Data Structures
Chapter 4

1. Singly Linked List
   - Pointer Reviewed & Linked
   - Linked List (1)
   - Linked List (2)
   - **Reverse Singly Linked List**
     - in-place O(n)
     - using stack O(n)
     - sub-list reverse O(n^2),
     - sub-list reverse O(n)

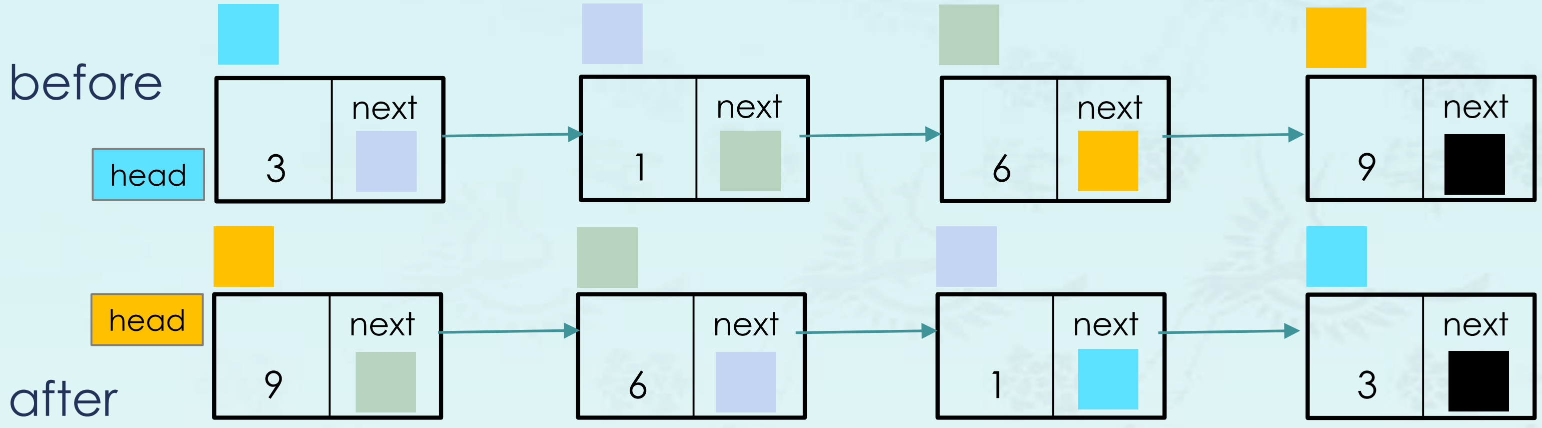
Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University

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

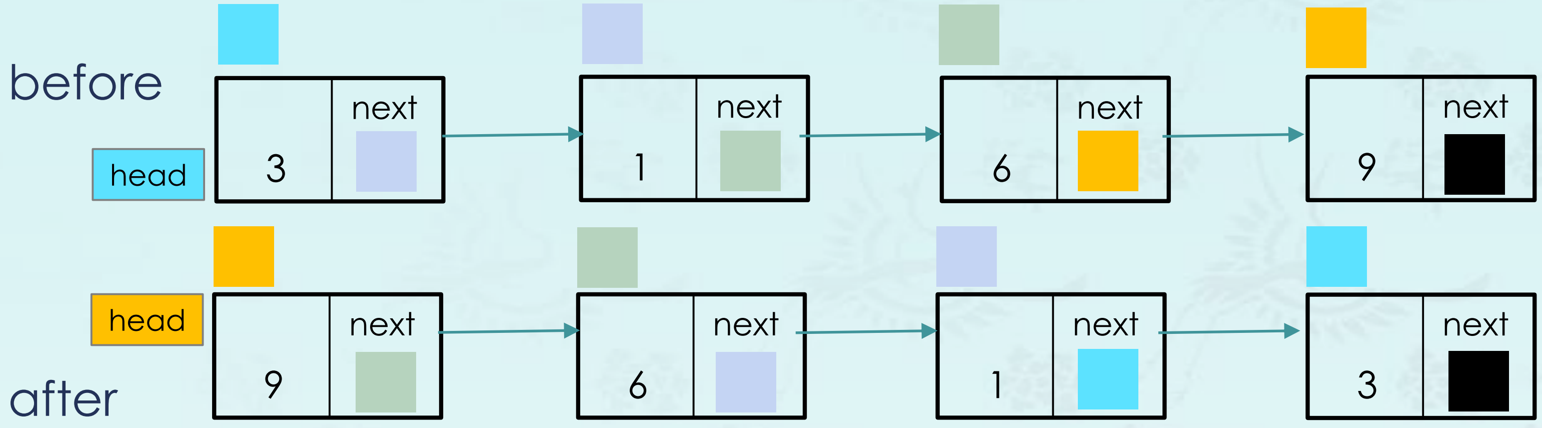
하나님이 우리를 구원하사 거룩하신 소명으로 부르심은 우리의 행위대로 하심이 아니요 오직 자기의 뜻과 영원 전부터 그리스도 예수 안에서 우리에게 주신 은혜대로 하심이라 (딤후1:9)

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

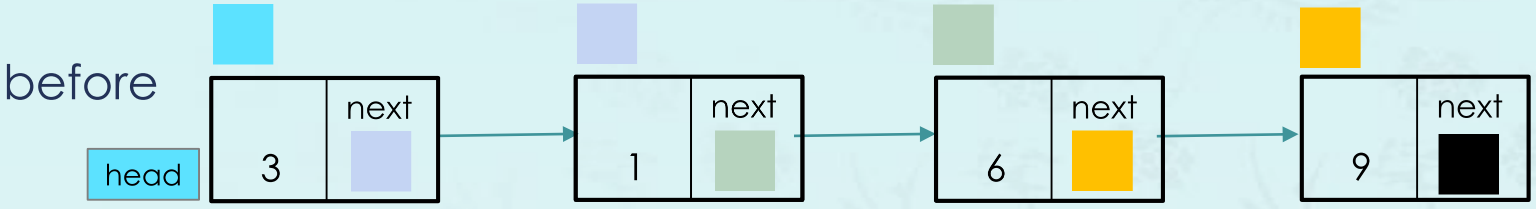


Linked List – reverse using stack

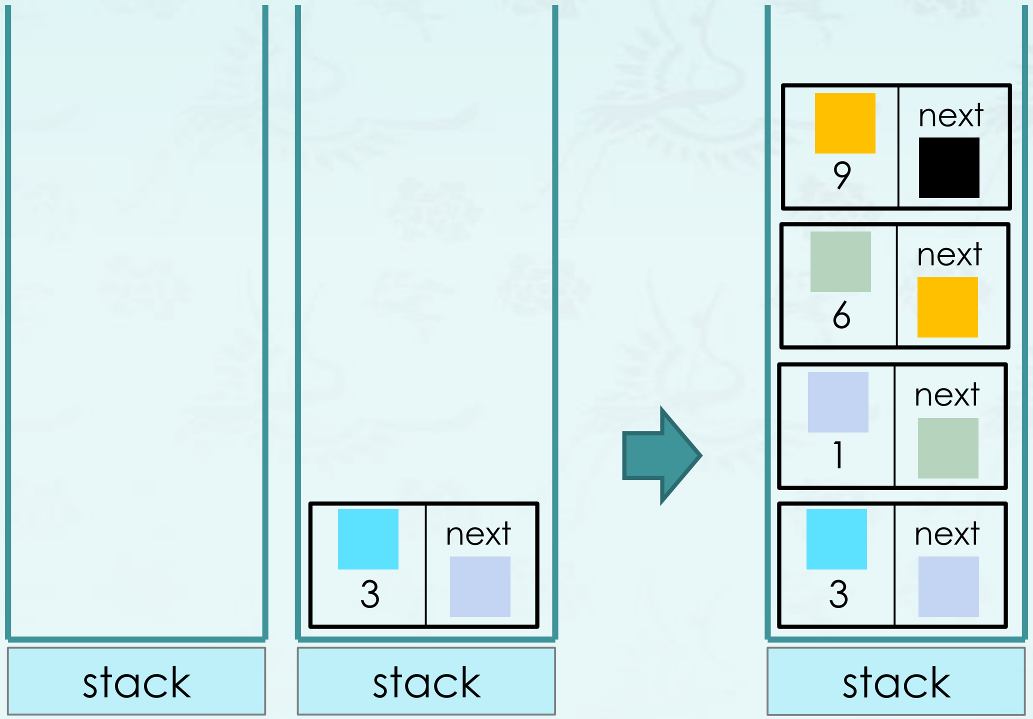


Algorithm:
Step 1. Push all nodes onto the stack.
Step 2. Top/pop all nodes and relink.

