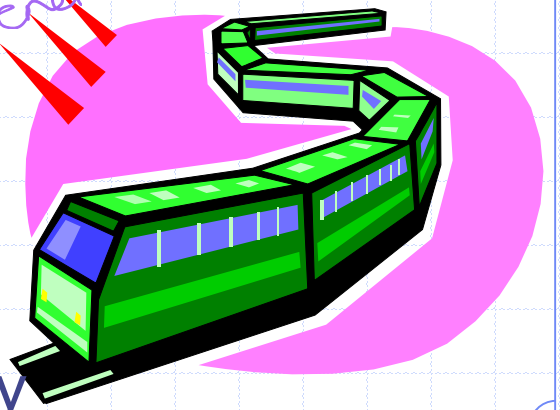
Link list→can travel in only 1 direction

Can insert/remove nodes at any time
- won't fall apart because of references
- can't find items with index- have to go one by one

Doubly-Linked Lists

↳can travel in both directions
↳address for next & previous
↳can insert/delete at either end
↳only 1 operation?

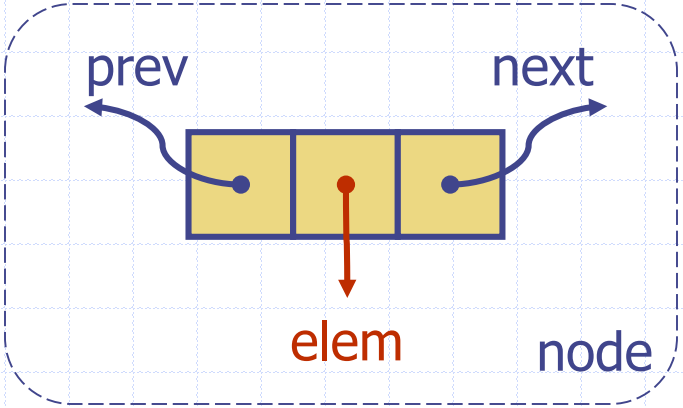


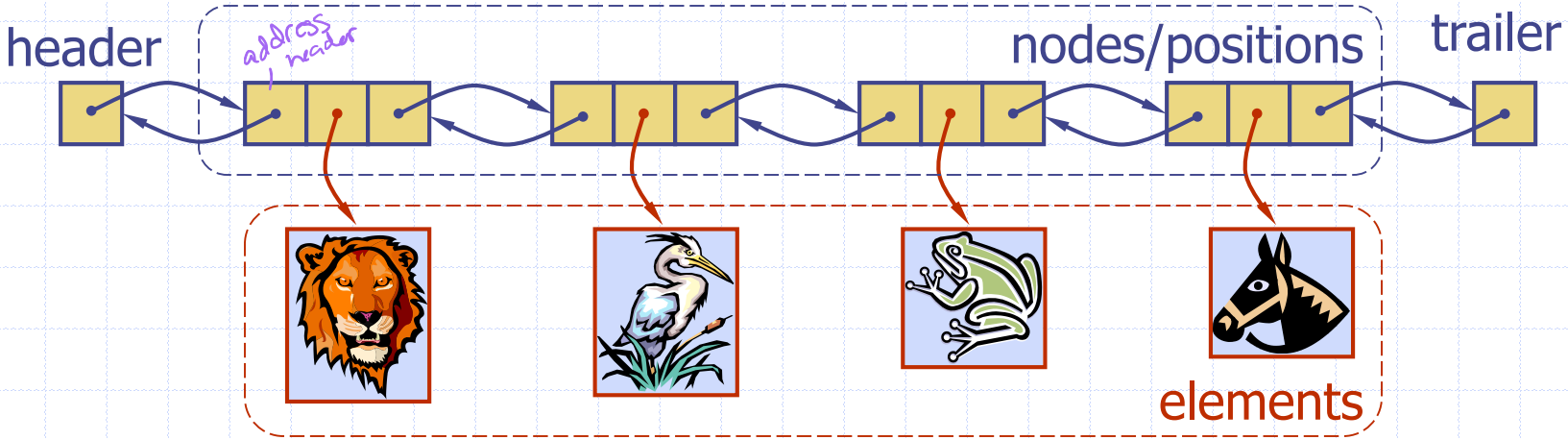
Sareh Taebi
COP3410 – Florida Atlantic University

