


## Data Structures

### Chapter 4

1. Singly Linked List
2. Doubly Linked List
  - Revisit – Singly Linked List
  - Sentinel Nodes & Basic Operations
  - Two Key Operations: erase, insert
  - **Advanced Operations: half(), sorted(), push\_sorted(), swap\_pairs()**



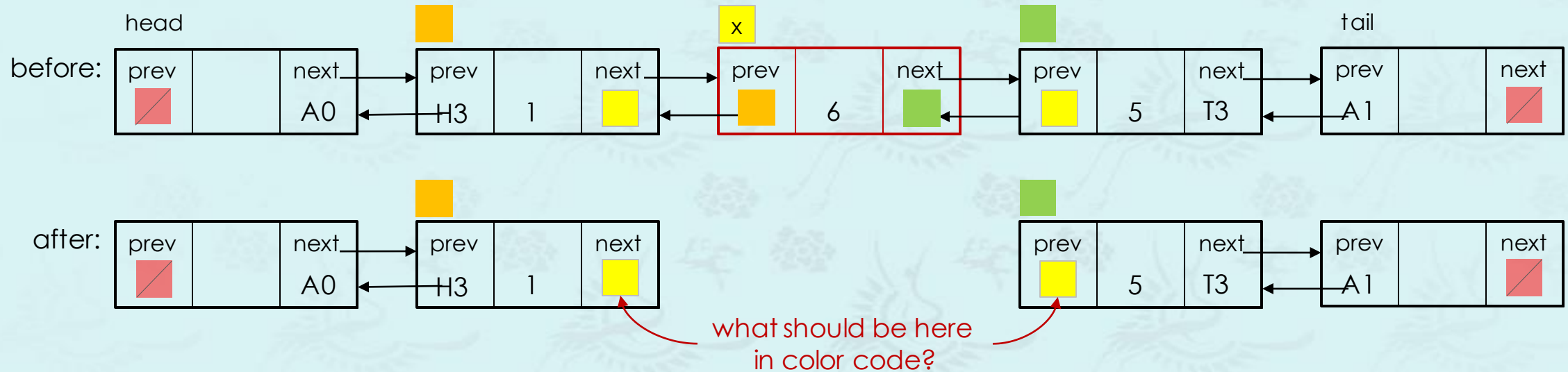
우리가 알거니와 하나님을 사랑하는 자 곧 그의 뜻대로 부르심을 입은 자들에겐 모든 것이  
합력하여 선을 이루느니라 (롬8:28)

And we know that in all things God works for the good of those who love him, who  
have been called according to his purpose. (Rom8:28)

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

## Revisit - erase()

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

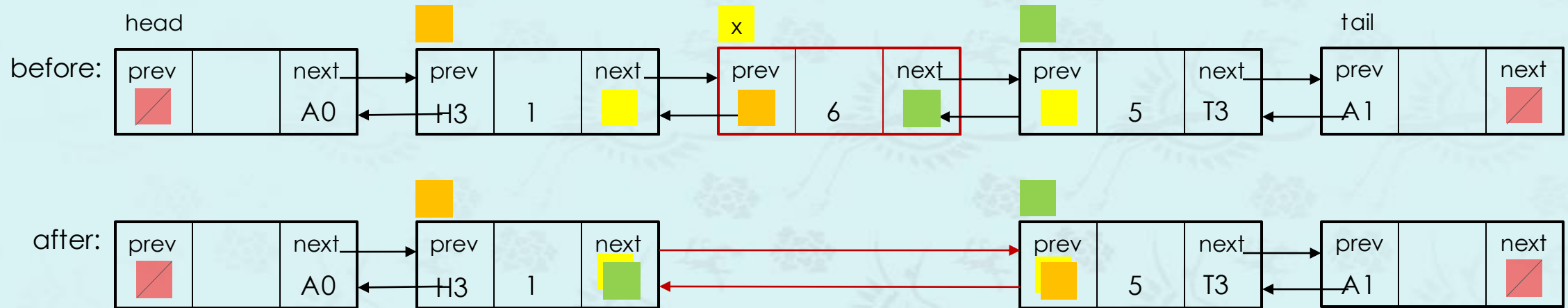


before: `x ==` [yellow square]  
[ ] == [ ]  
[ ] == [ ]  
[ ] == [ ]  
[ ] == [ ]  
express in terms of x

```
void erase(pNode x) {  
  
}  

