ENG 346
Data Structures and Algorithms for Artificial Intelligence
Linked Lists

Dr. Mehmet PEKMEZCİ

mpekmezci@gtu.edu.tr

https://github.com/mehmetpekmezci/GTU-ENG-346

ENG-346-FALL-2025 Teams code is **4b108kr**

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

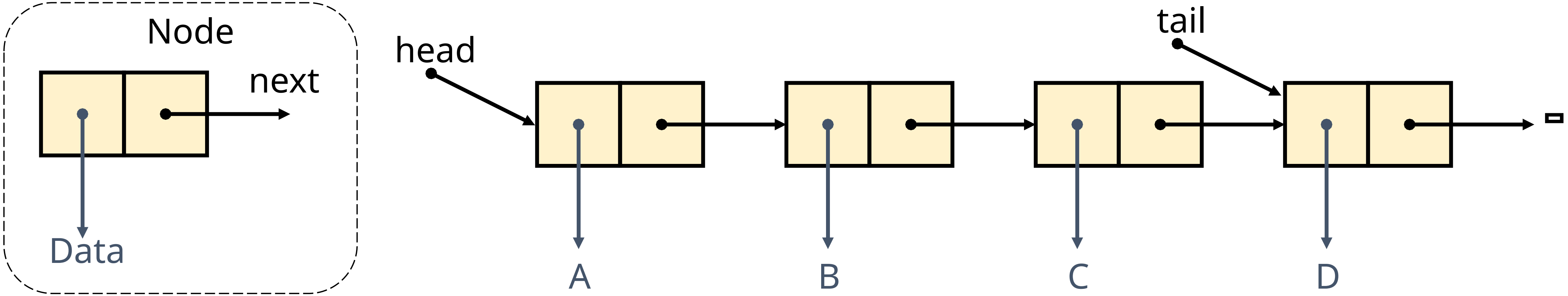
- Dynamic Size: Can easily grow or shrink in size by adding or removing nodes.
- Efficient Insertions and Deletions: Insertions and deletions at the beginning or middle of a linked list can be done in constant time.
- Memory Efficiency: Linked lists use memory more efficiently since they allocate memory 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.

