

## 07 Circular Doubly Linked Lists and Deques

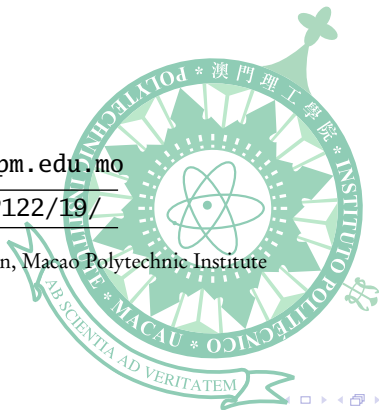
*Instructor* : Ke Wei (柯韋)

➡ A319    ☎ Ext. 6452    ✉ wke@ipm.edu.mo

<http://brouwer.ipm.edu.mo/COMP122/19/>

Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

February 01, 2019



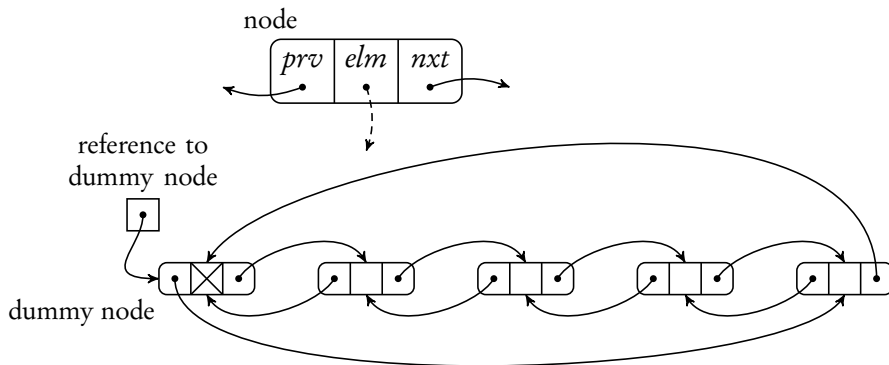
# Outline

- 1 Circular Doubly Linked Lists
- 2 Implementing Circular Doubly Linked Lists
- 3 Double-Ended Queues
- 4 Joining and Splitting

# Circular Doubly Linked Lists and Dummy Nodes

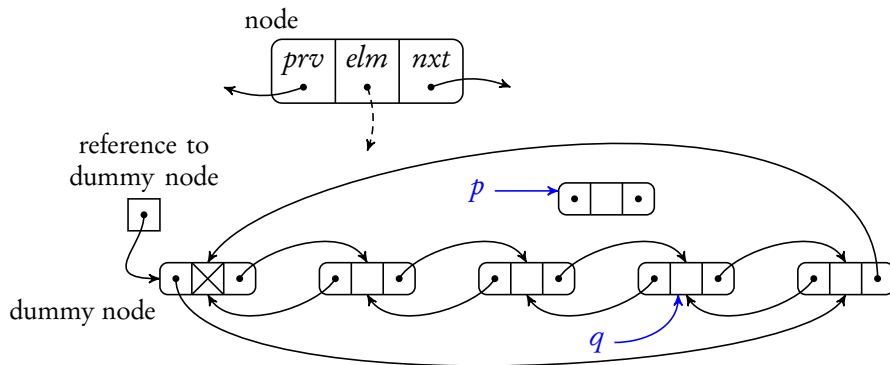
- In a node of a linked list, besides a link to the next node, it is natural to introduce a link to the previous node. This setting results *doubly linked lists*.
- The first node in a list does not have a *predecessor*, and the last node does not have a *successor*.
- We can link the first node and the last node together using the spare links. This setting results *circular linked lists*.
- A circular linked list must have at least one node. To unify the empty list, we introduce an extra *dummy node* (or *sentinel*) to each circular linked list, i.e., the dummy node stores only the links, but no element, and the empty list can be represented by a circular list with only a dummy node linking to itself.
- We put these altogether to give the very convenient circular doubly linked lists.

