

# CS 32 Week 3

## Discussion 1B

Yiyou Chen / Ian Galvez

# Announcement

Office hour changes:

Yiyou Chen's MW office hours **used to be** online. **Now changed to be in person.**

For the most accurate OH information, please check out the main course website.

# Topics

- Linked List:
  - Singly-linked list
  - Dummy nodes and singly-linked list with dummy nodes.
  - Doubly-linked list with dummy nodes.
  - Circular doubly linked list with dummy nodes.

# Singly Linked List: define

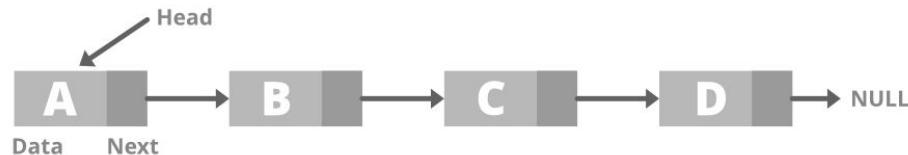
A sequential data structure.

Advantage:

Easy to insert and delete without knowing the length.

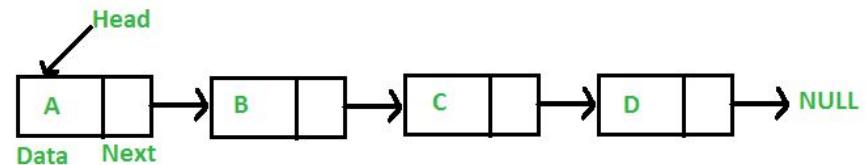
```
221 struct Node {  
222     int val;  
223     Node* next;  
224 };  
225  
226 Node* head = nullptr;
```

Singly Linked List



# Singly Linked List: search by value

```
Node* Find_Val(Node* head, int x) {  
}
```



# Singly Linked List: search by value

```
-->
230 Node* Find_Val(Node* head, int x) {
231     Node* p = head;
232     while (p != nullptr) {
233         if (p->val == x) break;
234         p = p->next;
235     }
236     return p;
237 }
```

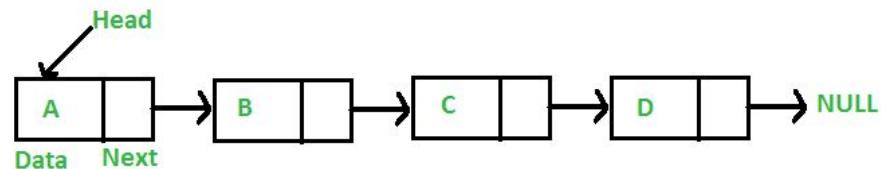


Image :<https://www.geeksforgeeks.org>

# Singly Linked List: search by index

```
239 Node* Find_k_th(Node* head, int k) {  
240  
241 }
```

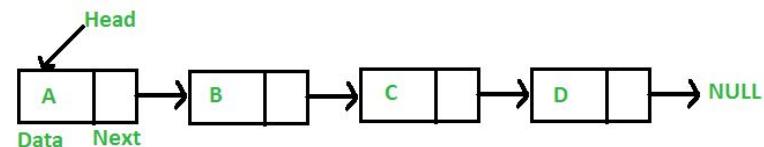


Image :<https://www.geeksforgeeks.org>

# Singly Linked List: search by index

```
---  
239 Node* Find_k_th(Node* head, int k) {  
240     Node* p = head;  
241     while (p != nullptr) {  
242         --k;  
243         if (k == 0) break;  
244         p = p->next;  
245     }  
246     return p;  
247 }
```

