Data Structures
Chapter 4

1. Singly Linked List
2. Doubly Linked List
   - Revisit – Singly Linked List
   - Doubly Linked List with Sentinels
   - Basic Operations
   - **Advanced Operations: pop_all(), half(), shuffle()**

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University
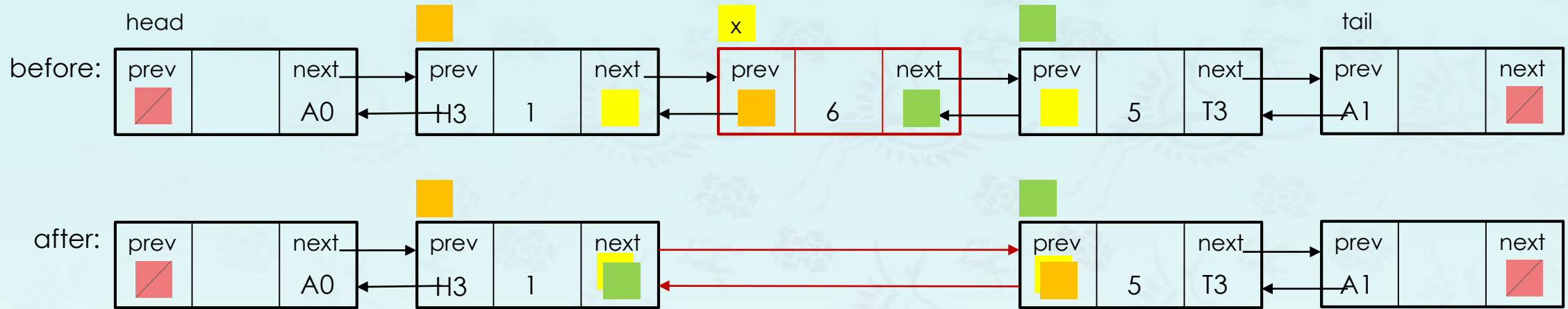
다윗은 당시에 하나님의 뜻을 따라 섬기다가 잠들어 그 조상들과 함께 묻혀 썩음을 당하였으되. 행13:36

For when David had served God's purpose in his own generation, he fell asleep; he was buried with his fathers and his body decayed. Acts 13:36

우리가 알거니와 하나님을 사랑하는 자 곧 그의 뜻대로 부르심을 입은 자들에겐 모든 것이 합력하여 선을 이루느니라 (롬8:28)

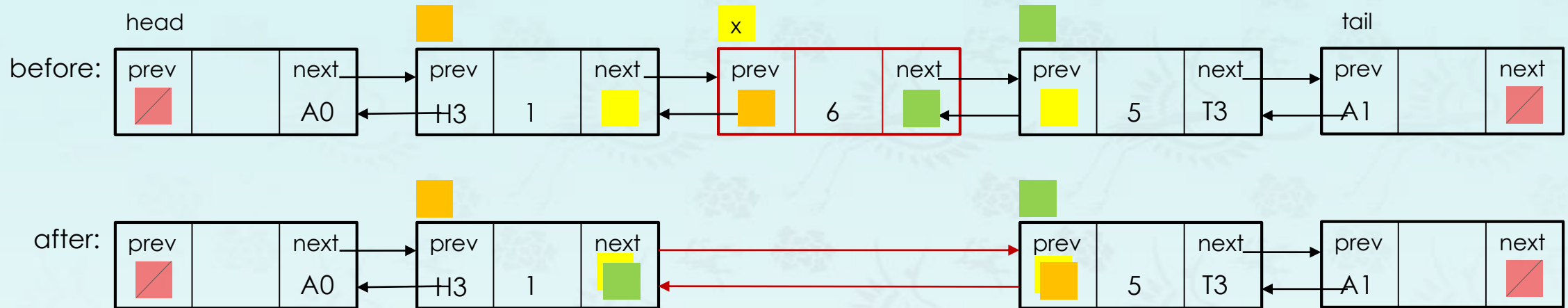하나님이 우리를 구원하사 거룩하신 소명으로 부르심은 우리의 행위대로 하심이 아니요 오직 자기의 뜻과 영원 전부터 그리스도 예수 안에서 우리에게 주신 은혜대로 하심이라 (딤후1:9)

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace school Rm304, Handong Global University

2

Revisit - erase()

The node x is to be erased or removed. Then, which nodes are changed and where?

what should be here in color code?

before:

x ==	[yellow]
==	
==	
express in terms of x ==	
==	

```
void erase(pNode x) {




}
```

Revisit - erase()

head

before:

prev		next
		A0

prev		next
H3	1	

x

prev		next
	6	

prev		next
	5	T3

tail

prev		next
A1		

after:

prev		next
		A0

prev		next
H3	1	

prev		next
	5	T3

prev		next
A1		

before:

x == ▢

☐ == ▢

☐ == ▢

express in terms of x == ▢

== ▢

```
void erase(pNode x) {



}
```

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

4

Revisit - erase()

before:

head

prev		next
		A0

prev		next
H3	1	

x

prev		next
	6	

prev		next
	5	T3

tail

prev		next
A1		

after:

prev		next
		A0

prev		next
H3	1	

prev		next
	5	T3

prev		next
A1		

before:

```
        x == 
  x->prev == 
  x->next == 
x->prev->next == 
x->next->prev == 
```

```
void erase(pNode x) {



}   It should be coded using info in x only.
```

