

# 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(), unique(), reverse(), randomize()**





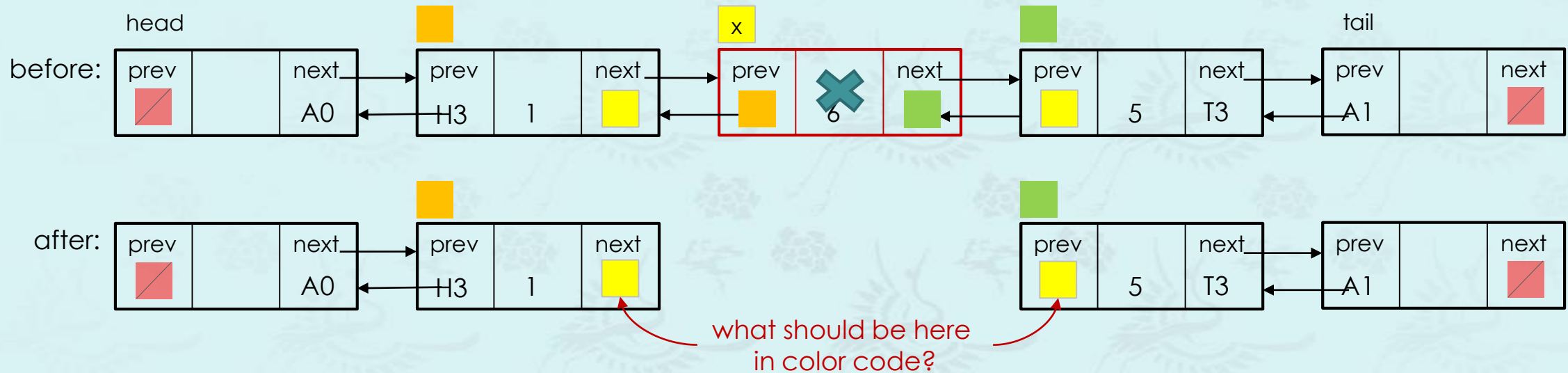
**우리가 알거니와 하나님을 사랑하는 자 곧 그의 뜻대로 부르심을 입은 자들에게는 모든 것이 합력하여 선을 이루느니라 (롬8:28)**

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?

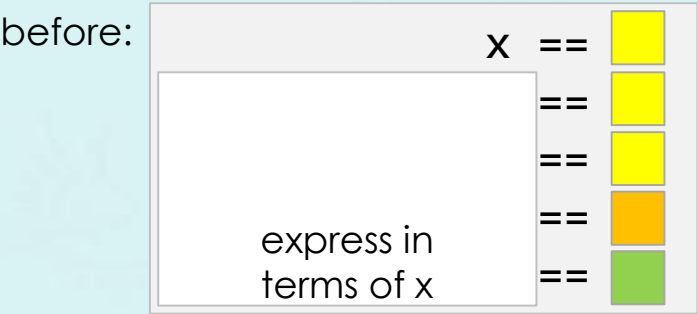
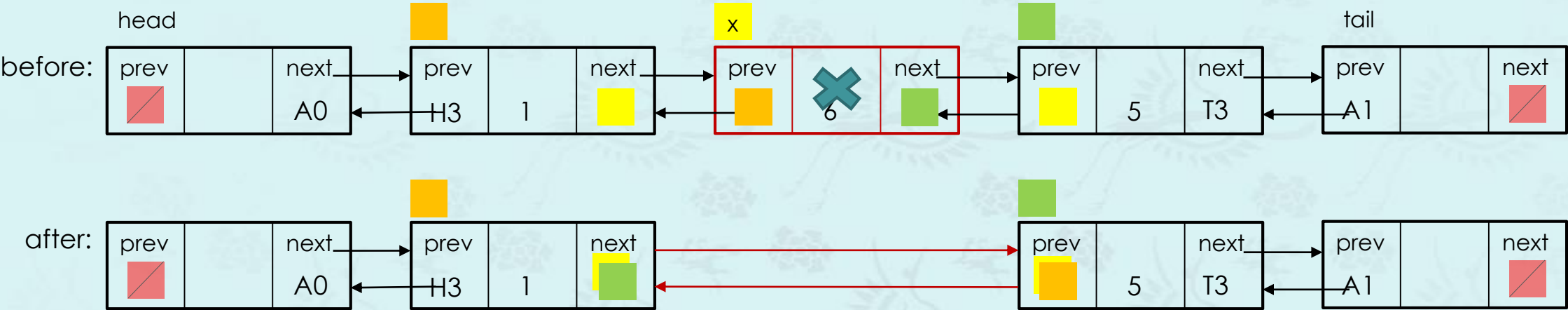