# Circular Doubly Linked Lists — Illustrated



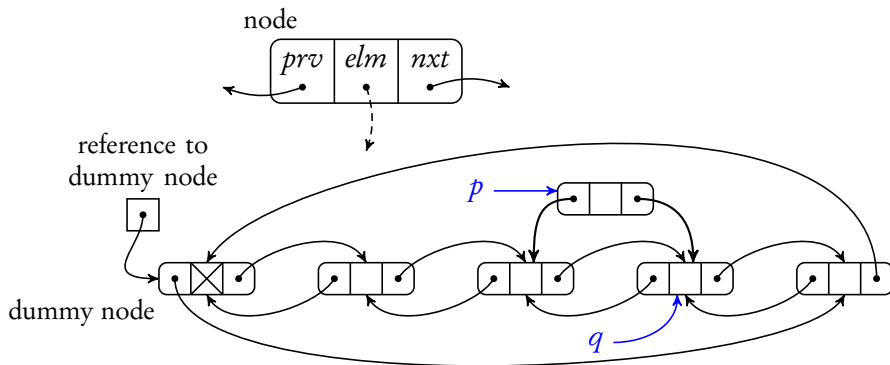
```
def insert_node(p, q): # insert p in front of q
    p.next, p.prev = q, q.prev
    p.prev.next = p.next.prev = p
```

# Circular Doubly Linked Lists — Illustrated



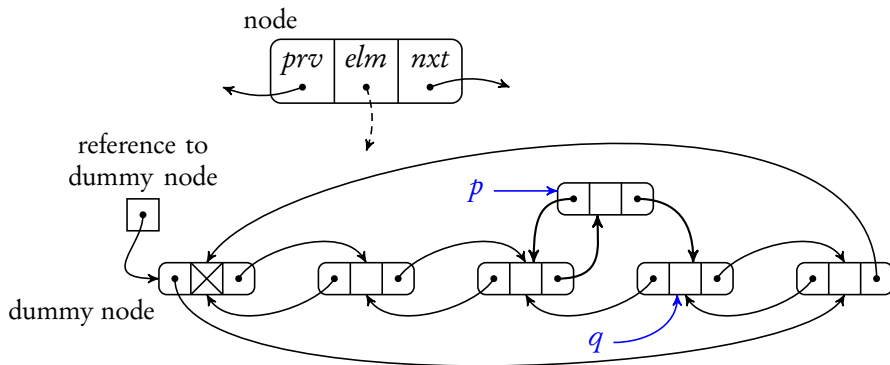
```
def insert_node(p, q): # insert p in front of q
    p.next, p.prev = q, q.prev
    p.prev.next = p.next.prev = p
```

# Circular Doubly Linked Lists — Illustrated



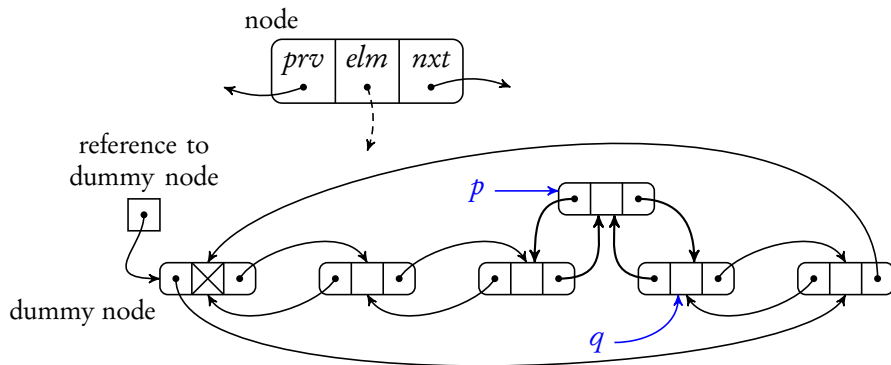
```
def insert_node(p, q): # insert p in front of q
    p.next, p.prev = q, q.prev
    p.prev.next = p.next.prev = p
```

# Circular Doubly Linked Lists — Illustrated



```
def insert_node(p, q): # insert p in front of q
    p.next, p.prev = q, q.prev
    p.prev.next = p.next.prev = p
```

# Circular Doubly Linked Lists — Illustrated

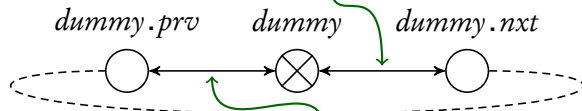


```
def insert_node(p, q): # insert p in front of q
    p.nxt, p.prev = q, q.prev
    p.prev.nxt = p.nxt.prev = p
```

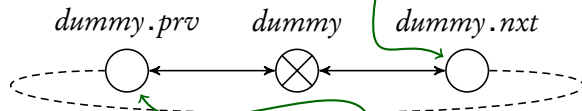


# Advantages of Circular Doubly Linked Lists

- Nodes at both ends are immediately accessible.
- Insertions and deletions at both ends are very efficient, independent to the length of the list.
- To add an element at the first position, we insert it before *dummy.next*.