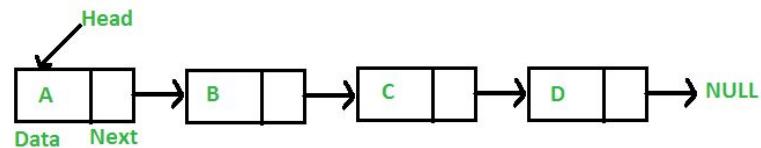
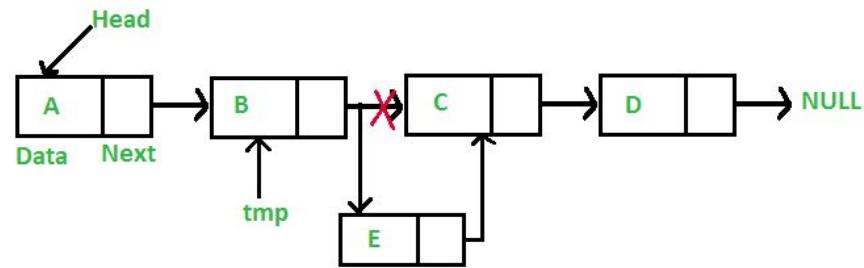


Image :<https://www.geeksforgeeks.org>

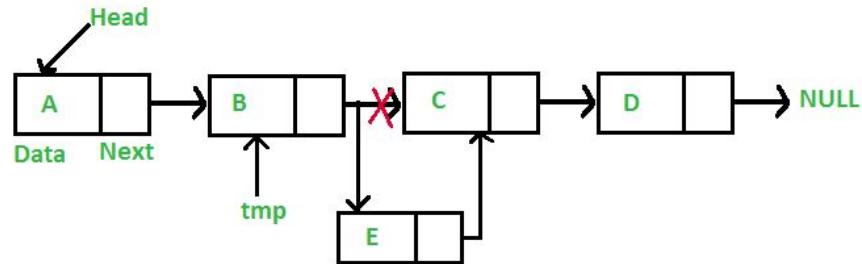
# Singly Linked List: insert an element after p

```
250 void Add_After(Node* p, int newval) {  
251  
252 }
```



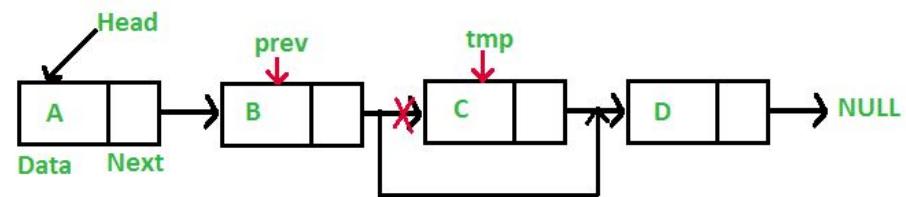
# Singly Linked List: insert an element after p

```
void Add_After(Node* p, int x) {  
    Node* newnode = new Node;  
    newnode->val = x;  
    if (p == head && head == nullptr) {  
        newnode->next = nullptr;  
        head = newnode;  
    }  
    else {  
        newnode->next = p->next;  
        p->next = newnode;  
    }  
}
```



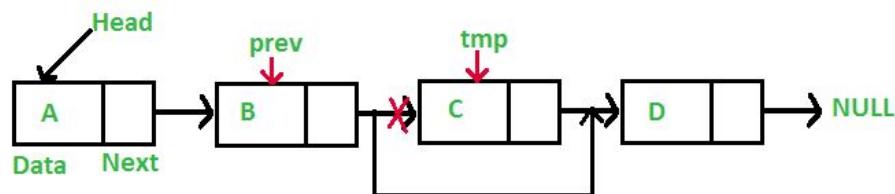
# Singly Linked List: delete first node with value x

```
256 int Delete_One_Val(Node* head, int x) {  
257  
258 }
```



# Singly Linked List: delete first node with value x

```
262 int Delete_One_Val(Node* head, int x) {  
263     if (head == nullptr) return -1;  
264     if (head->val == x) {  
265         Node* p = head->next;  
266         delete head;  
267         head = p;  
268     }  
269     else {  
270         Node* p = head;  
271         while (p->next != nullptr) {  
272             if(p->next->val == x) {  
273                 break;  
274             }  
275             p = p->next;  
276         }  
277         if (p->next == nullptr) { //x not found  
278             return -1;  
279         }  
280         Node* q = p->next;  
281         p->next = q->next;  
282         delete q;  
283     }  
284     return 1;  
285 }
```