before:

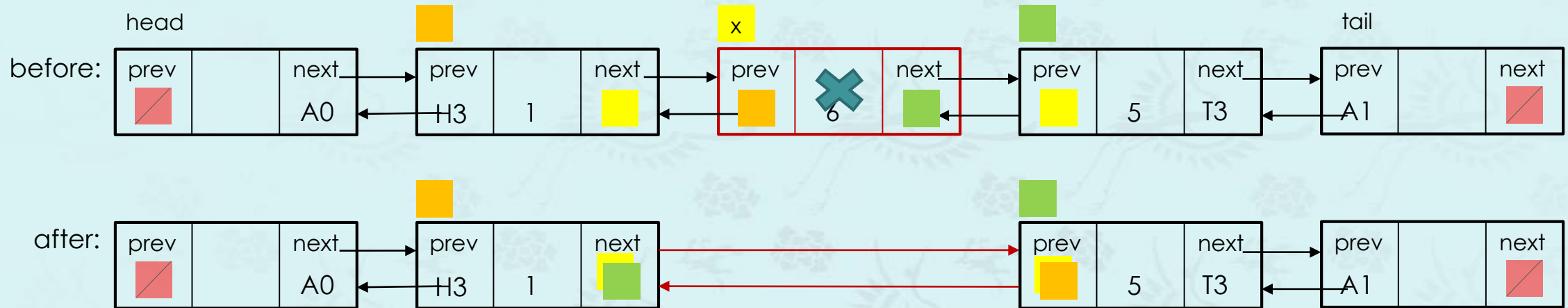
```
x == [yellow box]
x->prev->next == [white box]
x->next->prev == [white box]
x->prev == [white box]
x->next == [white box]
```

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

# Revisit - erase()



## Revisit - erase()

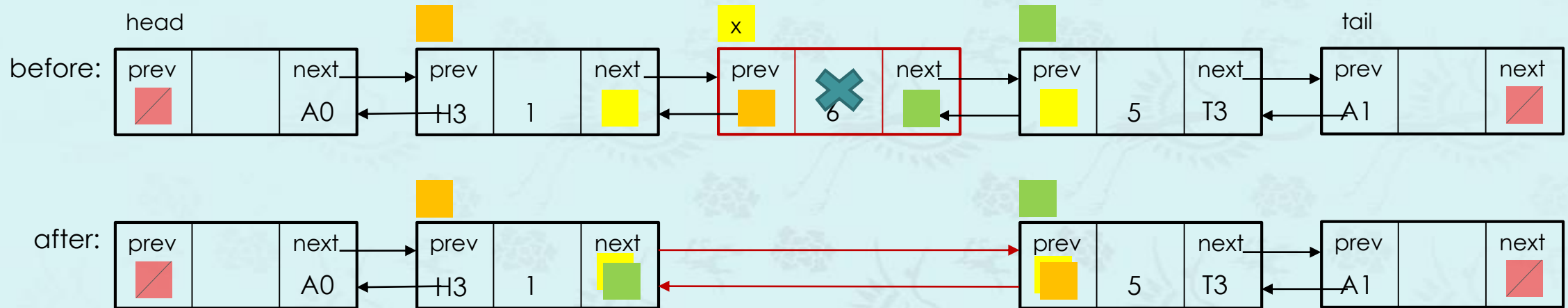


before:

<code>x ==</code>	yellow square
<code>x-&gt;prev-&gt;next ==</code>	yellow square
<code>x-&gt;next-&gt;prev ==</code>	yellow square
<code>x-&gt;prev ==</code>	orange square
<code>x-&gt;next ==</code>	green square



## Revisit - erase()



before:

```

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

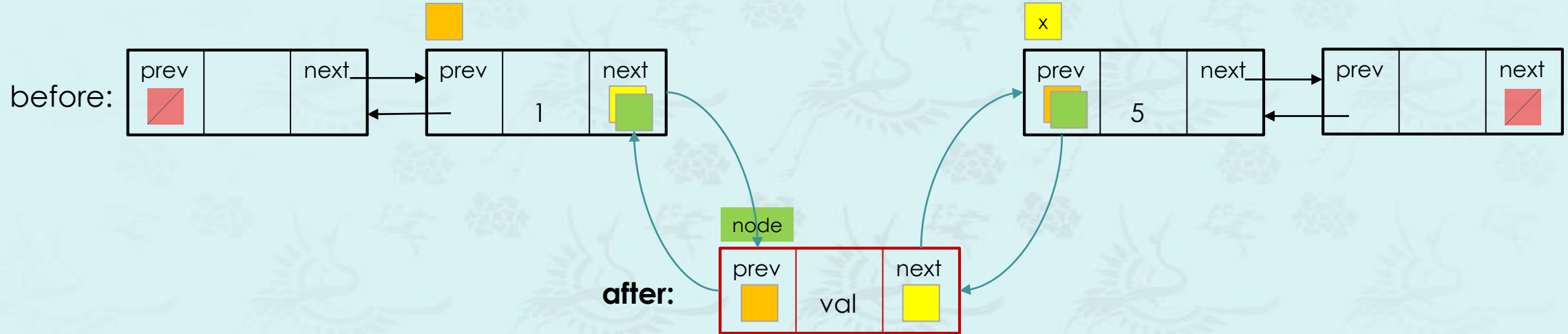
```

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.*

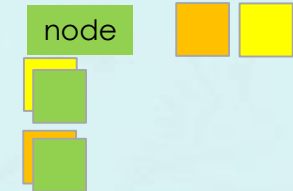
## Revisit - insert()

Identify where are to be changed or set?



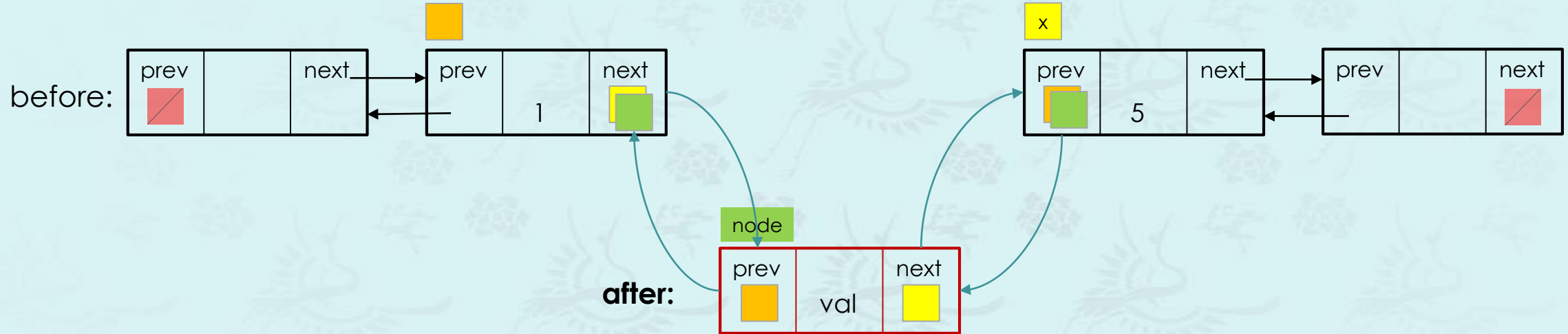
```
void insert(pNode x, int value) {
```

```
    // ...  
}
```