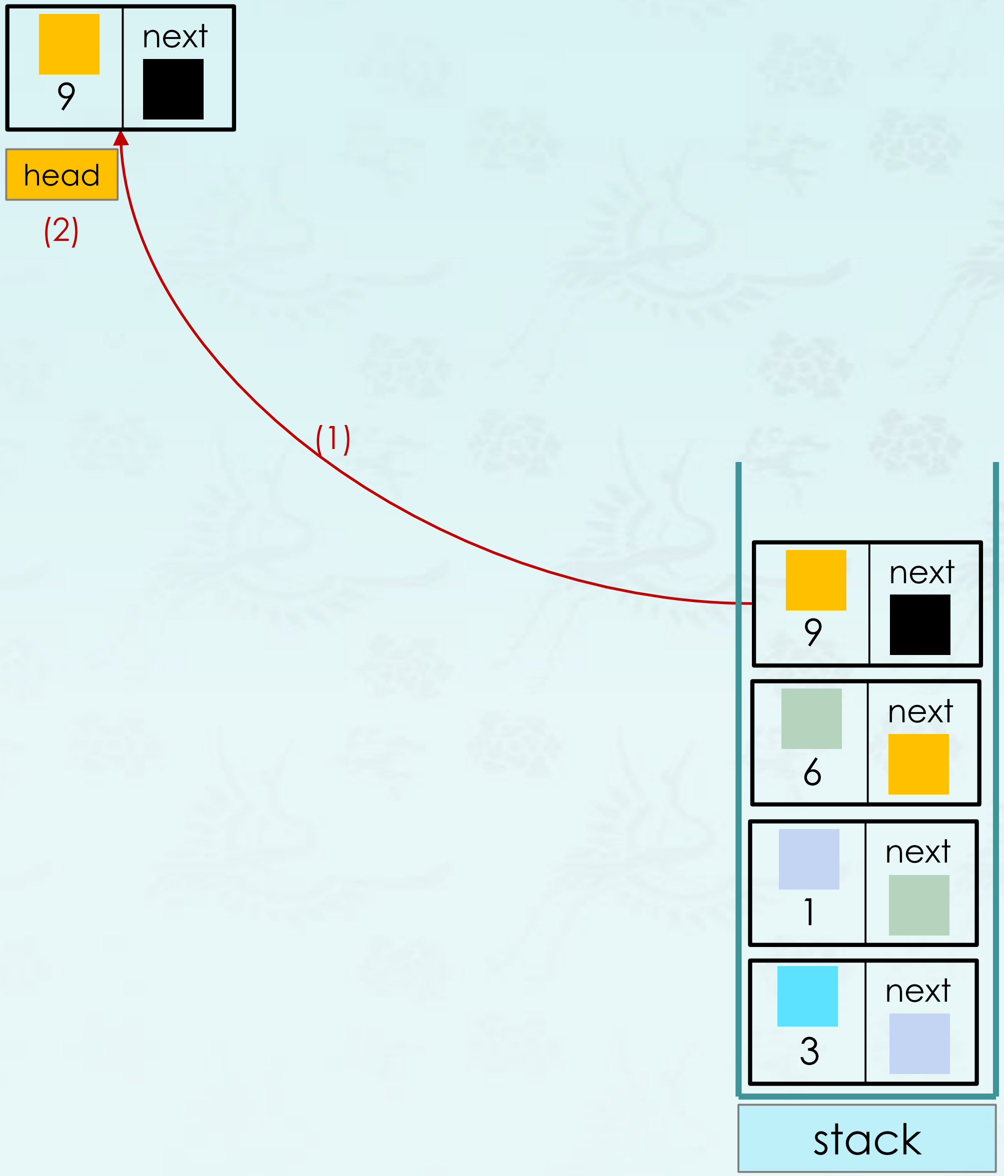
Linked List – reverse using stack



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

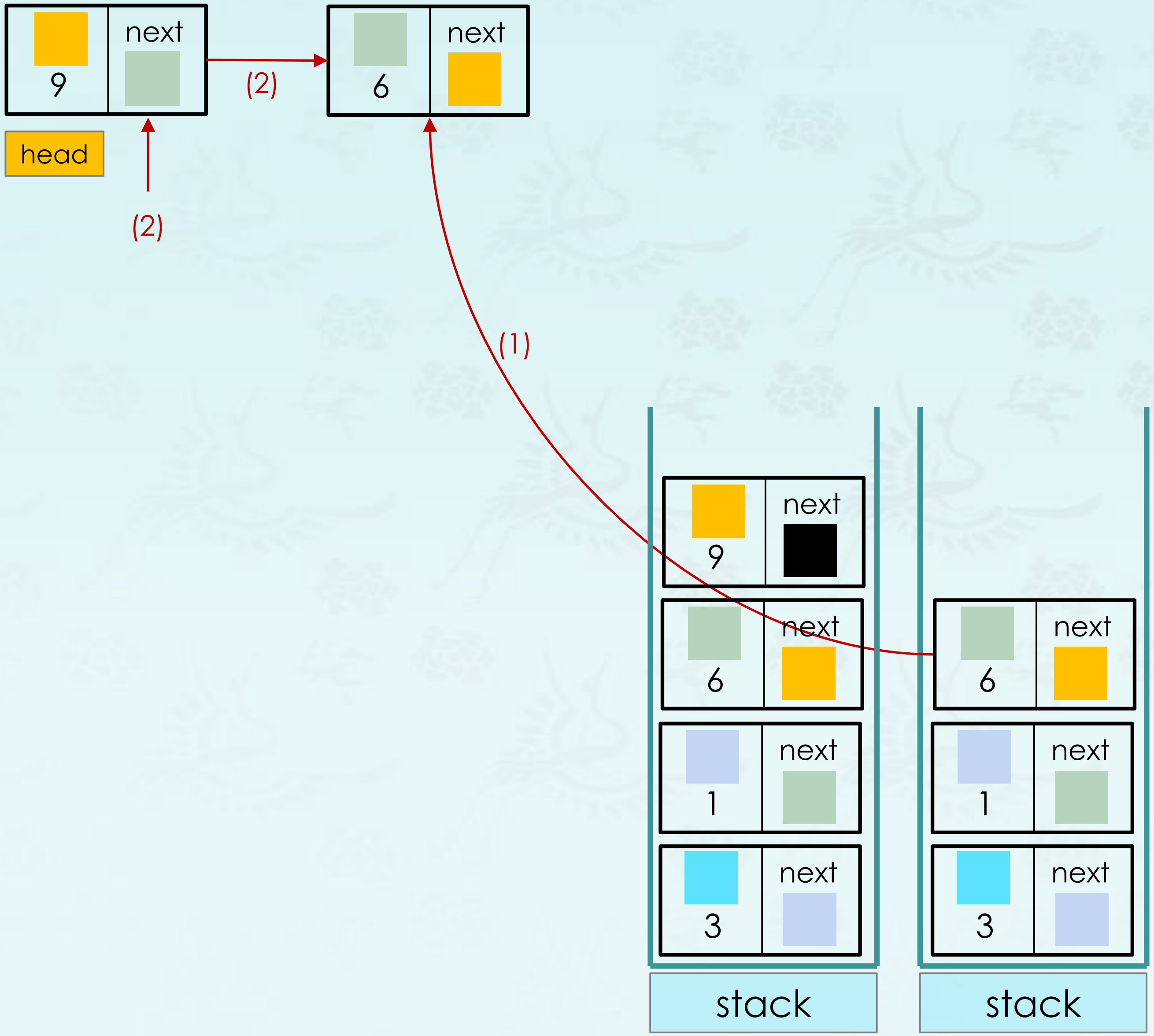


Linked List – reverse using stack



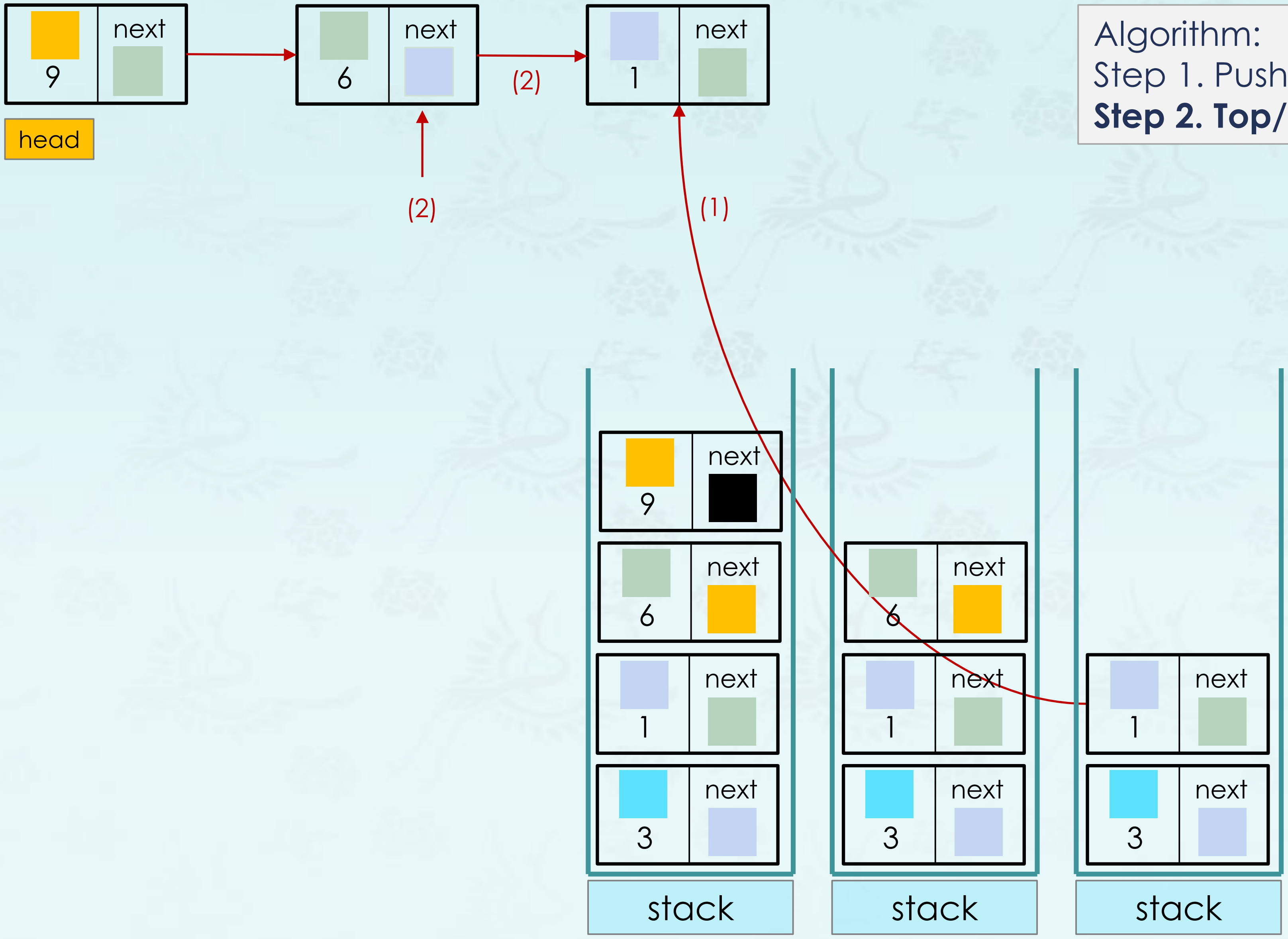
Algorithm:
Step 1. Push all nodes onto the stack.
Step 2. Top/pop all nodes and relink.

