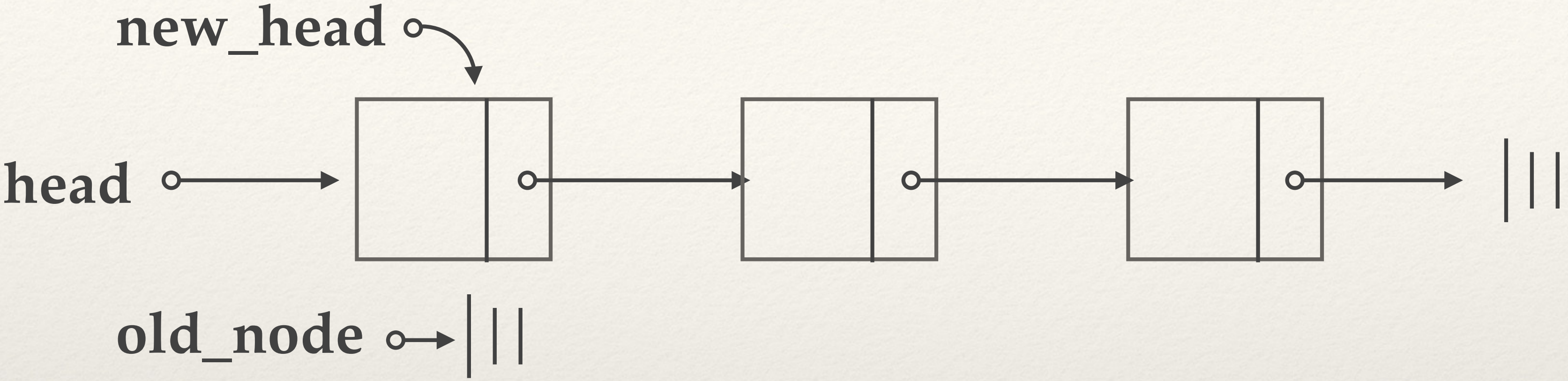
Freeing a List

- Start at the head and `free()` elements until the end of the list is reached.

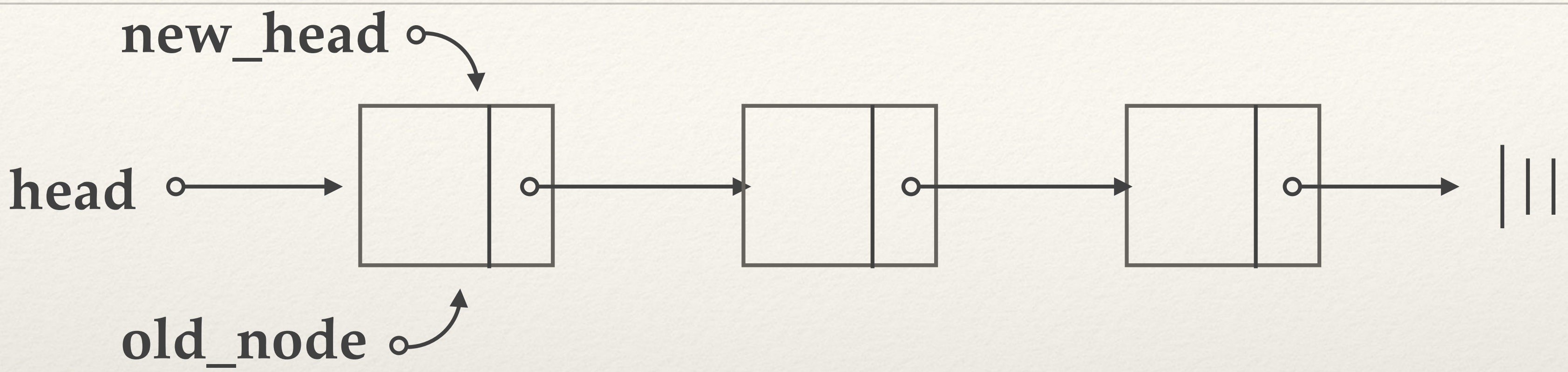
```
void free_list(Node * head) {
   Node * new_head = head, * old_node = NULL;
   while ( new_head != NULL ) {
        old_node = new_head;
        new_head = new_head->next;
        free ( old_node );
    }
}
```