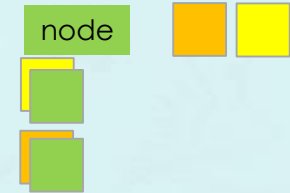


*It should be coded using value and info in x only.*

## Revisit - insert()

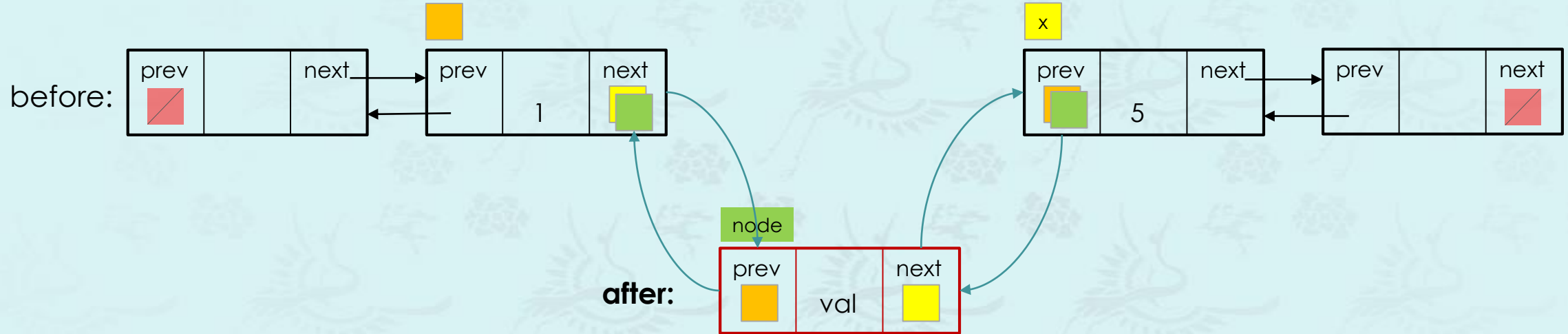


```
void insert(pNode x, int value) {  
    pNode node = new Node{value, x->prev, x};  
    x->prev->next = node;  
    x->prev = node;  
}
```



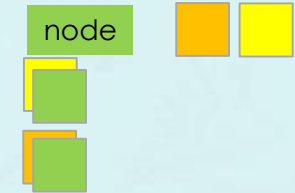


## Revisit - insert()

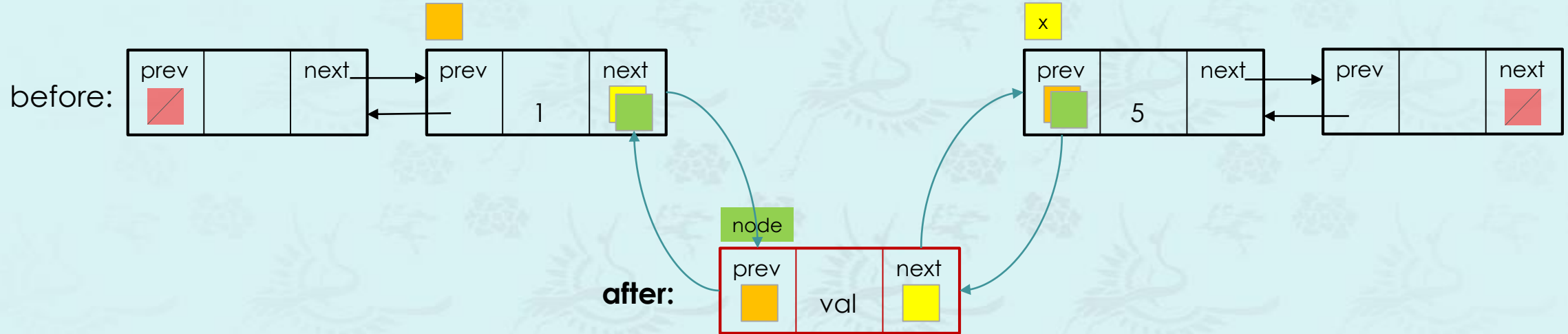


```
void insert(pNode x, int value) {
    pNode node = new Node{value, x->prev, x};
    x->prev->next = node;
    x->prev = node;
}
```

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



## Revisit - insert()



```
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  
}
```

## 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?

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

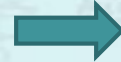


This code may not work some cases.  
How can you fix it?

