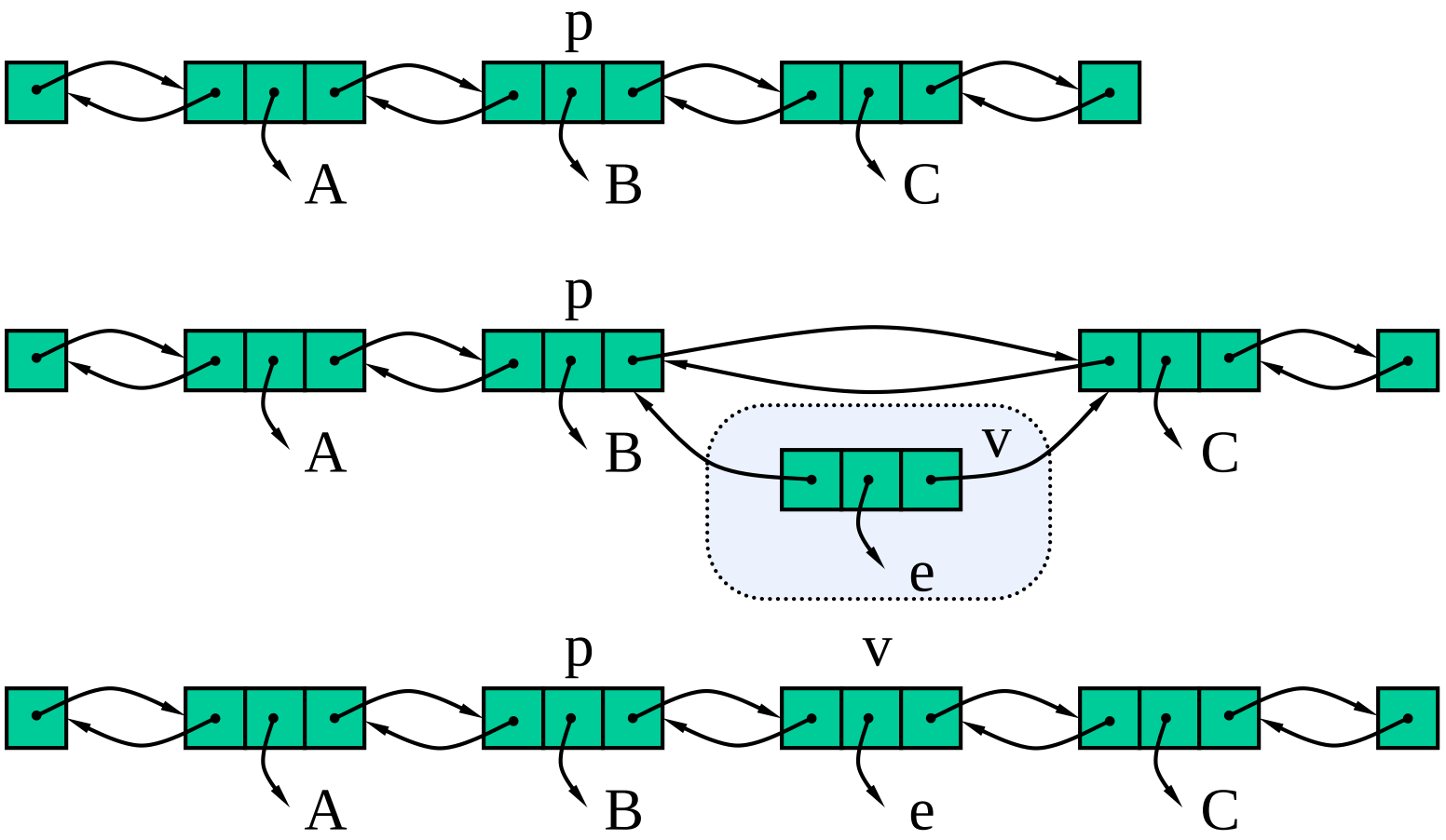
Algorithm insertAfter(*p*, *e*) - Doubly linked