Freeing a List



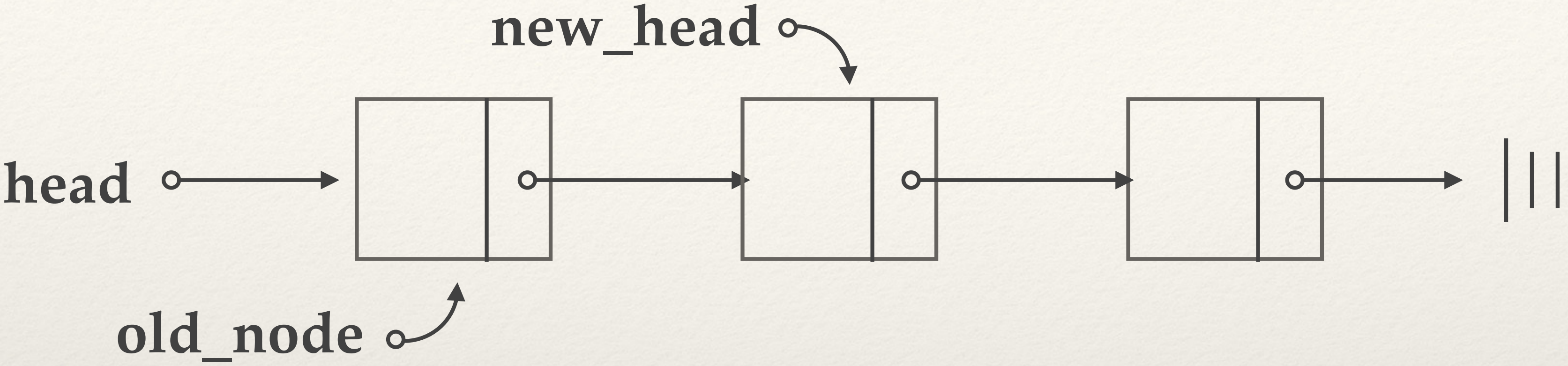
```
void free_list(Node * head) {
    Node * new_head = head, * old_node = NULL;
    while ( new_head != NULL ) {
         old_node = new_head;
         new_head = new_head->next;
         free ( old_node );
    }
}
```

Freeing a List



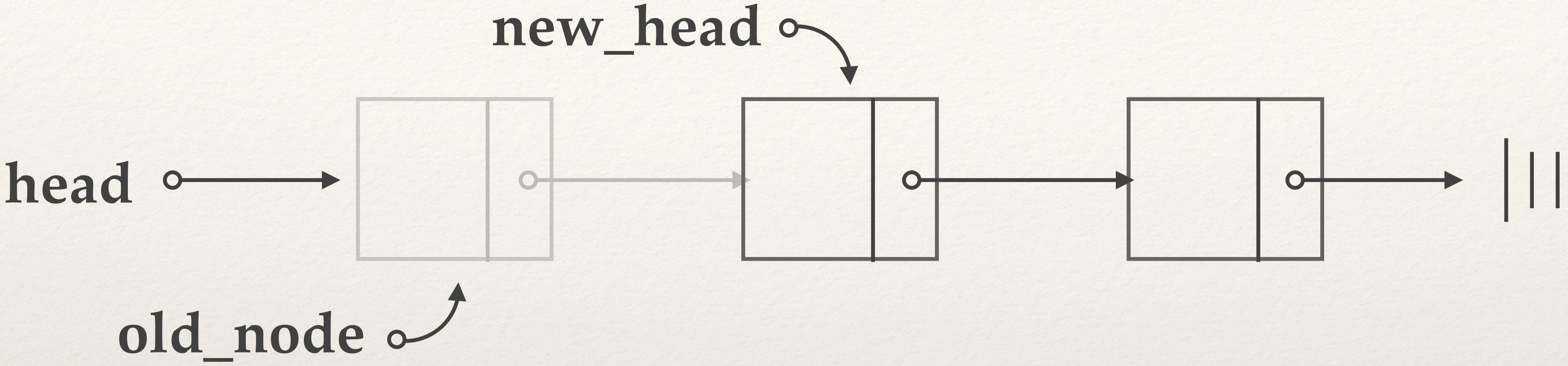
```
void free_list(Node * head) {
    Node * new_head = head, * old_node = NULL;
    while ( new_head != NULL ) {
        old_node = new_head;
        new_head = new_head->next;
        free ( old_node );
    }
}
```