Linked List – reverse using stack



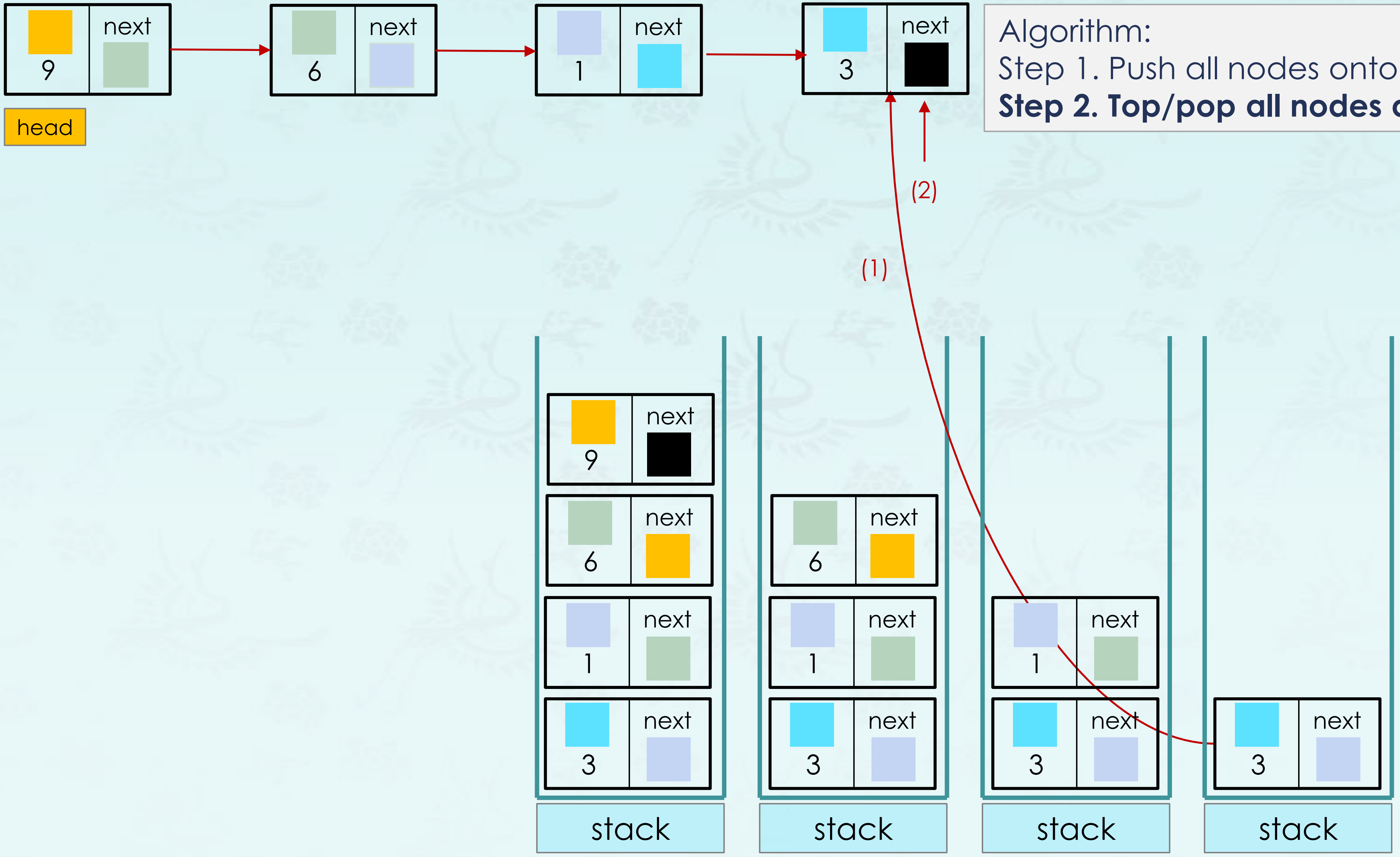
Algorithm:
Step 1. Push all nodes onto the stack.
Step 2. Top/pop all nodes and relink.

Linked List – reverse using stack



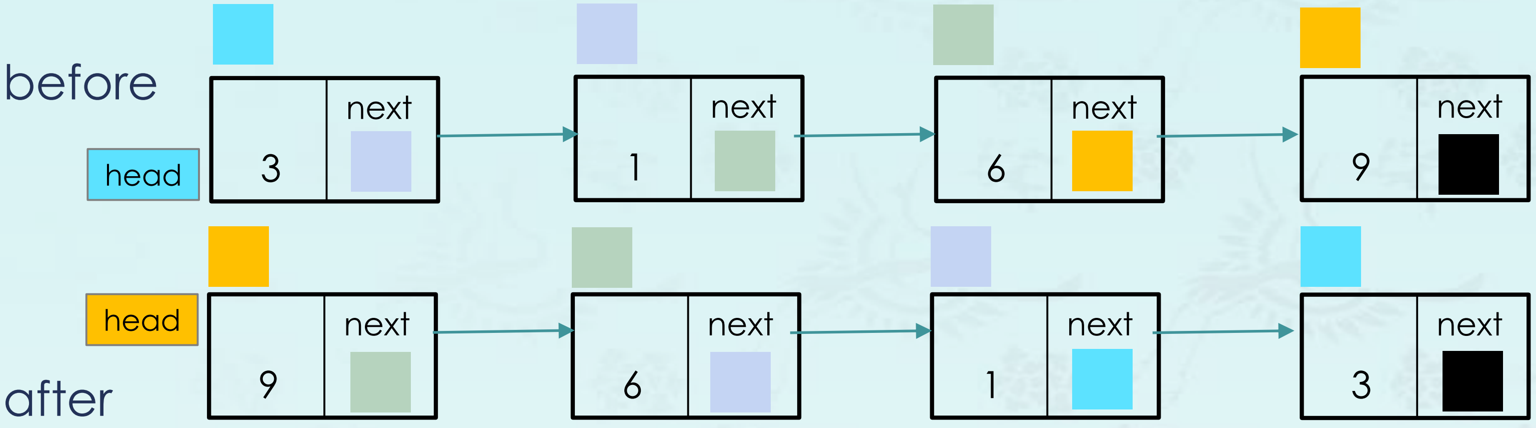
Algorithm:
Step 1. Push all nodes onto the stack.
Step 2. Top/pop all nodes and relink.