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




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






## 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);  
}
```



```
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);  
}
```



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

```
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;
}
```

- What does find() return if value not found?  
**The "end" node which is not nullptr.**
- Can we reduce the lines above by two?
- How about using for loop?

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

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

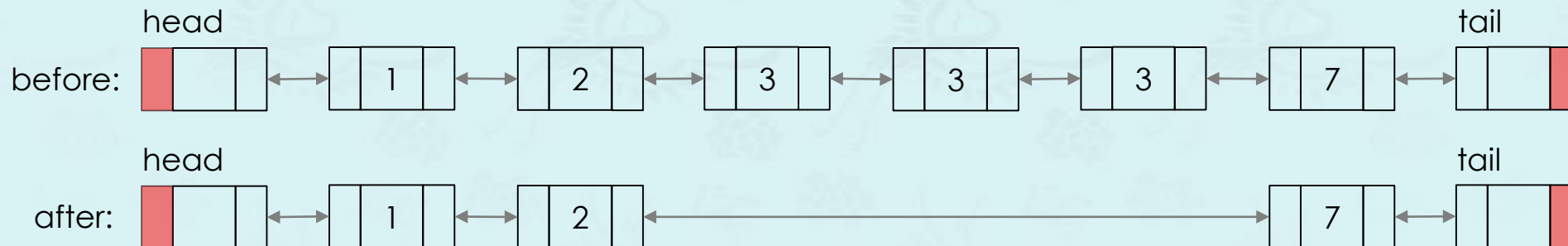
$O(n)$

$O(n)$



$O(n^2)$

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

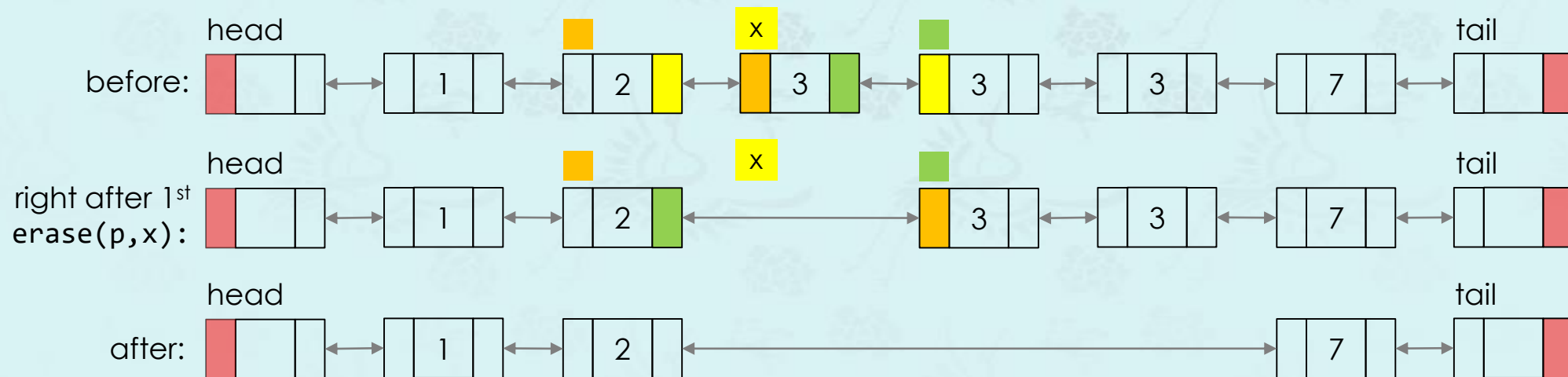


## doubly linked list – **pop\_all()**\*

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
```