```

## Revisit - erase()



before:

`x ==` [yellow square]

`==` [orange square]

`==` [green square]

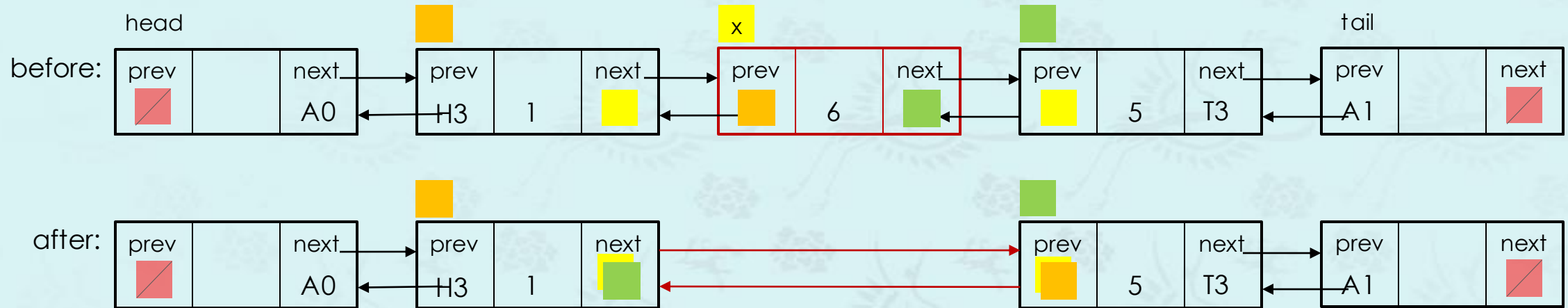
`==` [yellow square]

`==` [yellow square]

express in terms of x

```
void erase(pNode x) {
    // ...
}
```

## Revisit - erase()



before:

```

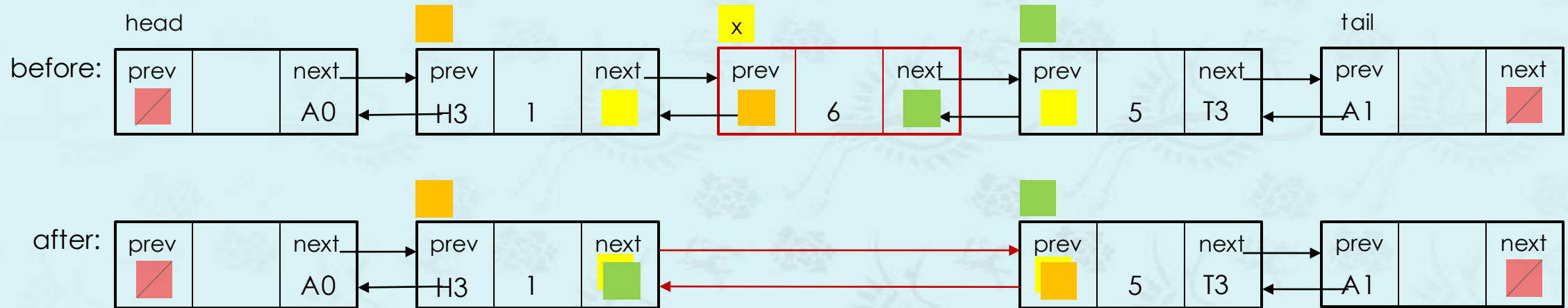
x == [yellow]
x->prev == [orange]
x->next == [green]
x->prev->next == [yellow]
x->next->prev == [yellow]
    
```

```

void erase(pNode x) {
    [ ]
}
    
```

*It should be coded using info in x only.*

## Revisit - erase()



before:

```

x == [yellow box]
x->prev == [orange box]
x->next == [green box]
x->prev->next == [yellow box]
x->next->prev == [yellow box]

```

```

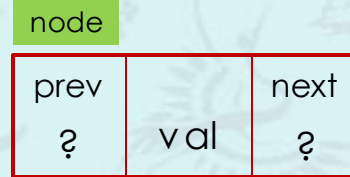
void erase(pNode x) {
    x->prev->next = x->next;
    x->next->prev = x->prev;
    delete x;
}

```

*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?