- To add an element at the last position, we insert it before *dummy*.
- To remove an element at the first position, we delete *dummy.next*.



- To remove an element at the last position, we delete *dummy.prv*.


# Nodes in Doubly Linked Lists

- In addition to the *elm* and *nxt* attributes, we also include the *prv* attribute, pointing to the previous node.
- We introduce these attributes in the *insert\_node* and *insert\_elm* functions, leaving the *Node* class empty.


```
class Node:
    def __init__(self, elm):
        self.elm = elm
```

```
def insert_elm(x, q):
    p = Node(x)
    insert_node(p, q)
```

```
def delete_elm(p):
    delete_node(p)
    return p.elm
```

 Try to complete the deletion operation following the illustration on Slide 4.

```
def delete_node(p):
    ...
```

 What happens if this deletion is applied to the node of a list that has only this node?

## Defining a Dummy Node in $CLnLs$

- We need to define a dummy node and initialize it to point to itself. We do this in the *constructor*.
- The list is empty when there is only the dummy node, that is, when the dummy node points to itself.

---

```

1 class CLnLs:
2     def __init__(self):
3         self.dummy = Node(None)
4         self.dummy.prv = self.dummy.nxt = self.dummy
5     def __bool__(self):
6         return self.dummy.nxt is not self.dummy
7     def check_empty(self):
8         if not self:
9             raise IndexError

```

---

## Forward and Backward Iterators

- While a singly linked list only iterates elements forward, with the *prv* pointers, a doubly linked list is also able to iterate elements backward.
- Python formulates the backward iterator as a special method `__reversed__(self)`.

---

```

10     def __iter__(self):
11         p = self.dummy.nxt
12         while p is not self.dummy:
13             yield p.elm
14             p = p.nxt

```

---



---

```

16     def __reversed__(self):
17         p = self.dummy.prv
18         while p is not self.dummy:
19             yield p.elm
20             p = p.prv

```

---

- To obtain a backward iterator of a collection *s*, we should call `reversed(s)`.

## Defining the *CLnLs* as a *Deque*

- A *double-ended queue* or deque, pronounced “deck”, is a linear structure that can add and remove elements at both ends.
- The *Deque* ADT has more general methods than the *Stack* and *Queue*:

*push*, *pop*, *top*, *push\_back*, *pop\_back* and *back*

---

```

20  def push(self, x):
21      insert_elm(x, self.dummy.next)
22  def pop(self):
23      self.check_empty()
24      x = delete_elm(self.dummy.next)
25      return x
26  def top(self):
27      self.check_empty()
28      return self.dummy.next.elm
  
```

---



---

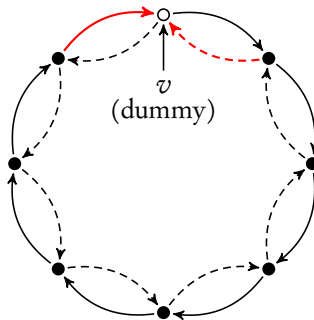
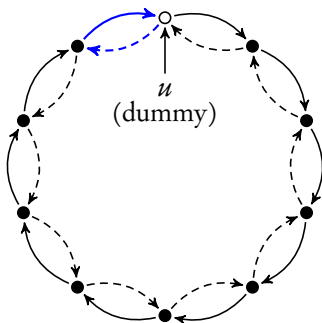
```

28  def push_back(self, x):
29      insert_elm(x, self.dummy)
30  def pop_back(self):
31      self.check_empty()
32      x = delete_elm(self.dummy.prev)
33      return x
34  def back(self):
35      self.check_empty()
36      return self.dummy.prev.elm
  
