

Data Structures

Chapter 4

1. Singly Linked List
 - Pointer & Linking
 - **Singly Linked List (SLL)**
 - SLL Basic Operations
 - SLL Advanced Operations
2. Doubly Linked List

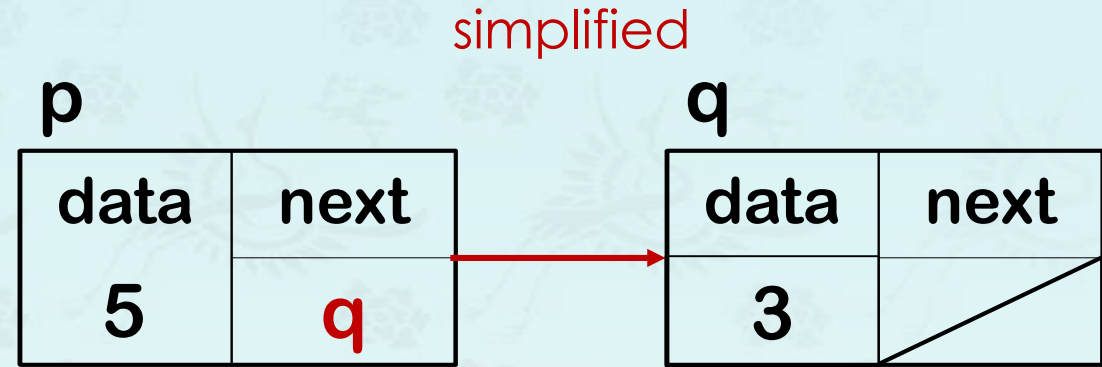


내 아들들을 먼 곳에서 이끌며 내 딸들을 땅 끝에서 오게 하며 내 이름으로 불려지는 모든 자 곧 내가 내 영광을 위하여 창조한 자를 오게 하라 그를 내가 지었고 그를 내가 만들었노라 (사43:6-7)

수고하고 무거운 짐 진 자들아 다 내게로 오라 내가 너희를 쉬게 하리라 나는 마음이 온유하고 겸손하니 나의 멍에를 메고 내게 배우라 그리하면 너희 마음이 쉼을 얻으리니 이는 내 멍에는 쉽고 내 짐은 가벼움이라 하시니라 (마11:28-30)

Pointers Linked

```
class Node {  
public:  
    int    data;  
    Node* next;  
};  
  
int main( ) {  
    Node* q = new Node{3, nullptr};  
    Node* p = new Node{5, q};  
}
```



Pointers Linked

```
class Node {  
public:  
    int    data;  
    Node* next;  
};
```

← constructor, destructor

```
int main( ) {  
    Node* p = new Node;  
    ...  
}
```

constructor →

destructor →

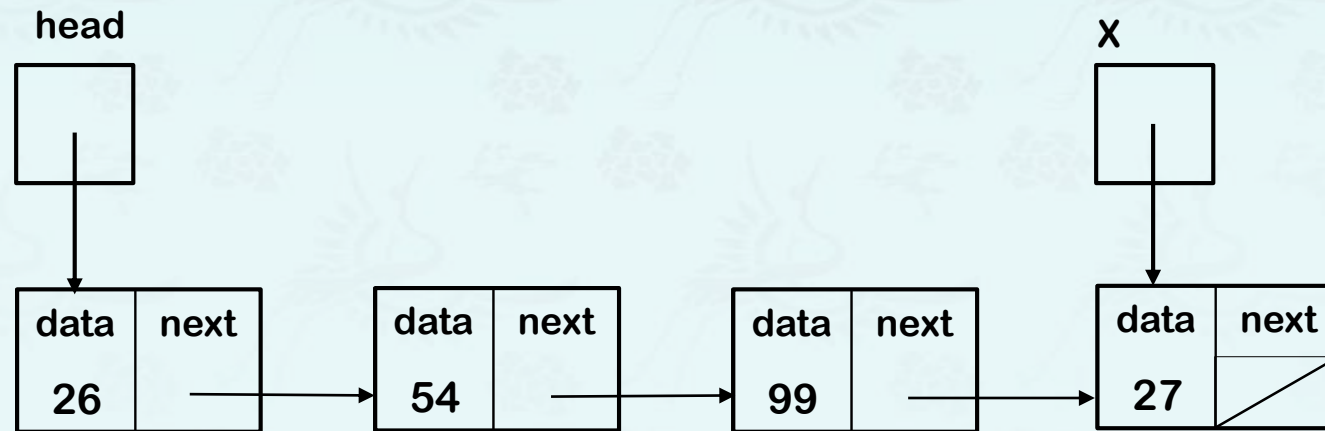
```
struct Node {  
    int    data;  
    Node* next;  
  
    Node(int i=0, Node* n=nullptr){  
        data = i, next = n;  
    }  
  
    ~Node() {};  
};  
  
int main( ) {  
    Node* p = new Node;  
    ...  
}
```