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

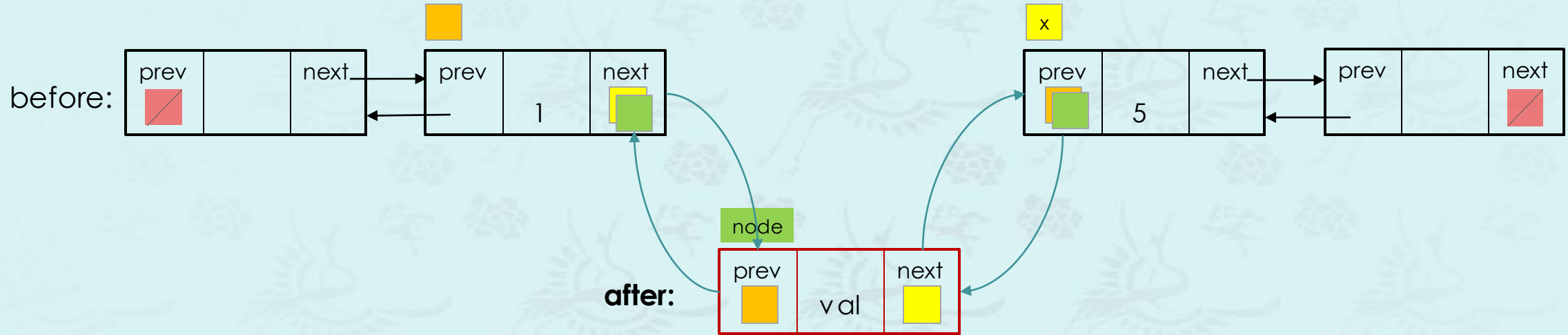
```
}
```

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



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

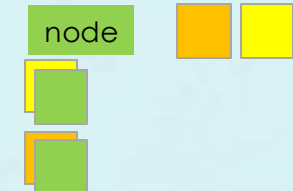
Identify where are to be changed or set?



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

```
    // Empty space for code
```

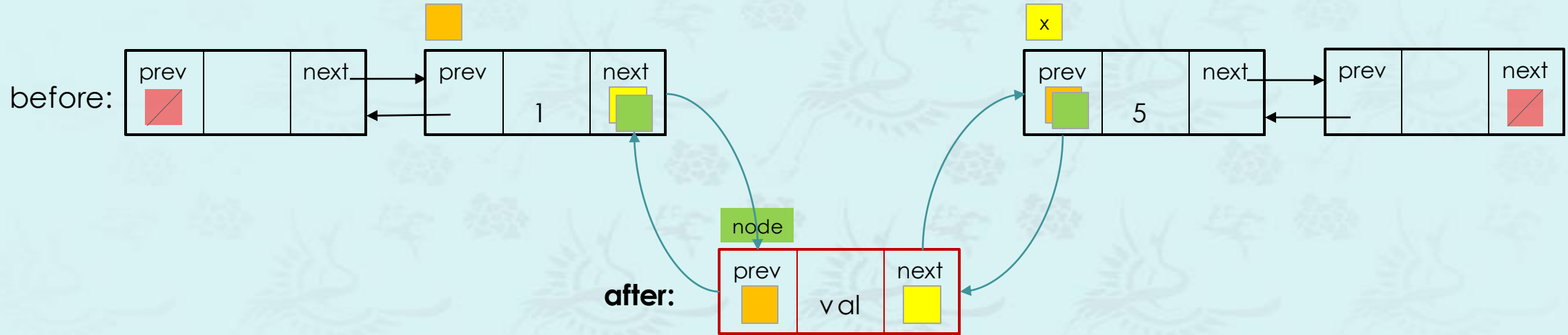
```
}
```



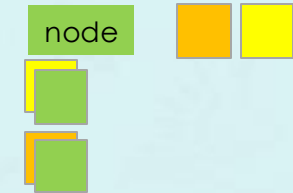
*It should be coded using value and info in x only.*