```

---

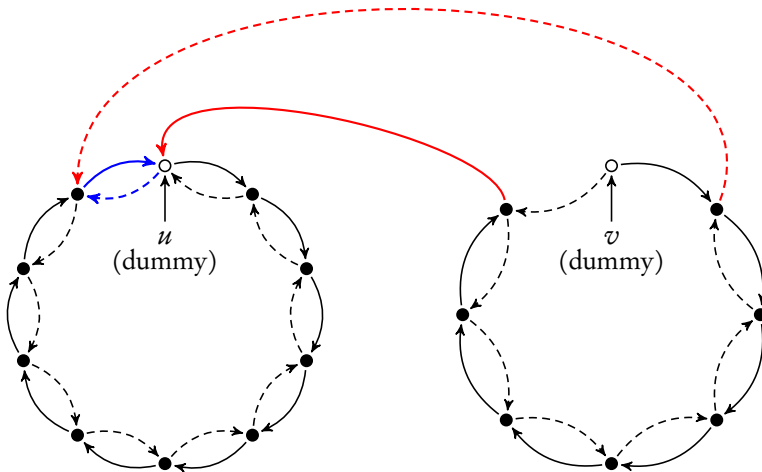
# Joining Two Lists

Here illustrates how to join a list  $v$  to the end of another list  $u$ .



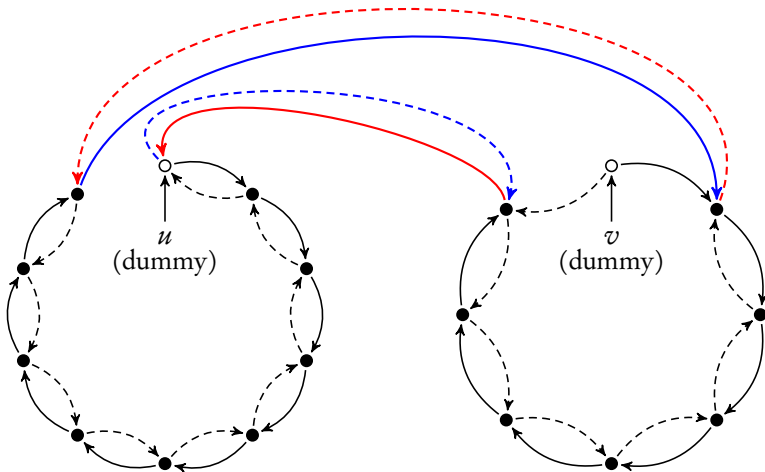
# Joining Two Lists

Here illustrates how to join a list  $v$  to the end of another list  $u$ .



# Joining Two Lists

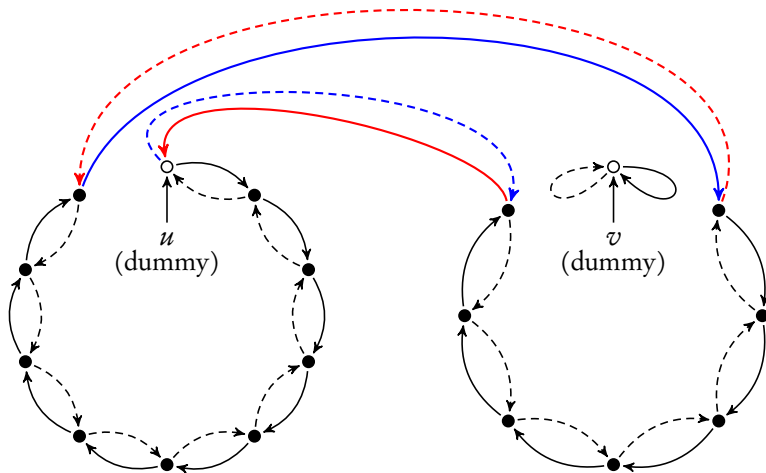
Here illustrates how to join a list  $v$  to the end of another list  $u$ .





# Joining Two Lists

Here illustrates how to join a list  $v$  to the end of another list  $u$ .



## Joining Two Lists — Code

The following method joins a list with dummy node  $v$  before node  $q$  in another list.