Linked List – reverse using stack



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

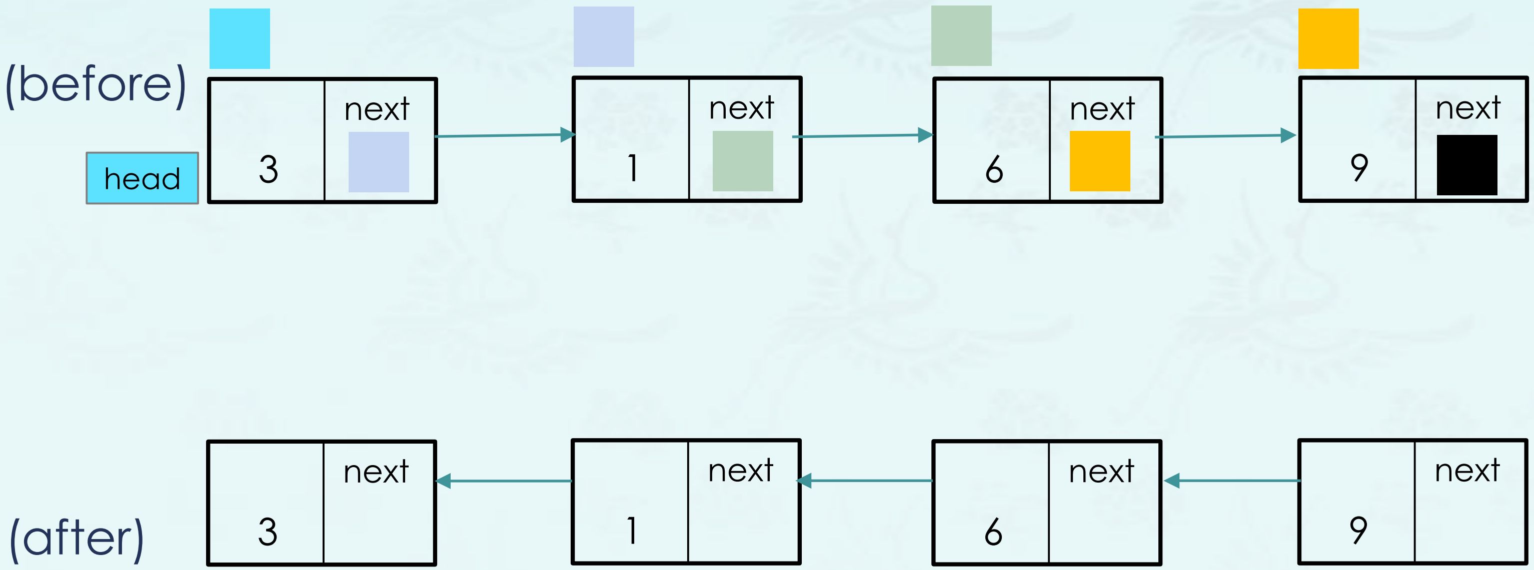
Linked List – reverse using stack




```
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

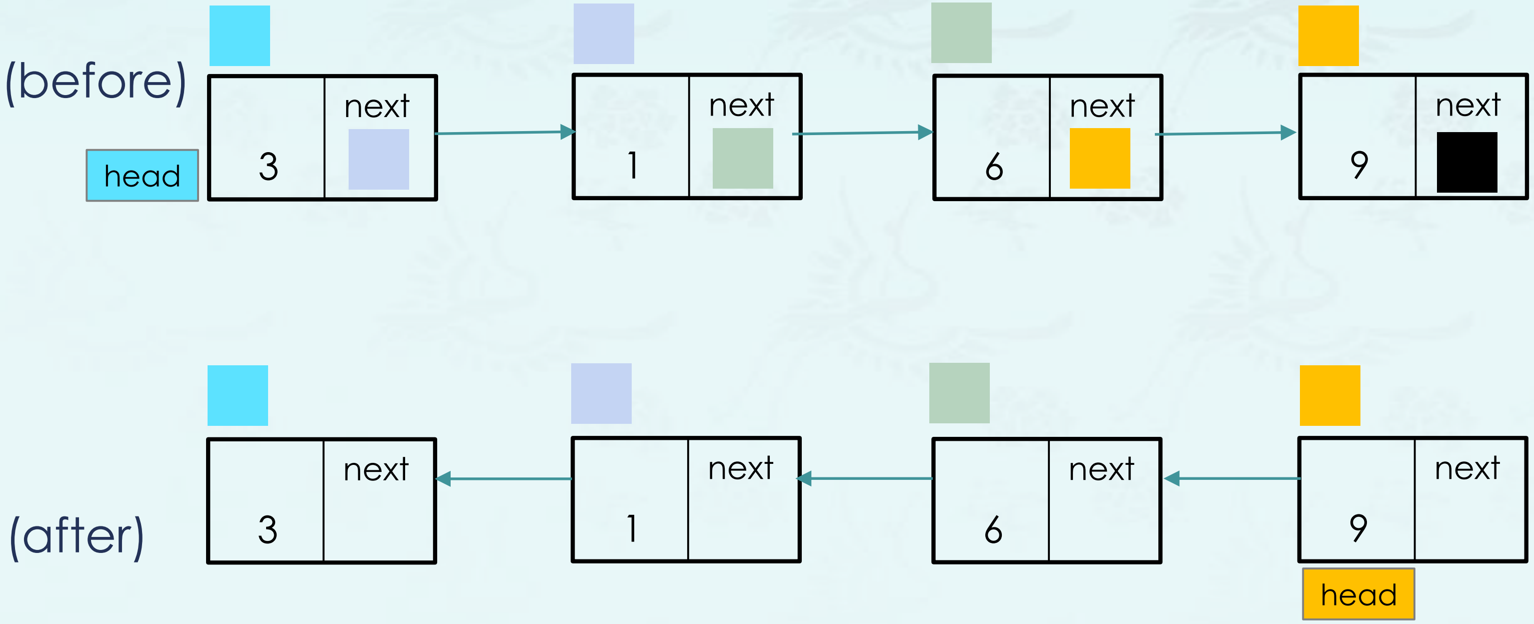


Recall that this is the only one we know inside the function

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



return ________;
```


Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University

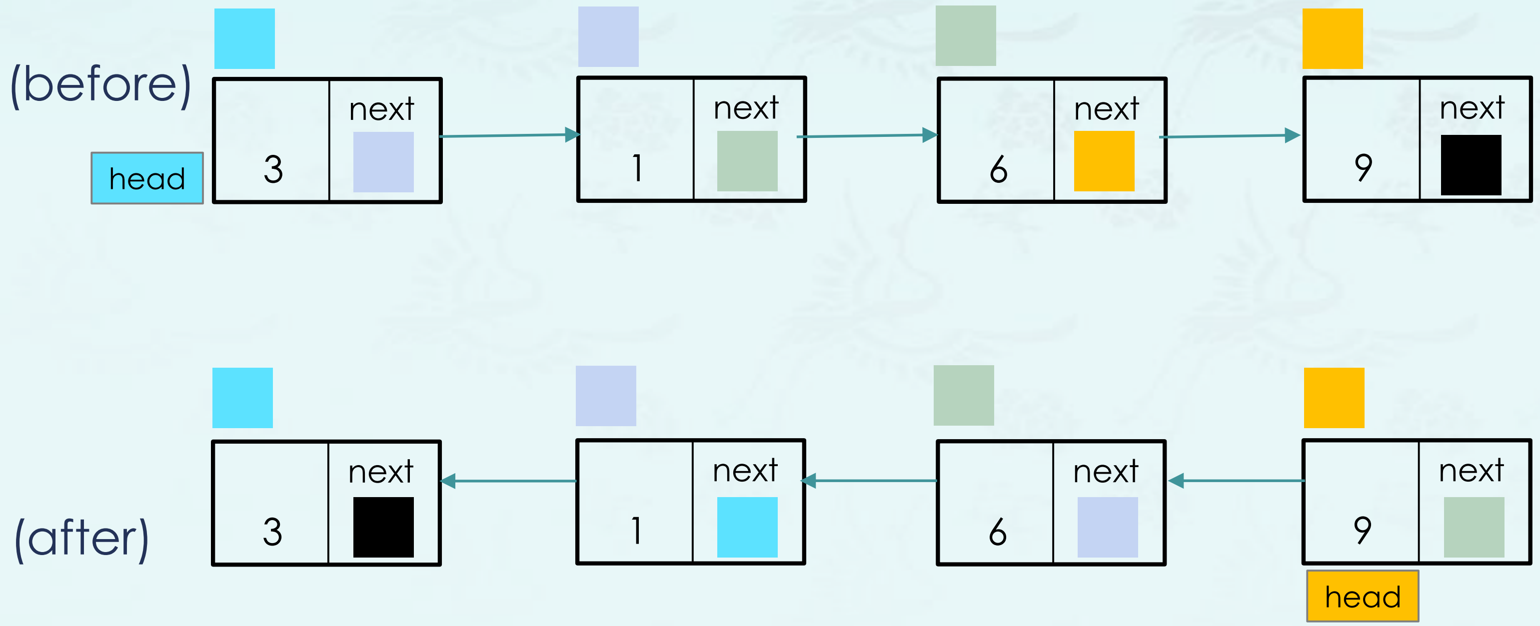
Linked List – reverse in-place



Recall that this is the only one we know inside the function

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

return ________;
```

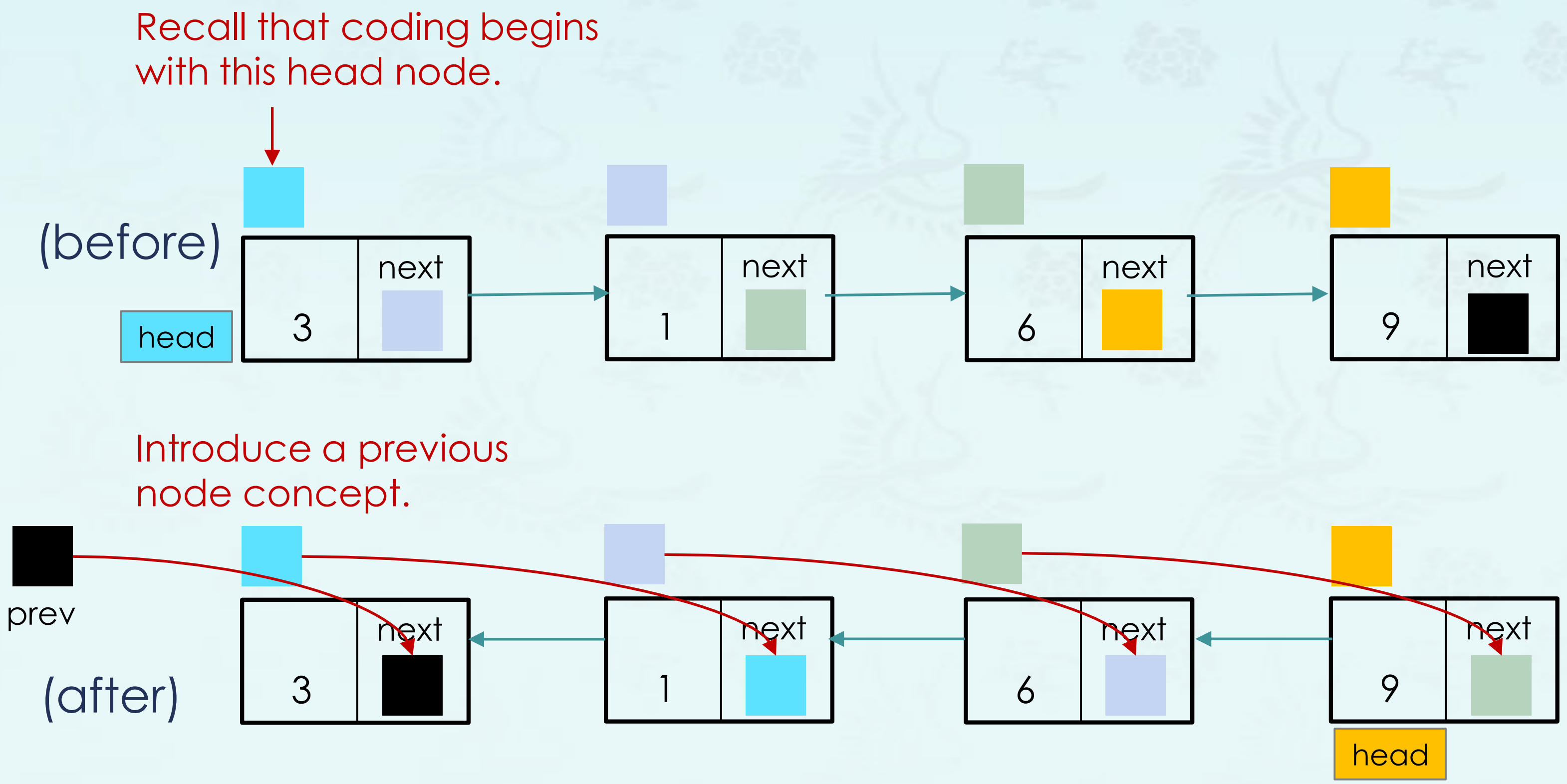
Linked List – reverse in-place



Recall that this is the only one we know inside the function

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

return ________;
```

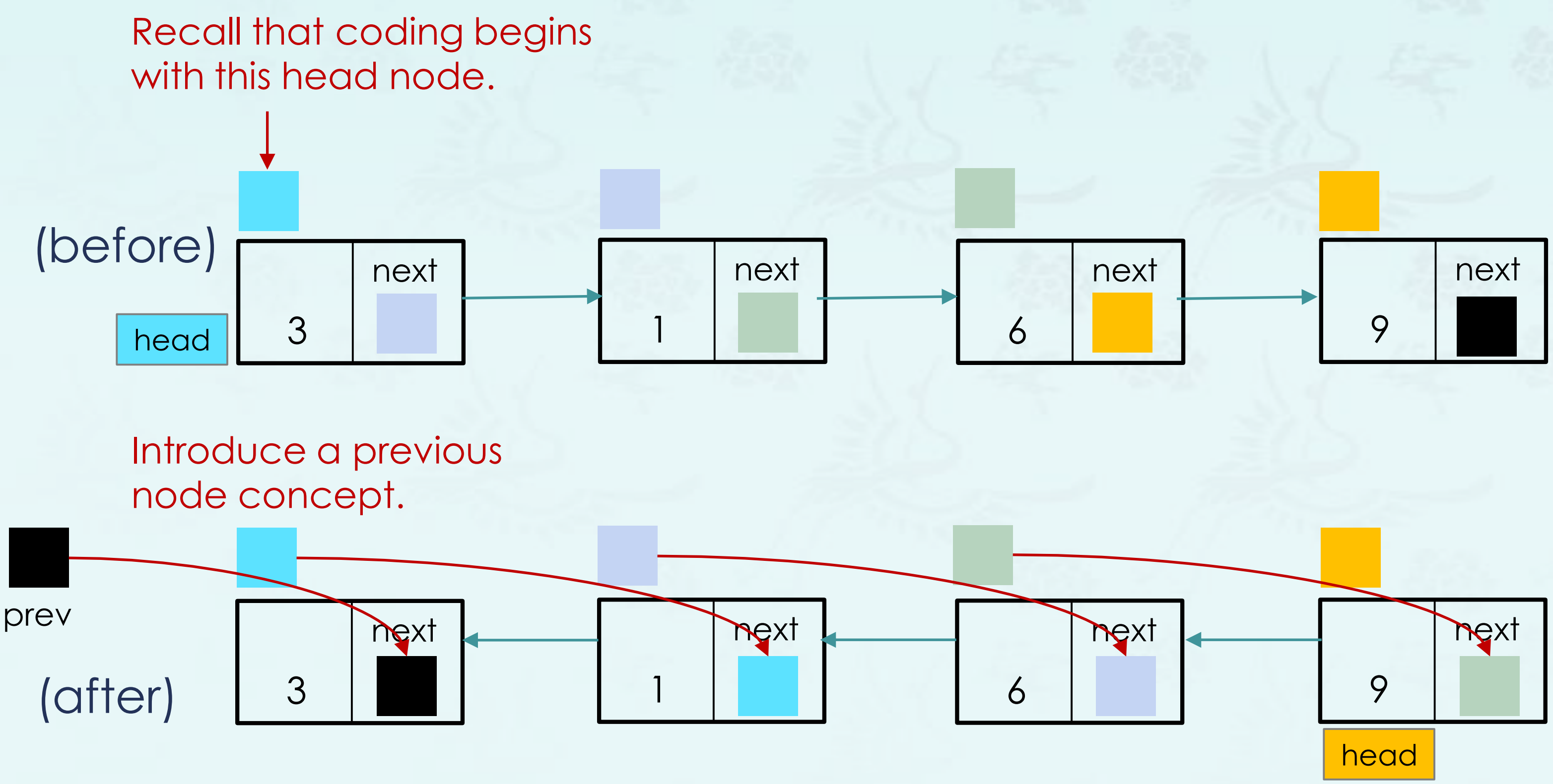
Linked List – reverse in-place



Recall that this is the only one we know inside the function

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

}
return ________;
```

Linked List – reverse in-place



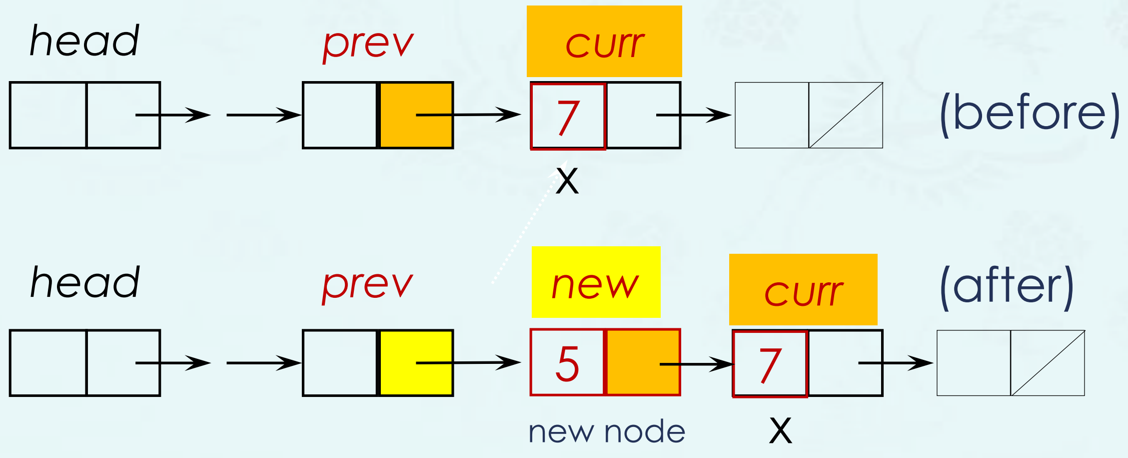
Recall that this is the only one
we know inside the function

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

pNode prev = nullptr;
while (head != nullptr) {




return ________;
```

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.