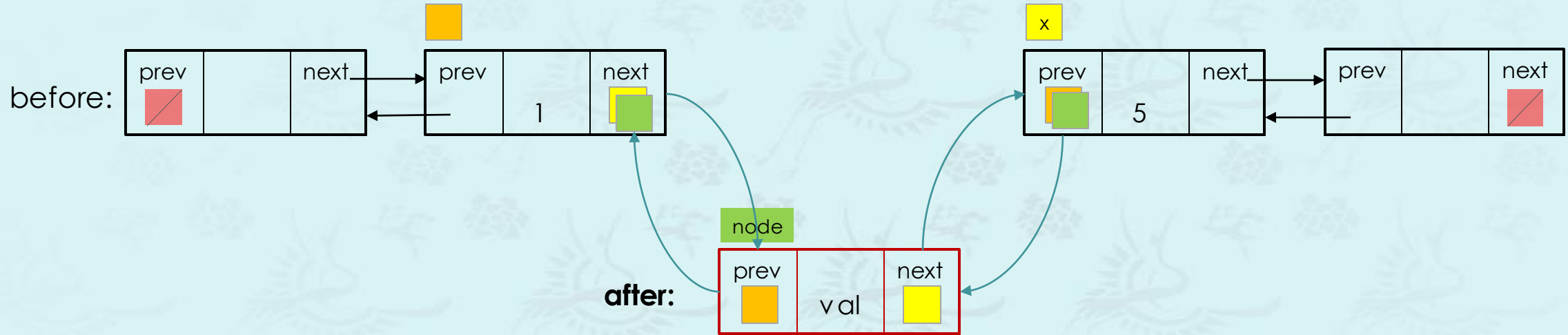
## Revisit - insert a new node(value) in place of the **node x**



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



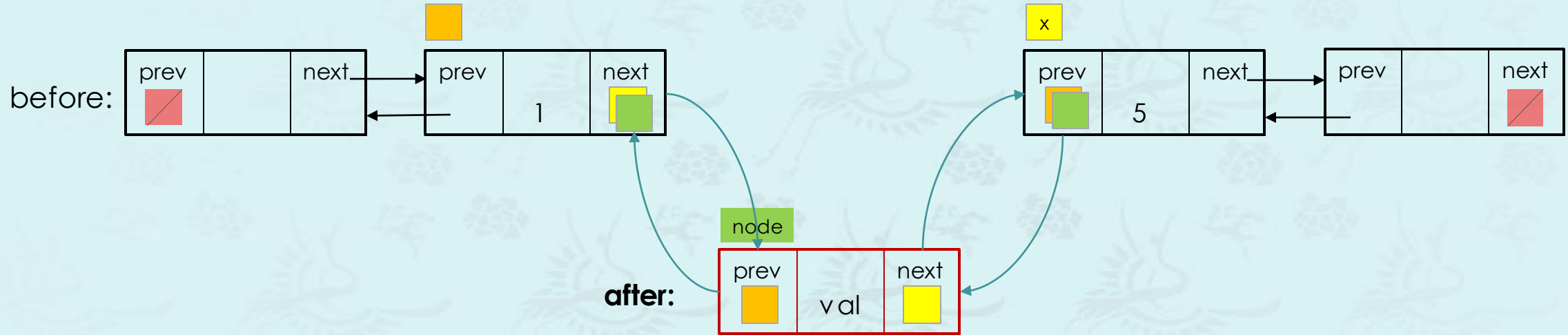
## Revisit - insert a new node(value) in place of the **node x**



```
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  
}
```

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



```
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( , value);
}
```



## 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( );
}
```



## 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?

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

Is this good enough?

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

```
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;  
    x->prev->next = x->next;  
    x->next->prev = x->prev;  
    delete x;  
} erase() function overloading
```

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

*find() returns 'tail' node or end(p) if not found.*




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 == end(p)) return;  
    erase(node);  
}
```




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


```
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);  
} erase() function overloading
```



```
void pop(pList p, int value){  
    erase(find(p, value));  
} find() returns 'tail' node or end(p) if not found.
```




```
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 == end(p)) 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 above by 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){

    return x;
}
```

