

ENG 346

Data Structures and

Algorithms for Artificial

Intelligence

Linked Lists

Dr. Mehmet PEKMEZCİ
mpekmezci@gtu.edu.tr

Linked List

- Organize and store a collection of elements, called nodes, in a linear sequence
- Each node in a linked list has
 - Data: Actual value or information to store in the list.
 - Reference(s): Represents the connection(s) between nodes in the sequence.
- Type of linked lists:
 - Singly linked list
 - Doubly linked list
 - Circular linked list

Advantages of Linked Lists

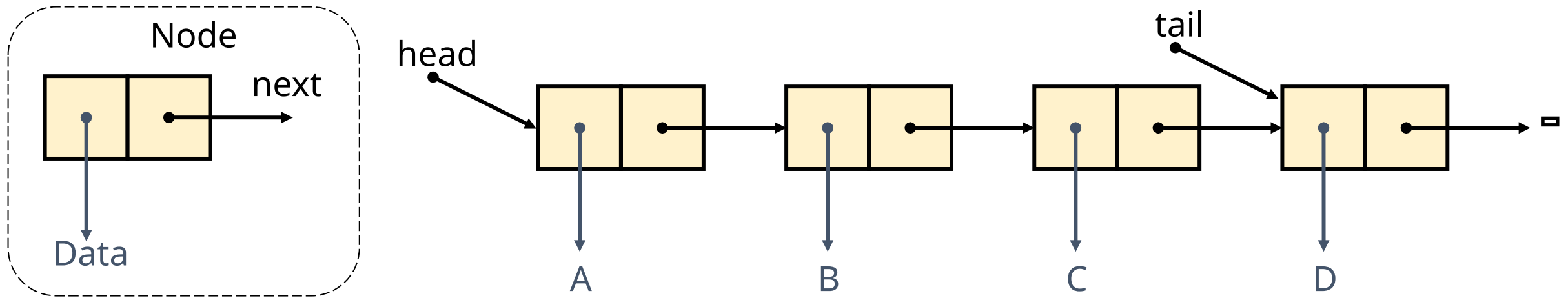
- **Dynamic Size:** Can easily grow or shrink in size by adding or removing nodes.
- **Efficient Insertions and Deletions:** Insertions and deletions at the beginning or middle of a linked list can be done in constant time.
- **Memory Efficiency:** Linked lists use memory more efficiently since they allocate memory for each node as needed.
- **No Need for Pre-allocation:** Linked lists don't require you to specify the size of the list in advance.
- **Constant-Time Insertions at the Head.**
- **Easy Implementation of Other Data Structures** like stacks, queues, and hash tables.

Disadvantages of Linked Lists

- Inefficient Random Access: Accessing an element at a specific index in a linked list can be inefficient, taking $O(n)$ time in the worst case.
- Increased Memory Overhead: Each node in a linked list requires additional memory to store the reference to the next node.
- Slower Traversal: Traversing a linked list can be slower than iterating through an array because it involves following references from one node to the next.