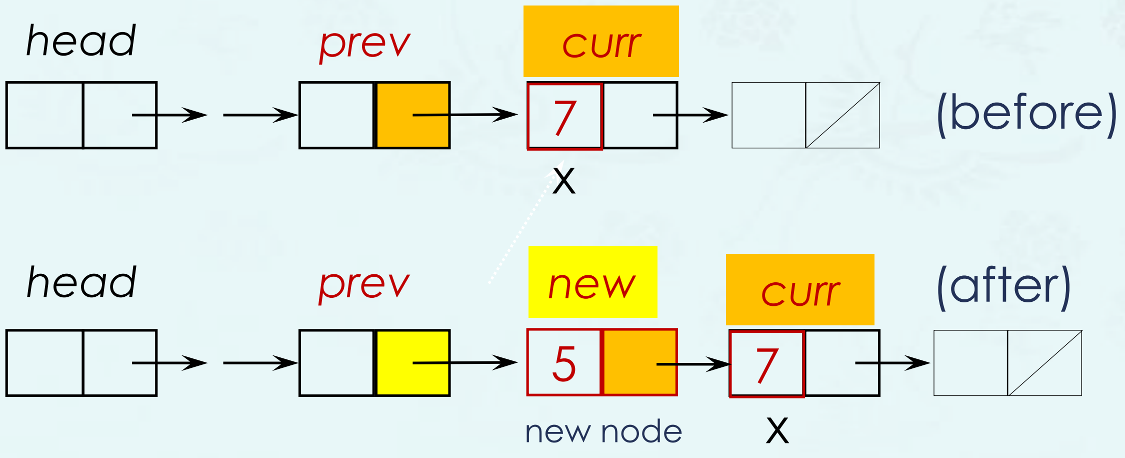



```
pNode insert(pNode head, int val, int x)
if (head->data == x)
  return push_front(val, head);

pNode curr = head;
pNode prev = nullptr;
while (curr != nullptr) {



  prev = curr;
  curr = curr->next;
}
return head;
```

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.

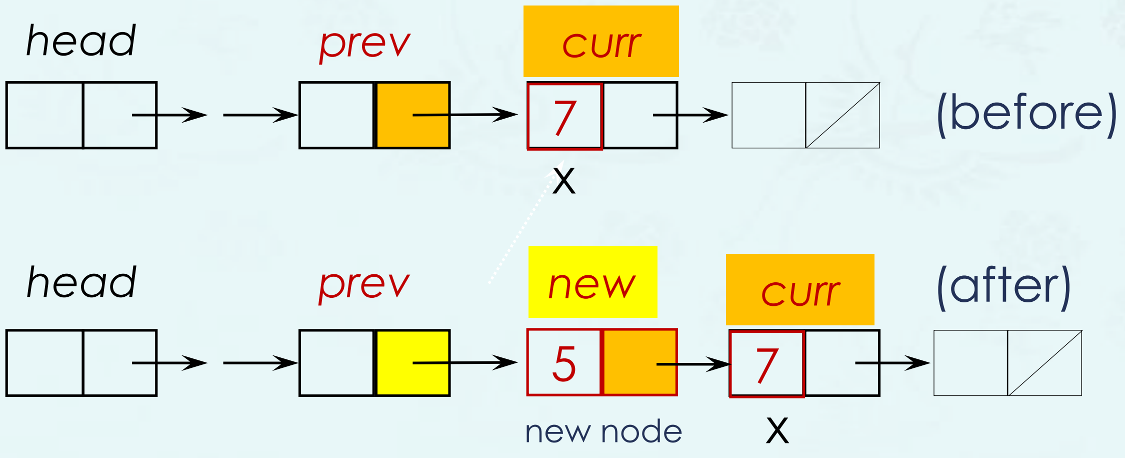



```
pNode insert(pNode head, int val, int x)
if (head->data == x)
  return push_front(val, head);

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

Linked List – insert()

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

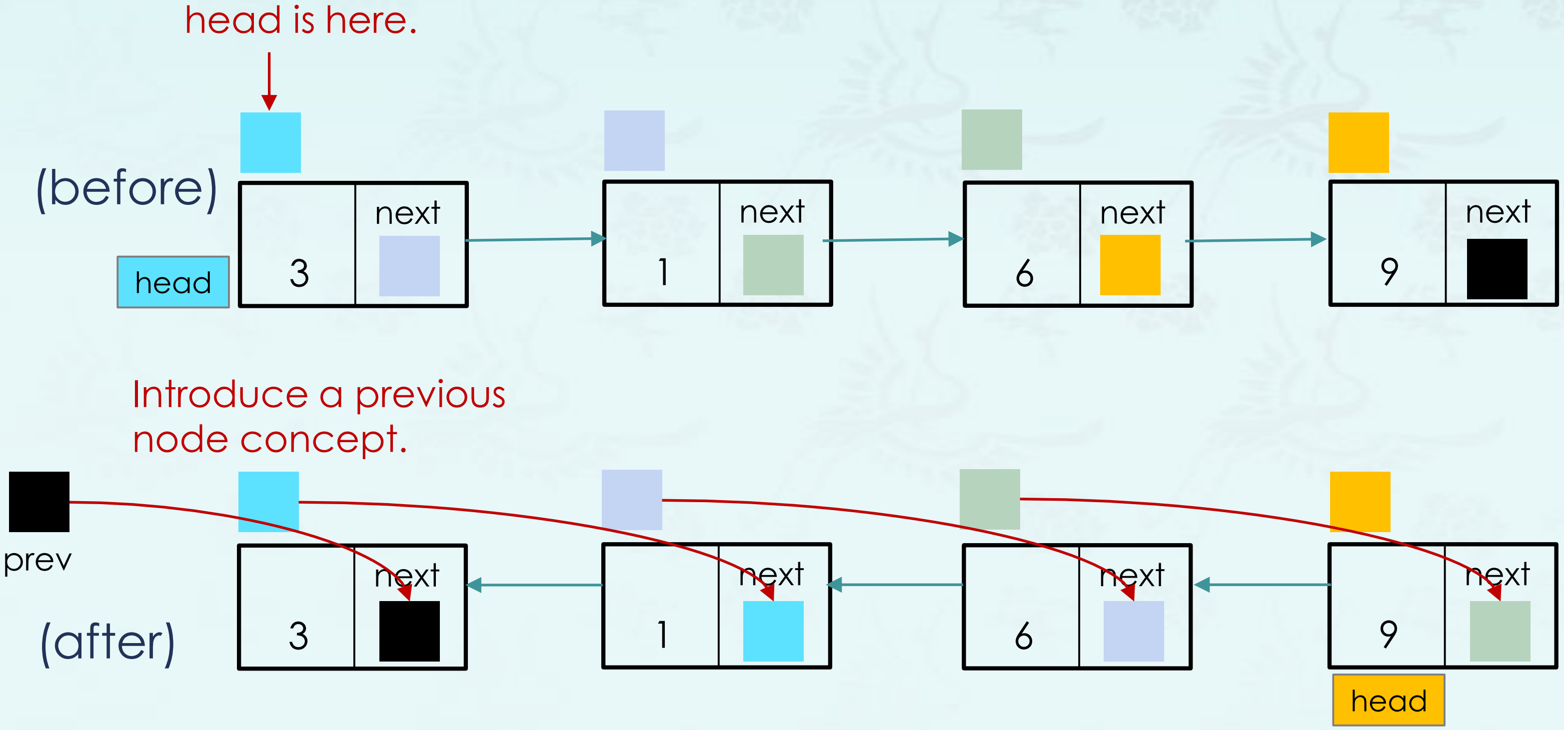
- If the first node(or **head**) is the position, then just invoke **push_front().**
- As observed below, we must to know **the pointer x** which is stored in the **previous node** of node x.