Revisit - erase()
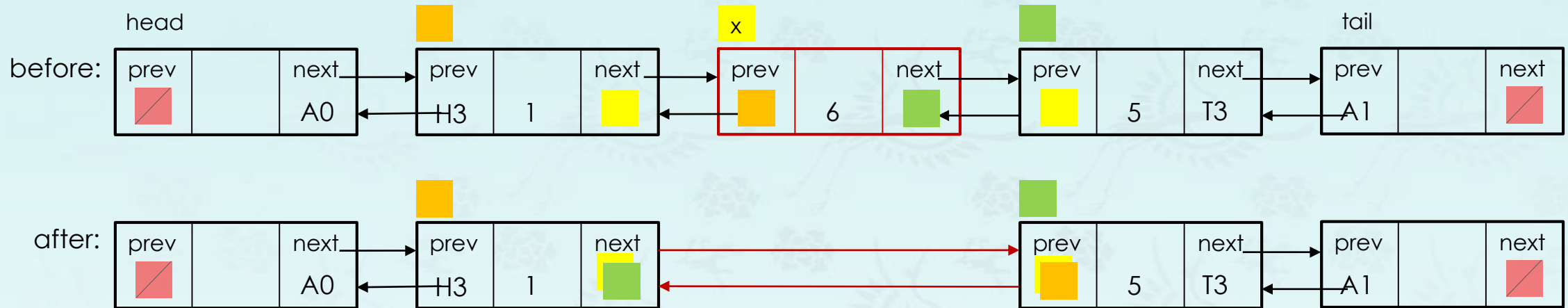
head

before:

prev		next
⬜		A0

→

prev		next
H3	1	🟨

→

x

prev		next
🟧	6	🟩

→

prev		next
🟨	5	T3

→

tail

prev		next
A1		⬜

after:

prev		next
⬜		A0

→

prev		next
H3	1	🟩

→

prev		next
🟧	5	T3

→

prev		next
A1		⬜

before:

```
        x == 🟨
  x->prev == 🟧
  x->next == 🟩
x->prev->next == 🟨
x->next->prev == 🟨
```

```
void erase(pNode x) {
  x->prev->next = x->next; 🟩
  x->next->prev = x->prev; 🟧
  delete x;           ✖
}  It should be coded using info in x only.
```

Revisit - insert a new node(value) in place of the node x

Identify where are to be changed or set?

before:


```
void insert(pNode x, int value) {


}
```

It should be coded using value and info in x only.

Revisit - insert a new node(value) in place of the node x

Identify where are to be changed or set?


```
void insert(pNode x, int value) {



}
```

It should be coded using value and info in x only.

before:

prev		next
⧄		

prev		next
	1	

prev		next
	5	

prev		next
		⧄

node

after:

prev	val	next
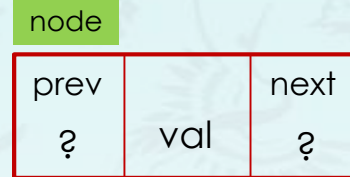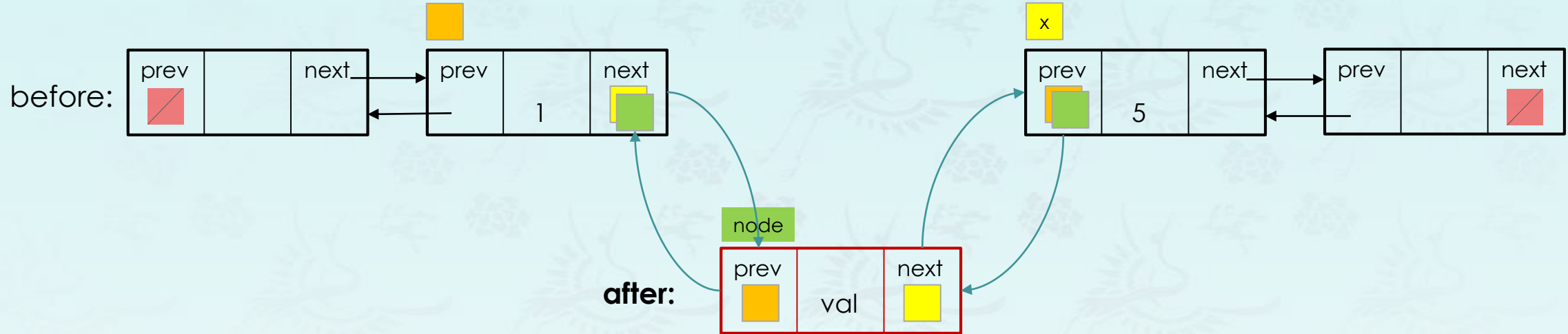
```
void insert(pNode x, int value) {
   pNode node = new Node{value, x->prev, x};
   x->prev->next = node;
   x->prev = node;
}
```

Revisit - insert a new node(value) in place of the node x

before:

after:
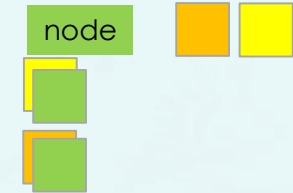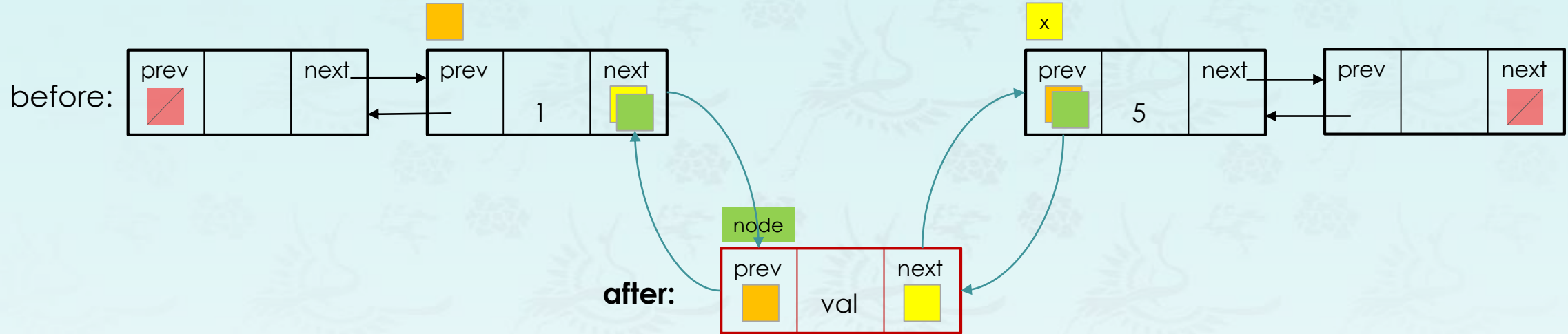
```
void insert(pNode x, int value) {
    pNode node = new Node{value, x->prev, x};
    x->prev->next = node;
    x->prev = node;
}
```

```
void insert(pNode x, int value) {
    pNode node = new Node{value, x->prev, x};
    x->prev = x->prev->next = node;    ←  this is OK
}
```