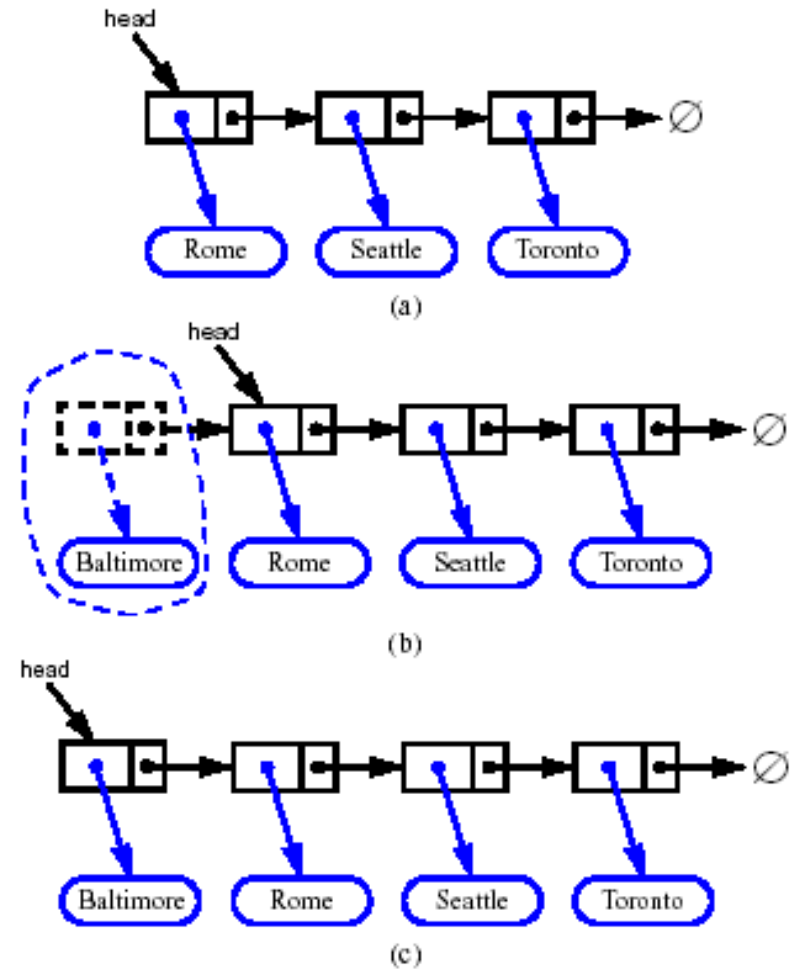
Singly Linked Lists

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer.
- Each node stores
 - Data
 - Link to the next node



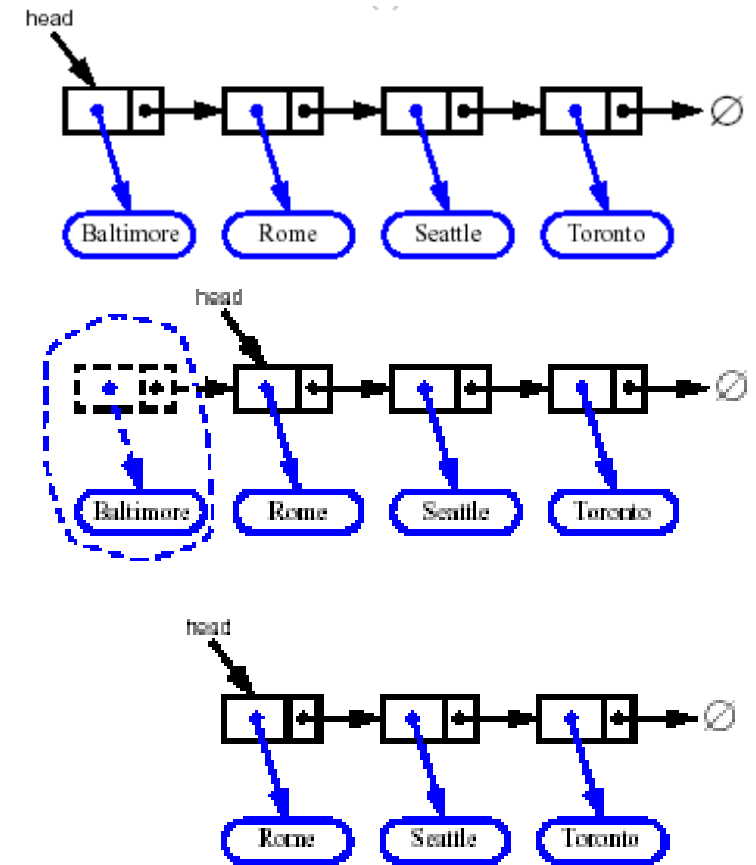
Inserting at the Head

- Allocate a new node
- Insert new element
- Have new node point to old head
- Update head to point to new node



