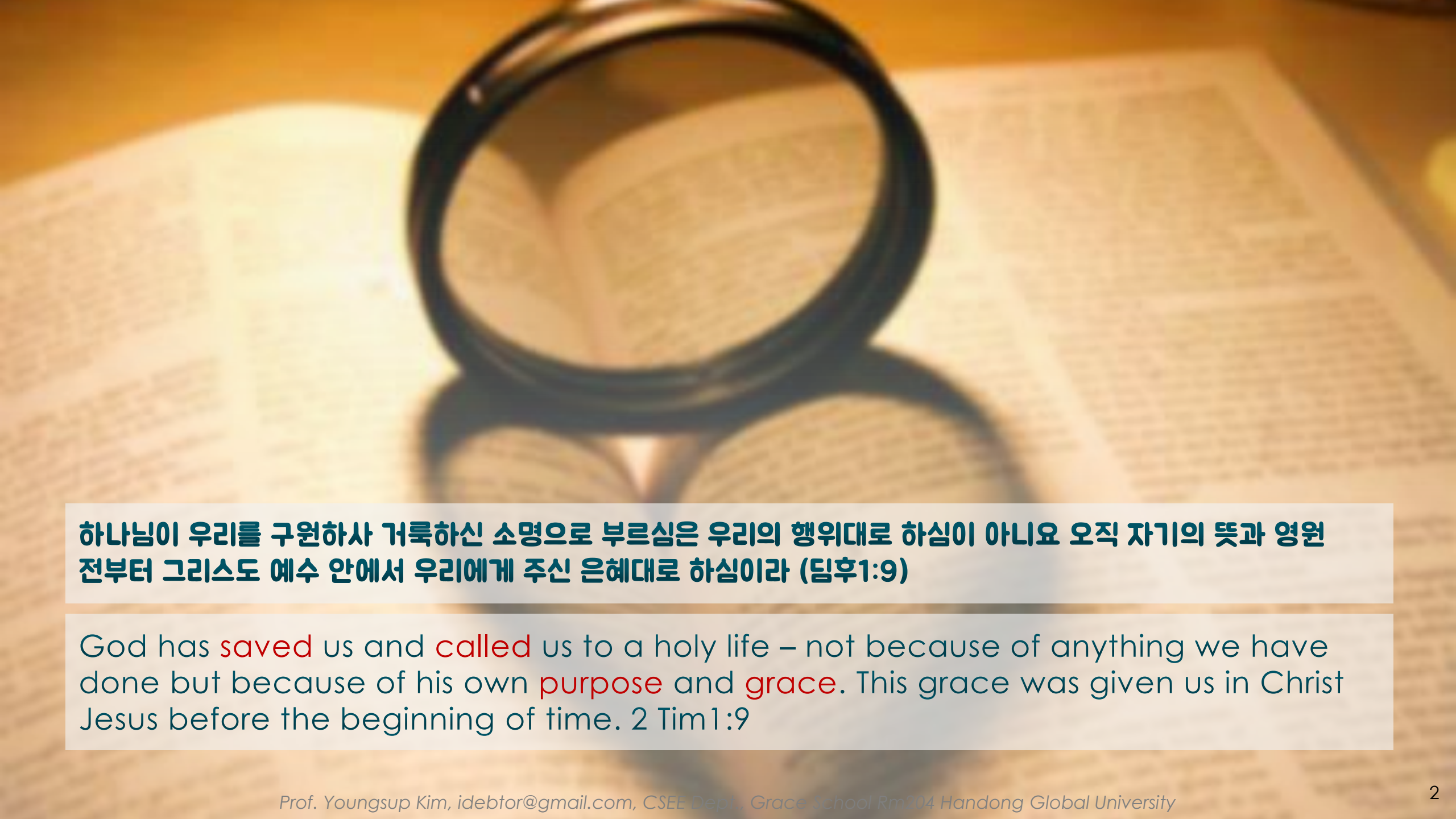


Data Structures

Chapter 4

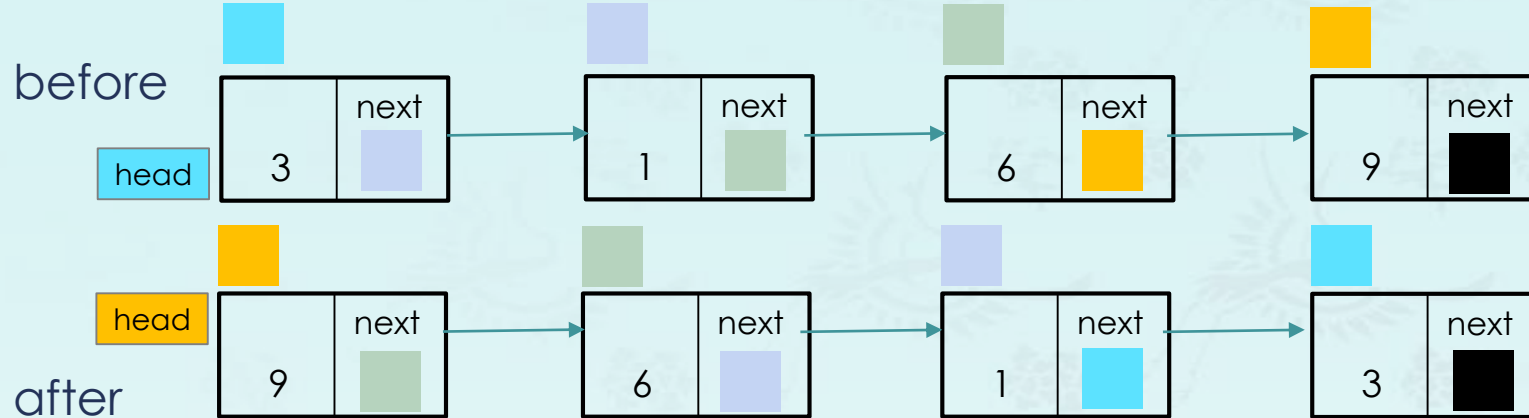
1. Singly Linked List
 - ◆ Pointer & Linking
 - ◆ Singly Linked List (SLL)
 - ◆ SLL Basic Operations (2)
 - ◆ **SLL advanced Operations**
2. Doubly Linked List



**하나님이 우리를 구원하사 기록하신 소명으로 부르심은 우리의 행위대로 하심이 아니요 오직 자기의 뜻과 영원
전부터 그리스도 예수 안에서 우리에게 주신 은혜대로 하심이라 (딤후1:9)**

God has **saved** us and **called** us to a holy life – not because of anything we have done but because of his own **purpose** and **grace**. This grace was given us in Christ Jesus before the beginning of time. 2 Tim1:9

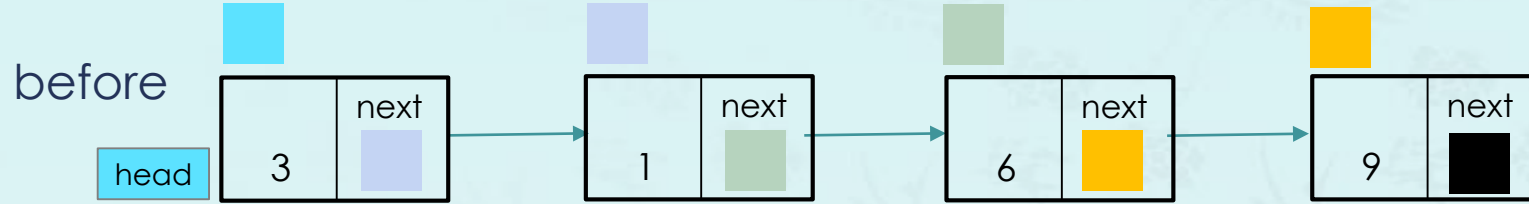
Linked List – reverse using stack



Algorithm:

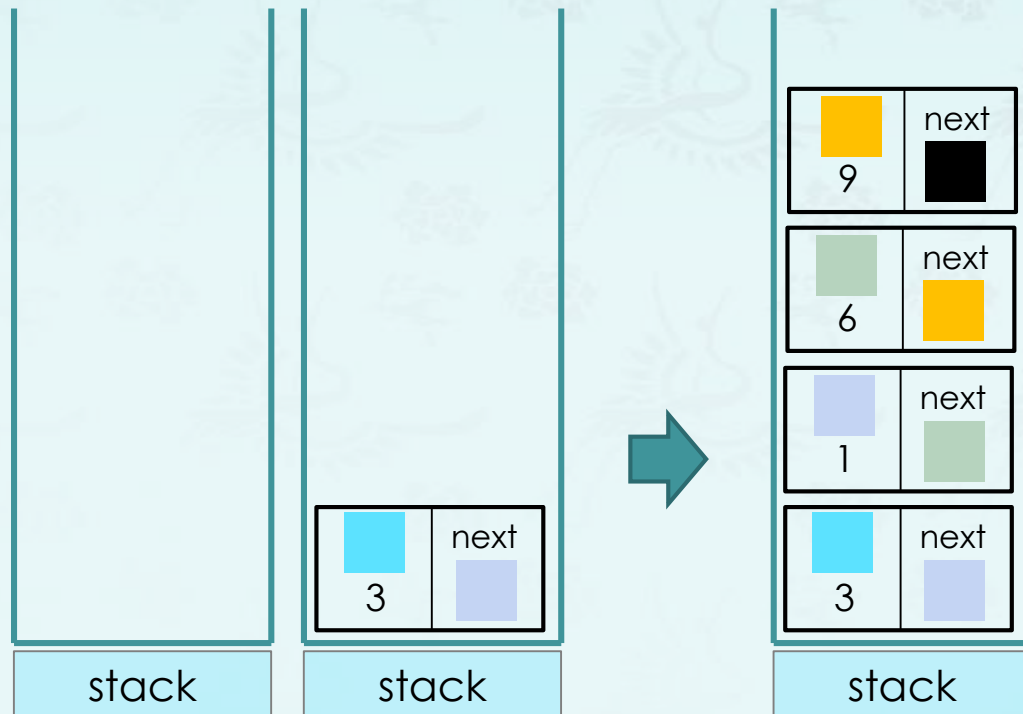
Step 1. Push all nodes onto the stack.
Step 2. Top/pop all nodes and relink.

Linked List – reverse using stack

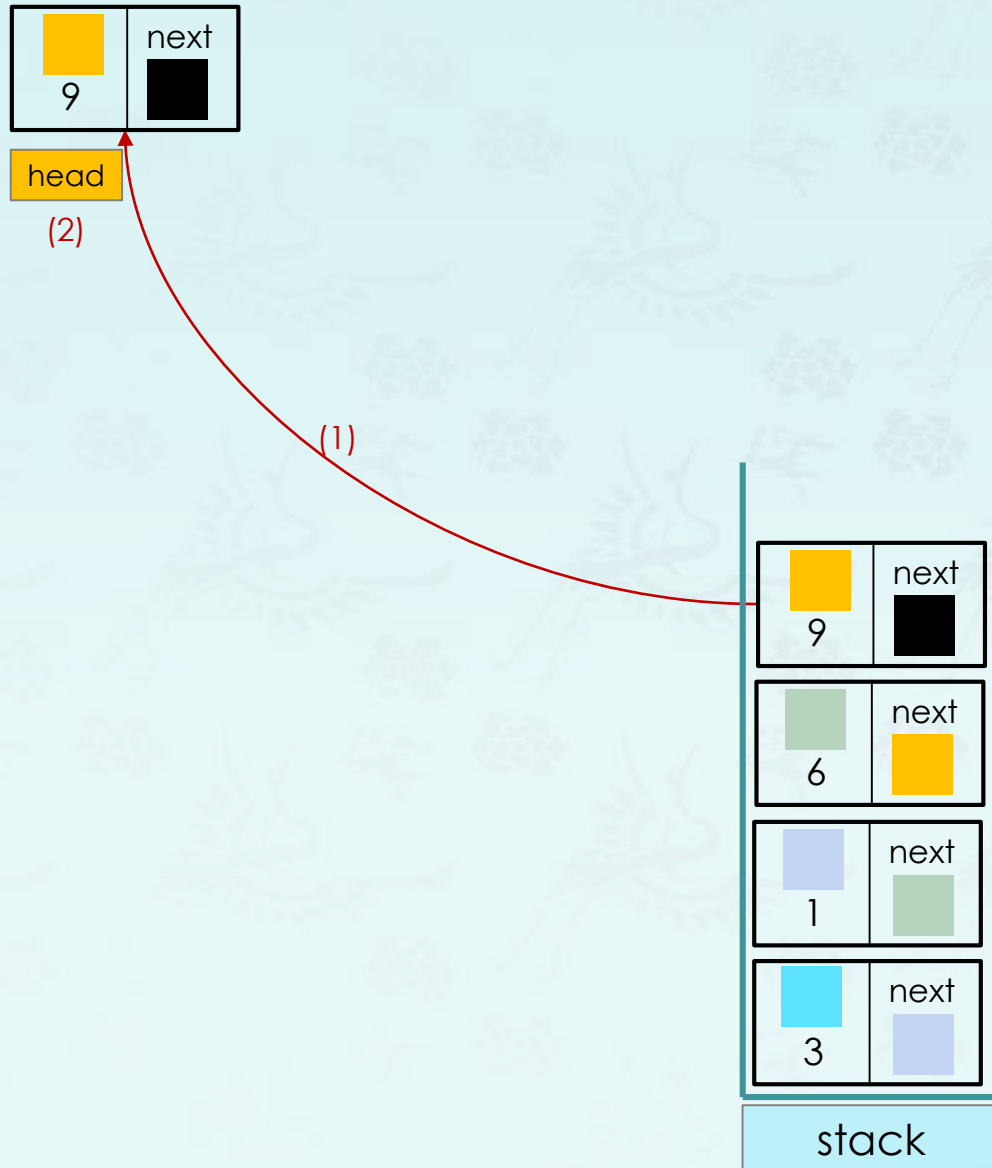


Algorithm:

Step 1. Push all nodes onto the stack.
Step 2. Top/pop all nodes and relink.