### Exercise 3: How about using 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)) {
        pop(p, value);
    }
} // version.1
```

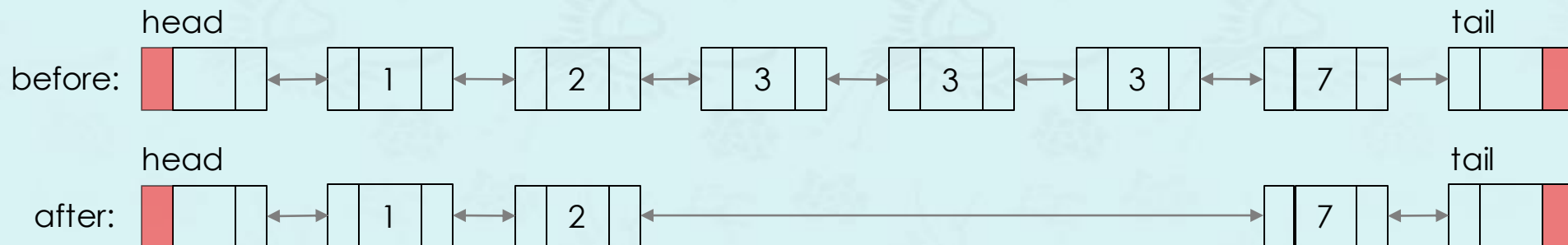
$O(n)$

$O(n)$



$O(n^2)$

The code above works, then what is the problem?  
What is the time complexity of each line and overall?



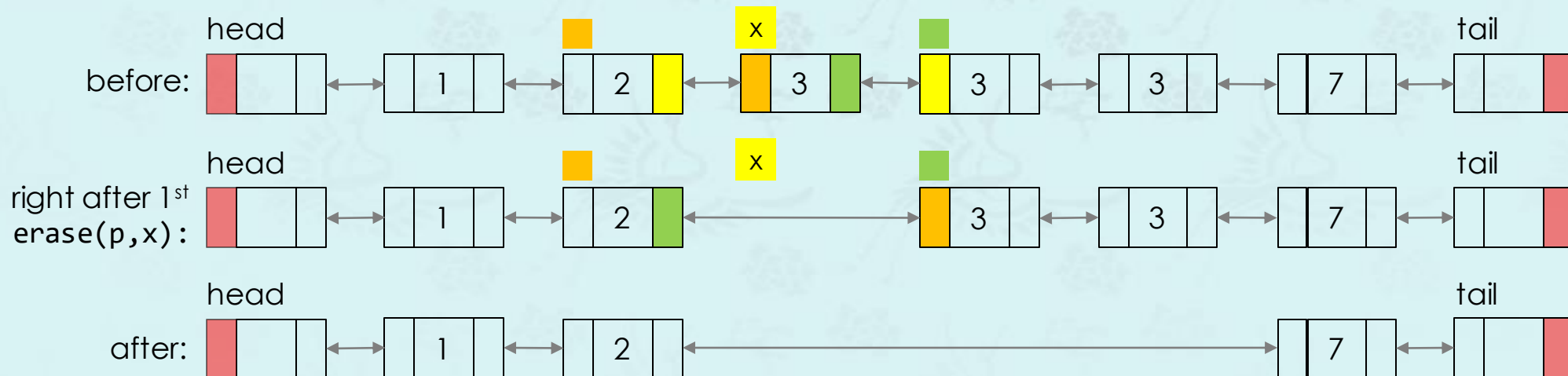


## doubly linked list – **pop\_all()**\*

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
```

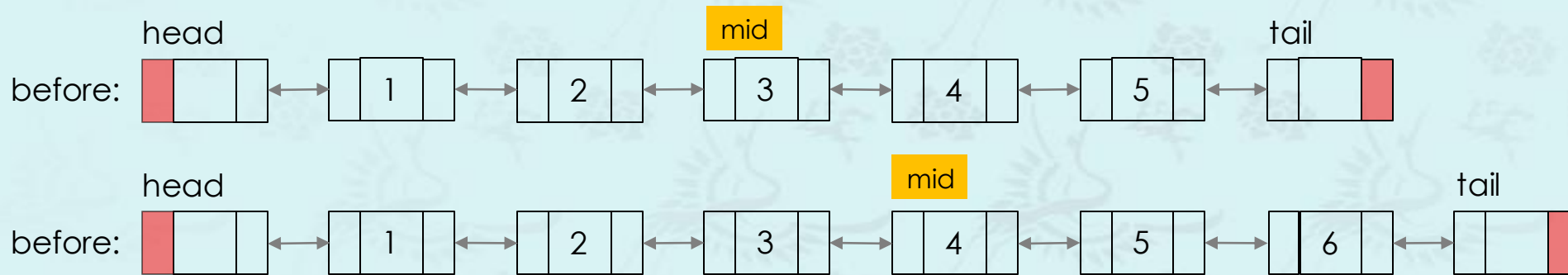
- Does `x` point to the next node right after the first `erase(p, x)` call finishes? Are you sure?
- If you have not figured it out completely, you review `erase()` source code.
- Be able to answer why the code above may work in some machines or with small number of nodes.