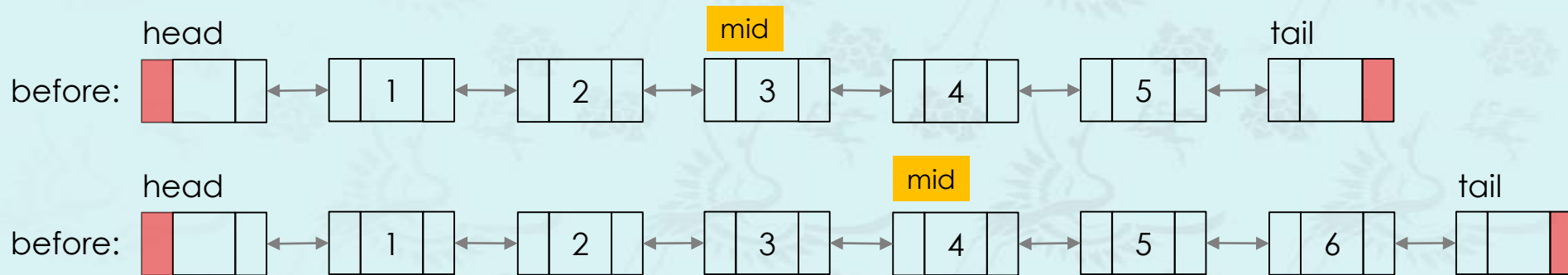
- Does `x` point to the next node right after the first `erase(p, x)` call finishes? Are you sure?
- If you have not figured it out completely, you review `erase()` source code.
- Be able to answer why the code above may work in some machines or with small number of nodes.



## doubly linked list – **half()**

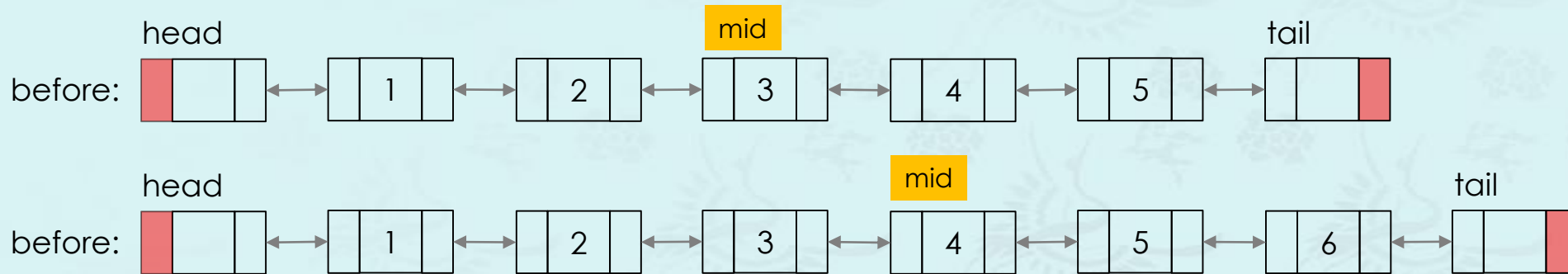
Write `half()` function that returns the mid node of the list.