Linked List

```
struct Node {  
    int    data;  
    Node* next;  
};  
  
...  
  
Node* head, *x, *y;
```

basic member functions

- push_front()
- push_back()
- pop_front()
- pop_back()
- insert()
- remove()
- clear()

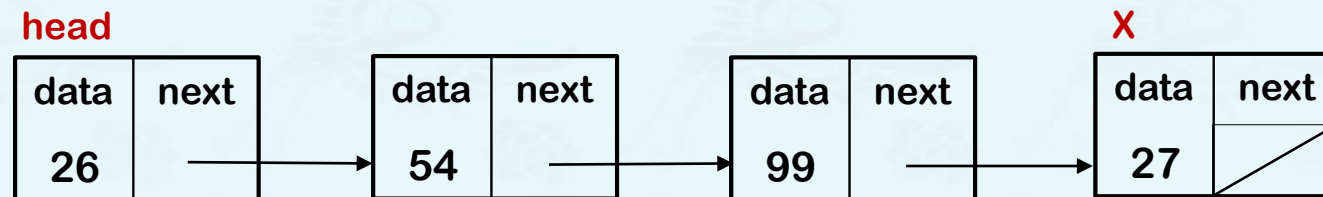


Linked List

```
struct Node {  
    int    data;  
    Node* next;  
};  
  
...  
  
Node* head, *x, *y;
```

basic member functions

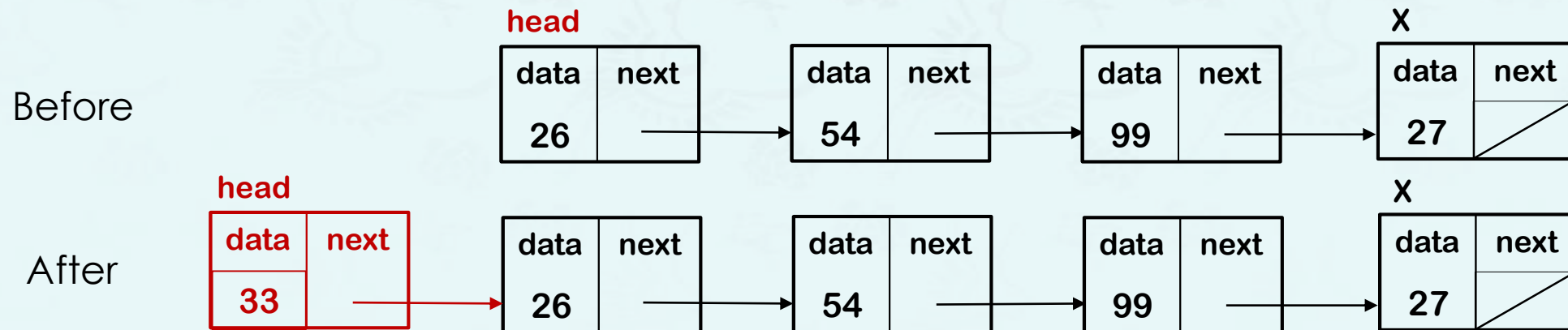
- push_front()
- push_back()
- pop_front()
- pop_back()
- insert()
- remove()
- clear()



Linked List - push_front()

Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (data = 33) at the **head of list**.