Doubly Linked List

- A doubly linked list provides symmetry
- Nodes implement Position and store:
  - element
  - link to the previous node → add to node class
  - link to the next node
- Special trailer and header nodes

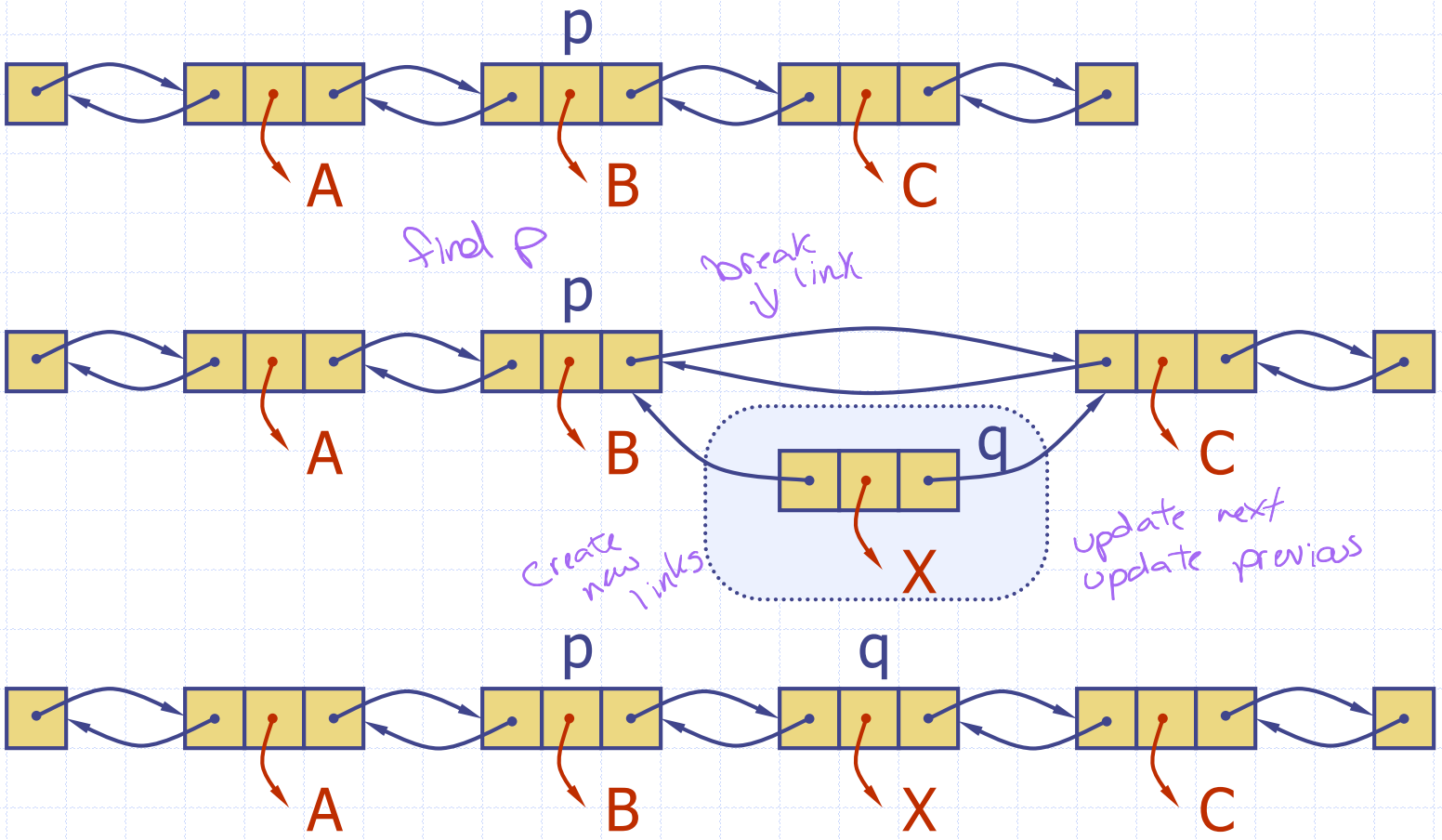
add node method, remove node method





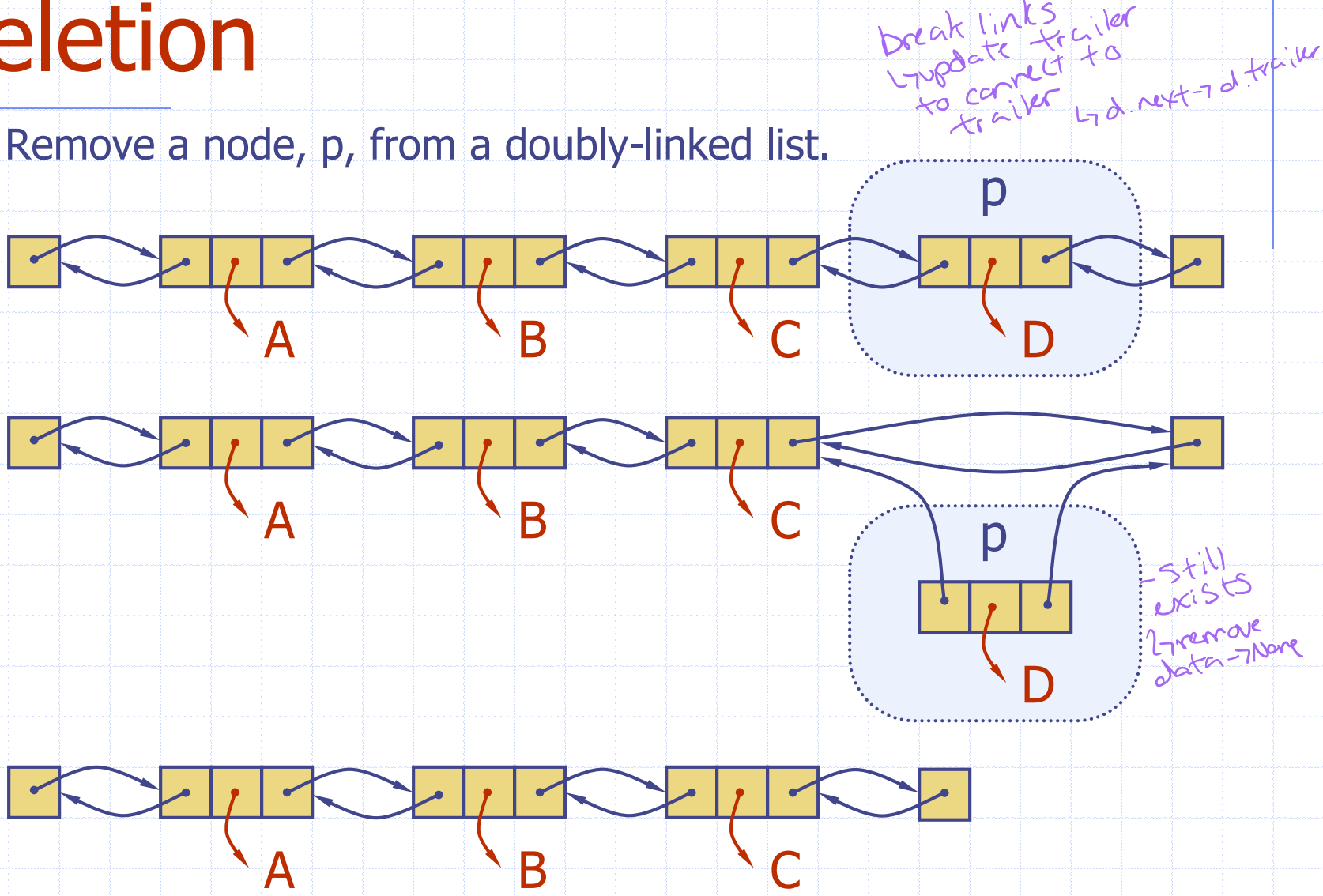
Insertion

- Insert a new node, q, between p and its successor.



Deletion

- Remove a node, p, from a doubly-linked list.



Doubly-Linked List in Python

```
1  class _DoublyLinkedBase:
2    """A base class providing a doubly linked list representation."""
3
4    class _Node:
5      """Lightweight, nonpublic class for storing a doubly linked node."""
6      (omitted here; see previous code fragment)
7
8    def __init__(self):
9      """Create an empty list."""
10     self._header = self._Node(None, None, None)
11     self._trailer = self._Node(None, None, None)
12     self._header._next = self._trailer                # trailer is after header
13     self._trailer._prev = self._header                # header is before trailer
14     self._size = 0                                    # number of elements
15
16   def __len__(self):
17     """Return the number of elements in the list."""
18     return self._size
19
20   def is_empty(self):
21     """Return True if list is empty."""
22     return self._size == 0
23
```

```
24   def _insert_between(self, e, predecessor, successor):