- 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()**

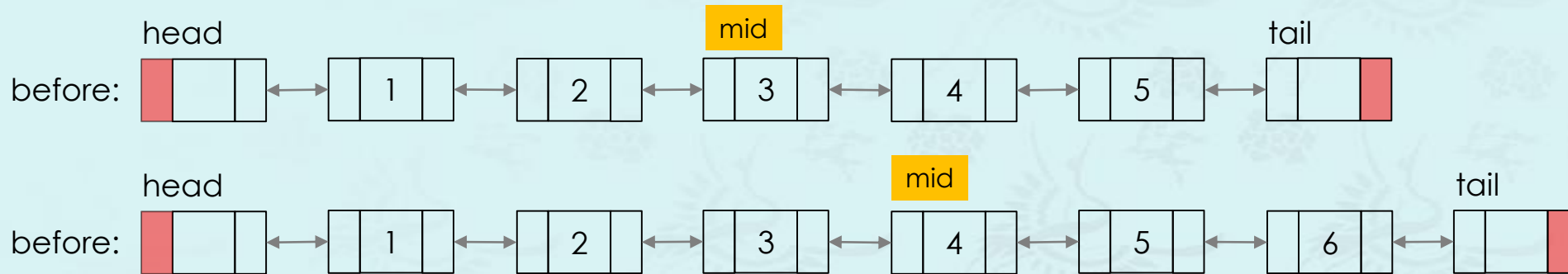


- **Method 1:** Get the size of the list  $O(n)$ . Then scan to the halfway, breaking the last link followed.
- If the list size  $n = 2$  million nodes, What is  $T(n)$ ?

```
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
```

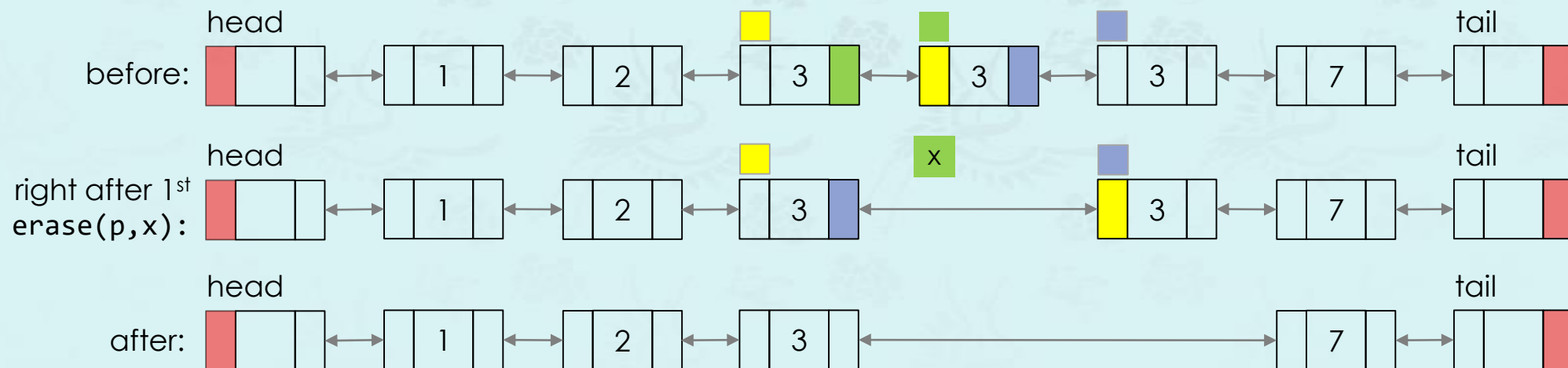
- If the list size  $n = 2$  million nodes, What is  $T(n)$ ?

## doubly linked list – **unique()**\*

An `unique()` function removes extra nodes in sub-lists of equal value of nodes from the list. It removes all but the first node from every consecutive groups of equal nodes. This function is useful for sorted lists. The list should only be traversed once to have the time complexity,  $O(n)$ .

```
void unique(pList p) {  
    if (size(p) <= 1) return;  
    for (pNode x = begin(p); x != end(p); x = x->next)  
        if (x->data == x->prev->data) erase(p, x);  
} // version.1 buggy – it may not work in some machines or for a large list.
```

We can proceed down the list and compare adjacent nodes. When adjacent nodes are the same, remove the second one. But you need to do something before and after the deletion. That is a tricky part of coding.