before:

after:

node

after:
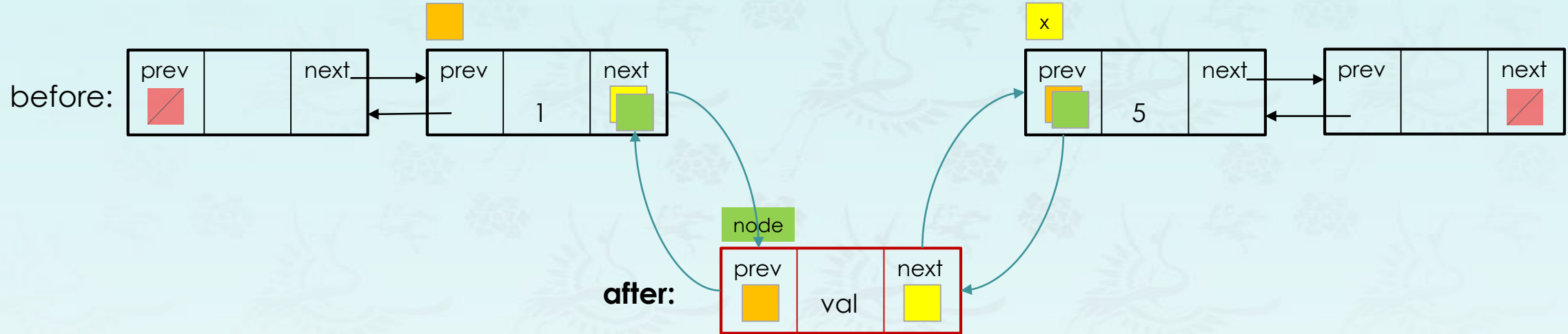
```
void insert(pNode x, int value) {
    pNode node = new Node{value, x->prev, x};
    x->prev = x->prev->next = node;
}
```

insert() extends the list by inserting a new node with value **before** the node at the specified position **x**. For example, if **begin(p)** is specified as an insertion position, the new node becomes the first one.

push_front()

```
// Inserts a new node at the beginning of the list, right before its
// current first node. The content of data item is copied(or moved) to the
// inserted node. This effectively increases the container size by one.
void push_front(pList p, int value) {
  insert(begin(p), value);
}
```
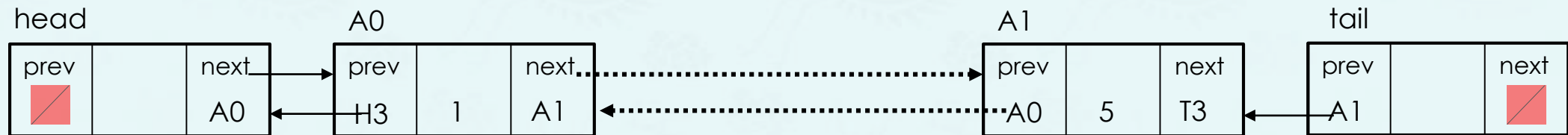
pop_front()

```
// Removes the first node in the list container, effectively reducing
// its size by one. This destroys the removed node.
void pop_front(pList p) {
  if (!empty(p)) erase(begin(p));
}
```
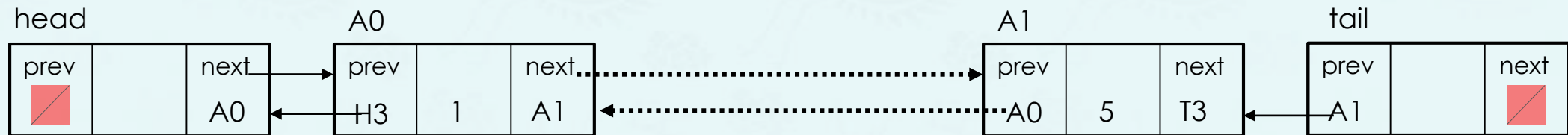
Revisit: erase(), pop(), and find()

```
void erase(pNode x){
  x->prev->next = x->next;
  x->next->prev = x->prev;
  delete x;
}
```

```
void pop(pList p, int value){
  erase(find(p, value));
}
```
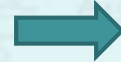
This code may not work some cases.
How can you fix it?

Is this good enough?

```
void pop(pList p, int value){
  pNode node = find(p, value);
  if (node == p->tail) return;
  erase(node);
}
```

# Revisit: erase(), pop(), and find()

Funtion overloading in C++

```cpp
void erase(pNode x){
  x->prev->next = x->next;
  x->next->prev = x->prev;
  delete x;
}
```
○

```cpp
void erase(pList p, pNode x){
  if (x == end(p)) return;
  x->prev->next = x->next;
  x->next->prev = x->prev;
  delete x;
}
```

```cpp
void pop(pList p, int value){
  erase(find(p, value));
}
```
✗

This code may not work some cases.
How can you fix it?

```cpp
void pop(pList p, int value){
  pNode node = find(p, value);
  if (node == p->tail) return;
  erase(node);
}
```
✗

Revisit: erase(), pop(), and find()