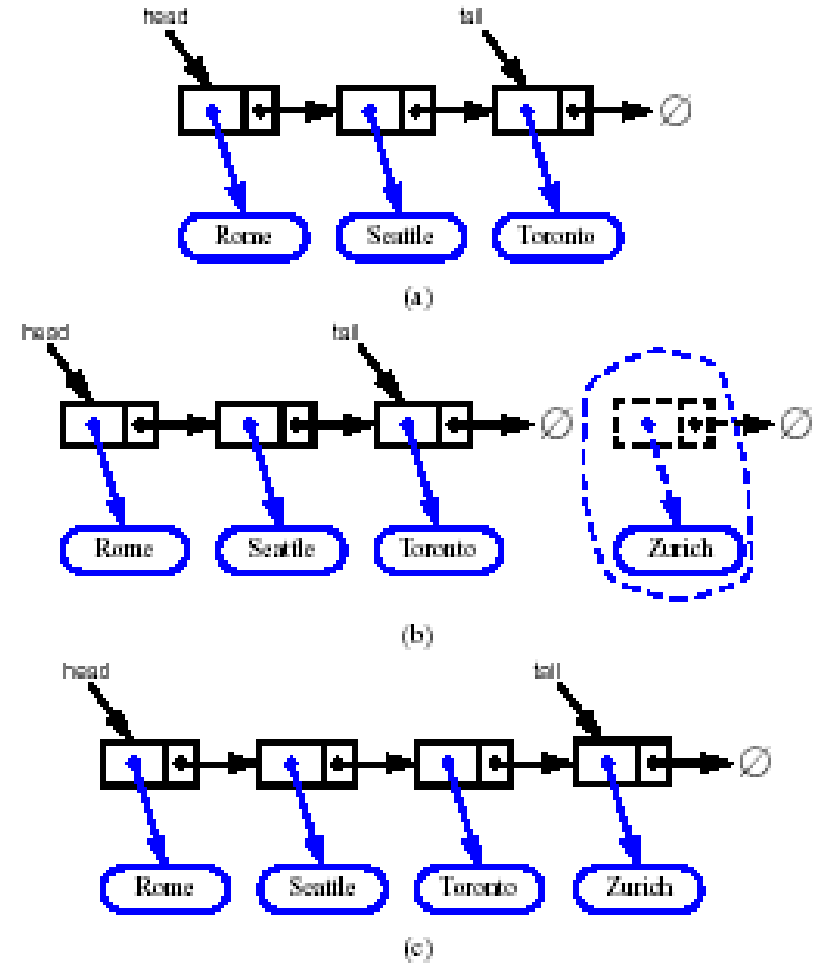
Removing from the Head

- Update head to point to next node in the list
- Allow garbage collector to reclaim the former first node



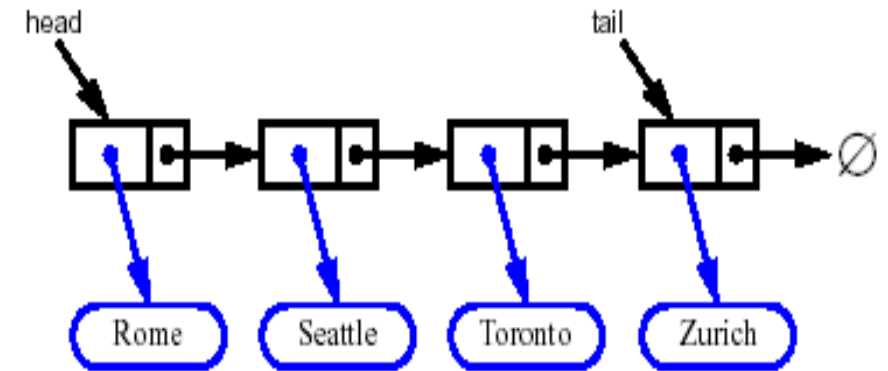
Inserting at the Tail

- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node



Removing from the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



Stack as a Linked List

- Implement a stack with a singly linked list
- The top element is The first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time

Linked-List Stack in Python

```
1 class LinkedStack:
2     """ LIFO Stack implementation using a singly linked list for storage. """
3
4     #----- nested _Node class -----
5     class _Node:
6         """ Lightweight, nonpublic class for storing a singly linked node. """
7         __slots__ = '_element', '_next'      # streamline memory usage
8
9         def __init__(self, element, next):    # initialize node's fields
10             self._element = element          # reference to user's element
11             self._next = next                # reference to next node
12
13     #----- stack methods -----
14     def __init__(self):
15         """ Create an empty stack. """
16         self._head = None                    # reference to the head node
17         self._size = 0                       # number of stack elements
18
19     def __len__(self):
20         """ Return the number of elements in the stack. """
21         return self._size
22
```

```
23     def is_empty(self):
24         """ Return True if the stack is empty. """
25         return self._size == 0
26
27     def push(self, e):
28         """ Add element e to the top of the stack. """
29         self._head = self._Node(e, self._head)    # create and link a new node
30         self._size += 1
31
32     def top(self):
33         """ Return (but do not remove) the element at the top of the stack.
34
35         Raise Empty exception if the stack is empty.
36         """
37         if self.is_empty():
38             raise Empty('Stack is empty')
39         return self._head._element                # top of stack is at head of list

