COMP122/20 - Data Structures and Algorithms

07 Circular Doubly Linked Lists and Deques

Instructor : Ke Wei 〚柯韋〛

➠ A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/20/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

February 7, 2020

Outline

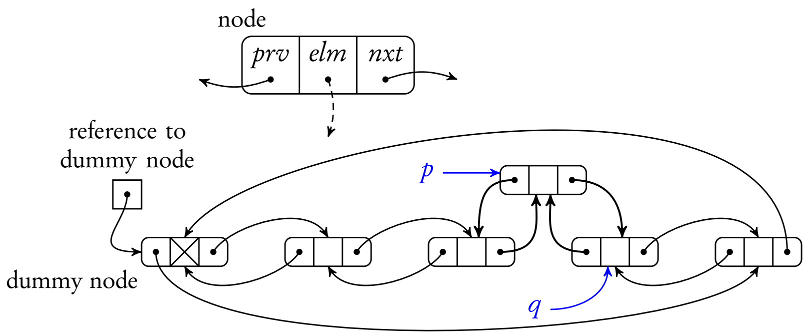
1. **Circular Doubly Linked Lists**
2. **Implementing Circular Doubly Linked Lists**
3. **Double-Ended Queues**
4. **Joining and Splitting**

👁 *Textbook §6.3, 7.2 – 7.3.*

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.

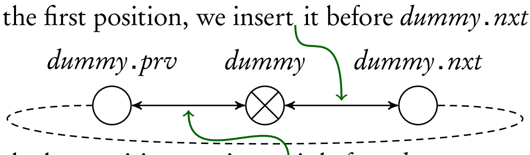
Circular Doubly Linked Lists — Illustrated



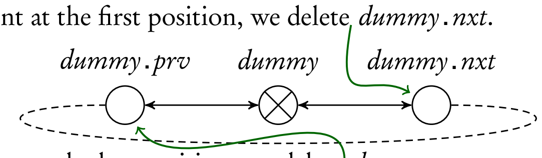
```
def insert_node(p, q): # insert p in front of q
    p.nxt, p.prv = q, q.prv
    p.prv.nxt = p.nxt.prv = p
```

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.nxt*.



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



- 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.

```
class CLnLs:
    def __init__(self):
        self.dummy = Node(None)
        self.dummy.prv = self.dummy.nxt = self.dummy
    def __bool__(self):
        return self.dummy.nxt is not self.dummy
    def check_empty(self):
        if not self:
            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*).

```
    def __iter__(self):
        p = self.dummy.nxt
        while p is not self.dummy:
            yield p.elm
            p = p.nxt
```

```
    def __reversed__(self):
        p = self.dummy.prv
        while p is not self.dummy:
            yield p.elm
            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*:

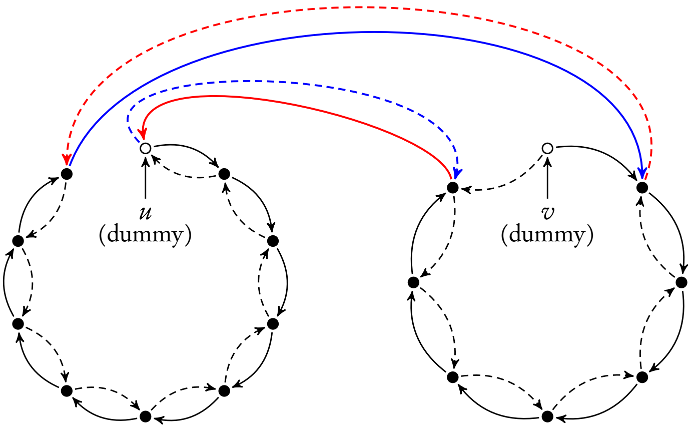
  push, *pop*, *top*, *push_back*, *pop_back* and *back*

```
    def push(self, x):
        insert_elm(x, self.dummy.nxt)
    def pop(self):
        self.check_empty()
        x = delete_elm(self.dummy.nxt)
        return x
    def top(self):
        self.check_empty()
        return self.dummy.nxt.elm
```

```
    def push_back(self, x):
        insert_elm(x, self.dummy)
    def pop_back(self):
        self.check_empty()
        x = delete_elm(self.dummy.prv)
        return x
    def back(self):
        self.check_empty()
        return self.dummy.prv.elm
```

Joining Two Lists

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



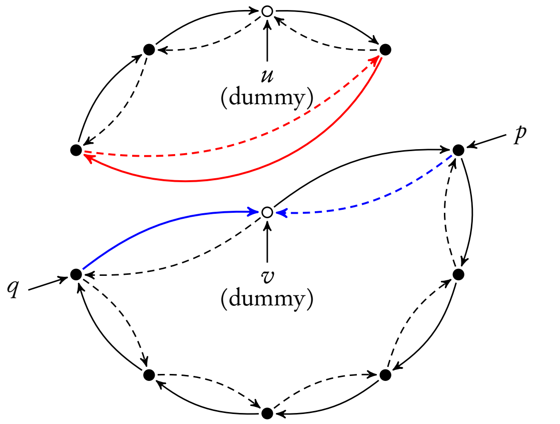
Joining Two Lists — Code

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

```
def join_clist(v, q):
    if v.nxt is not v:
        v.nxt.prv = q.prv
        v.prv.nxt = q
        v.nxt.prv.nxt = v.nxt
        v.prv.nxt.prv = v.prv
        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.


u
(dummy)
p
q
v
(dummy)