Funtion overloading in C++

```
void erase(pNode x){
  x->prev->next = x->next;
  x->next->prev = x->prev;
  delete x;
}
```
○
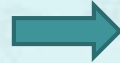
```
void erase(pList p, pNode x){
  if (x == end(p)) return;
  erase(x);
}
```
○

```
void pop(pList p, int value){
  erase(find(p, value));
}
```
✗
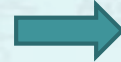
```
void pop(pList p, int value){
  erase(p, find(p, value));
}
```
○

This code may not work some cases.
How can you fix it?

```
void pop(pList p, int value){
  pNode node = find(p, value);
  if (node == p->tail) return;
  erase(node);
}
```
✗

Exercise 1: What does find() return if value not found?

```
pNode find(pList p, int value){
  pNode x = begin(p);
  while(x != end(p)) {
    if (x->data == value) return x;
    x = x->next;
  }
  return x;
}
```

Exercise 2: Can we reduce the lines below by one or two?

```
pNode find(pList p, int value){
  pNode x = begin(p);
  while(x != end(p)) {
    if (x->data == value) return x;
    x = x->next;
  }
  return x;
}
```

```
pNode find(pList p, int value){
  pNode x = begin(p);


  return x;
}
```

Exercise 3: Implement find() using a for-loop?

```
pNode find(pList p, int value){
  pNode x = begin(p);
  while(x != end(p)) {
    if (x->data == value) return x;
    x = x->next;
  }
  return x;
}
```

```
pNode find(pList p, int value){



  return x;
}
```

doubly linked list – **pop_all()***

Write a pop_all() which takes a list and deletes any nodes with a value given from the list. Ideally, the list should only be traversed once to have the time complexity, O(n).

```
void pop_all(pList p, int value) { // value = 3 in this example
  while (find(p, value) != end(p)) {        O(n)
    pop(p, value);                          O(n)     →   O(n^2)
  }
} // version.1
```

The code above works, then what is the problem?
What is the time complexity of each line and overall?
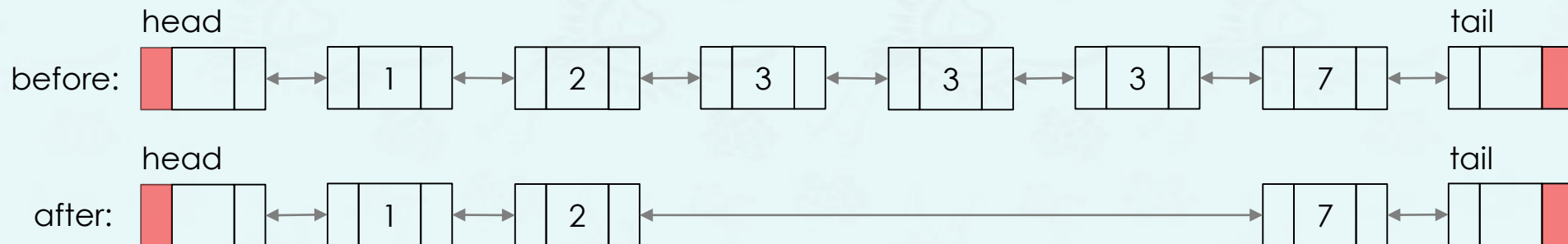
doubly linked list – **pop_all()***

Write a pop_all() which takes a list and deletes any nodes with a value given from the list. Ideally, the list should only be traversed once to have the time complexity, O(n).

```
void pop_all(pList p, int value) { // value = 3 in this example
  for (pNode x = begin(p); x != end(p); x = x->next)
    if (x->data == value) erase(p, x);
} // version.2 – fast, but buggy
```

- Does x point to the next node right after the first `erase(p, x)` call finishes? Are you sure?
- If you have not figured it out completely, you review `erase()` source code.
- Be able to answer why the code above may work in some machines or with small number of nodes.
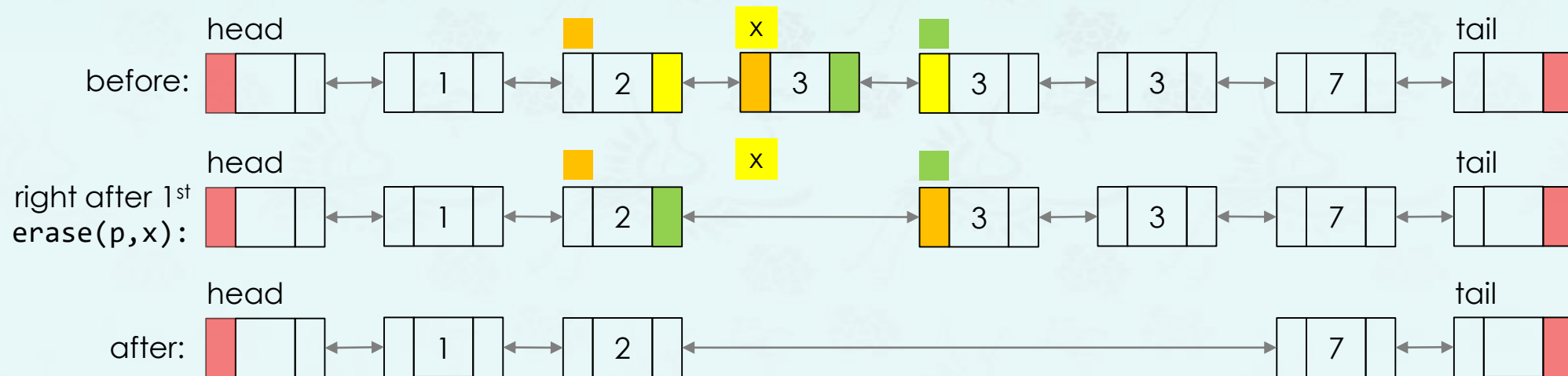
doubly linked list – **half()**