Linked List – reverse using stack

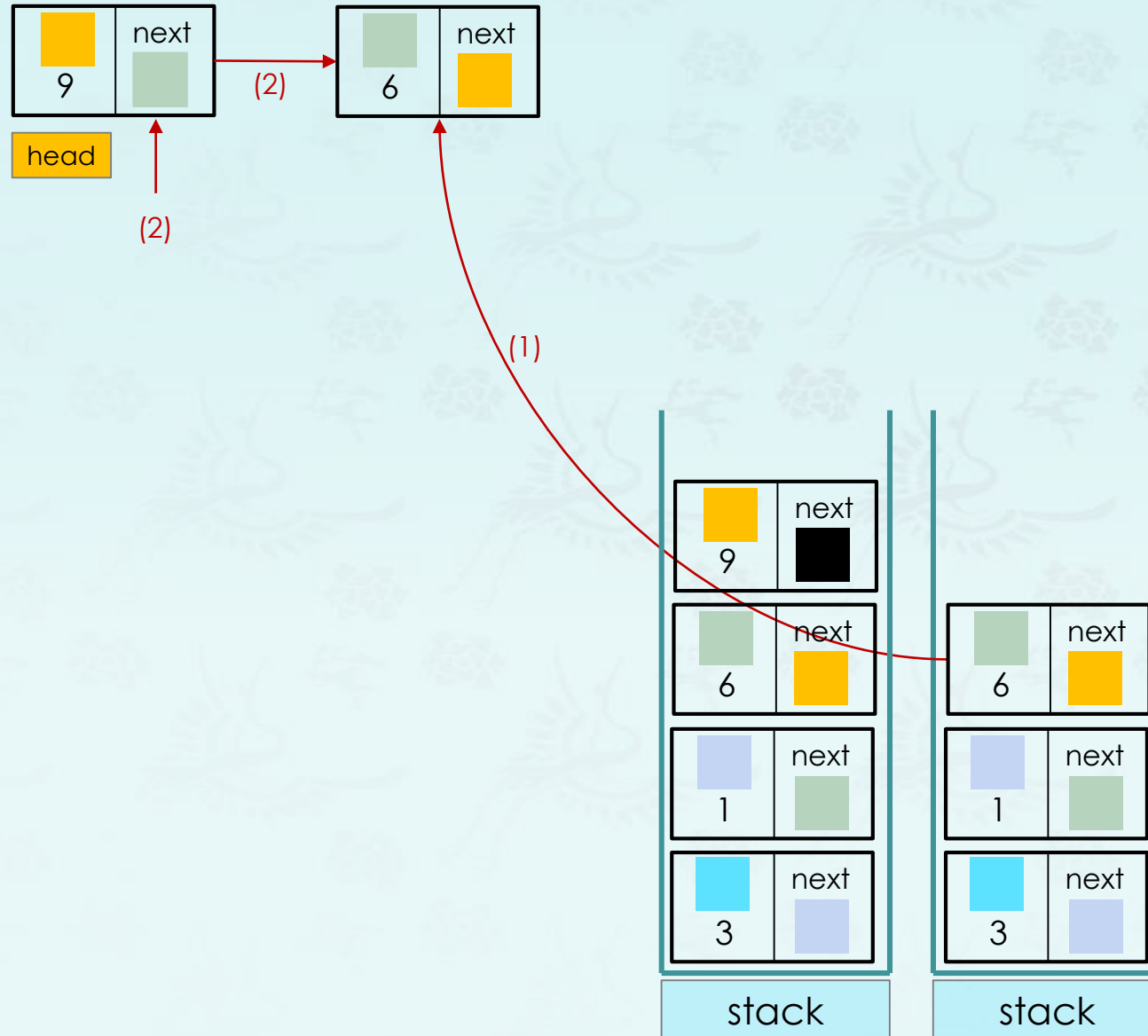


Algorithm:

Step 1. Push all nodes onto the stack.

Step 2. Top/pop all nodes and relink.

Linked List – reverse using stack

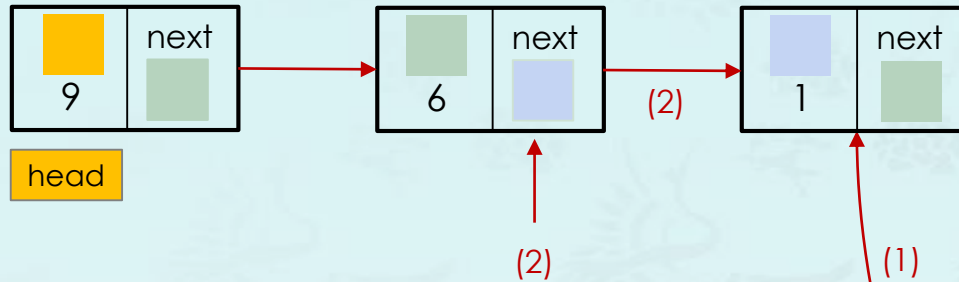


Algorithm:

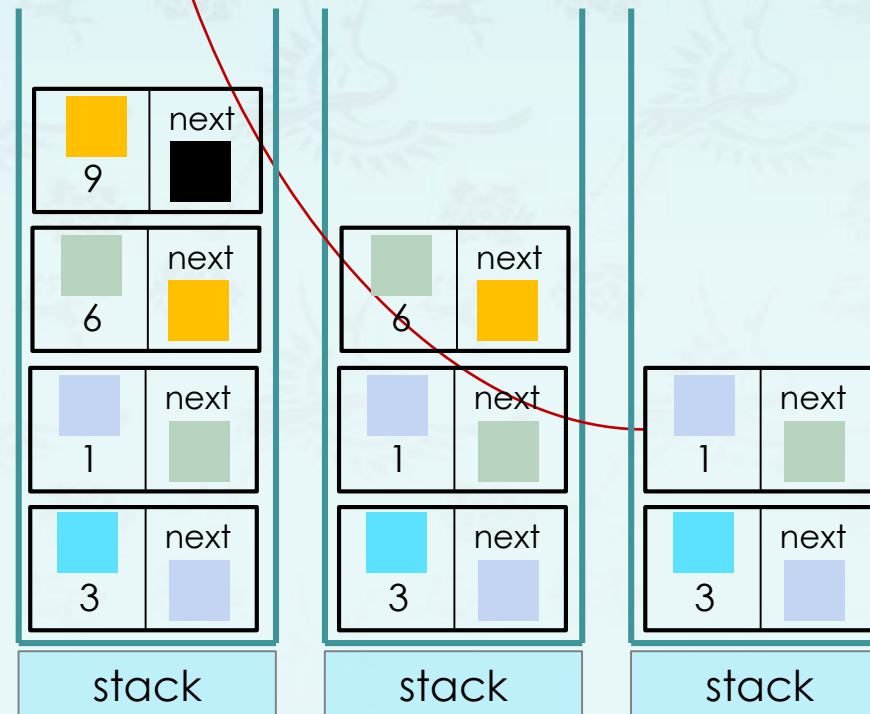
Step 1. Push all nodes onto the stack.

Step 2. Top/pop all nodes and relink.