Freeing a List



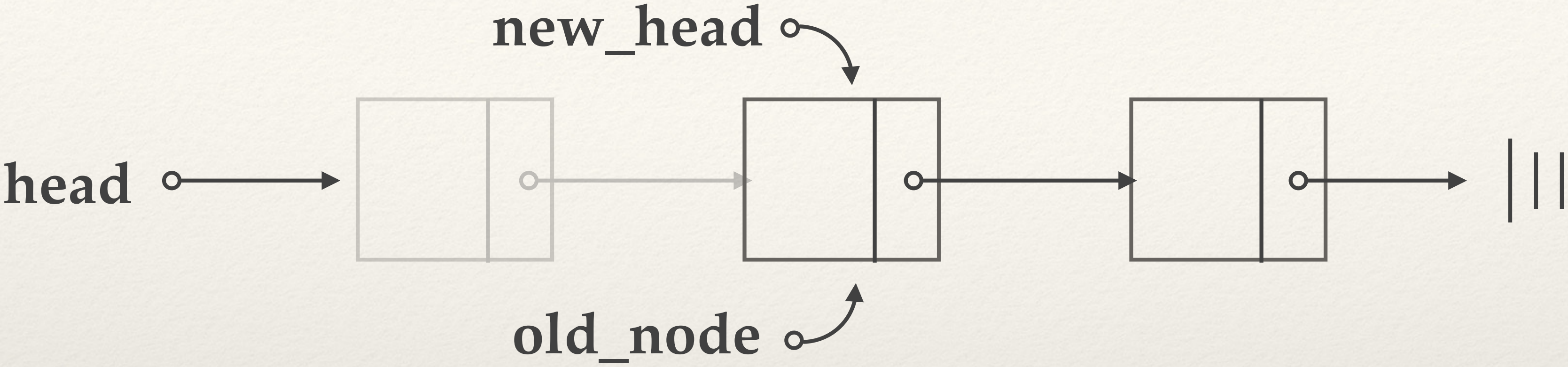
```
void free_list(Node * head) {
    Node * new_head = head, * old_node = NULL;
    while ( new_head != NULL ) {
         old_node = new_head;
         new_head = new_head->next;
         free ( old_node );
    }
}
```

Freeing a List



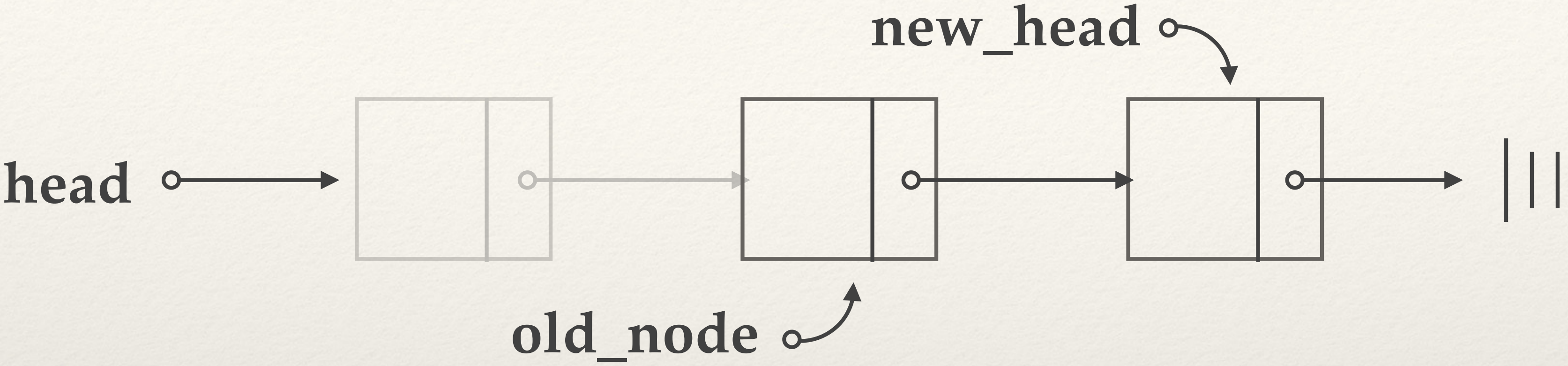
```
void free_list(Node * head) {
    Node * new_head = head, * old_node = NULL;
    while ( new_head != NULL ) {
          old_node = new_head;
          new_head = new_head->next;
          free ( old_node );
     }
}
```