```

```
40     def pop(self):
41         """ Remove and return the element from the top of the stack (i.e., LIFO).
42
43         Raise Empty exception if the stack is empty.
44         """
45         if self.is_empty():
46             raise Empty('Stack is empty')
47         answer = self._head._element
48         self._head = self._head._next            # bypass the former top node
49         self._size -= 1
50         return answer

```

Linked

Queue as a Linked List

- Implement a queue with a singly linked list
 - The front element is the first node
 - The rear element is the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time

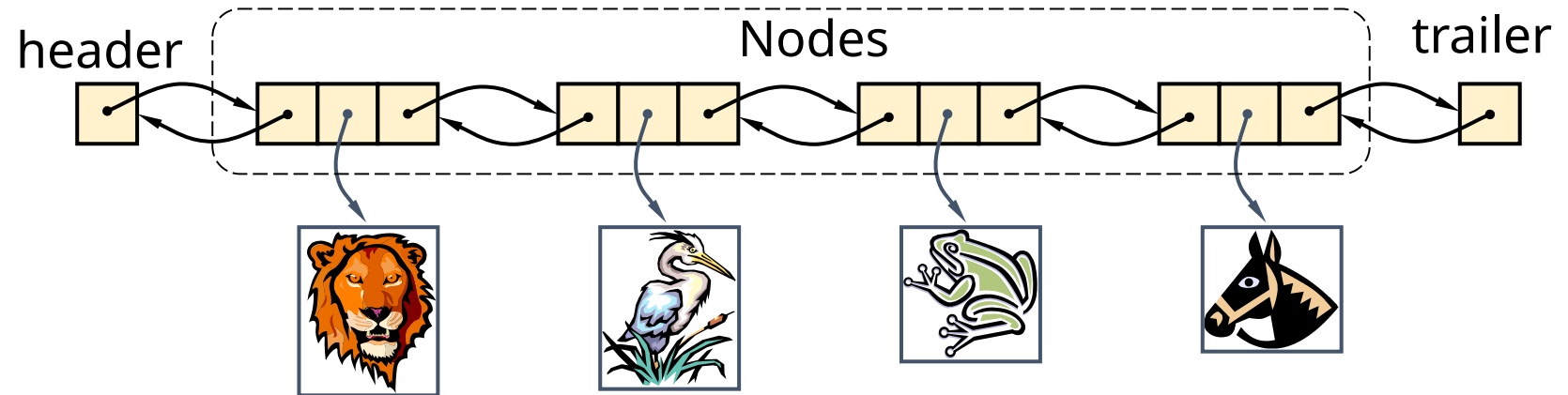
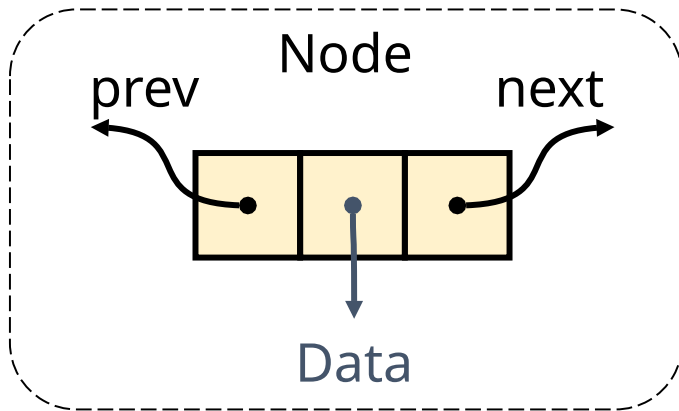
Linked-List Queue in Python

```
1 class LinkedQueue:
2     """FIFO queue implementation using a singly linked list for storage."""
3
4     class _Node:
5         """Lightweight, nonpublic class for storing a singly linked node."""
6         (omitted here; identical to that of LinkedStack._Node)
7
8     def __init__(self):
9         """Create an empty queue."""
10        self._head = None
11        self._tail = None
12        self._size = 0                # number of queue elements
13
14    def __len__(self):
15        """Return the number of elements in the queue."""
16        return self._size
17
18    def is_empty(self):
19        """Return True if the queue is empty."""
20        return self._size == 0
21
22    def first(self):
23        """Return (but do not remove) the element at the front of the queue."""
24        if self.is_empty():
25            raise Empty('Queue is empty')
26        return self._head._element    # front aligned with head of list
```

```
27    def dequeue(self):
28        """Remove and return the first element of the queue (i.e., FIFO).
29
30        Raise Empty exception if the queue is empty.
31        """
32        if self.is_empty():
33            raise Empty('Queue is empty')
34        answer = self._head._element
35        self._head = self._head._next
36        self._size -= 1
37        if self.is_empty():           # special case as queue is empty
38            self._tail = None         # removed head had been the tail
39        return answer
40
41    def enqueue(self, e):
42        """Add an element to the back of queue."""
43        newest = self._Node(e, None)   # node will be new tail node
44        if self.is_empty():
45            self._head = newest        # special case: previously empty
46        else:
47            self._tail._next = newest
48            self._tail = newest        # update reference to tail node
49            self._size += 1
```