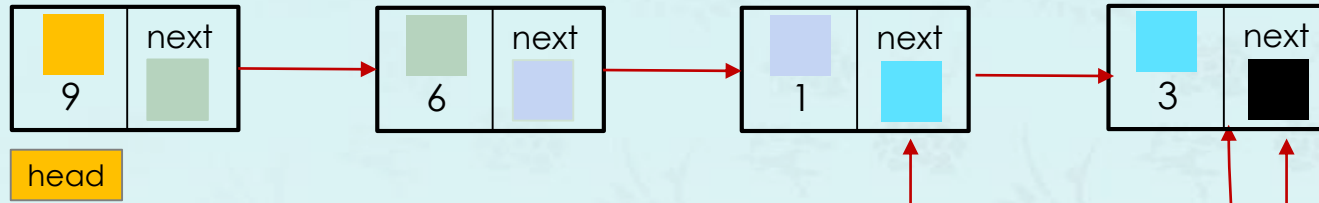
Linked List – reverse using stack



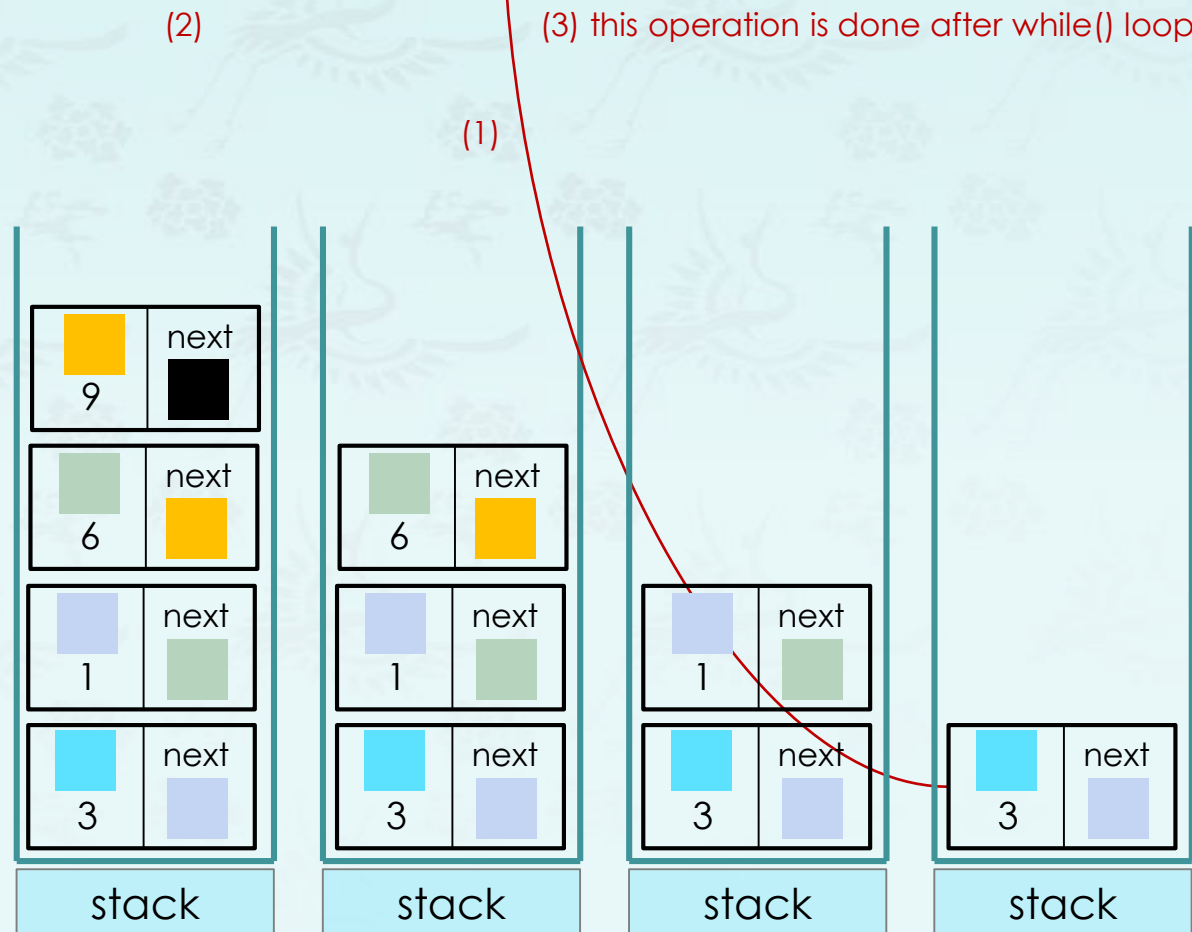
Algorithm:
Step 1. Push all nodes onto the stack.
Step 2. Top/pop all nodes and relink.



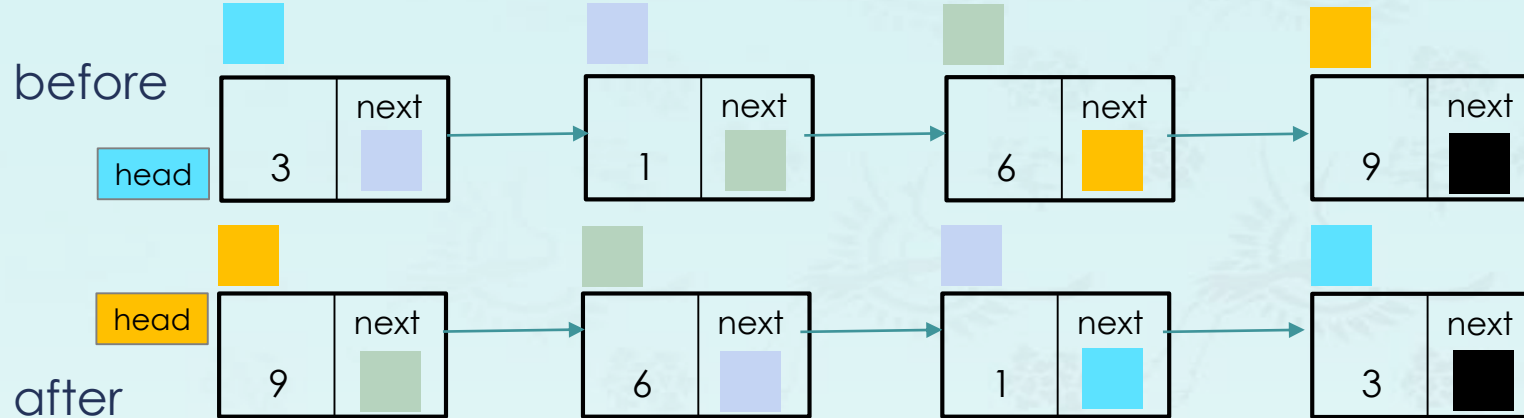
Linked List – reverse using stack



Algorithm:
Step 1. Push all nodes onto the stack.
Step 2. Top/pop all nodes and relink.

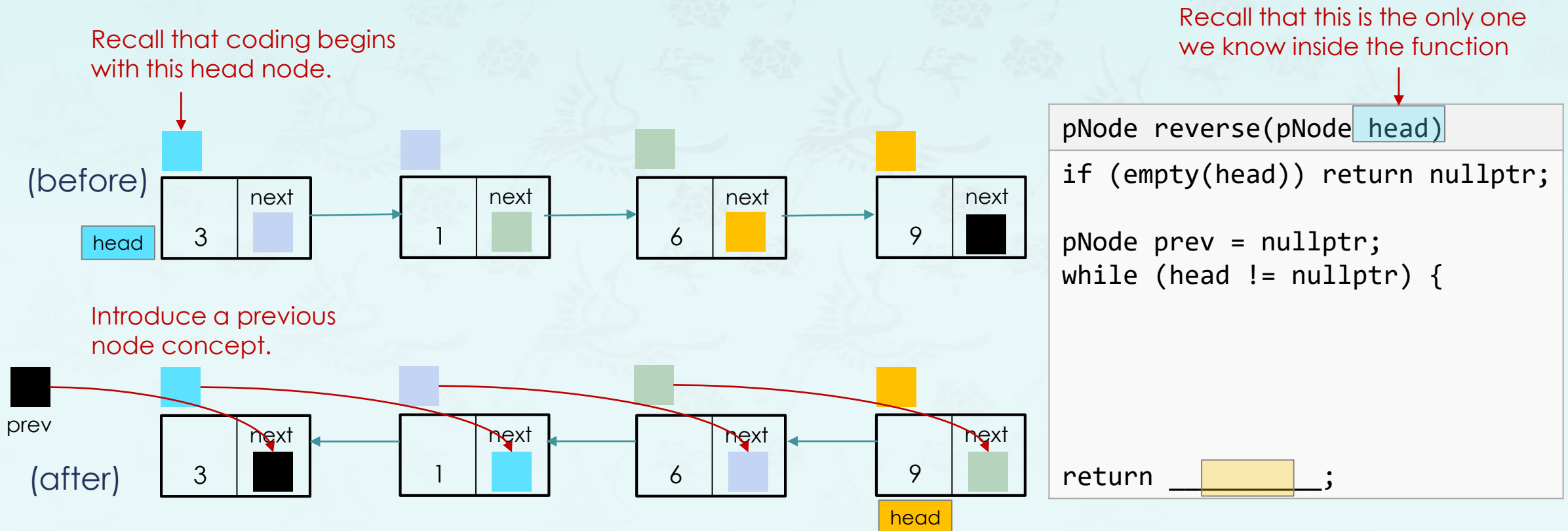


Linked List – reverse using stack



```
pNode reverse(pNode head)
{
    if (empty(head)) return nullptr;
    while( list is not empty )
        get a node from list
        push it onto the stack
    }
    while( stack is not empty )
        get a node from the stack
        relink it back the new list
    }
    return head; // new head
```

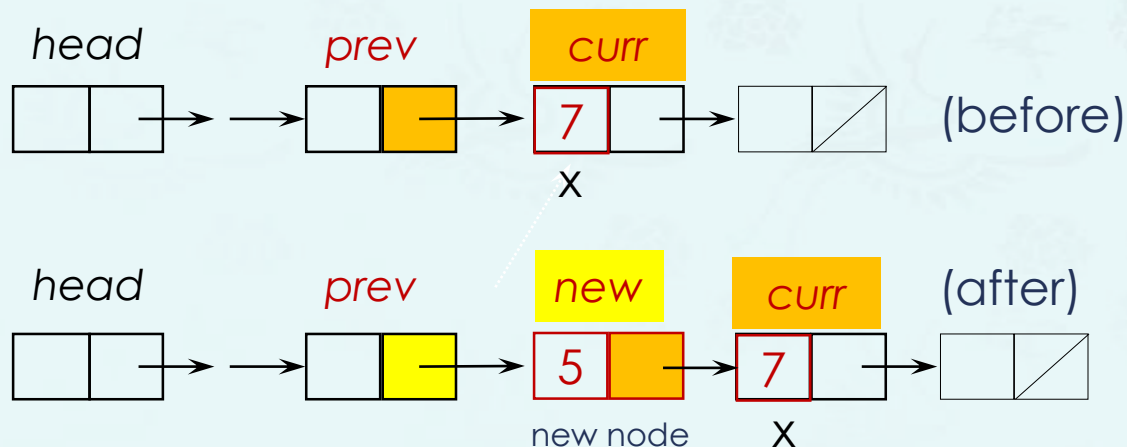
Linked List – reverse in-place



Linked List – insert()

TASK: Code a function that inserts a node(5) **at a node position x** specified by a value(7).