Freeing a List



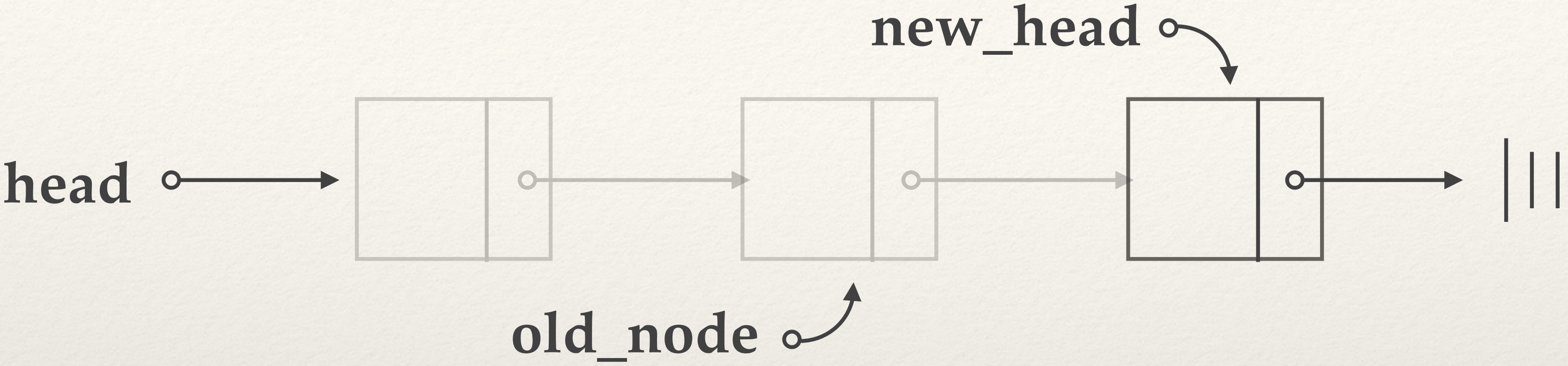
```
void free_list(Node * head) {
    Node * new_head = head, * old_node = NULL;
    while ( new_head != NULL ) {
        old_node = new_head;
        new_head = new_head->next;
        free ( old_node );
    }
}
```

Freeing a List



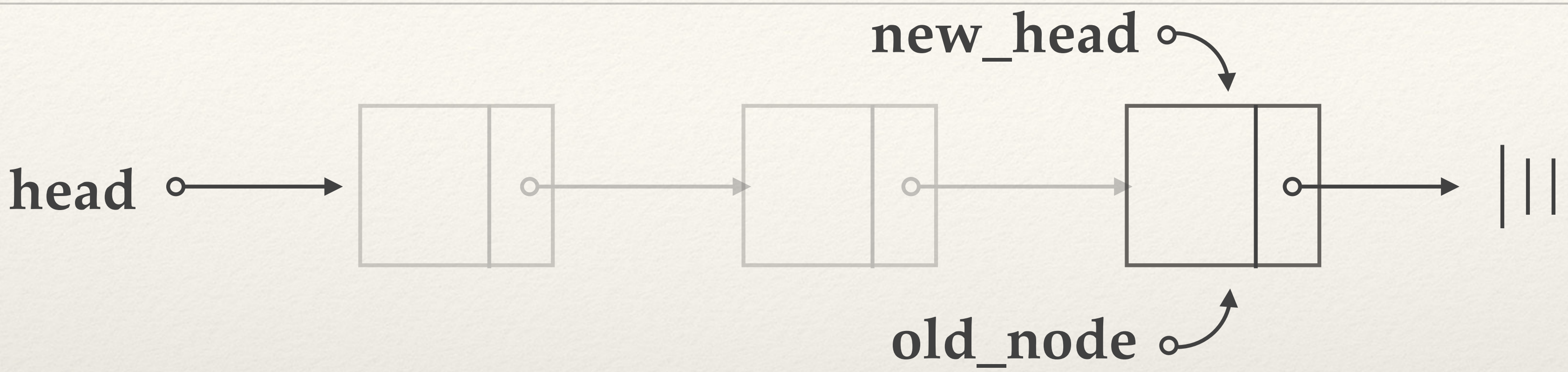
```
void free_list(Node * head) {
    Node * new_head = head, * old_node = NULL;
    while ( new_head != NULL ) {
        old_node = new_head;
        new_head = new_head->next;
        free ( old_node );
    }
}
```