Algorithm insertAfter(*p*, *e*):

```
v ← new Node()
v.element ← e
v.prev ← p
v.next ← p.next
(p.next).prev ← v
p.next ← v
return v
end
```

Insertion

- We visualize operation insertAfter(*p*, *e*), which returns position *v*



Algorithm remove(*p*) - Doubly linked

Algorithm remove(*p*):

$t \leftarrow p$.element

$(p$.prev$)$.next $\leftarrow p$.next

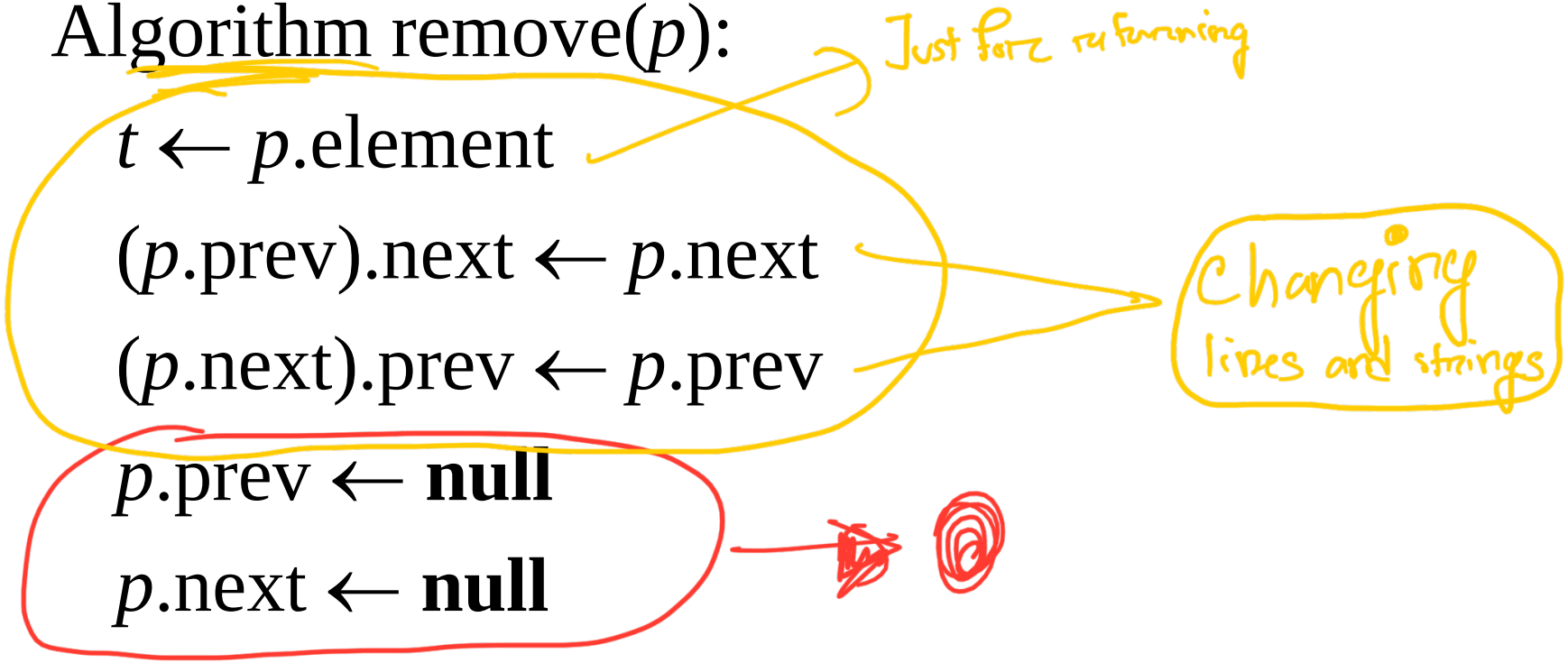