Linked List - push_front()

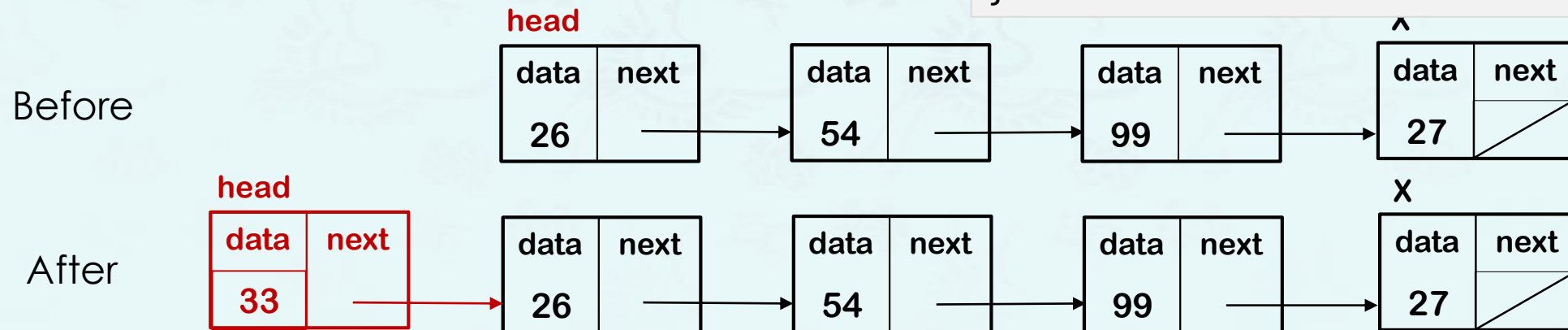
Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (data = 33) at the head of list.

```
Node* push_front(Node* head, int data) {  
    Node *y = new Node;  
    y->data = data;  
    y->next = head;  
    return y;  
}
```

```
Node* push_front(Node* head, data) {  
    Node *y = new Node {data, head};  
    return y;  
}
```

```
Node* push_front(Node* head, data) {  
    return new Node {data, head};  
}
```



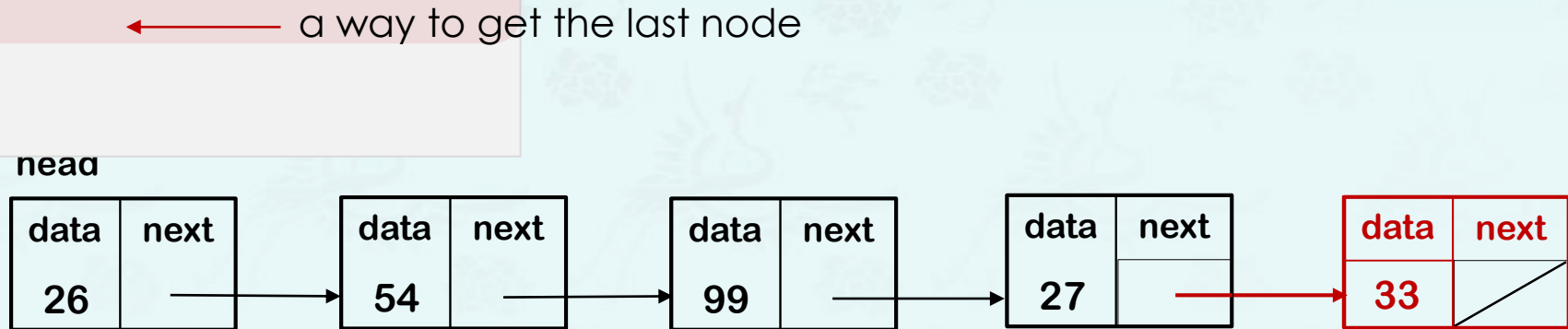
Linked List - push_back()

Let us imagine that we have created a linked list, where **head** points to the head of the list and **x** at the last item in the list (i.e. the one with the nullptr pointer) as shown below.

- Add a node (data = 33) at the end of list.

```
Node* push_back(Node* head, int data) {  
    Node *y = new Node {data, nullptr};  
    Node *x = head;  
    while (x->next != nullptr)  
        x = x->next;  
    x->next = y;  
    return head;  
}
```