Freeing a List



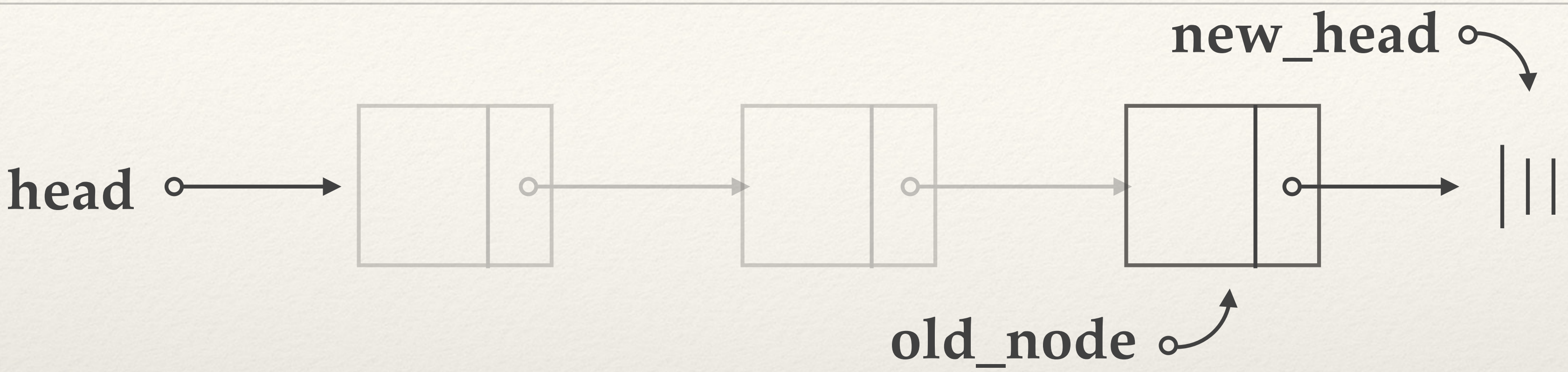
```
void free_list(Node * head) {
    Node * new_head = head, * old_node = NULL;
    while ( new_head != NULL ) {
        old_node = new_head;
        new_head = new_head->next;
        free ( old_node );
    }
}
```

Freeing a List



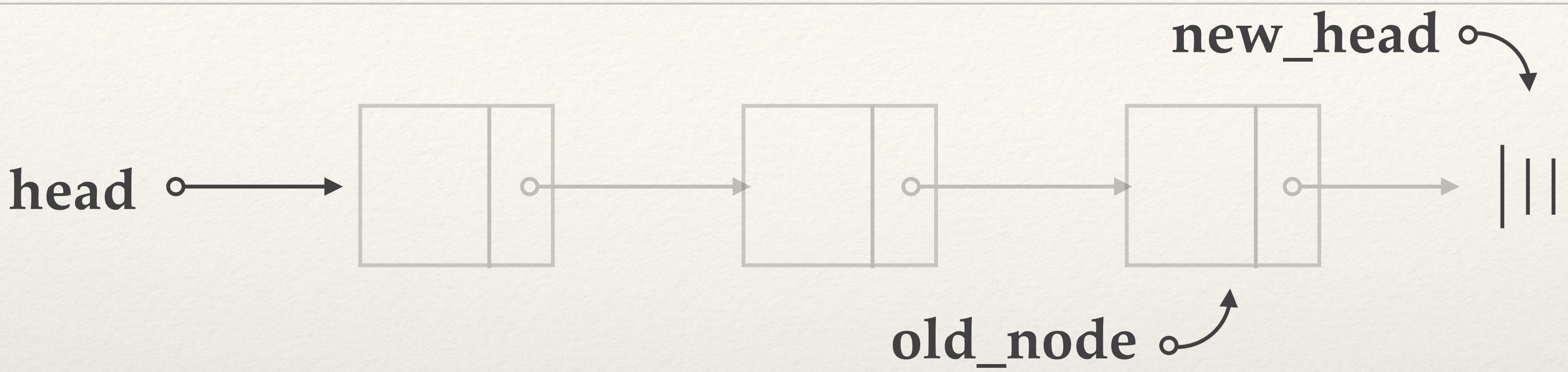
```
void free_list(Node * head) {
    Node * new_head = head, * old_node = NULL;
    while ( new_head != NULL ) {
        old_node = new_head;
        new_head = new_head->next;
        free ( old_node );
    }
}
```

Freeing a List



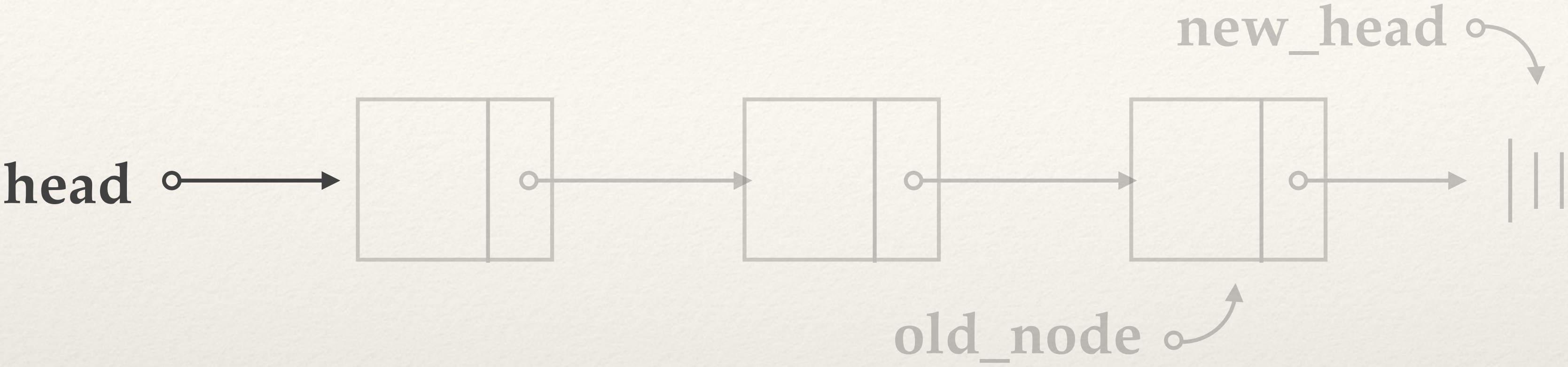
```
void free_list(Node * head) {
    Node * new_head = head, * old_node = NULL;
    while ( new_head != NULL ) {
        old_node = new_head;
        new_head = new_head->next;
        free ( old_node );
    }
}
```