# Singly Linked List: reverse the order of a linked list

```
95 void Reverse_Order(Node* head) {  
96  
97 }
```

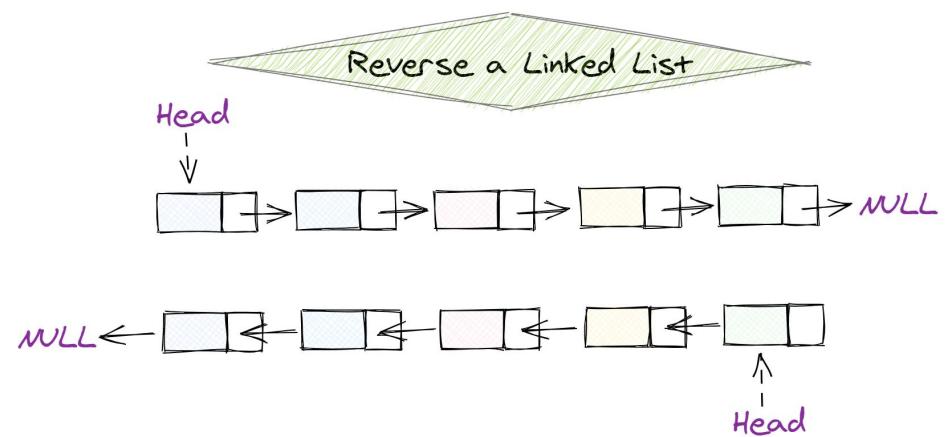


Image: <https://algodaily.com/challenges/reverse-a-linked-list>

# Singly Linked List: reverse the order of a linked list

```
323 void Reverse_Order(Node* head) {  
324     if (head == nullptr || head->next == nullptr) return;  
325     Node* p = head;  
326     Node* q = head->next;  
327     p->next = nullptr;  
328     while (q != nullptr) {  
329         Node* temp = q->next;  
330         q->next = p;  
331         p = q;  
332         q = temp;  
333     }  
334     head = p;  
335 }
```

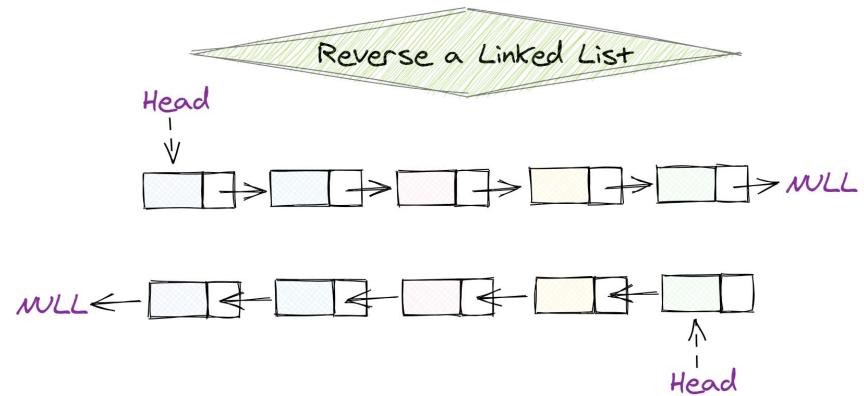


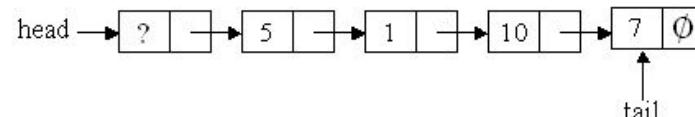
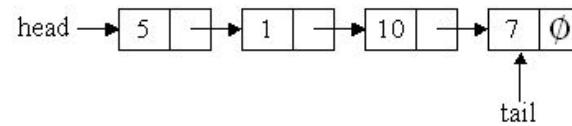
Image: <https://algodaily.com/challenges/reverse-a-linked-list>

# Singly Linked List: dummy node for head

Sometimes, adding a dummy node simplifies the code. The first element becomes head->next.