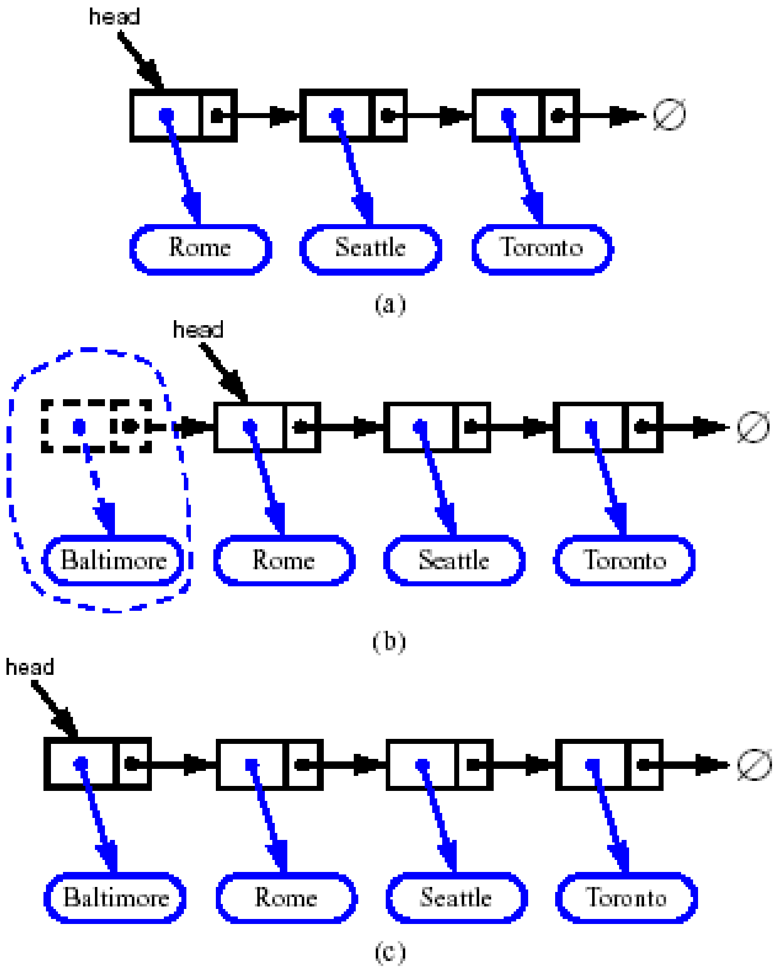
Singly Linked Lists

- 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



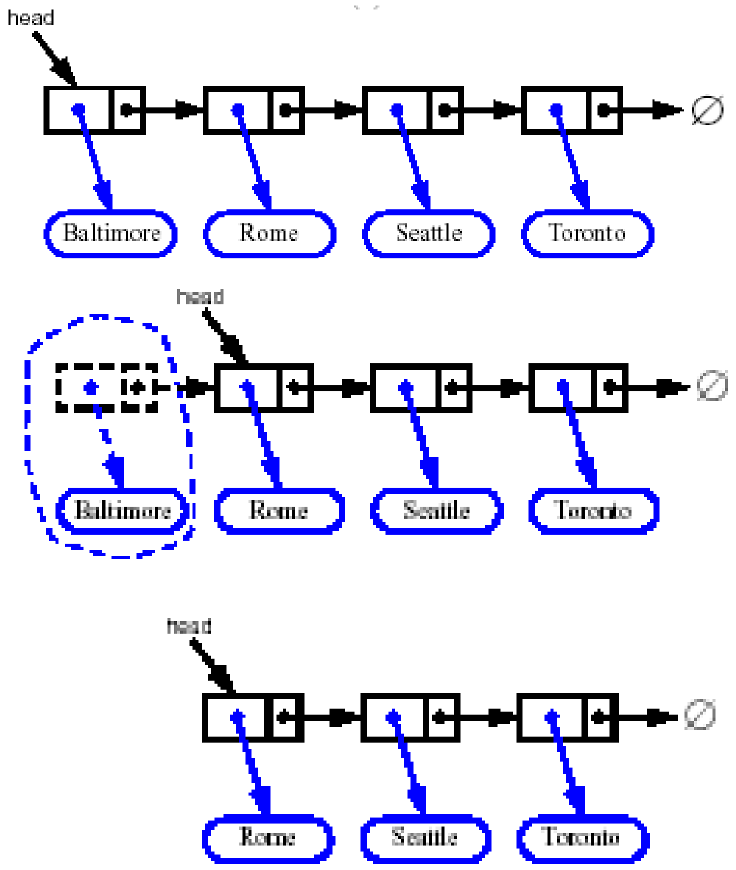
Inserting at the Head

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



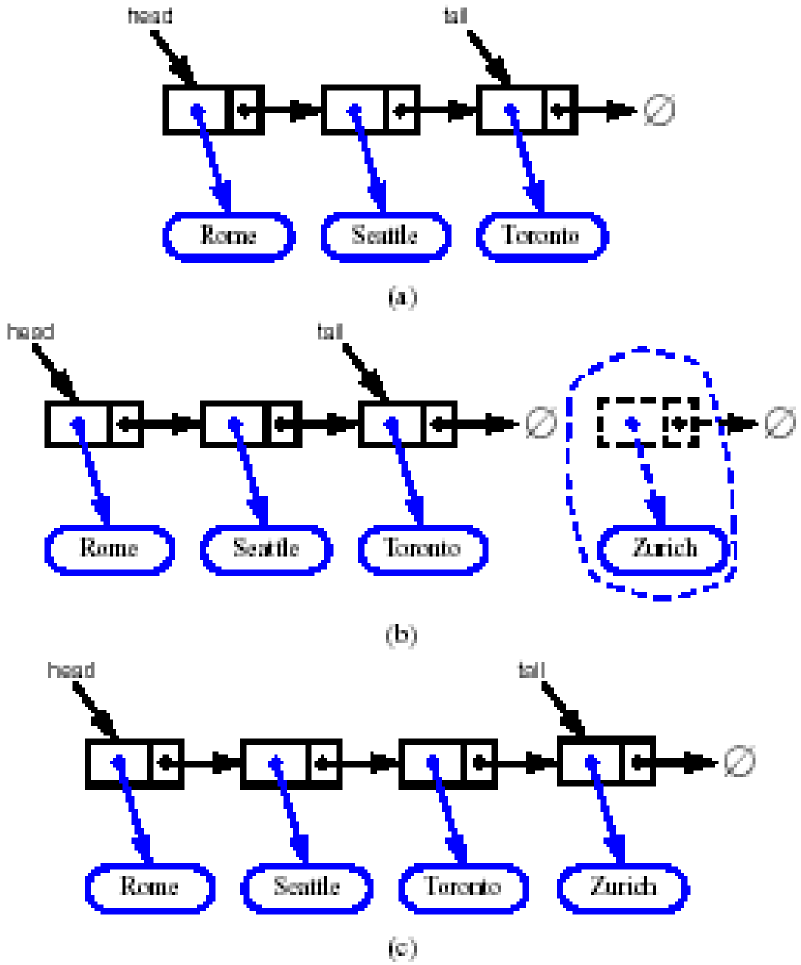
Removing from the Head

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



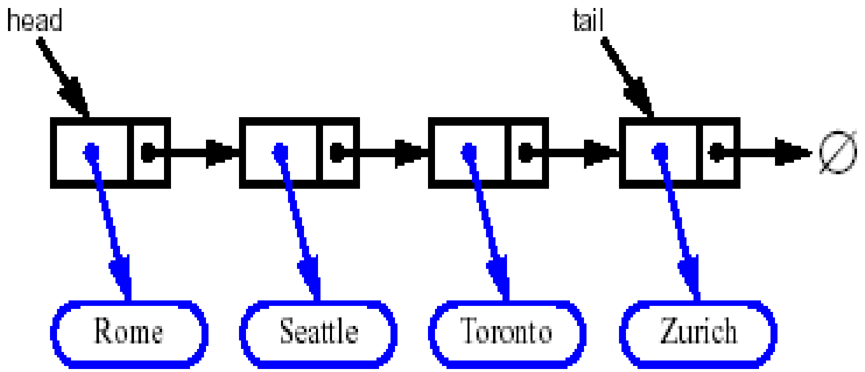
Inserting at the Tail

- 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 ㅍ 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

```python
class LinkedStack:
  """LIFO Stack implementation using a singly linked list for storage."""

  #-------------------------- nested _Node class --------------------------
  class _Node:
    """Lightweight, nonpublic class for storing a singly linked node."""
    __slots__ = '_element', '_next'         # streamline memory usage

    def __init__(self, element, next):      # initialize node's fields
      self._element = element               # reference to user's element
      self._next = next                     # reference to next node

  #------------------------------- stack methods -------------------------------
  def __init__(self):
    """Create an empty stack."""
    self._head = None                       # reference to the head node
    self._size = 0                          # number of stack elements

  def __len__(self):
    """Return the number of elements in the stack."""
    return self._size

  def is_empty(self):
    """Return True if the stack is empty."""
    return self._size == 0

  def push(self, e):
    """Add element e to the top of the stack."""
    self._head = self._Node(e, self._head)  # create and link a new node
    self._size += 1

  def top(self):
    """Return (but do not remove) the element at the top of the stack.

    Raise Empty exception if the stack is empty.
    """
    if self.is_empty():
      raise Empty('Stack is empty')
    return self._head._element              # top of stack is at head of list

  def pop(self):
    """Remove and return the element from the top of the stack (i.e., LIFO).

    Raise Empty exception if the stack is empty.
    """
    if self.is_empty():
      raise Empty('Stack is empty')
    answer = self._head._element
    self._head = self._head._next           # bypass the former top node
    self._size -= 1
    return answer
```

Queue as a Linked List

- Implement a queue with a singly linked list
  - The front element ㅍ the first node
  - The rear element ㅍ 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