## doubly linked list – **half()**

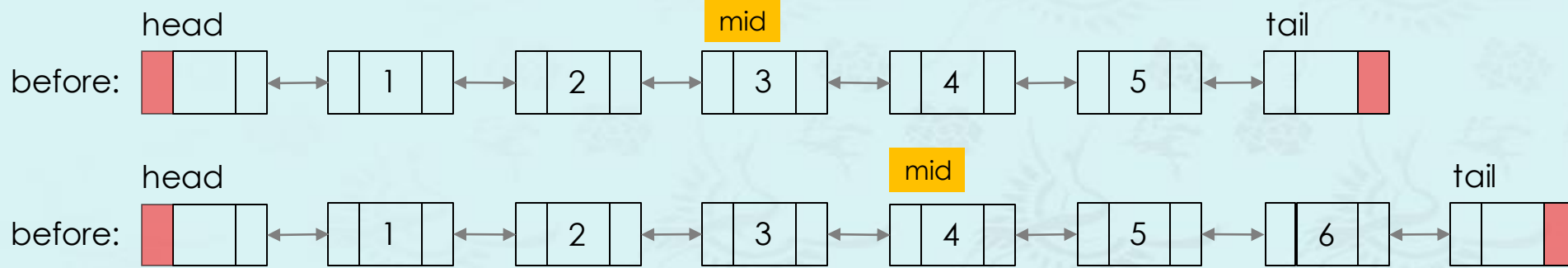
Write `half()` function that returns the mid node of the list.

- 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.

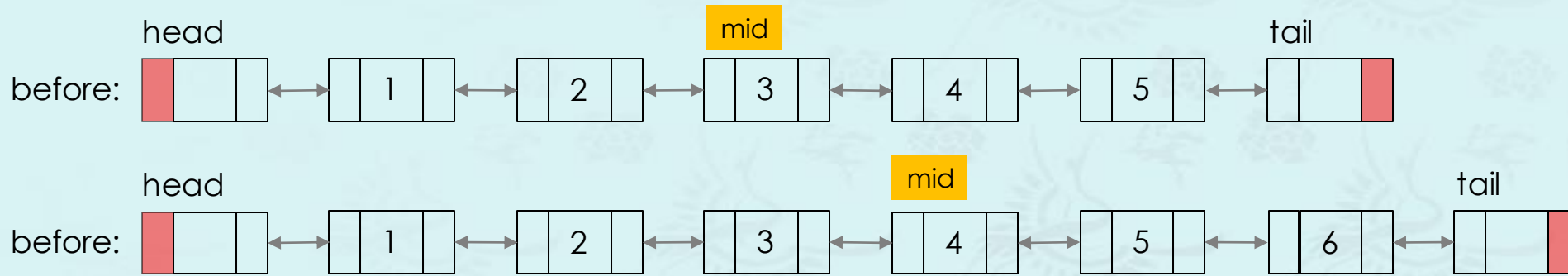
## doubly linked list – **half()**



- **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  $T(n)$ ?

```
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  
}
```

## doubly linked list – **half()**



- **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  $T(n)$ ?

## doubly linked list – **sorted()**

Implement two `sorted()` functions that returns `true` if a list is sorted and `false` otherwise.