- If the first node(or **head**) is the position, then just invoke **push_front()**.
- As observed below, we must to know **the pointer x** which is stored in the **previous node** of node x.

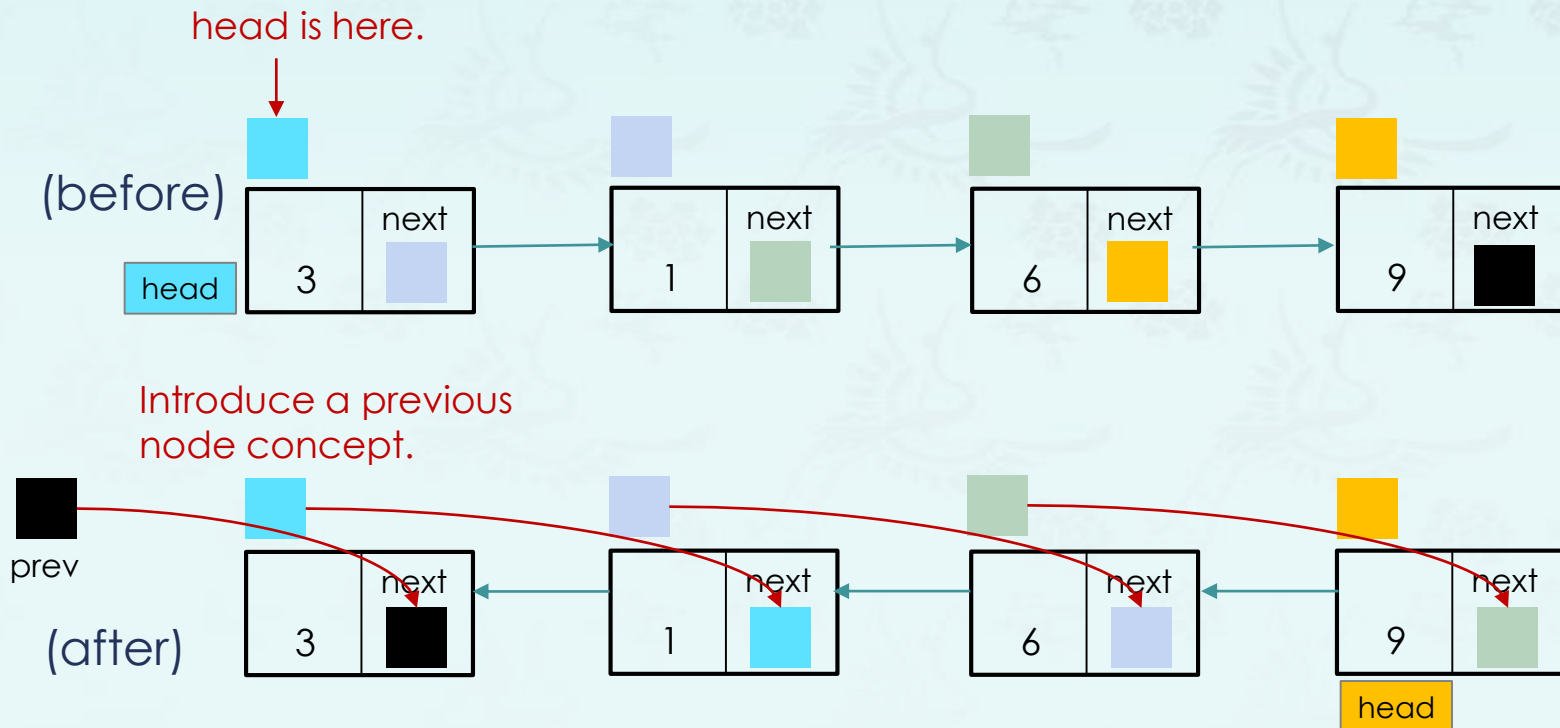


```
pNode insert(pNode head, int val, int x)
{
    if (head->data == x)
        return push_front(val, head);

    pNode curr = head;
    pNode prev = nullptr;
    while (curr != nullptr) {
        if (curr->data == x) {
            prev->next = new Node{val, prev->next};
            return head;
        }
        prev = curr;
        curr = curr->next;
    }
    return head;
}
```

Linked List – reverse in-place

1. We want to overwrite **head->next** with **prev**. But before overwriting we must save **head->next** because we it is the next node we need to process.
2. Once we overwrite **head->next** with **prev**,
3. be ready to process the next node by setting **prev** as and **head** as .



```
pNode reverse(pNode head)
if (empty(head)) return nullptr;

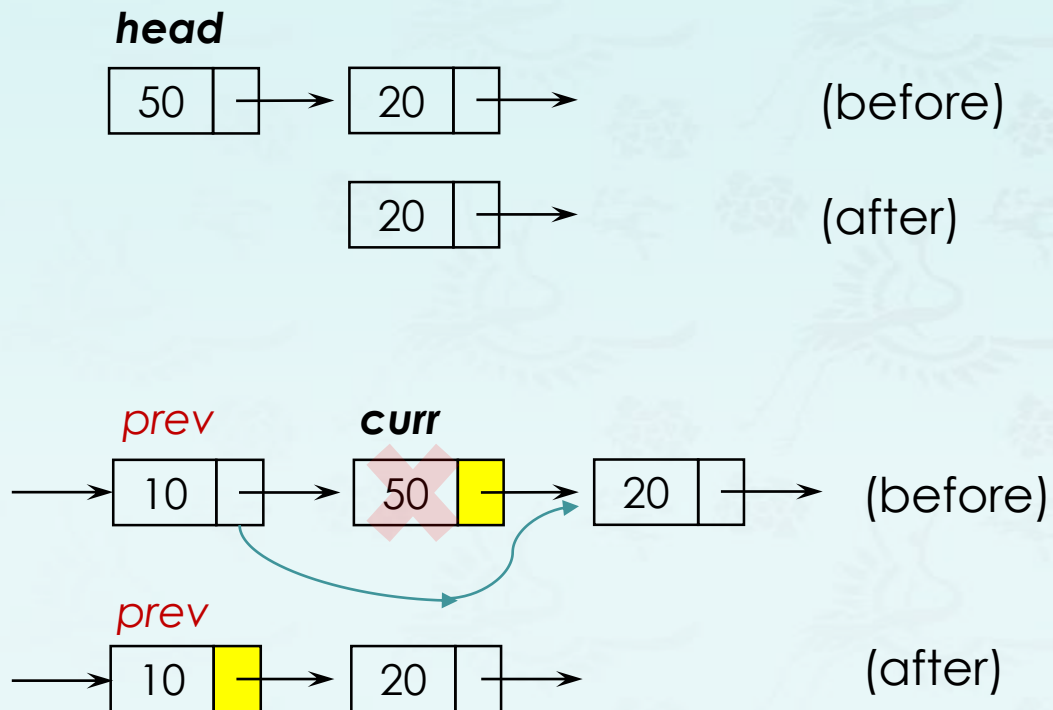
pNode prev = nullptr;
while (head != nullptr) {
    (1)
    (2)
    (3)
    (3)
}
return head;
```

Which one has the last node address to return as a new head after while loop?