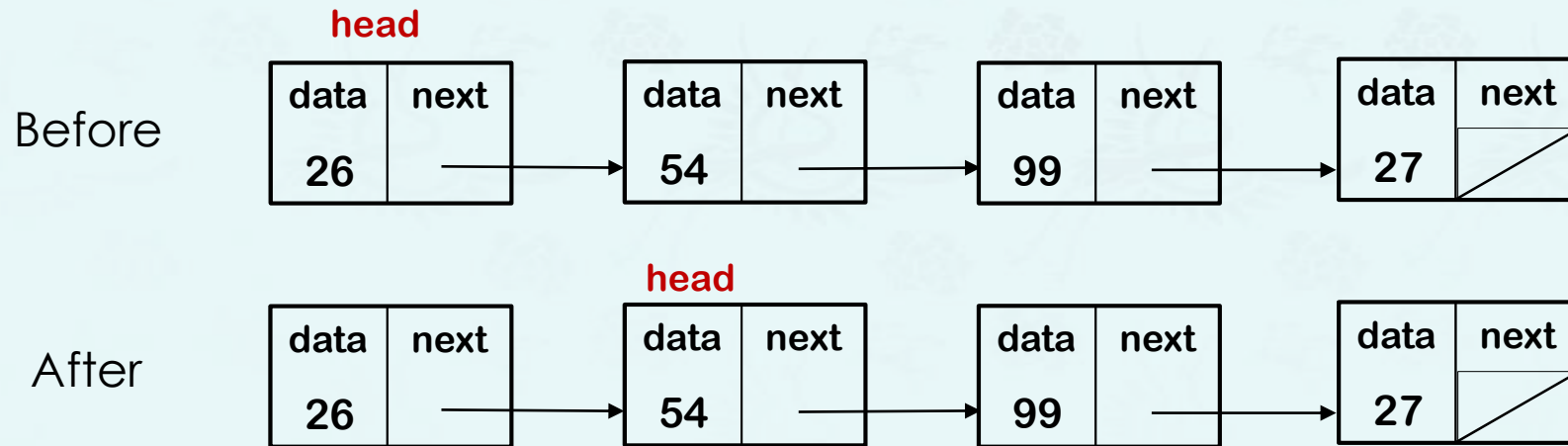
- To get to the tail we have to scroll along the list until the end. We want a pointer that will stop while still pointing at the last node. Thus our termination condition is that the node's next field is **nullptr**. Once we have a pointer to the end of the list, we can make it point to the node we want to add:



Linked List - pop_front()

- Remove the first node or move head to the next node.
Then what is wrong with the following code?

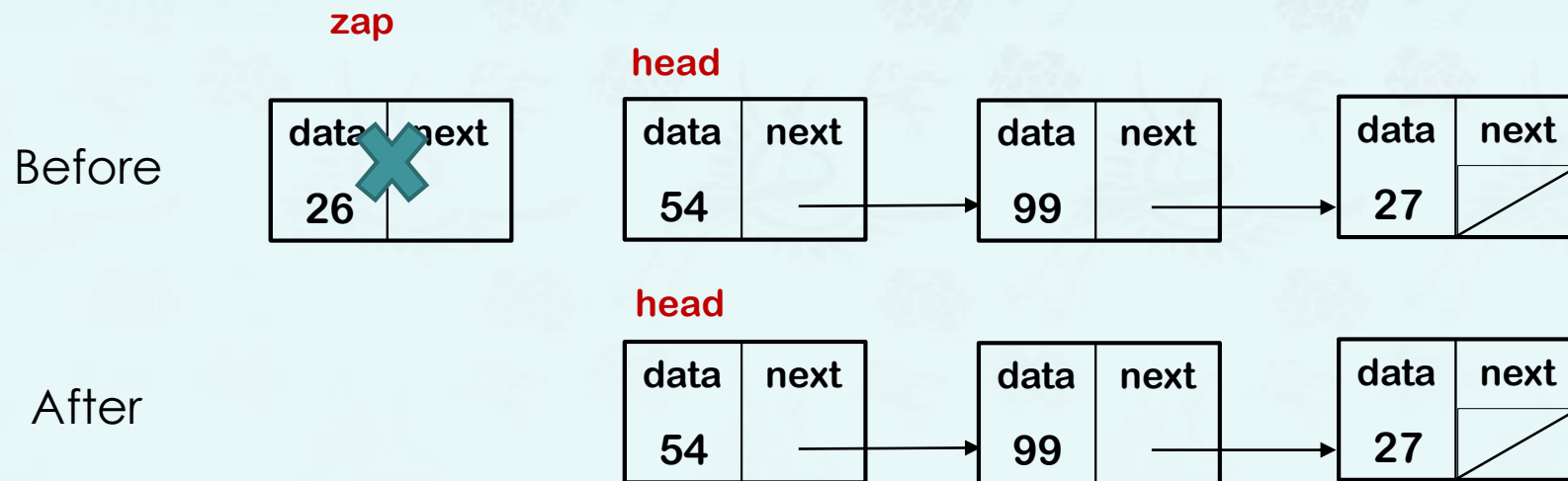
```
Node* pop_front(Node* head) {  
    head = head->next;  
    return head;  
}
```