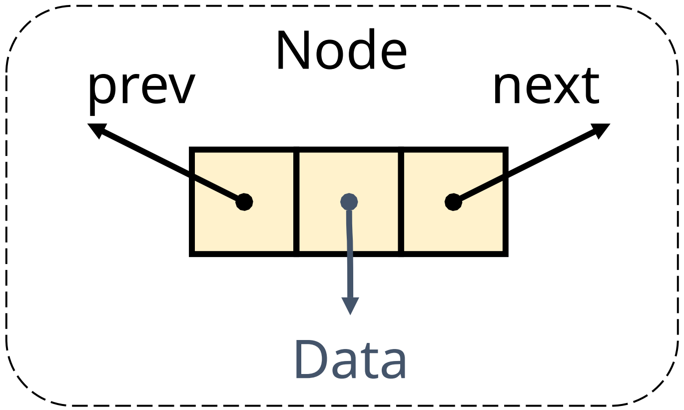
GEBZE TECHNICAL UNIVERSITY

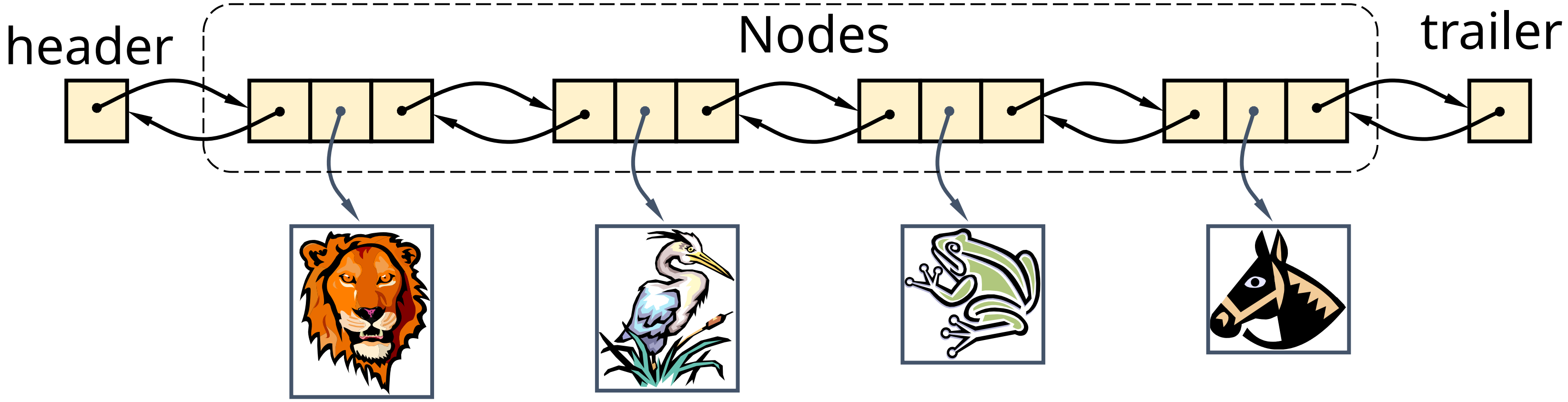
```python
class LinkedQueue:
  """FIFO queue implementation using a singly linked list for storage."""

  class _Node:
    """Lightweight, nonpublic class for storing a singly linked node."""
    (omitted here; identical to that of LinkedStack._Node)

  def __init__(self):
    """Create an empty queue."""
    self._head = None
    self._tail = None
    self._size = 0                          # number of queue elements

  def __len__(self):
    """Return the number of elements in the queue."""
    return self._size

  def is_empty(self):
    """Return True if the queue is empty."""
    return self._size == 0

  def first(self):
    """Return (but do not remove) the element at the front of the queue."""
    if self.is_empty():
      raise Empty('Queue is empty')
    return self._head._element              # front aligned with head of list
```

```python
  def dequeue(self):
    """Remove and return the first element of the queue (i.e., FIFO).

    Raise Empty exception if the queue is empty.
    """
    if self.is_empty():
      raise Empty('Queue is empty')
    answer = self._head._element
    self._head = self._head._next
    self._size -= 1
    if self.is_empty():                     # special case as queue is empty
      self._tail = None                     # removed head had been the tail
    return answer

  def enqueue(self, e):
    """Add an element to the back of queue."""
    newest = self._Node(e, None)            # node will be new tail node
    if self.is_empty():
      self._head = newest                   # special case: previously empty
    else:
      self._tail._next = newest
    self._tail = newest                     # update reference to tail node
    self._size += 1
```