Linked List – pop()

TASK: Code a function that deletes a node with a value specified.

- If the first node(or **head**) is the one to delete, then just invoke **pop_front()**.
- As observed below, we must to know **the pointer x** which is stored in the **previous node** of node x.



```
pNode pop(pNode head, int val)
```

```
if (head->data == val)
    return pop_front(head);
```

```
pNode curr = head;
pNode prev = nullptr;
while (curr != nullptr) {
    if (curr->data == val) {
        prev->next = curr->next;
        delete curr;
        return head;
    }
    prev = curr;
    curr = curr->next;
}
return head;
```

Simplifying this while() loop is left as a part of Problem Set.

Linked List – pop()

TASK: Code a function that deletes a node with a value specified.

- If the first node(or **head**) is the one to delete, then just invoke **pop_front()**.
- As observed below, we must to know **the pointer x** which is stored in the **previous node** of node x.

- Simplify the code by rewriting while loop.



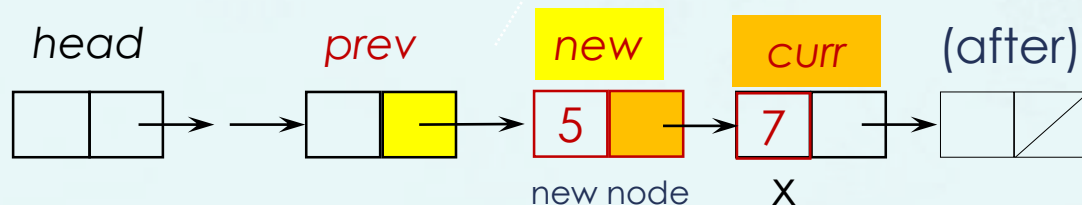
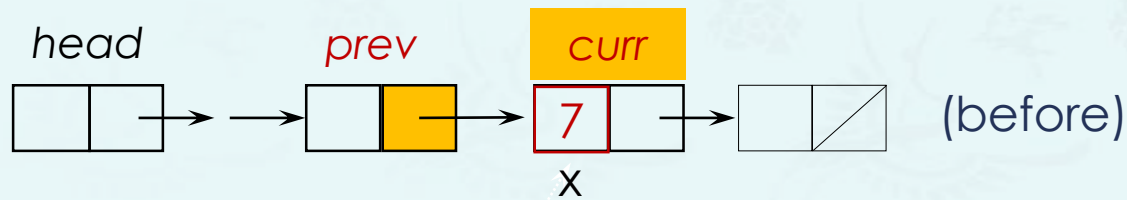
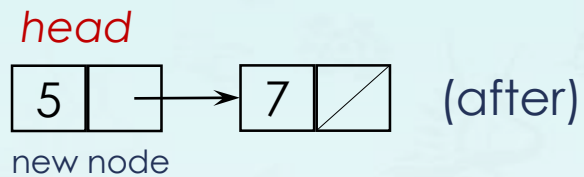
```
pNode pop(pNode head, int val)
{
    if (head->data == val)
        return pop_front(head);

    pNode curr = head;
    pNode prev = nullptr;
    while (curr != nullptr) {
        if (curr->data == val) {
            prev->next = curr->next;
            delete curr;
            return head;
        }
        prev = curr;
        curr = curr->next;
    }
    return head;
}
```


Linked List – insert() or push()

TASK: Code a function that inserts a node(5) **at a node position x** specified by a value(7).

- If the first node(or **head**) is the position, then just invoke **push_front()**.
- As observed below, we must to know **the pointer x** which is stored in the **previous node** of node x.



```
pNode insert(pNode head, int val, int x)
{
    if (head->data == x)
        return push_front(val, head);

    pNode curr = head;
    pNode prev = nullptr;
    while (curr != nullptr) {
        if (curr->data == x) {
            prev->next = new Node{val, prev->next};
            return head;
        }
        prev = curr;
        curr = curr->next;
    }
    return head;
}
```

Simplifying this while() loop is left as a part of Problem Set.

Linked List – insert() or push()

TASK: Code a function that inserts a node(5) **at a node position x** specified by a value(7).

- If the first node(or **head**) is the position, then just invoke **push_front()**.
- As observed below, we must to know **the pointer x** which is stored in the **previous node** of node x.

- Simplify the code by rewriting while loop.



```
pNode insert(pNode head, int val, int x)
{
    if (head->data == x)
        return push_front(val, head);

    pNode curr = head;
    pNode prev = nullptr;
    while (curr != nullptr) {
        if (curr->data == x) {
            prev->next = new Node{val, prev->next};
            return head;
        }
        prev = curr;
        curr = curr->next;
    }
    return head;
}
```

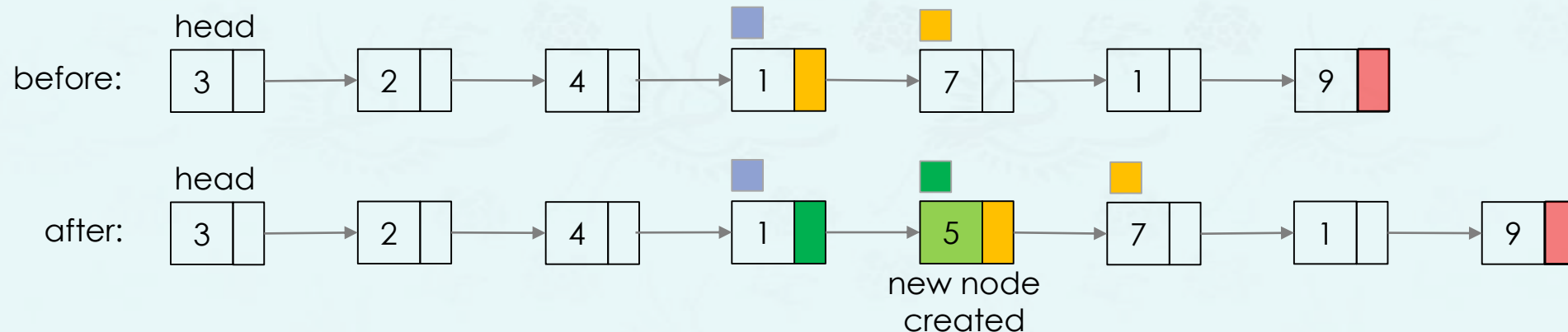
Linked List – push_sorted()

- **inserts a new node in sorted ascending order.**

The basic strategy is to iterate down the list looking for the place to insert the new node. That could be the end of the list, or a point just before a node.

```
// inserts a new node with value in sorted order
Node* push_sorted(Node* p, int value) {
    if (empty(p) || value <= p->data) return push_front(p, value);
    // your code here
    return p;
}
```

- Which node should be located to invoke push_sorted() if value = 5?

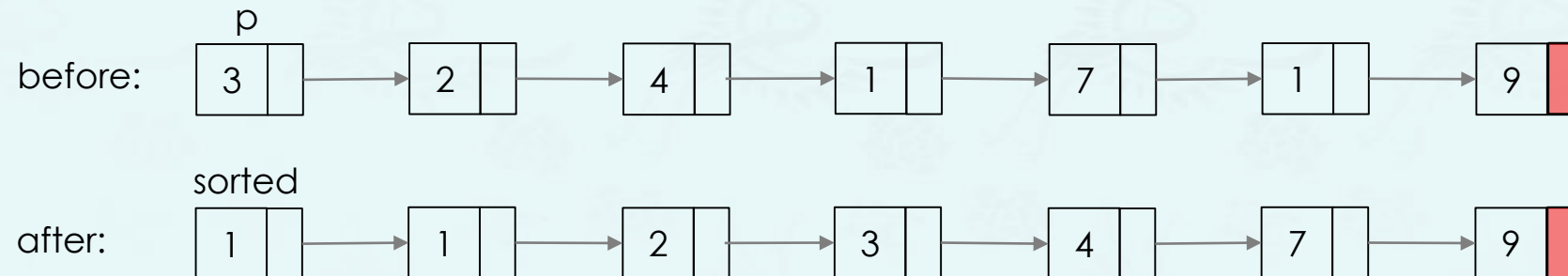


Linked List – insertion_sort()

- sorts the singly linked list using insertion sort and returns a new list sorted.
 - Repeatedly, invoke push_sorted() with a value in the list such that push_sorted() returns a newly formed list head. Deallocate the old list.