```
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

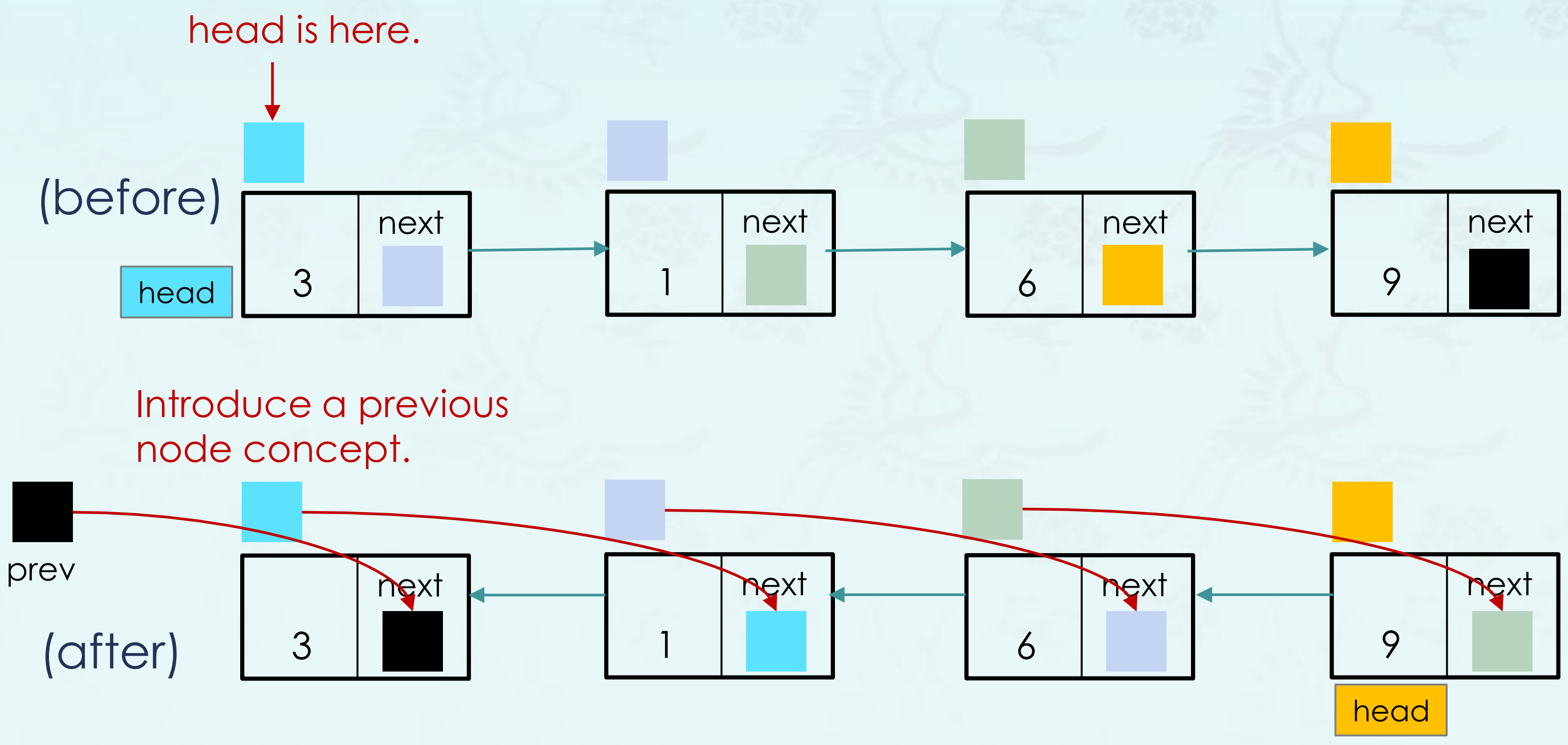
1. We want to overwrite **head->next** with **prev**
   But before overwriting we must save **head->next** .
   because we it is the next node we need to process.




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

pNode prev = nullptr;
while (head != nullptr) {
(1)



}
return _____________;
```

Linked List – reverse in-place

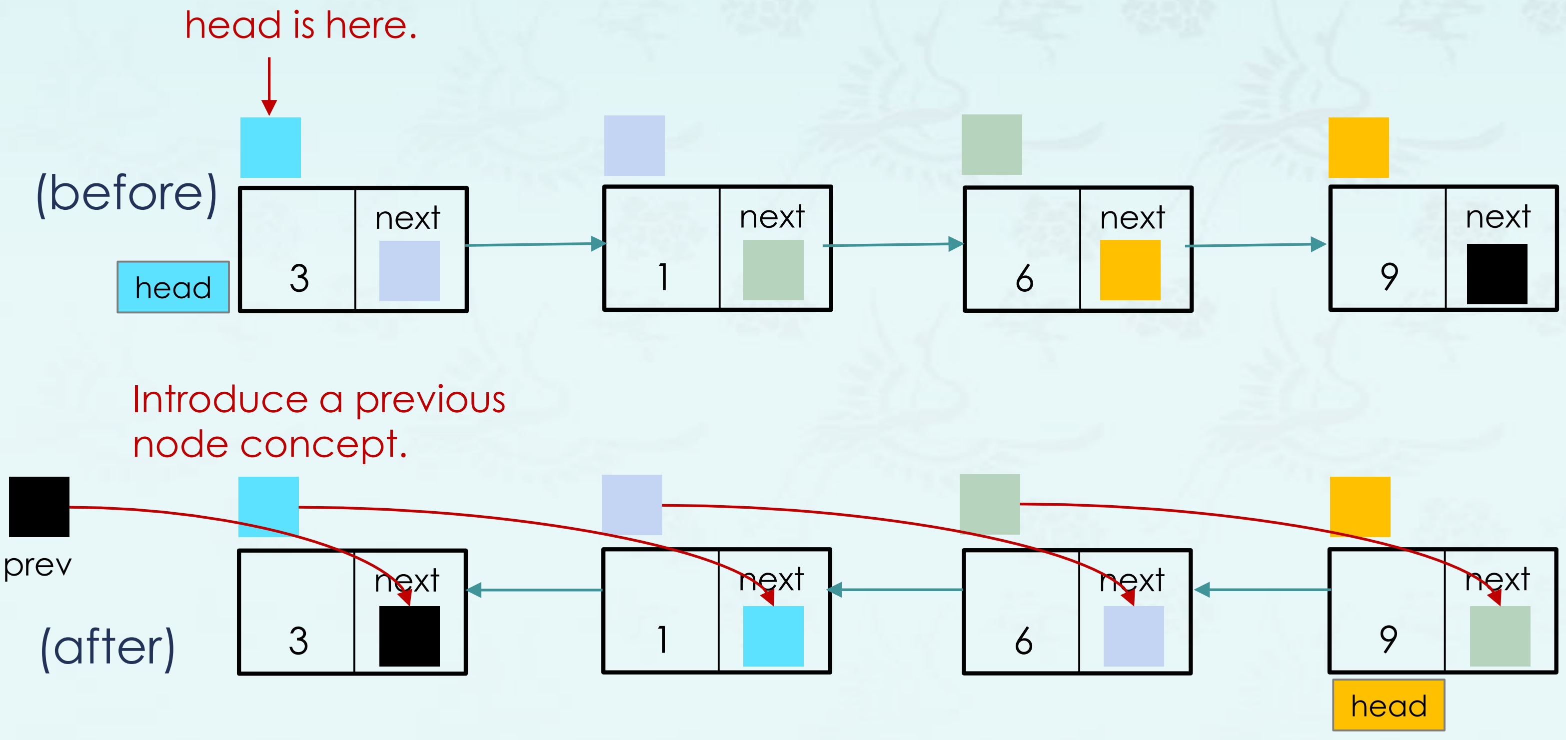
1. We want to overwrite **head->next** with **prev**
   But before overwriting we must save **head->next** .
   because we it is the next node we need to process.
2. Once we overwrite **head->next** with **prev** ,




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

pNode prev = nullptr;
while (head != nullptr) {
(1)
(2)


}
return ________;
```