$(p$.next$)$.prev $\leftarrow p$.prev

p.prev $\leftarrow$ **null**

p.next $\leftarrow$ **null**

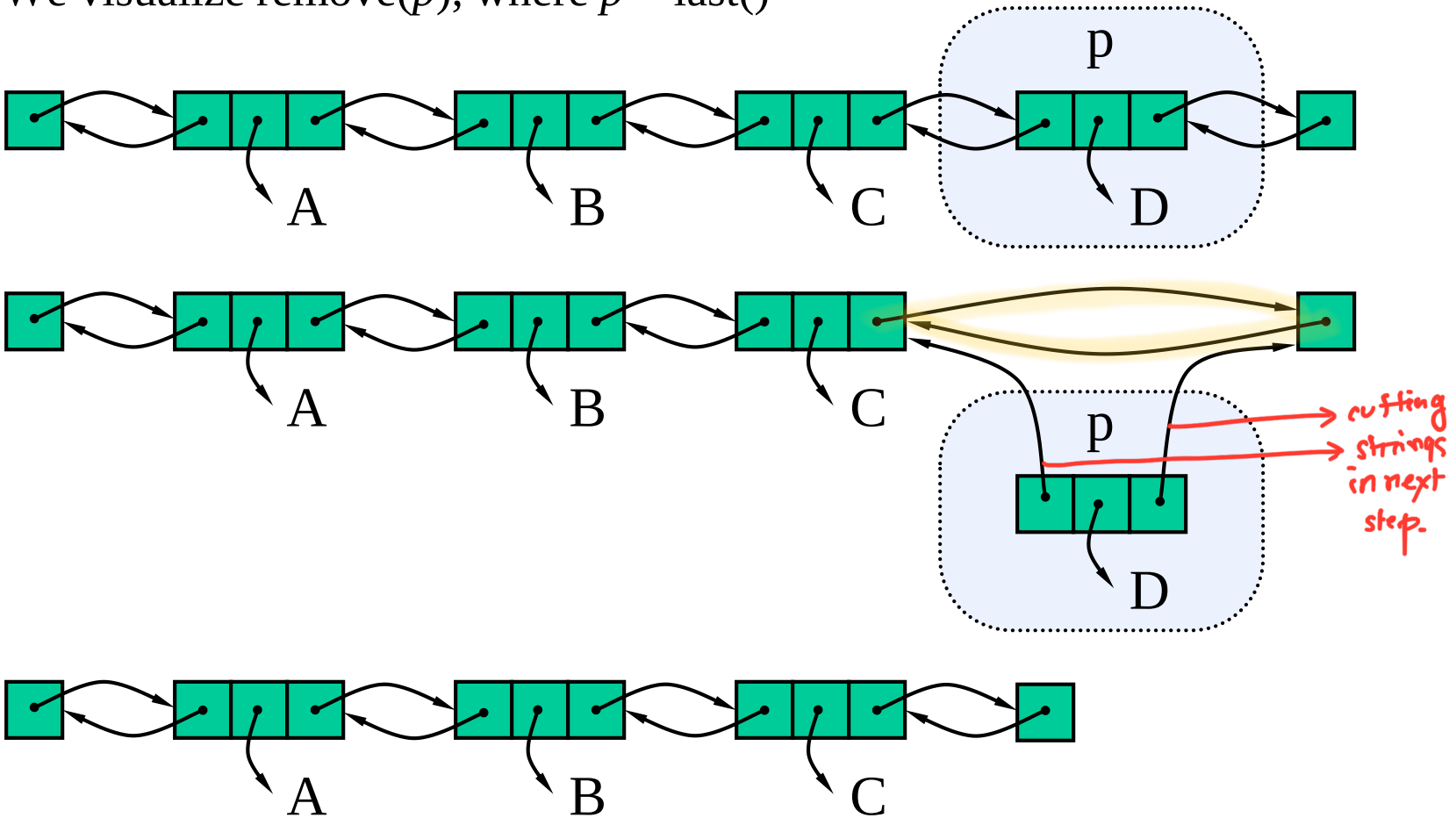
return *t*

end



Deletion

- We visualize remove(*p*), where *p* = last()



Running Times

- first(): O(1)
- last(): O(1)
- before(*p*): O(1)
- after(*p*): O(1)
- element(): O(1)

- insertBefore(*p*, *e*): O(1)
- insertAfter(*p*, *e*): O(1)
- remove(*p*): O(1)