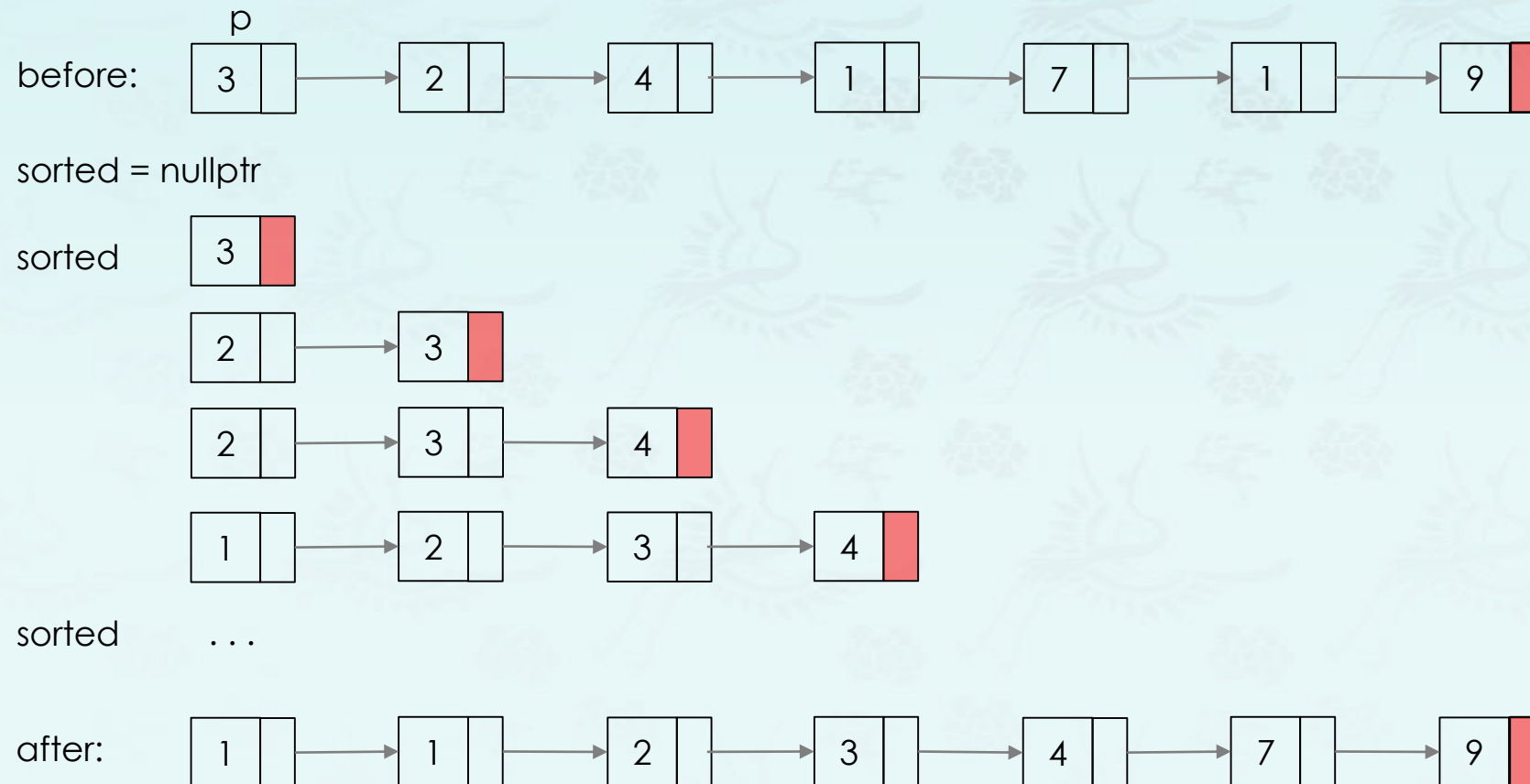
```
// inserts a new node with value in sorted order
Node* insertion_sort(Node* p) {
    if (empty(p)) return nullptr;
    if (size(p) < 2) return p;

    Node* sorted = nullptr; // sorted will be newly formed using push_sorted()
    // your code here - a while loop invokes push_sorted() for all nodes in p
    return sorted;
}
```



Linked List – insertion_sort()

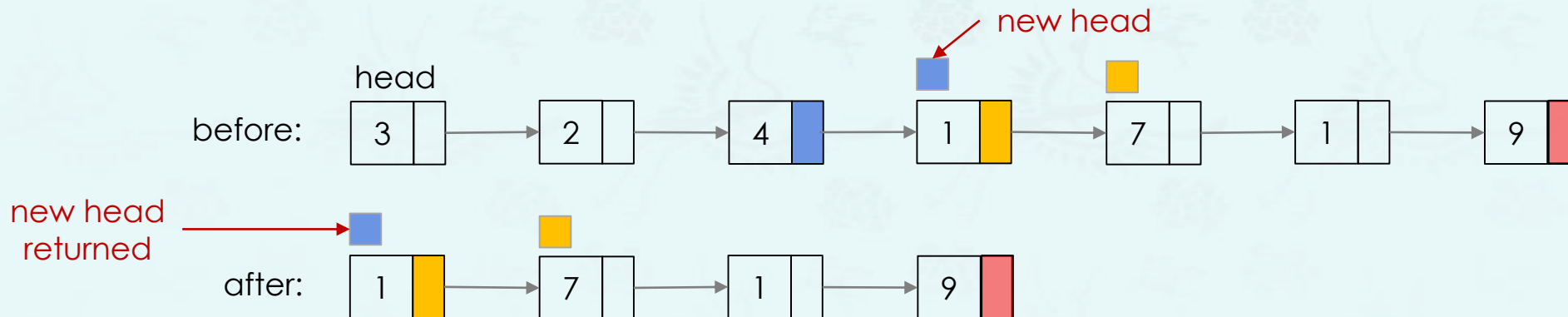
- sorts the singly linked list using insertion sort and returns a new list sorted.
 - Repeatedly, invoke push_sorted() with a value in the list such that push_sorted() returns a newly formed list head. Deallocate the old list.



Linked List – keep_second_half()