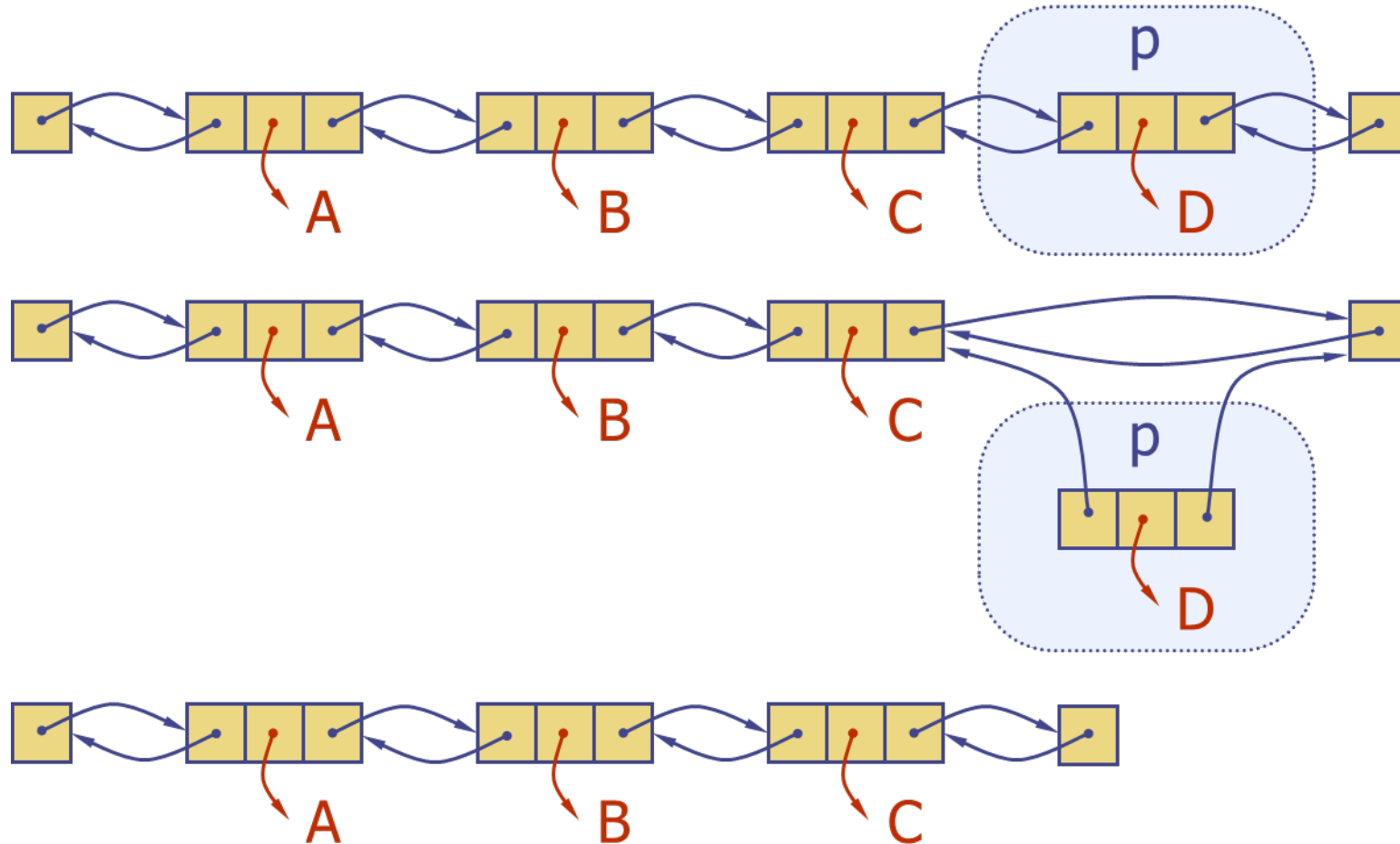
Doubly Linked List

- A linked list which can be traversed in both directions.
- Each Node stores:
 - Data
 - Link to the previous node
 - Link to the next node