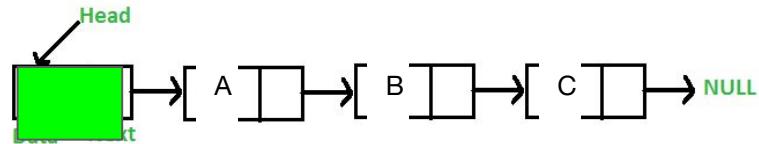
```
290 Node* dummyhead = new Node;  
291 dummyhead->val = INF; //Optional  
292 dummyhead->next = nullptr;  
...
```

Most of the time, we don't care about dummy head's value, so we don't even need to initialize it. (here we initialize to infinity).



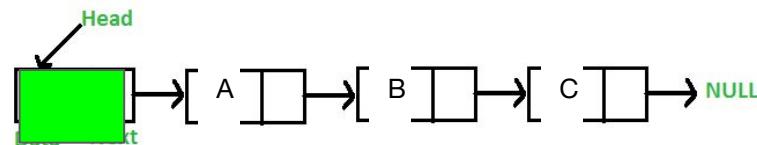
## Singly Linked List: insert a node after p (dummy node)

```
void Add_After(Node* p, int newval) {  
    Node* newnode = new Node;  
    newnode->val = newval;  
    newnode->next = p->next;  
    p->next = newnode;  
}
```



## Singly Linked List: delete first node with value x (dummy node)

```
int Delete_One_Val(Node* head, int x) {  
    Node* p = head;  
    while (p->next != nullptr) {  
        if (p->next->val == x)  
            break;  
        p = p->next;  
    }  
    if (p->next == nullptr) return -1;  
    Node* q = p->next;  
    p->next = q->next;  
    delete q;  
    return 1;  
}
```



# Singly Linked List: delete all nodes with value x

```
279 void Delete_ALL_Val(Node* head, int x) {  
280  
281 }
```

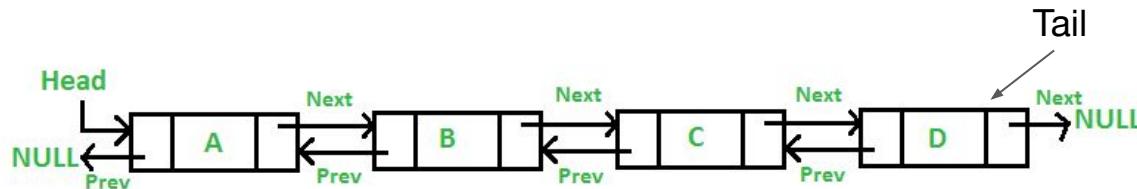
# Singly Linked List: delete all nodes with value x

```
---  
283 void Delete_All_Val(Node* head, int x) {  
284     while (Delete_One_Val(head, x) > 0) {}  
285 }
```

Exercise: faster way?

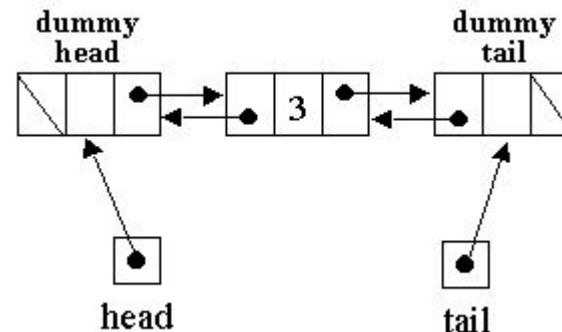
# Doubly Linked List

```
struct DNode {  
    int val;  
    DNode *prev, *next;  
};  
  
DNode* head = nullptr;  
DNode* tail = nullptr;
```



# Doubly Linked List (dummy nodes)

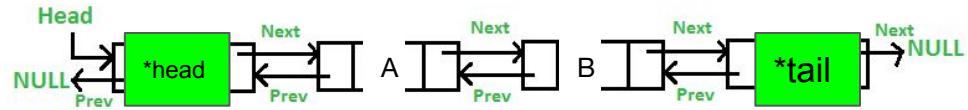
```
struct DNode {  
    int val;  
    DNode *prev, *next;  
};  
  
DNode* dummyhead = new DNode;  
dummyhead->val = INF; //Optional  
dummyhead->next = dummytail;  
dummyhead->prev = nullptr;  
DNode* dummytail = new DNode;  
dummytail->val = INF; //Optional  
dummytail->next = nullptr;  
dummytail->prev = dummyhead;
```