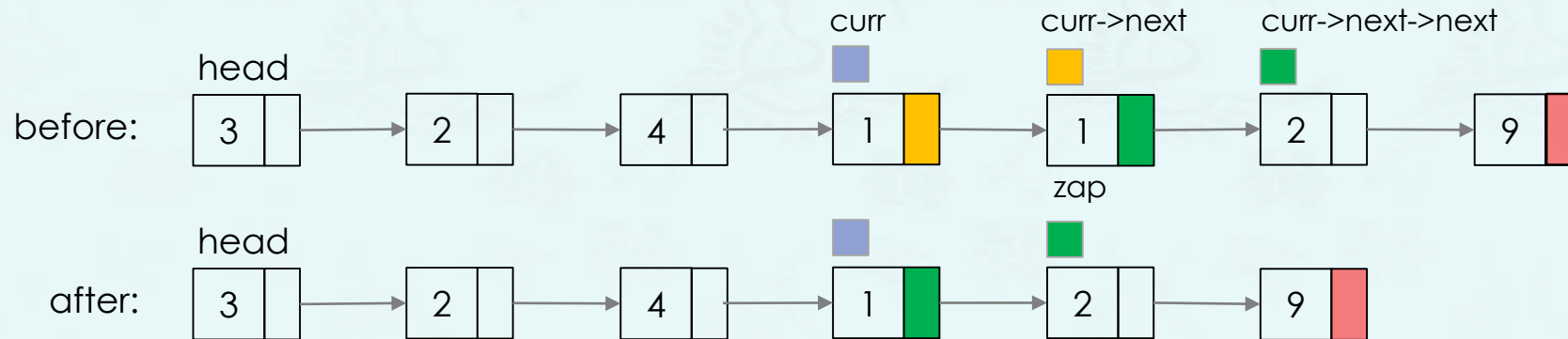
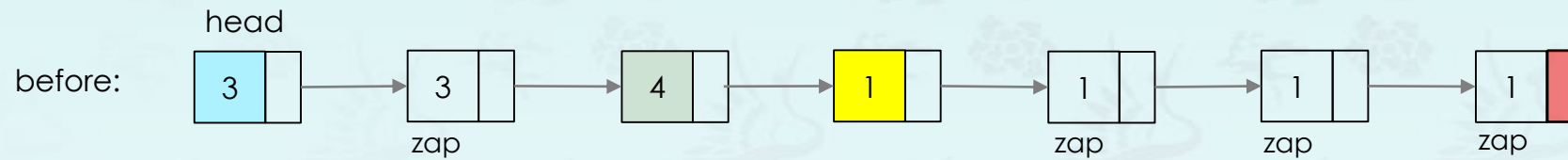
- removes the first half of the list and returns the new head of the list which begins with the second half. If there are odd number of nodes, remove less and keep more. For example, keep 5 nodes if there are 9 nodes.

```
Node* keep_second_half(Node* p) {    // hint: copy and modify clear() provided.  
    if (empty(p)) return nullptr;  
    // your code here  
    return new head ... ;  
}
```



Linked List – zap_duplicates()

- Removes consecutive items in the list, and leaves its neighbors unique. We can proceed down the list and compare adjacent nodes.
- When adjacent nodes are the same, remove the second one. There's a tricky case where the node after the next node needs to be noted before the deletion.
- Your implementation must go through the list **only once**.



Data Structures

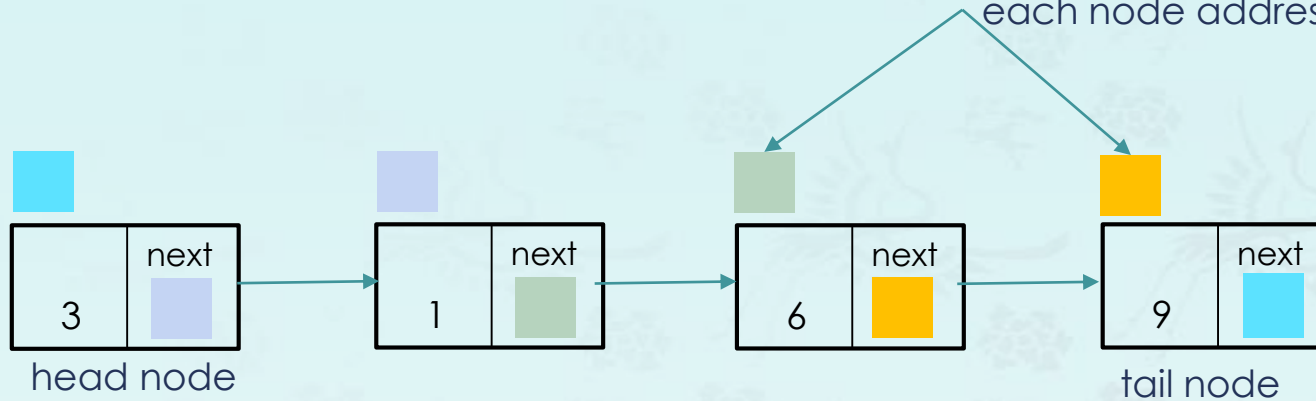
Chapter 4

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

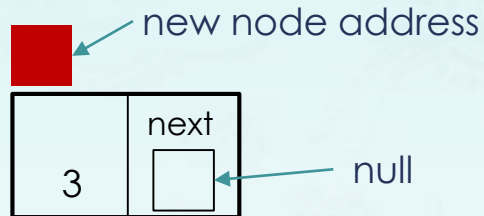
Summary &
quaestio quaestio 90 < 9 9 ? ?

Circular list using singly linked-List

each node address: 각 node address는 그 앞에 노드가 저장하고 있다.



Either head or tail node only is provided.



head node정보만 가지고 있을 때,

예를 들면, new node를 head 앞에 넣으려면, tail node를 찾아내야 $O(n)$, tail node와 head node사이 insert할 수 있다. tail node 다음에 삽입하는 것도 마찬가지다. 항상 그 앞에 노드의 주소 값을 알아야 삽입이나 삭제가 가능하고 그 둘을 연결할 수 있으니까, Singly linked list에서는 그 앞에 노드를 찾는데 worst case 를 고려하면 항상 $O(n)$ 된다.

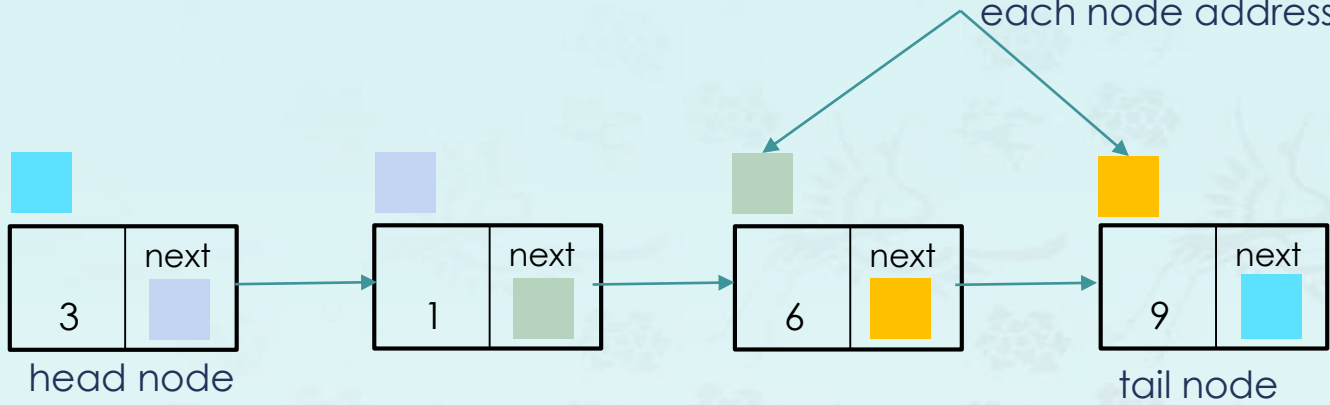
반면에, tail node 정보를 갖고 있다면,

tail node 다음에 insert하거나 delete하는 것은 $O(1)$ 으로 가능하다. 왜냐하면, tail node 자체는 그 다음 노드 (head node)의 address를 이미 갖고 있으니 얼마든지 new node를 연결할 수 있겠다. 대부분 작업이 한번에 $O(1)$ 가능하다. 다만, 자기 자신(tail node)를 삭제하는 것을 제외하고 말이다. 즉 tail node를 삭제하려면 그 앞에 있는 노드(그림에서 6 노드)를 찾아내야 ($O(n)$ 으로), tail node를 삭제하고 노드 6과 노드 3을 연결할 수 있다.

기억할 점: Singly listed list에서 한 노드를 삭제하거나 삽입하려면, 그 앞에 노드 정보가 필요하다. circular list 이므로 tail node는 이미 start node 정보도 가지고 있으므로 더 유리하다.

Circular list using singly linked-List

each node address: 각 node address는 그 앞에 노드가 저장하고 있다.



Either head or tail node only is provided.

| Time complexity | function | x = 3 | x = 9 |
|------------------|-----------------|-------|-------|
| Using head and x | insert(head, x) | 0(n) | 0(n) |
| | delete(head, x) | 0(n) | 0(n) |
| Using tail and x | insert(tail, x) | 0(1) | 0(1) |
| | delete(tail, x) | 0(1) | 0(n) |