Write `half()` function that returns the mid node of the list.
- Even number of nodes, it returns the first node of the second half which is 6th node if there are ten nodes.
- If there are five (odd number) nodes in the list, it returns the third one (or middle one).
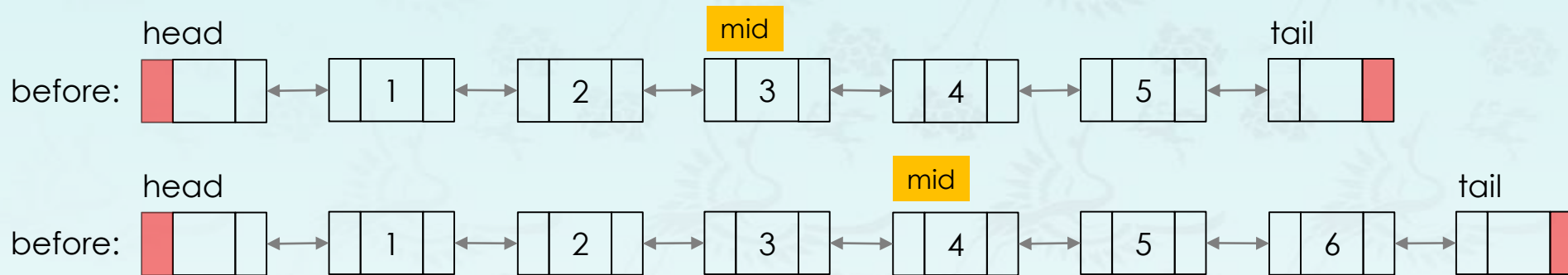
- **Method 1:** Get the size of the list O(n). Then scan to the halfway, breaking the last link followed.

- **Method 2:** It works by sending rabbit and turtle down the list: turtle moving at speed one, and rabbit moving at speed two. As soon as the rabbit hits the end, you know that the turtle is at the halfway point as long as the rabbit gets asleep at the halfway.
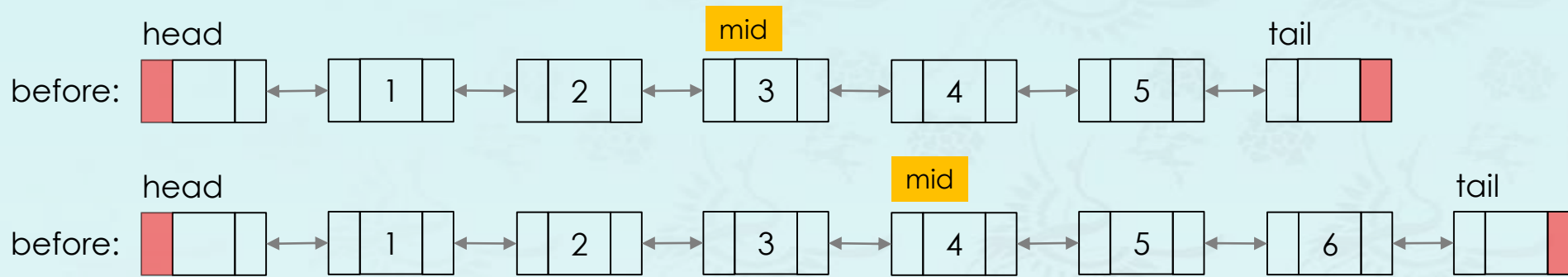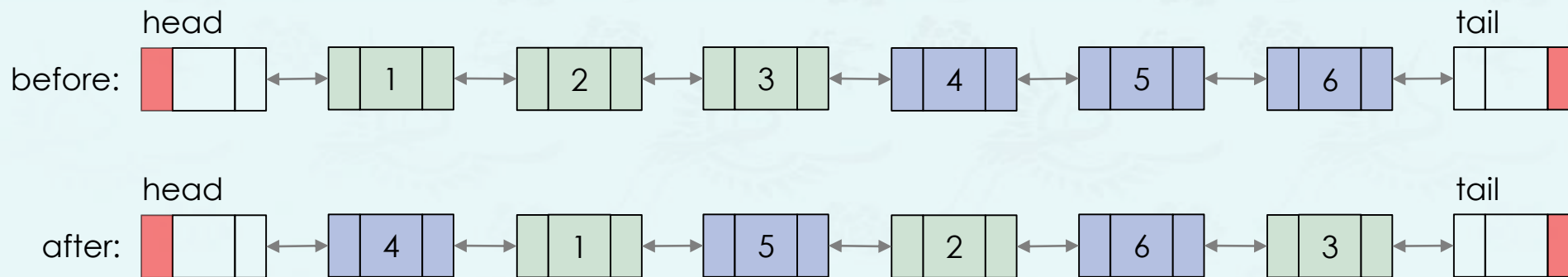
doubly linked list – **half()**
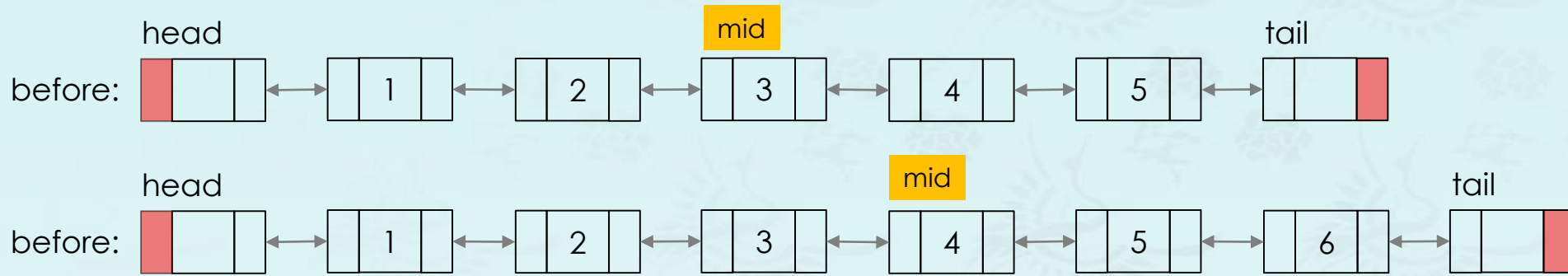
- **Method 1:** Get the size of the list O(n). Then scan to the halfway, breaking the last link followed.
- If the list size n = 2 million nodes, what is the number of nodes that you must check through?

```
pNode half(pList p) {
   int N = size(p);
   // go through the list and get the size
   // go through the list at the halfway
   // return the current pointer
}
```

head　　　　　　　　　　　　　mid　　　　　　　　　　tail

before: [ ] ↔ [1] ↔ [2] ↔ [3] ↔ [4] ↔ [5] ↔ [ ]

head　　　　　　　　　　　mid　　　　　　　　　　　　tail

before: [ ] ↔ [1] ↔ [2] ↔ [3] ↔ [4] ↔ [5] ↔ [6] ↔ [ ]