# Doubly Linked List: search by value or index

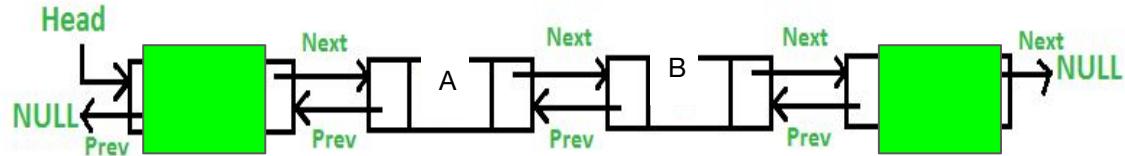
Searching by kth number or value are almost the same as for the singly linked list.

```
370 DNode* DFind_Val(DNode* head, DNode* tail, int x) {  
371     DNode* p = head->next;  
372     while (p != tail) {  
373         if (p->val == x) break;  
374         p = p->next;  
375     }  
376     return p;  
377 }  
378  
379 DNode* DFind_k_th(DNode* head, DNode* tail, int k) {  
380     DNode* p = head->next;  
381     while (p != tail) {  
382         --k;  
383         if (k <= 0) break;  
384         p = p->next;  
385     }  
386     return p;  
387 }
```



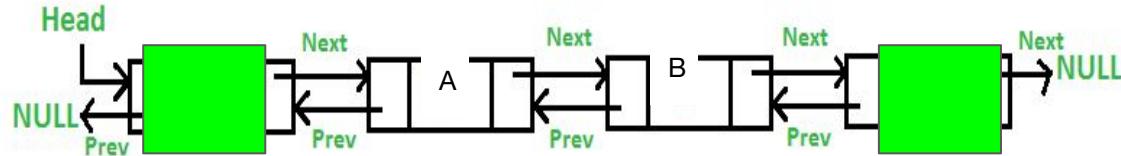
# Doubly Linked List: insert a new node after p

```
359 void DAdd_After(DNode* p, int newval) {  
360  
361 }
```



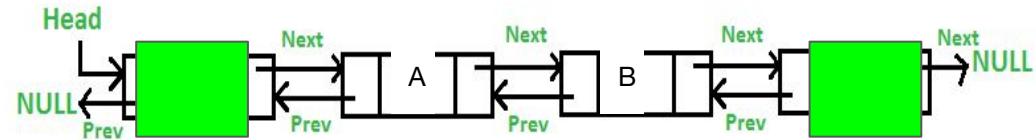
# Doubly Linked List: insert a new node after p

```
360 void DAdd_After(DNode* p, int newval) {  
361     DNode* newnode = new DNode;  
362     newnode->val = newval;  
363     newnode->next = p->next;  
364     newnode->prev = p;  
365     p->next->prev = newnode;  
366     p->next = newnode;  
367 }
```



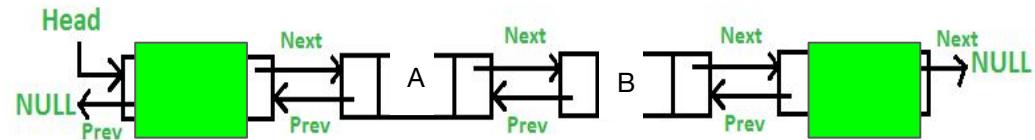
# Doubly Linked List: delete a DNode\* p

```
390 DNode* DDelete_One_Node(DNode* p) {  
391  
392 }
```



# Doubly Linked List: delete a DNode\* p

```
394 DNode* DDelete_One_Node(DNode* p) {  
395     p->next->prev = p->prev;  
396     p->prev->next = p->next;  
397     delete p;  
398 }
```



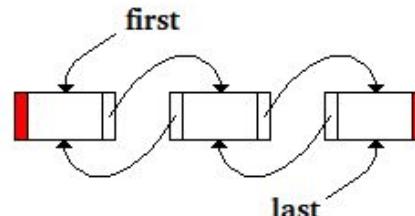
# Circular Doubly Linked List

Instead of having a head and a tail, why not connect the head and tail to make it circular?