Linked List - pop_front()

- Remove the first node or move head to the next node.
Then what is wrong with the following code?
- When removing a node, beware of memory leak; remember to give yourself a pointer to the node that is about to be removed before you lose your pointer to it:

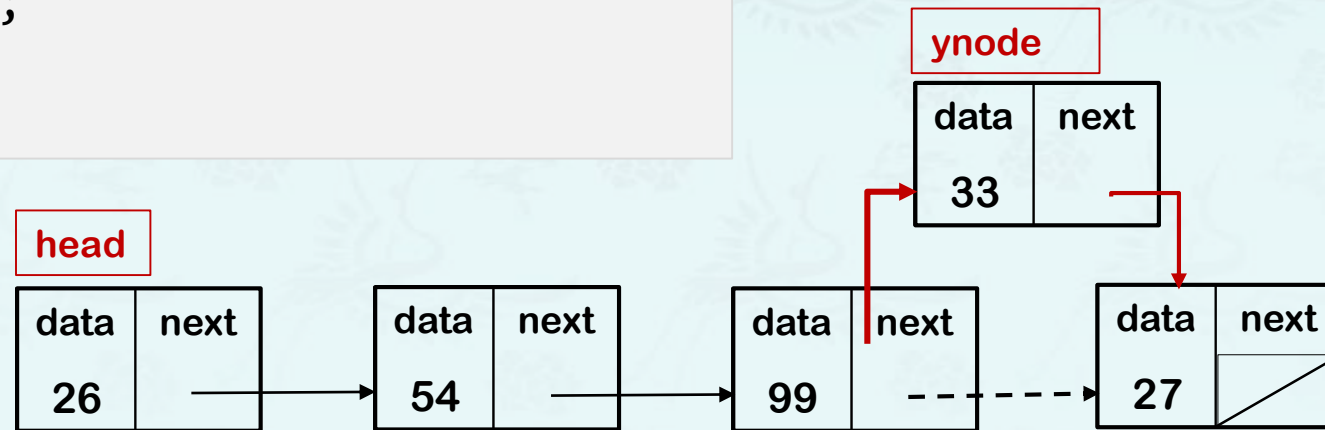
```
Node* pop_front(Node* head) {  
    Node* zap = head;  
    head = head->next;  
    delete zap;  
    return head;  
}
```



Linked List - insert()

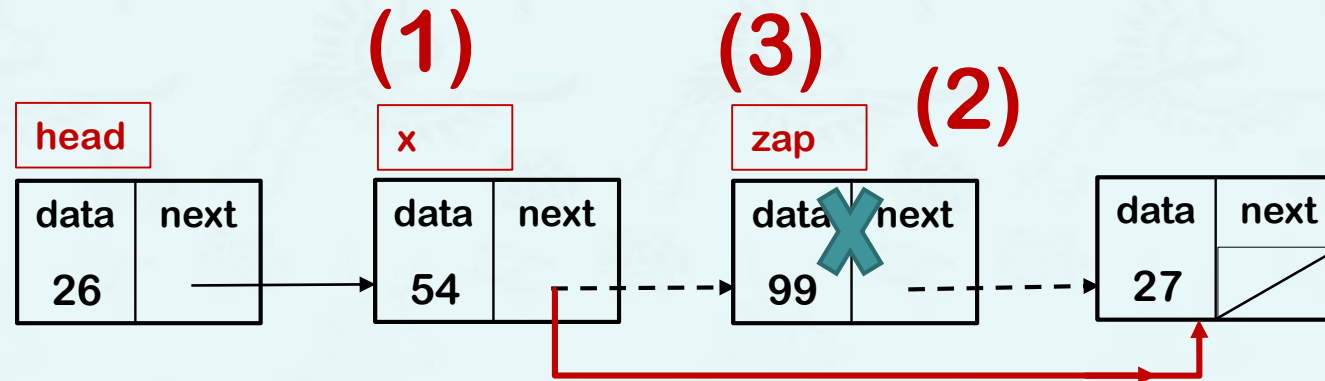
- Insert a new node(data = 33) **after** the node (key = 99) as shown below.
- Starting from the head node, we have to stop at the node (key = 99) before the insertion point. Remember that a singly-linked list is a one way street!

```
Node* insert(Node* head, int key, int data) {  
    Node* x = head;  
    while (x->data != key)           Where is x pointing after while()?  
        x = x->next;  
    Node* ynode = new Node {data, x->next};  
    x->next = ynode;  
    return head;  
}
```