Doubly Linked List

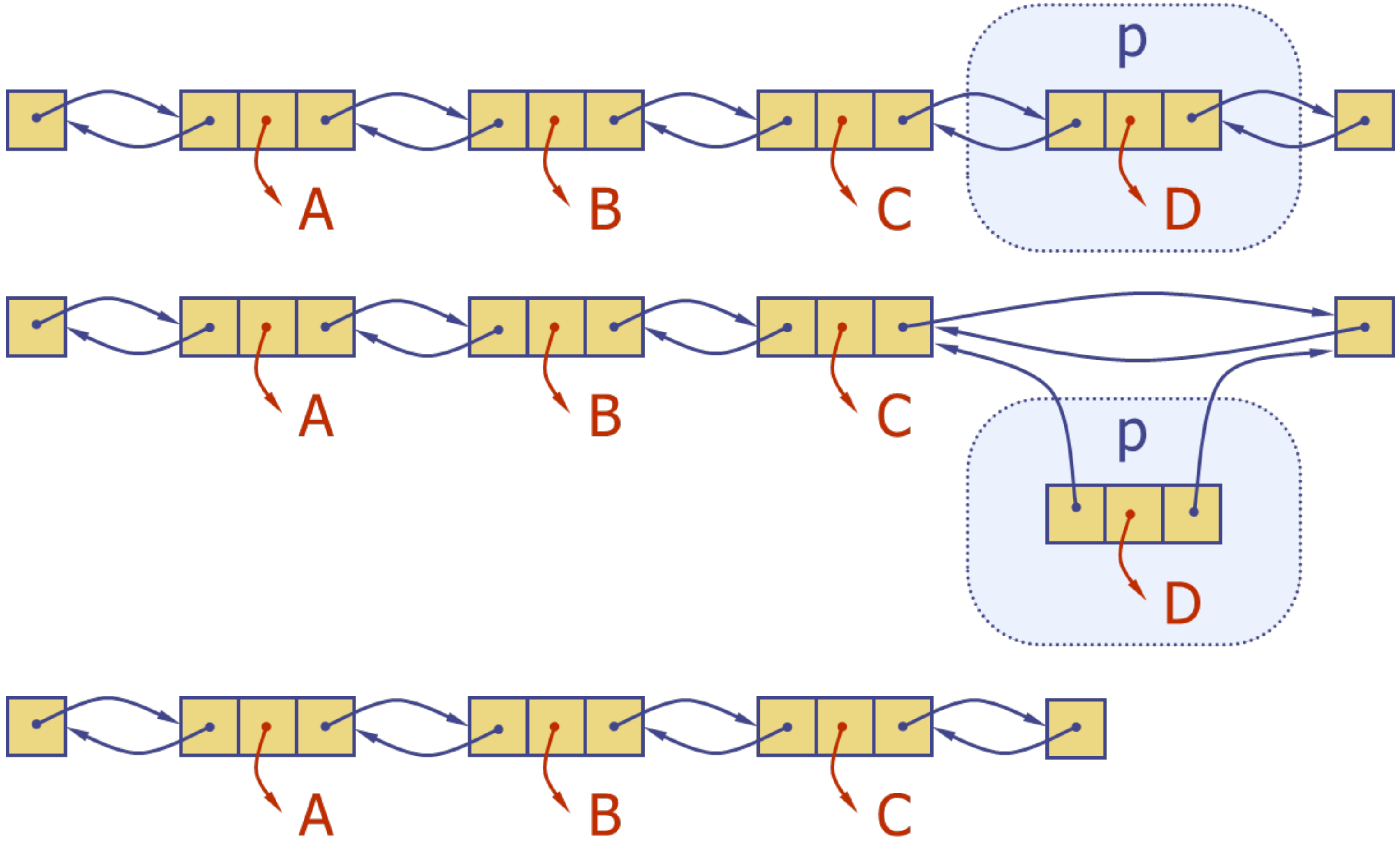
- A linked list which can be traversed in both directions.
- Each Node stores:
  - Data
  - Link to the previous node
  - Link to the next node