Freeing a List



```
void free_list(Node * head) {
   Node * new_head = head, * old_node = NULL;
   while ( new_head != NULL ) {
        old_node = new_head;
        new_head = new_head->next;
        free ( old_node );
    }
}
```

Freeing a List



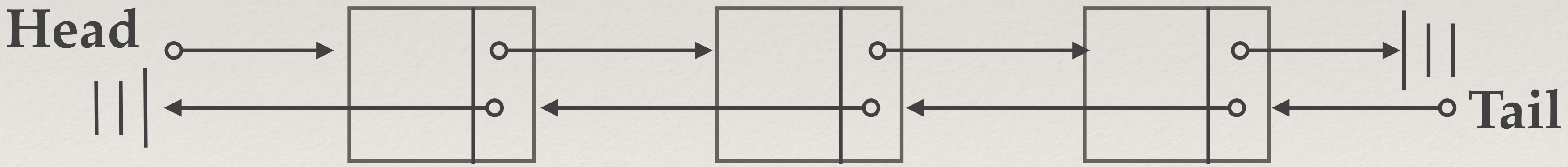
```
void free_list(Node * head) {
    Node * new_head = head, * old_node = NULL;
    while ( new_head != NULL ) {
         old_node = new_head;
         new_head = new_head->next;
         free ( old_node );
     }
}
```

Remember to set head to NULL
in the code that called `free_list`

Doubly Linked Lists

- Doubly linked lists are similar to singly linked lists except that there are pointers to the **next** and **previous** structures.

- It allows you to step through a list in either direction.



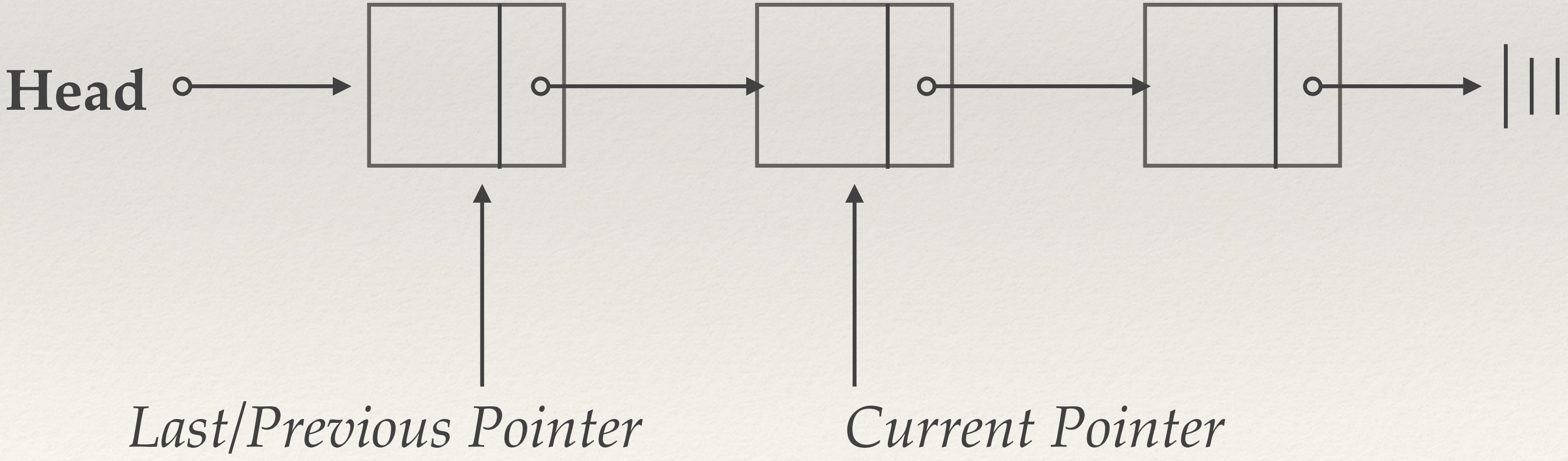
Doubly Linked Lists

- The doubly linked list uses two pointers in the structure:

```
struct element {
    int value;
    struct element *next, *prev;
}
```