Linked List - remove()

- Remove a node(key = 99) in the middle of list as shown below.



Linked List - remove()

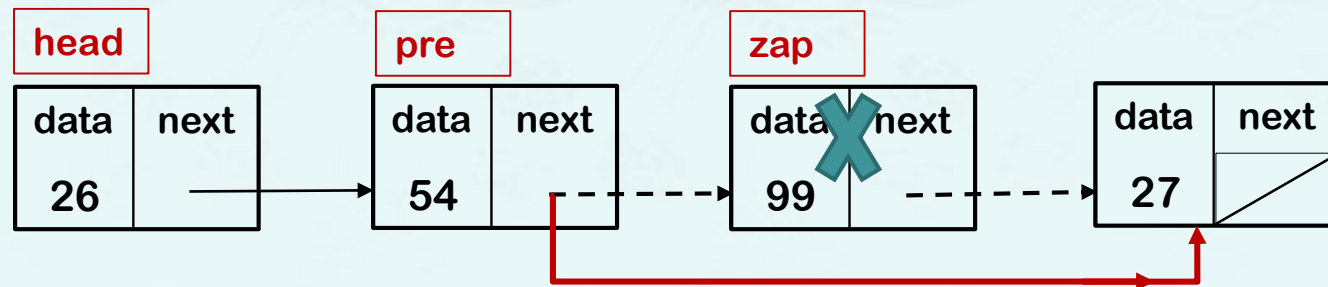
- Remove a node(key = 99) in the middle of list as shown below.
 - use a handle pointer (**zap** here) to keep hold of the unwanted node
 - find the node **before** the unwanted node and make links.
 - delete the unwanted node

```
Node* remove(Node* head, int key) {  
    node* pre = head,  
    node* zap = head->next;  
    while(zap->data != key) {  
        pre = zap;  
        zap = zap->next;  
    }  
    pre->next = zap->next;  
    delete zap;  
    return head;  
}
```

→ To find both pre and zap.

Assuming

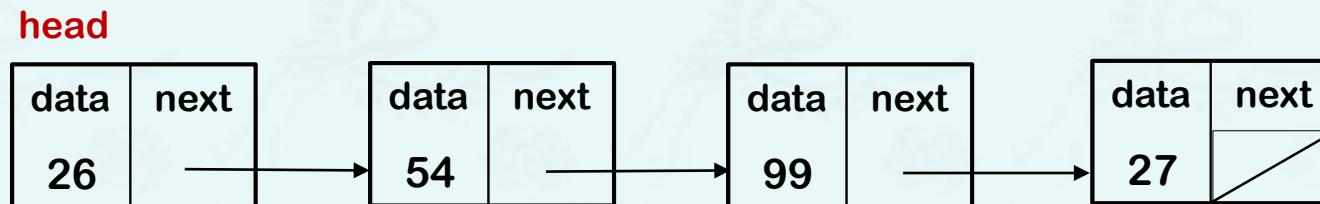
- (1) there are at least two nodes,
- (2) the key is not at the head node, and
- (3) there is a key node.



Linked List - clear()

Removes nodes from the list (which are destroyed), and leaving the list with a size of 0.

```
void clear(Node* head) {  
    Node *curr = head;  
    while(curr != nullptr) {  
        ...  
        delete ...  
    }  
}
```



Data Structures

Chapter 4

1. Singly Linked List
 - Pointer & Linking
 - **Singly Linked List (1)**
 - Singly Linked List (2)
 - Singly Linked List Operations
2. Doubly Linked List

Summary &
quaestio quaestio 90 < 9 9 ? ?
→