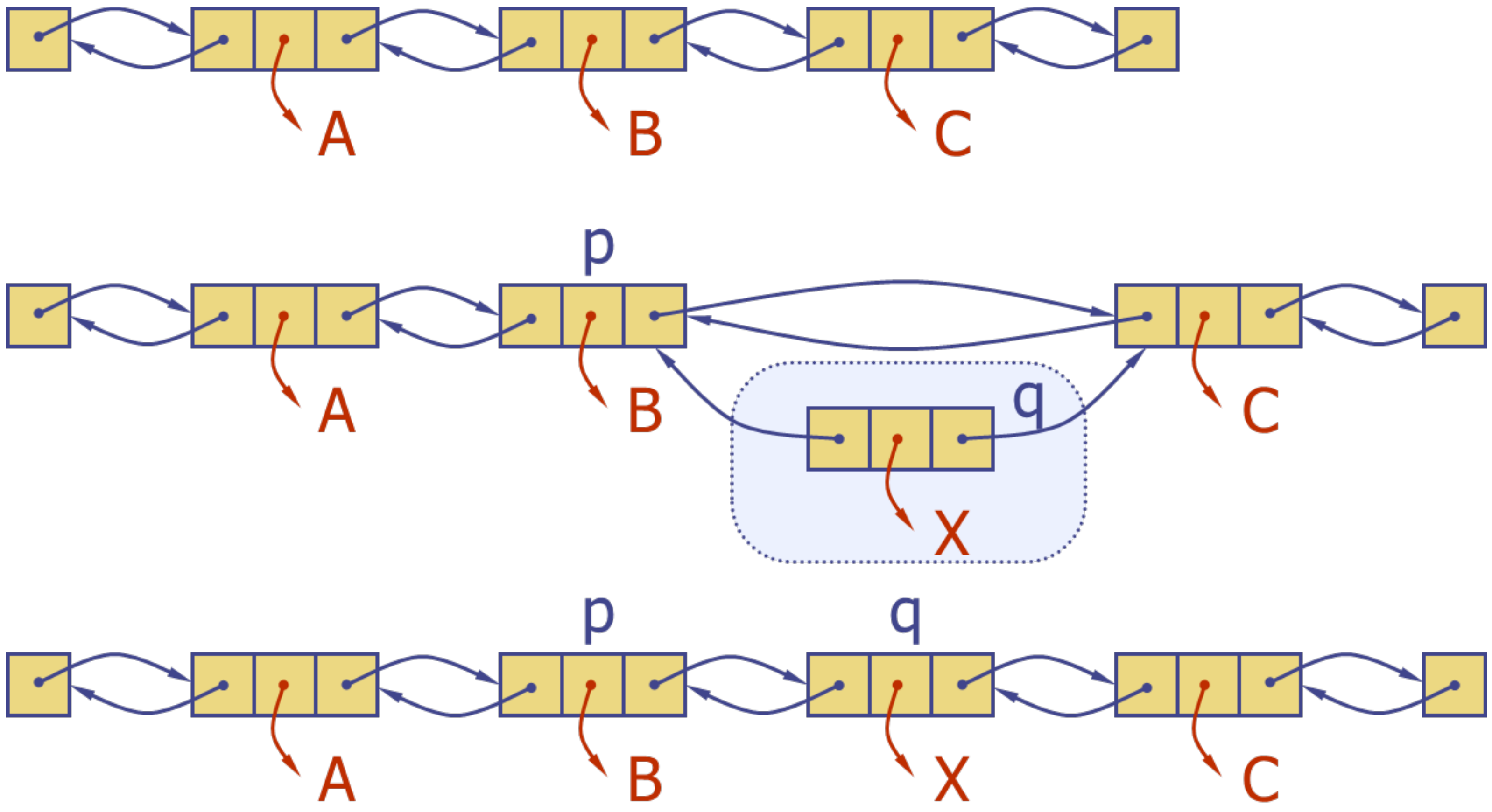
Deletion from Doubly Linked List

- 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

```
class _DoublyLinkedBase:
  """A base class providing a doubly linked list representation."""

  class _Node:
    """Lightweight, nonpublic class for storing a doubly linked node."""
    (omitted here; see previous code fragment)

  def __init__(self):
    """Create an empty list."""
    self._header = self._Node(None, None, None)
    self._trailer = self._Node(None, None, None)
    self._header._next = self._trailer                # trailer is after header
    self._trailer._prev = self._header                # header is before trailer
    self._size = 0                                    # number of elements

  def __len__(self):
    """Return the number of elements in the list."""
    return self._size

  def is_empty(self):
    """Return True if list is empty."""
    return self._size == 0

  def _insert_between(self, e, predecessor, successor):
    """Add element e between two existing nodes and return new node."""
    newest = self._Node(e, predecessor, successor)   # linked to neighbors
    predecessor._next = newest
    successor._prev = newest
    self._size += 1
    return newest

  def _delete_node(self, node):
    """Delete nonsentinel node from the list and return its element."""
    predecessor = node._prev
    successor = node._next
    predecessor._next = successor
    successor._prev = predecessor
    self._size −= 1
    element = node._element                           # record deleted element
    node._prev = node._next = node._element = None    # deprecate node
    return element                                    # return deleted element
```

Circular Linked List