25     """Add element e between two existing nodes and return new node."""
26     newest = self._Node(e, predecessor, successor)   # linked to neighbors
27     predecessor._next = newest
28     successor._prev = newest
29     self._size += 1
30     return newest
31
32   def _delete_node(self, node):
33     """Delete nonsentinel node from the list and return its element."""
34     predecessor = node._prev
35     successor = node._next
36     predecessor._next = successor
37     successor._prev = predecessor
38     self._size -= 1
39     element = node._element                          # record deleted element
40     node._prev = node._next = node._element = None   # deprecate node
41     return element                                   # return deleted element
```

Performance

- In a doubly linked list
  - The space used by a list with n elements is $O(n)$ (more space per node)
  - The space used by each position of the list is $O(1)$
  - All the standard operations of a list run in $O(1)$ time

Deque

- Stack + Queue = Deque
- We can add and remove elements from both the begin and end.
- Like a deck of cards
- Implementation can be done using the double linked list base class

Deque Implementation

```
1  class LinkedDeque(_DoublyLinkedBase):          # note the use of inheritance
2    """Double-ended queue implementation based on a doubly linked list."""
3
4    def first(self):
5      """Return (but do not remove) the element at the front of the deque."""
6      if self.is_empty( ):
7        raise Empty("Deque is empty")
8      return self._header._next._element          # real item just after header
9
10   def last(self):
11     """Return (but do not remove) the element at the back of the deque."""
12     if self.is_empty( ):
13       raise Empty("Deque is empty")
14     return self._trailer._prev._element         # real item just before trailer
15
16   def insert_first(self, e):
17     """Add an element to the front of the deque."""
18     self._insert_between(e, self._header, self._header._next)   # after header
19
20   def insert_last(self, e):
21     """Add an element to the back of the deque."""
22     self._insert_between(e, self._trailer._prev, self._trailer)   # before trailer
23
24   def delete_first(self):
25     """Remove and return the element from the front of the deque.
26
27     Raise Empty exception if the deque is empty.
28     """
29     if self.is_empty( ):
30       raise Empty("Deque is empty")
31     return self._delete_node(self._header._next)       # use inherited method
32
33   def delete_last(self):
34     """Remove and return the element from the back of the deque.
35
36     Raise Empty exception if the deque is empty.
37     """
38     if self.is_empty( ):
39       raise Empty("Deque is empty")
40     return self._delete_node(self._trailer._prev)       # use inherited method
```