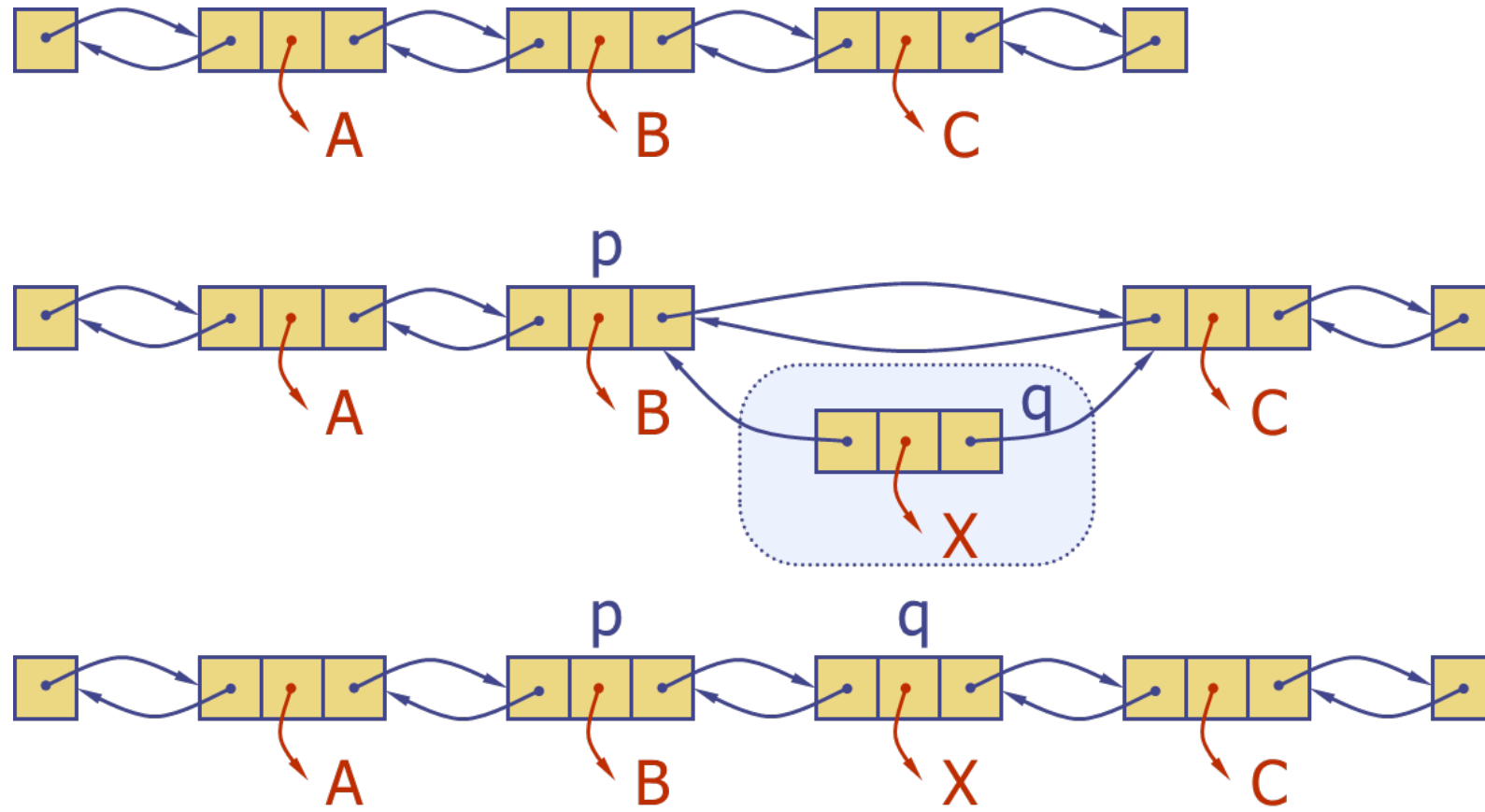
Deletion from Doubly Linked List

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



Insertion to Doubly Linked List

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



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
```

Circular Linked List