- **Method 2:** It works by sending rabbit and turtle down the list: turtle moving at speed one, and rabbit moving at speed two. As soon as the rabbit hits the end, you know that the turtle is at the halfway point as long as the rabbit gets asleep at the halfway. O(n)

```
pNode half(pList p) {
  pNode rabbit = turtle = begin(p);
  pNode turtle = begin(p);
  while (rabbit != end(p)) {
    rabbit = rabbit->next->next;
    turtle = turtle->next;
  }
  return turtle;
} // buggy on purpose
```

- If the list size n = 2 million nodes, what is the number of nodes that you must check through?

doubly linked list – **perfect shuffle()****

Implement shuffle()** function that returns so called "perfectly shuffled" list. The first half and the second half are interleaved each other. The shuffled list begins with the second half of the original. For example, 1234567890 returns 6172839405.
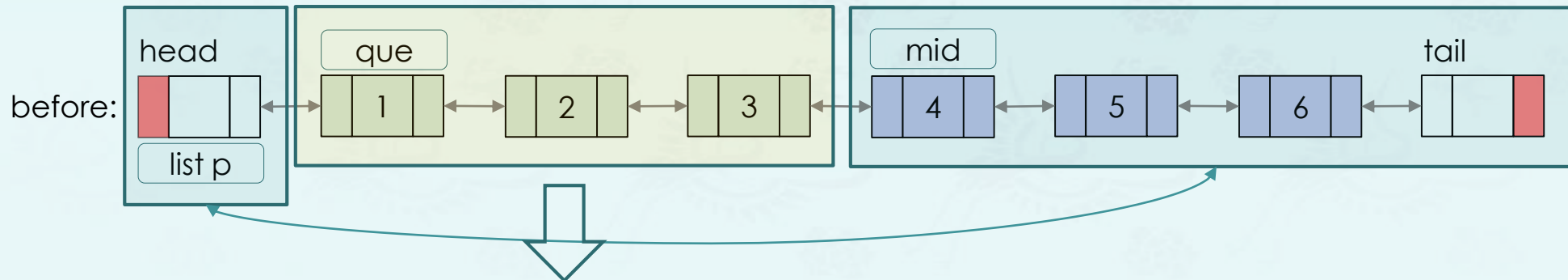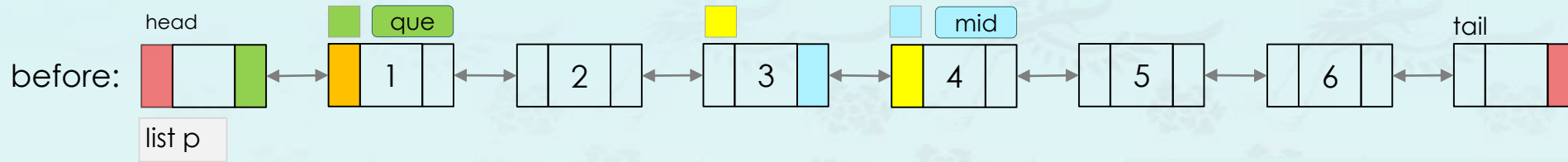
doubly linked list – **perfect shuffle()****

Implement shuffle()** function that returns so called "perfectly shuffled" list. The first half and the second half are interleaved each other. The shuffled list begins with the second half of the original. For example, 1234567890 returns 6172839405.

Algorithm:
1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.
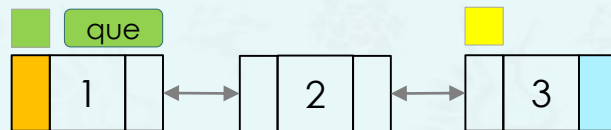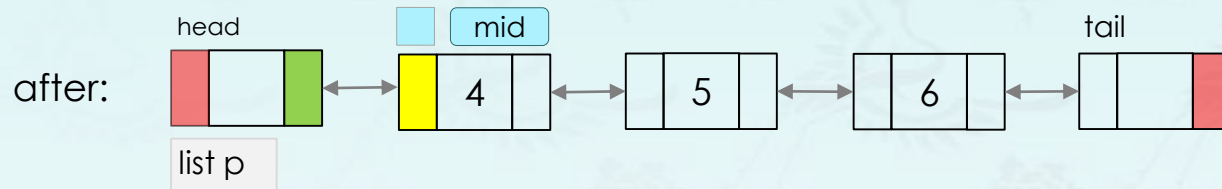4) keep on interleaving nodes until the "que" is exhausted.
   - save away next pointers of mid and que.
   - interleave nodes in the "que" into "mid" in the list of p.
     (insert the first node in "que" at the second node in "mid".)