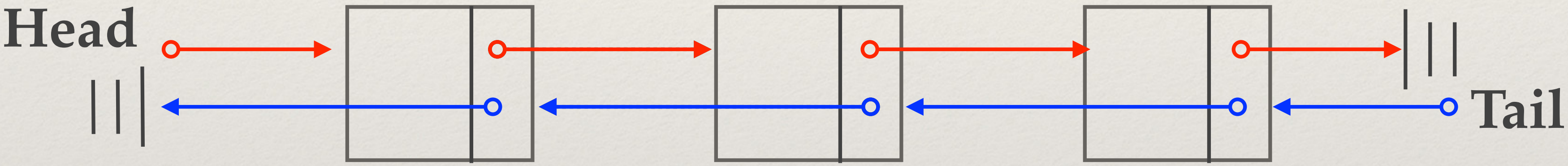
Benefits

- You can always move in either direction in the list.
- There is no need to keep an extra pointer around pointing to the last element visited.



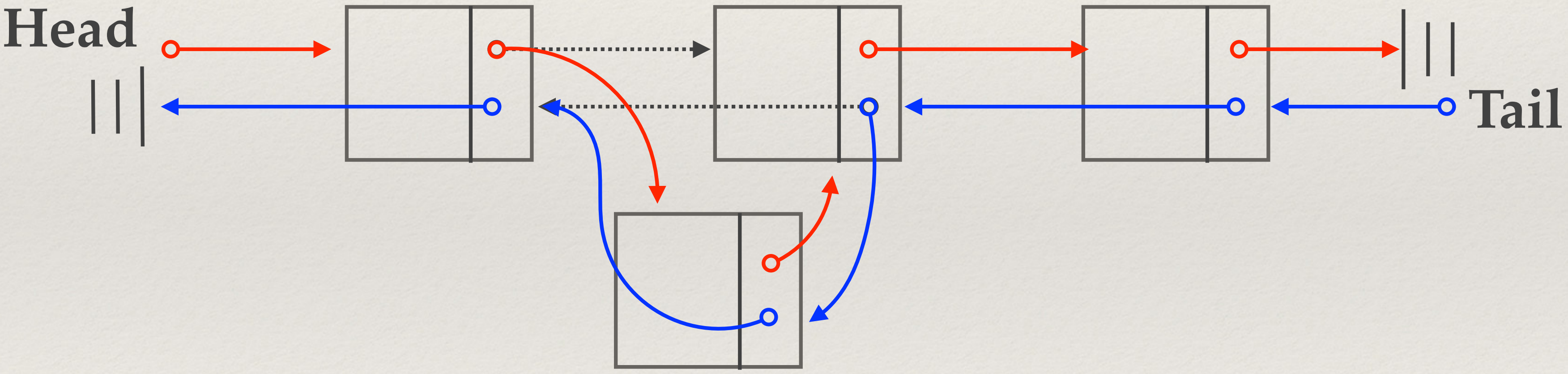
But…

- More pointers must be changed to add or delete an element in the list.