- The one takes a list and a compare function pointer to compare two nodes.  
Using this function, the user can distinguish between an ascending list and a descending list.
- The other takes just a list and returns `true` if it is a sorted list, either ascending or descending.

```
bool sorted(pList p, int(*comp)(int a, int b)) {  
    if (size(p) <= 1) return true;  
  
    int data = "set it to 1st node value"  
    for (pNode x = "starts from second node"; x != end(p); x = x->next) {  
        // your code here; return false as soon as out of order found  
        // compare data and x->data  
        data = x->data;  
    }  
    return true;  
}
```

```
bool sorted(pList p) {  
    return false;           // your code here;  
}
```

## doubly linked list – helper functions

- Write two compare functions which are used to invoke `sorted()`.

```
int ascending (int a, int b) { return a - b; };  
int descending(int a, int b) { return b - a; };
```

- Write two helper functions, `more()` and `less()` which are used to invoke `push_sorted()`.  
The `more()` or `less()` returns **the node** of which value is greater or smaller than a given value `z` firstly encountered, respectively.
- You may modify `find()` function to code these functions.

```
pNode more(pList p, int z) {  
    pNode x = begin(p);  
    // your code here  
  
    return x;  
}
```

```
pNode less(pList p, int z) {  
    pNode x = begin(p);  
    // your code here  
  
    return x;  
}
```

```
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;  
}
```

## doubly linked list – **push\_sorted()**

Write push\_sorted function that inserts a node with value into a "sorted" list.

```
// inserts a new node with value in sorted order
void push_sorted(pList p, int value) {
    if sorted(p, "ascending_function_ptr")
        insert("find a node more() than value", value);
    else
        insert("find a node less() than value", value);
}
```

Note:

- If you use "using namespace std, the less() function may have a name conflict.
- Then use the scope resolution operator :: to access the function we defined.



## doubly linked list – **push\_sorted()**

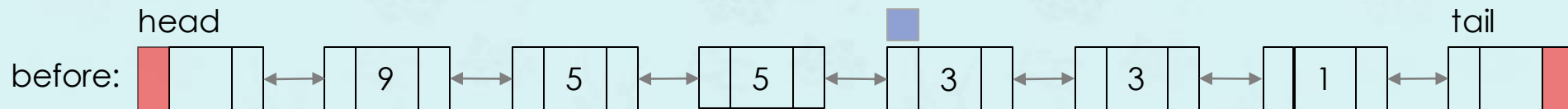
Write push\_sorted function that inserts a node with value into a "sorted" list.

```
// inserts a new node with value in sorted order
void push_sorted(pList p, int value) {
    if sorted(p, "ascending_function_ptr")
        insert("find a node more() than value", value);
    else
        insert("find a node less() than value", value);
}
```