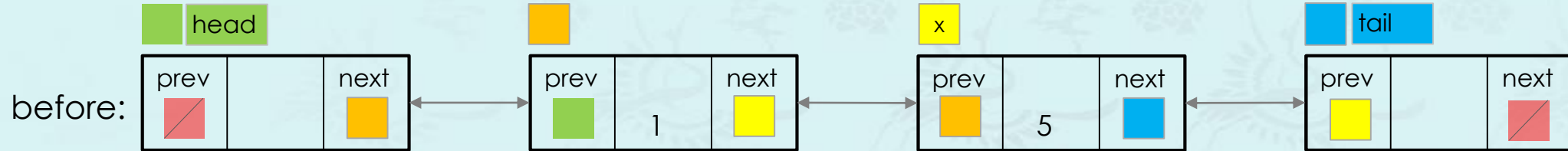
## doubly linked list – **reverse()**

Write `reverse()` function that reverses the order of the nodes in the list. The entire operation does not involve the construction or destruction of any element. Nodes are not moved, but pointers are moved within the list. Its time complexity is  $O(n)$ .

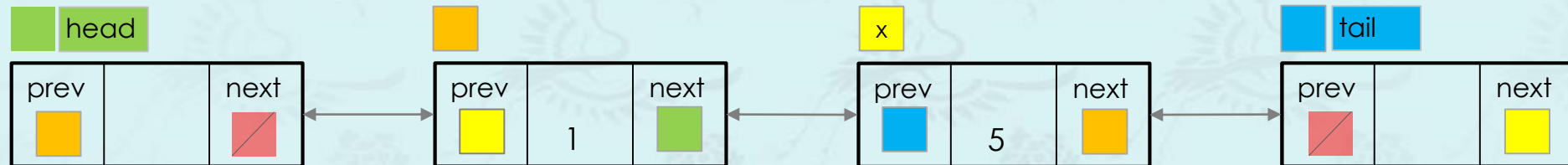
```
// reverses the order of the nodes in the list container. O(n)
void reverse(pList p) {
    if (size(p) <= 1) return;
    // hint: swap prev and next in every node including two sentinel nodes.
    // then, swap head and tail.
    // your code here

}
```

## doubly linked list – **reverse()**



Step 1: swap prev and next in every node **including two sentinel nodes**.



Step 2: swap head and tail node.



## doubly linked list – **randomize()**

There are many ways to implement this function. The most well-known algorithm is called the Fisher-Yates shuffle. You may refer to Wikipedia or random.cpp for detail.

- **This naive method we use here** swaps each element with another element chosen randomly from all elements. Even though this is not the best approach, however, it is acceptable at this point.

```
void randomize(pList p) {
    int N = size(p);
    if (N <= 1) return;
    srand((unsigned)time(nullptr));

    pNode curr = begin(p);
    while (curr != end(p)) {
        int x = rand_extended(N) % N;    O(1)
        pNode xnode = find_by_index(p, x);
        swap(curr->data, xnode->data);    O(1)
        curr = curr->next;
    }
}
```

- The time complexity of find\_by\_index(): \_\_\_\_\_
- The time complexity of randomize(): \_\_\_\_\_

```
// a helper function
pNode find_by_index(pList p, int n_th) {
    int n = 0;
    pNode curr = begin(p);
    while (curr != end(p) && n_th != n)
        curr = curr->next;
    return curr;
}
```

## doubly linked list – **randomize()**

Rewrite the randomize() such that it may have the time complexity of  $O(n)$ .

### Hints:

- Rewrite the code such that it does not use find\_by\_index() of which the time complexity is  $O(n)$ .
- How about extracting all the node values before while() loop and save them in an array?

```
void randomize(pList p) {  
    int N = size(p);  
    if (N <= 1) return;  
    srand((unsigned)time(nullptr));  
    // your code here  
    pNode curr = begin(p);  
    while (curr != end(p)) {  
        int x = rand_extended(N) % N;            $O(1)$   
        pNode xnode = find_by_index(p, x);  
        swap(curr->data, ... );                  $O(1)$   
        curr = curr->next;  
    }  
}
```

- The time complexity of extracting values: \_\_\_\_\_
- The time complexity of randomize(): \_\_\_\_\_



# 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**

*Summary &*  
*quaestio quaestio* 90 < 9 9 ? ?  
→