before:

doubly linked list – **perfect shuffle()****

1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.
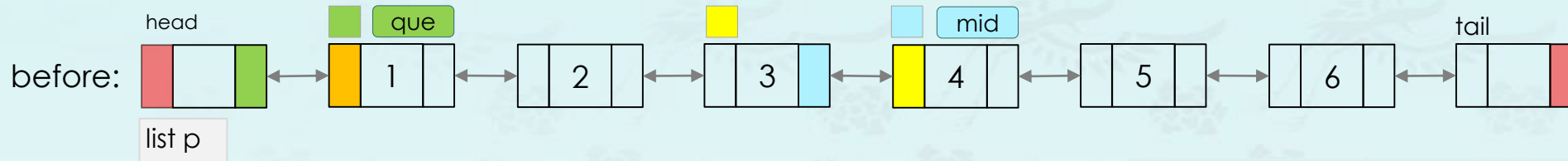4) keep on interleaving nodes until the "que" is exhausted

before:

head   que   mid   tail

list p

(1) Name functions to get **que** and **mid** nodes?

```
pNode mid = half(p);  // 4 ~ 6
pNode que = begin(p); // 1 ~ 6
```

doubly linked list – **perfect shuffle()****

1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.
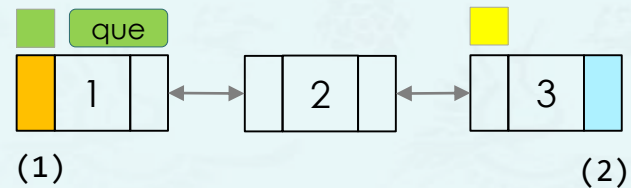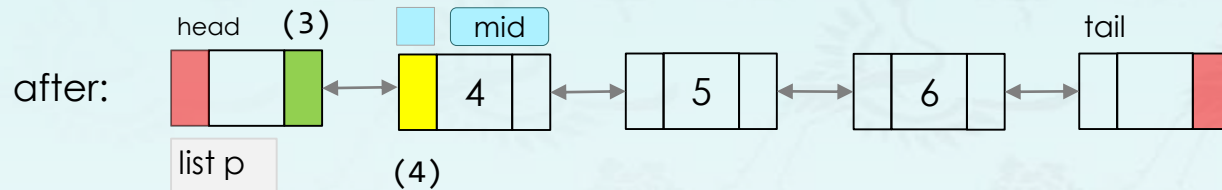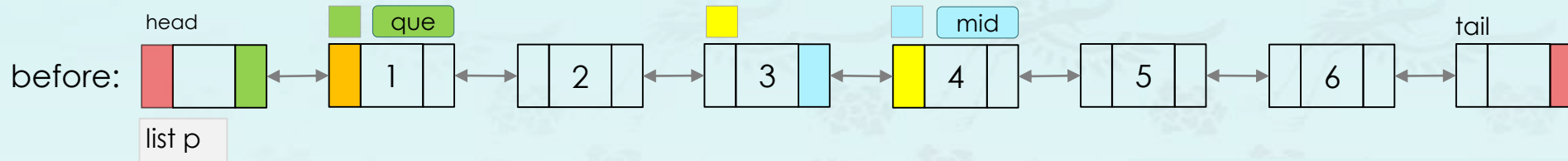4) keep on interleaving nodes until the "que" is exhausted

before:

after:

(1) Name functions to get **que** and **mid** nodes?
(2) Identify where the color should be updated.

```
pNode mid = half(p);  // 4 ~ 6
pNode que = begin(p); // 1 ~ 6
```

doubly linked list – **perfect shuffle()****

1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.
4) keep on interleaving nodes until the "que" is exhausted

before:

head   que   mid   tail
list p

| | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | |

list p

after:

head (3)   mid   tail

| | 4 | | 5 | | 6 | |

list p   (4)

que

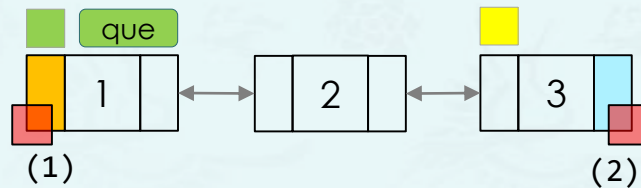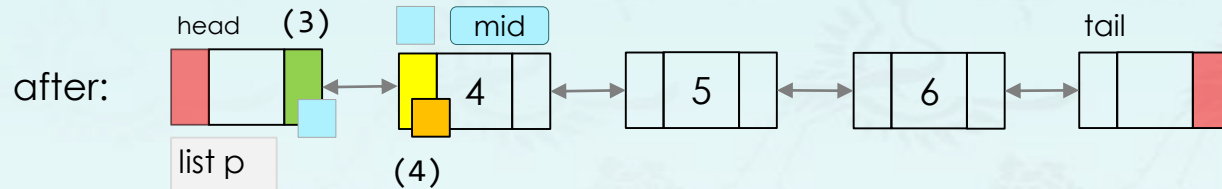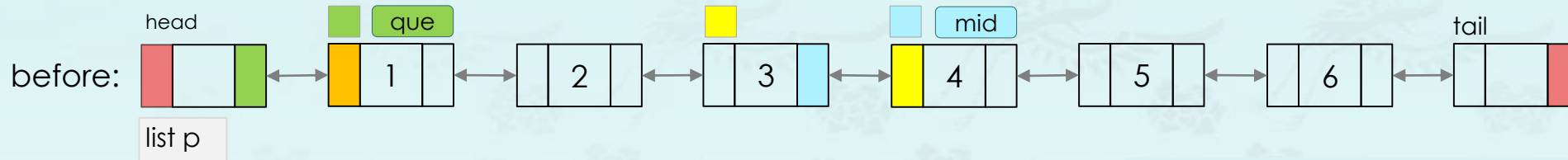| | 1 | | 2 | | 3 | |

(1)   (2)

(1) Name functions to get **que** and **mid** nodes?
(2) Identify where the color should be updated.
    How about them in code?

```
pNode mid = half(p);
pNode que = begin(p);
```