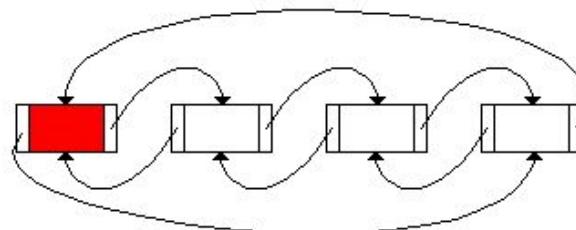
Now only one dummy head is needed.

```
struct CNode {  
    int val;  
    CNode *prev, *next;  
};
```

```
CNode* dummyhead = new CNode;  
dummyhead->val = INF; //Optional  
dummyhead->prev = dummyhead;  
dummyhead->next = dummyhead;
```

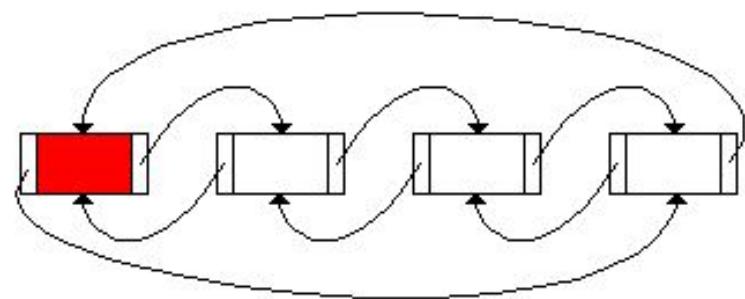


Using **NULL** to  
mark end of list



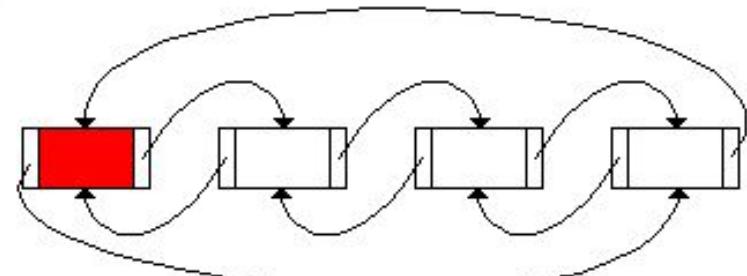
Using a special  
dummy node

## Circular Doubly Linked List: search by value or index

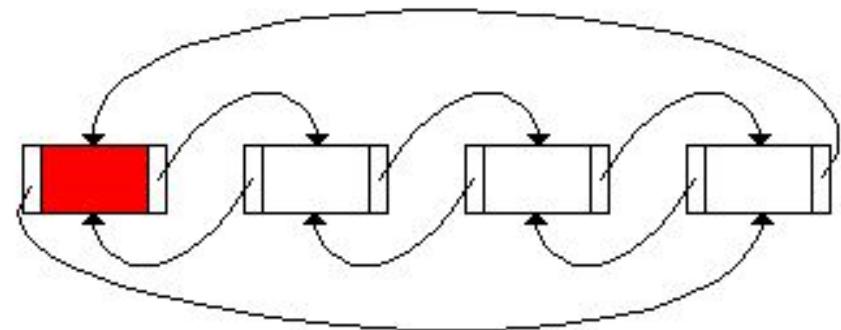


# Circular Doubly Linked List: search by value or index

```
411 CNode* CFind_Val(CNode* head, int x) { //head == dummyhead
412     CNode* p = head->next;
413     while (p != head) {
414         if (p->val == x) {
415             return p;
416         }
417     }
418     return nullptr;
419 }
420
421 CNode* CFind_k_th(CNode* head, int k) { //head == dummyhead
422     CNode* p = head->next;
423     while (p != head) {
424         --k;
425         if (k <= 0) break;
426         p = p->next;
427     }
428     if (p == head)
429         return nullptr;
430     return p;
431 }
```