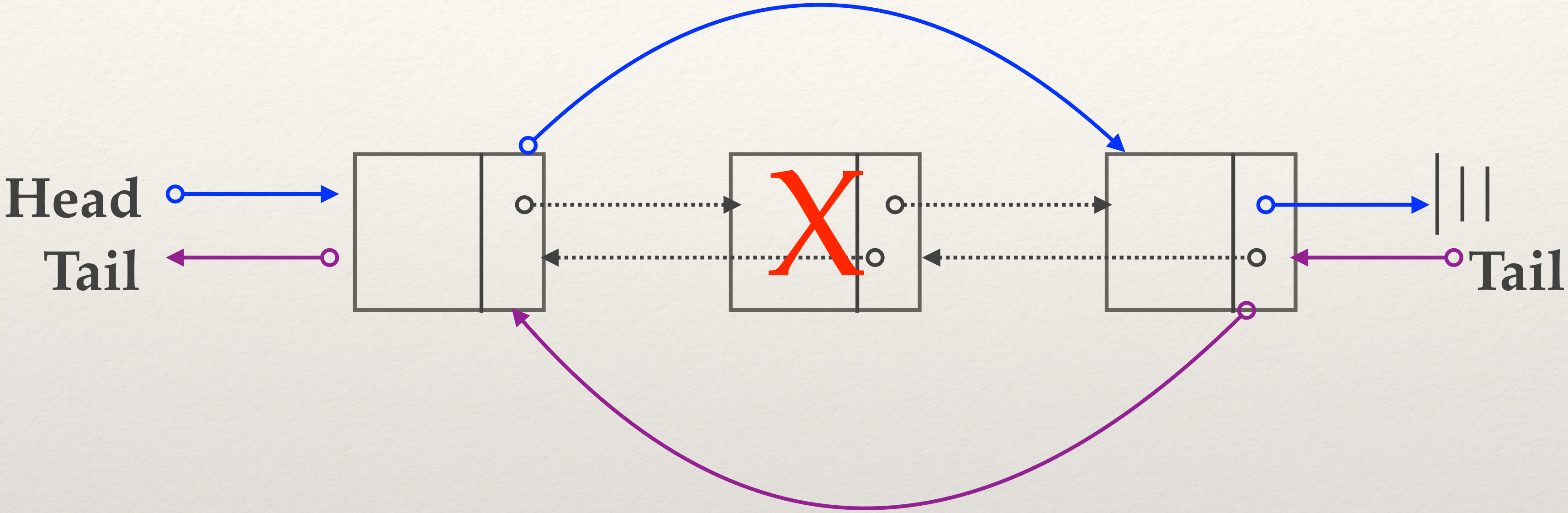


But…

- More pointers must be changed to add or delete an element in the list.



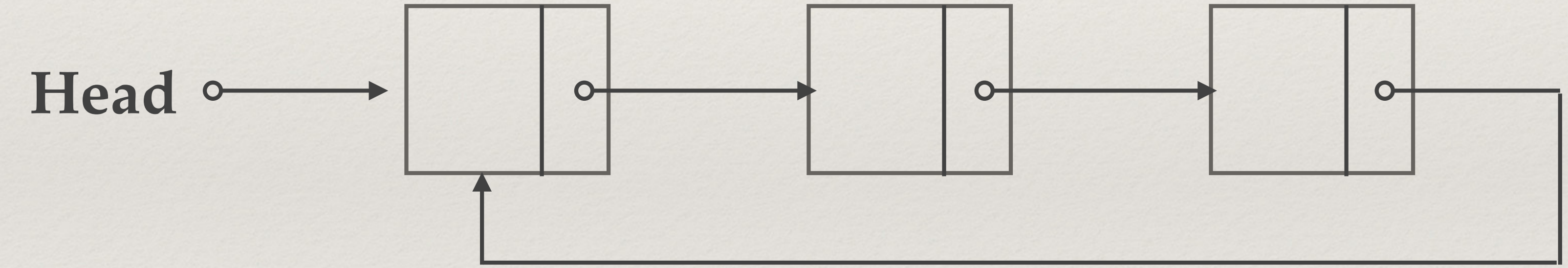
Deleting an Element

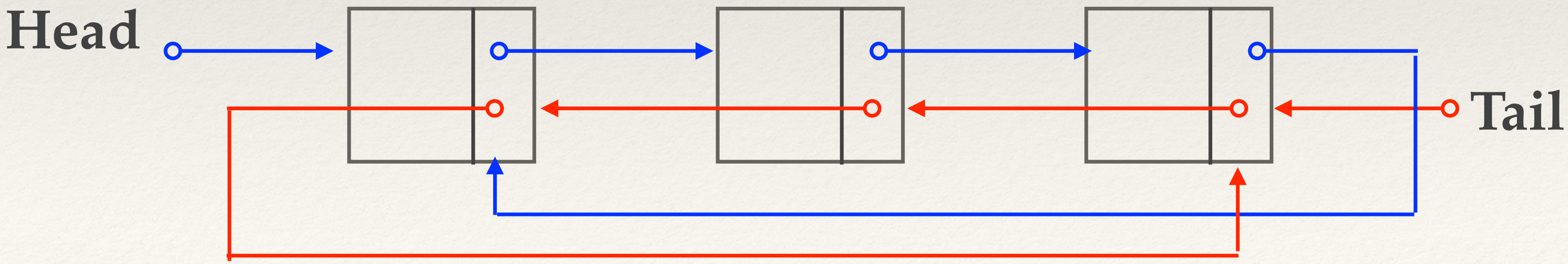


Circular Lists

- Circular lists are list in which the last element in the list points to the first element.

- It can be singly or doubly linked.





Circular Lists

- You never run into the end of the list.
    - Simpler transversal logic.
- But…you can get into an infinite traversal if you search for something that is not in the list.
    - You must check if you are in a loop
    - *e.g.* `if head == current_ptr`