(1)
(2)
(3)
(4)

doubly linked list – **perfect shuffle()****

1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.
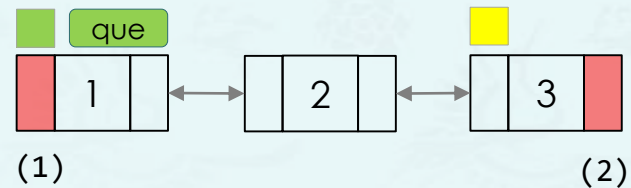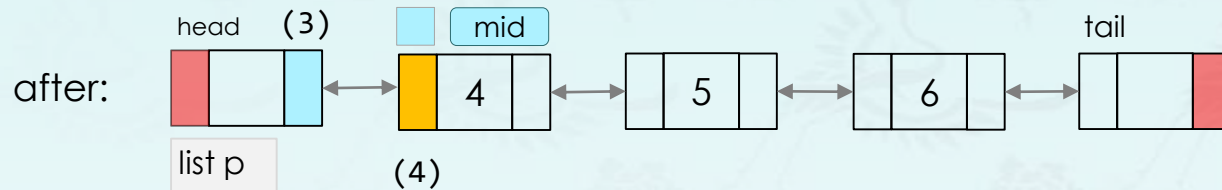4) keep on interleaving nodes until the "que" is exhausted

before:

list p

head   que   mid   tail
1   2   3   4   5   6

after:

head   (3)   mid   tail
4   5   6
list p
(4)

que
1   2   3
(1)   (2)

(1) Name functions to get **que** and **mid** nodes?
(2) Identify where the color should be updated.
    How about them in code?
(3) What color code must be there?
    How about them in code?

```
pNode mid = half(p);
pNode que = begin(p);
```

(1)
(2)
(3)
(4)

doubly linked list – **perfect shuffle()****

1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.
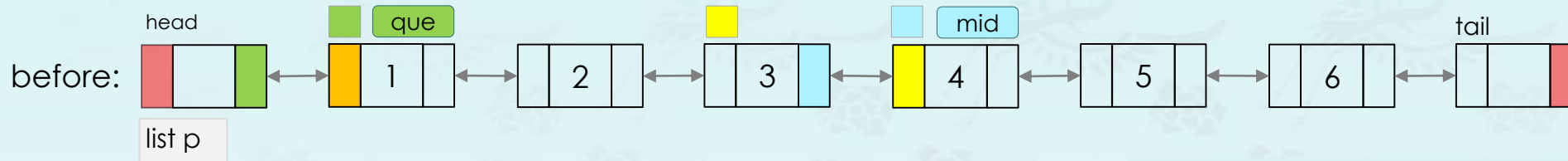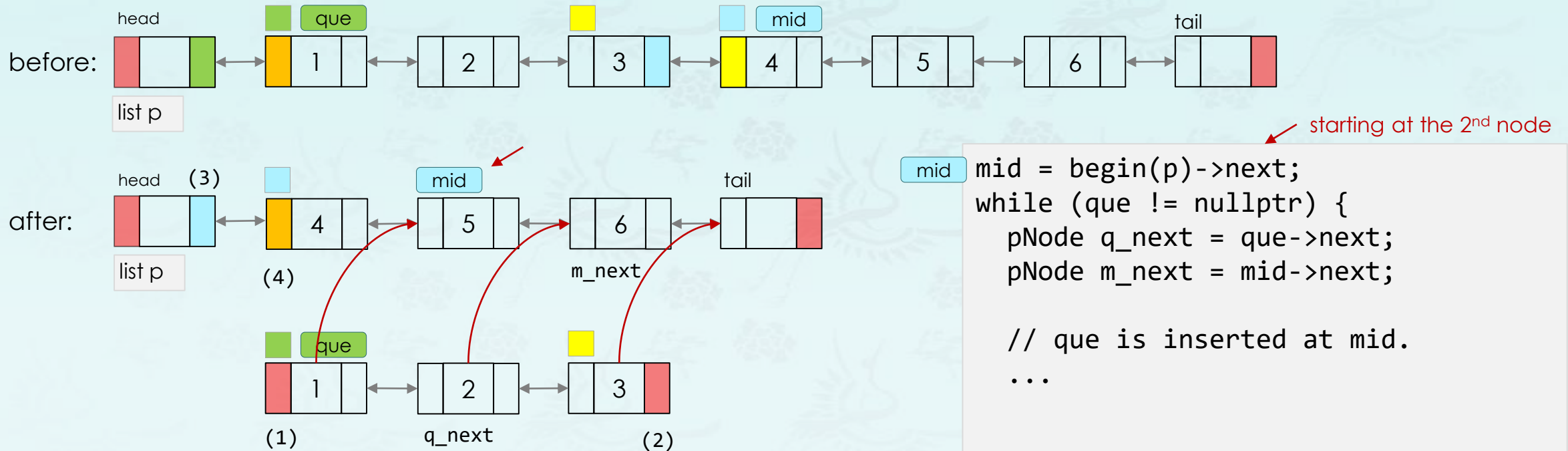4) keep on interleaving nodes until the "que" is exhausted

(1) Name functions to get **que** and **mid** nodes?
(2) Identify where the color should be updated.
    How about them in code?
(3) What color code must be there?
    How about them in code?

```
pNode mid = half(p);
pNode que = begin(p);
```

(1)
(2)
(3)
(4)

doubly linked list – **perfect shuffle()****

1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.

starting at the 2nd node

```
mid = begin(p)->next;
while (que != nullptr) {
    pNode q_next = que->next;
    pNode m_next = mid->next;

    // que is inserted at mid.
    ...



}
```

4) keep on interleaving nodes until the "que" is exhausted
   • save away next pointers of mid and que.
   • interleave nodes in the "que" into "mid" in the list of p.
     (start inserting the fist node in "que" at the second node in "mid".)

**Data Structures
Chapter 4**

1. Singly Linked List
2. Doubly Linked List
   - Revisit – Singly Linked List
   - Doubly Linked List with Sentinels
   - Basic Operations
   - **Advanced Operations**

Summary &

quaestio quaestio qo ? ? ? ?