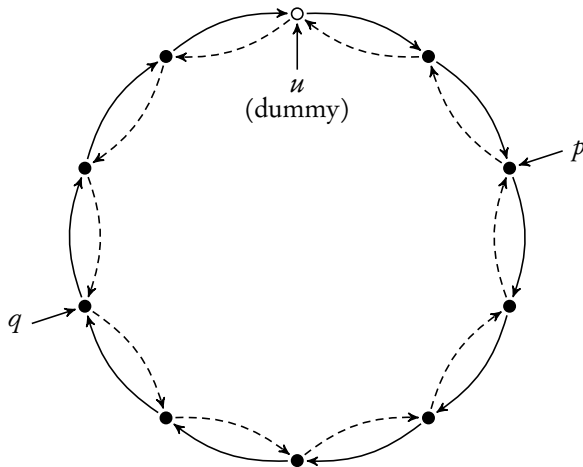
---

```
1 def join_clist( $v$ ,  $q$ ):  
2     if  $v.nxt$  is not  $v$ :  
3          $v.nxt.prv = q.prv$   
4          $v.prv.nxt = q$   
5          $v.nxt.prv.nxt = v.nxt$   
6          $v.prv.nxt.prv = v.prv$   
7          $v.nxt = v.prv = v$ 
```

---

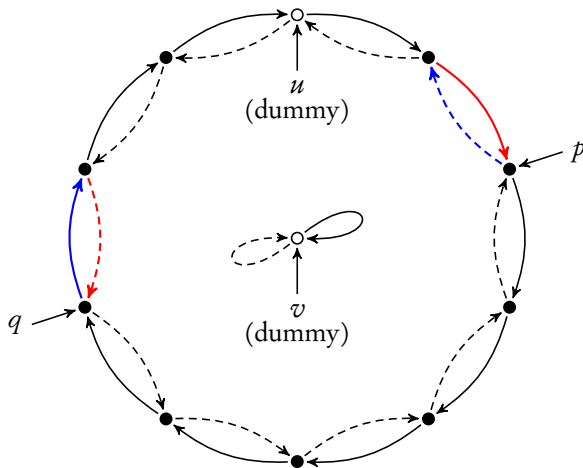
# Splitting a List

Here illustrates how to split out the portion between two nodes  $p$  and  $q$  from a list.



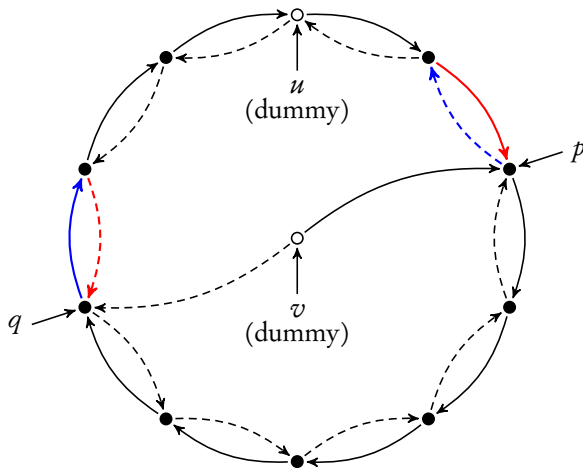
# Splitting a List

Here illustrates how to split out the portion between two nodes  $p$  and  $q$  from a list.



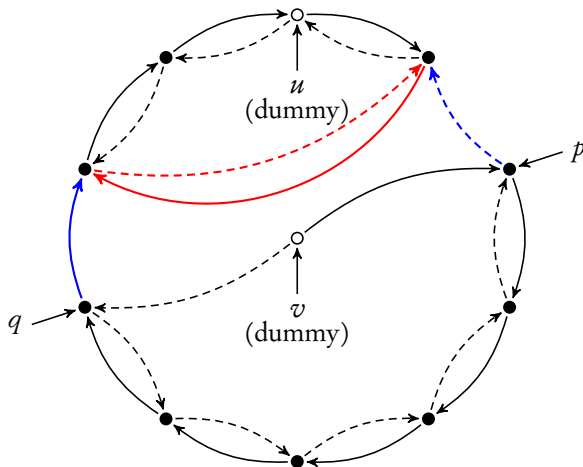
# Splitting a List

Here illustrates how to split out the portion between two nodes  $p$  and  $q$  from a list.



# Splitting a List

Here illustrates how to split out the portion between two nodes  $p$  and  $q$  from a list.



# Splitting a List

Here illustrates how to split out the portion between two nodes  $p$  and  $q$  from a list.