Linked List – reverse in-place

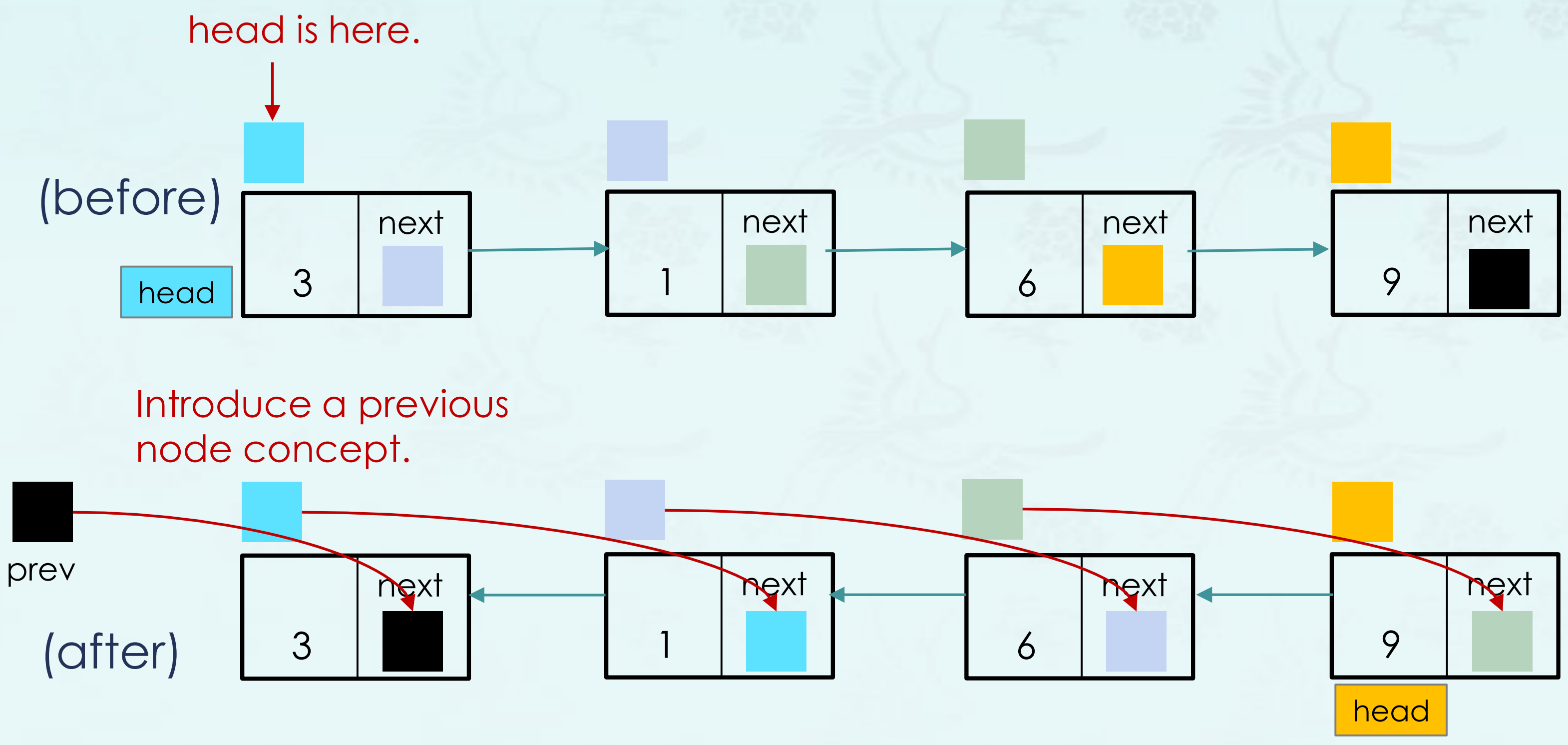
1. We want to overwrite **head->next** with **prev**
   But before overwriting we must save **head->next** .
   because we it is the next node we need to process.
2. Once we overwrite **head->next** with **prev** ,
3. be ready to process the next node by setting
   prev as and **head** as .




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

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

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 .

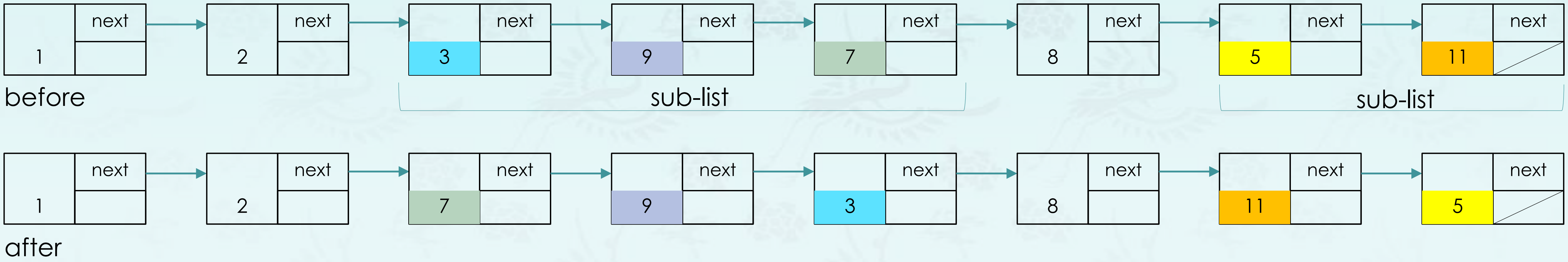



```
pNode reverse(pNode head)
if (empty(head)) return nullptr;

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

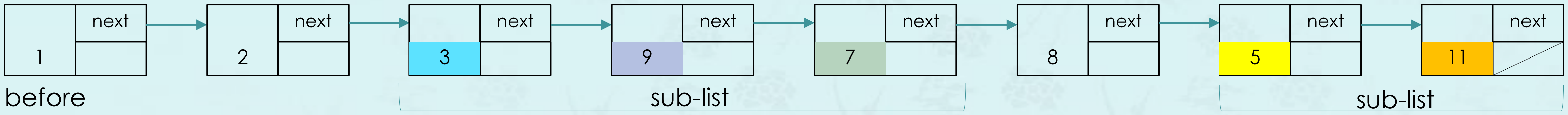
Which one has the last node address to return as a new head after while loop?

Linked List – reverse elements in sub-lists of odd numbers

TASK: Reverse elements in sub-lists of odd numbers only in a singly-linked list.
Given a linked list that contains N integers, select all the sub-lists contain only odd integers. Reverse elements in those sub lists only.
For example, if the list is {1, 2, 3, 9, 7, 8, 5, 11}, then the selected sub-lists are {3, 9, 7} and {5, 11}. Reverse elements in those list such as {7, 9, 3} and {11, 5}. Now, this function returns the original list except odd numbers reversed in the sub-lists. In this example, it returns {1, 2, 7, 9, 3, 8, 11, 5}



Linked List – reverse elements in sub-lists of odd numbers (version 1 – O(n^2))

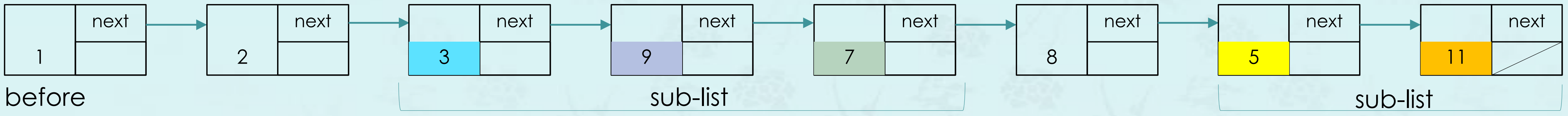


version 1 – O(n^2)

```
while (head != nullptr) {
  if the node is odd {
    push it to odd_stack
    go for the next node
    continue
  }
  // even node encountered
  while (odd_stack is not empty) {
    get top of odd_stack & pop
    push_back to the head2
  }
  add even node to head2          added
  go for the next node
}

while( odd_stack is not empty) {
  get top of odd_stack & pop
  push_back to the head2
}
clear head
return head2
```

- For the sake of the simplicity of coding, we use `push_back()`.
- `head` is the original list `head`.
- `head2` is the new list as a result.
- `odd_stack` stacks up odd(s) until an even shows up.
- You may use either `stack<Node*>` or `stack<int>` but recall that `push_back()` takes a data item, not a node itself.

Linked List – reverse elements in sub-lists of odd numbers (version 2 – O(n))



version 2 – O(n)

```
while (head != nullptr) {
  if the node is odd {
    push it to odd_stack
    go for the next node
    continue
  } // even node encountered
  while (odd_stack is not empty) {
    get top of odd_stack & pop
    head2 is null, set head2
    add it to the tail2 & set tail2
  }
  head2 is null, set head2
  add even node to tail2 & set tail2
  go for the next node
}
while( odd_stack is not empty) {
  get top of odd_stack & pop
  head2 is null, set head2
  add it to the tail2 & set tail2
}  // no clear head necessary
return head2
```

- For the sake of the speed of the code, do not use `push_back()`. Use almost the same algorithm, but manage to add a node at the `head2` and `tail2` by yourself instead of calling `push_back()`.
- `head` is the original list `head`.
- `head2` is the new list as a result.
- `tail2` is the tail node of the `head2`.
- `odd_stack` stacks up odd(s) until an even shows up.
- Do not use `stack<int>`, but `stack<Node*>` to reuse the nodes.
- Do not clear `head` since all nodes are reused.

Summary &

quaestio quaestio qo ? ? ? ?

Data Structures
Chapter 4

1. Singly Linked List
   - Pointer Reviewed & Linked
   - Linked List (1)
   - Linked List (2)
   - **Reverse**
2. Doubly Linked List

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University