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

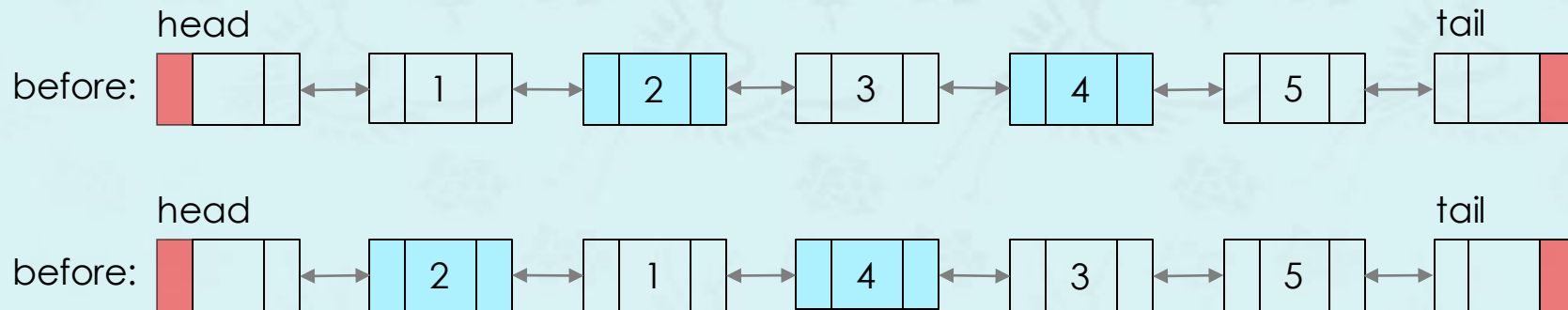


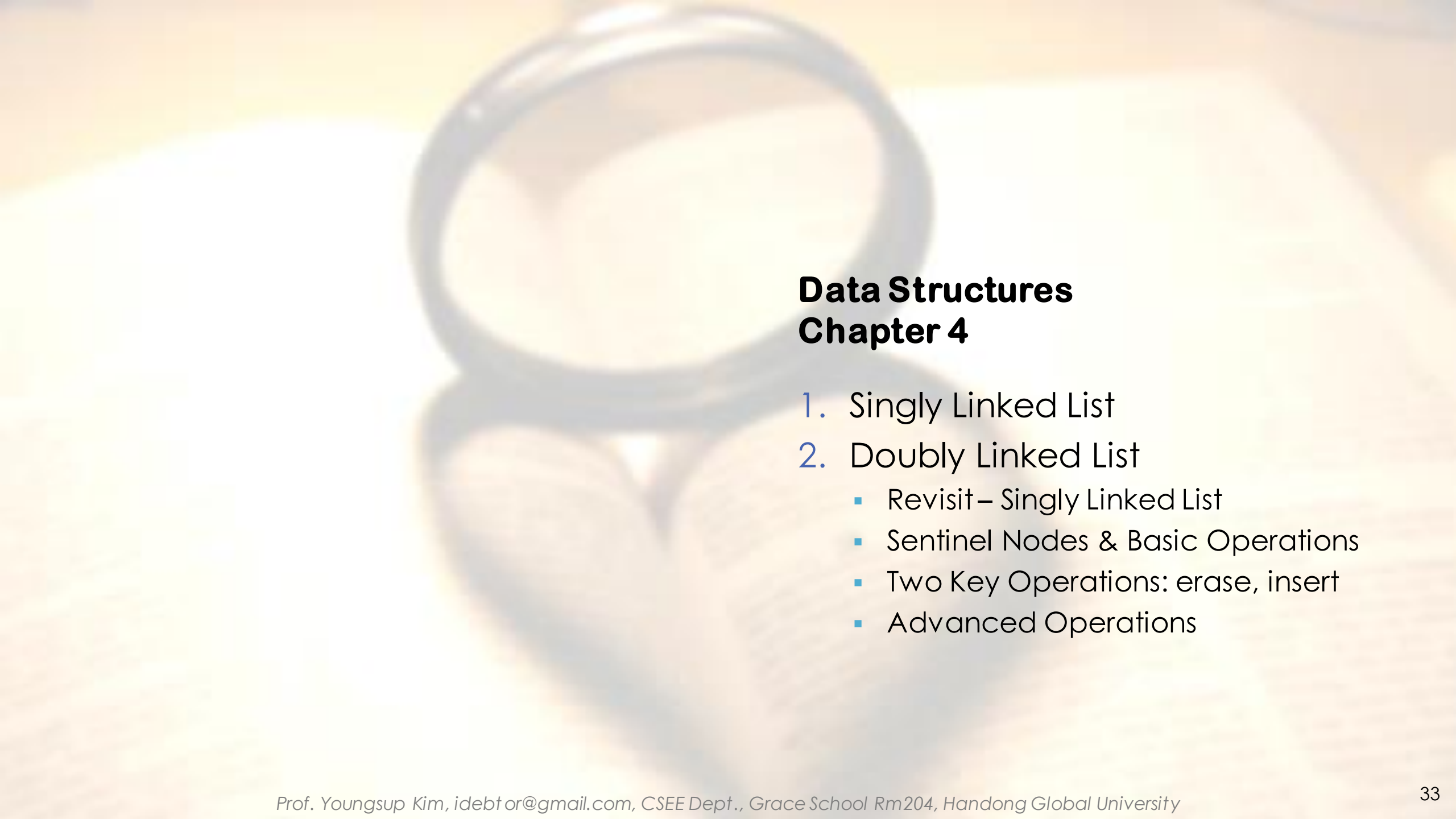
Which node should be located to invoke insert() if value = 5?



## doubly linked list – **swap\_pairs()**

It swaps two values in adjacent nodes of the list. It does not swap the links, but the values only. If the list has an odd number of nodes, then the last one remains as it is. It goes through the list once, its time complexity is  $O(n)$ .





## **Data Structures**

### **Chapter 4**

1. Singly Linked List
2. Doubly Linked List
  - Revisit – Singly Linked List
  - Sentinel Nodes & Basic Operations
  - Two Key Operations: erase, insert
  - Advanced Operations