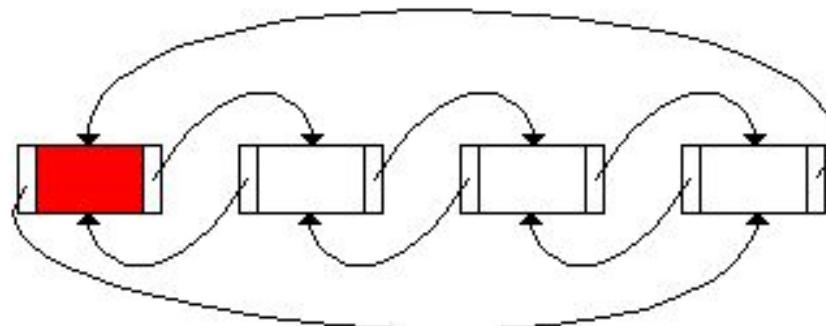


## Circular Doubly Linked List: insert after/before Node p

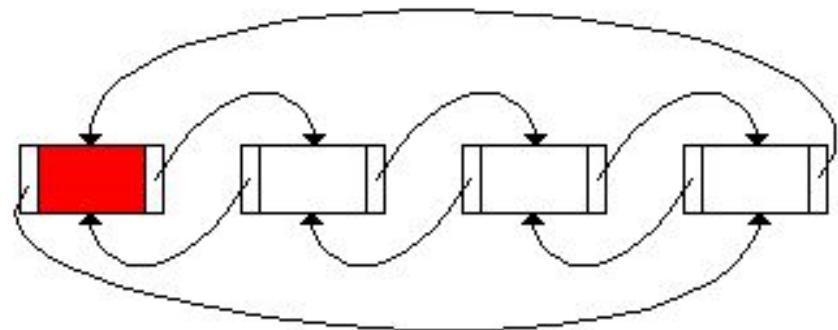


## Circular Doubly Linked List: insert after/before Node p

```
433 void CAdd_After(CNode* p, int newval) {  
434     CNode* newnode = new CNode;  
435     newnode->val = newval;  
436     newnode->next = p->next;  
437     newnode->prev = p;  
438     p->next->prev = newnode;  
439     p->next = newnode;  
440 }  
441  
442 void CAdd_Before(CNode* p, int newval) {  
443     CNode* newnode = new CNode;  
444     newnode->val = newval;  
445     newnode->next = p;  
446     newnode->prev = p->prev;  
447     p->prev->next = newnode;  
448     p->prev = newnode;  
449 }  
450
```

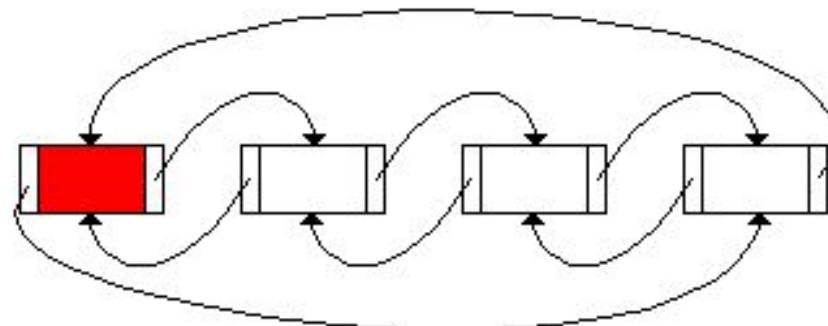


## Circular Doubly Linked List: delete Node p



## Circular Doubly Linked List: delete Node p

```
451 void CDelete_One_Node(CNode* p) {  
452     //assume p != dummyhead  
453     p->next->prev = p->prev;  
454     p->prev->next = p->next;  
455     delete p;  
456 }
```



# Linked Lists

1. Singly-linked list.
  - a. For simple tasks.
  - b. Keeping next only.
2. Doubly-linked list.
  - a. Bidirectional, making some tasks like deletion and reverse order easier.
  - b. Keeping prev and next.
3. Circular doubly linked list (my personal favorite).
  - a. Bidirectional and easy to implement.
  - b. Keeping prev and next.

Time complexity: with  $n$  nodes in the list,  $O(n)$  complexity for insertion and deletion.

Question: how to swap two linked lists?

# Exercise

Modify the insertion, Add\_After(), to make the linked list sorted by value.