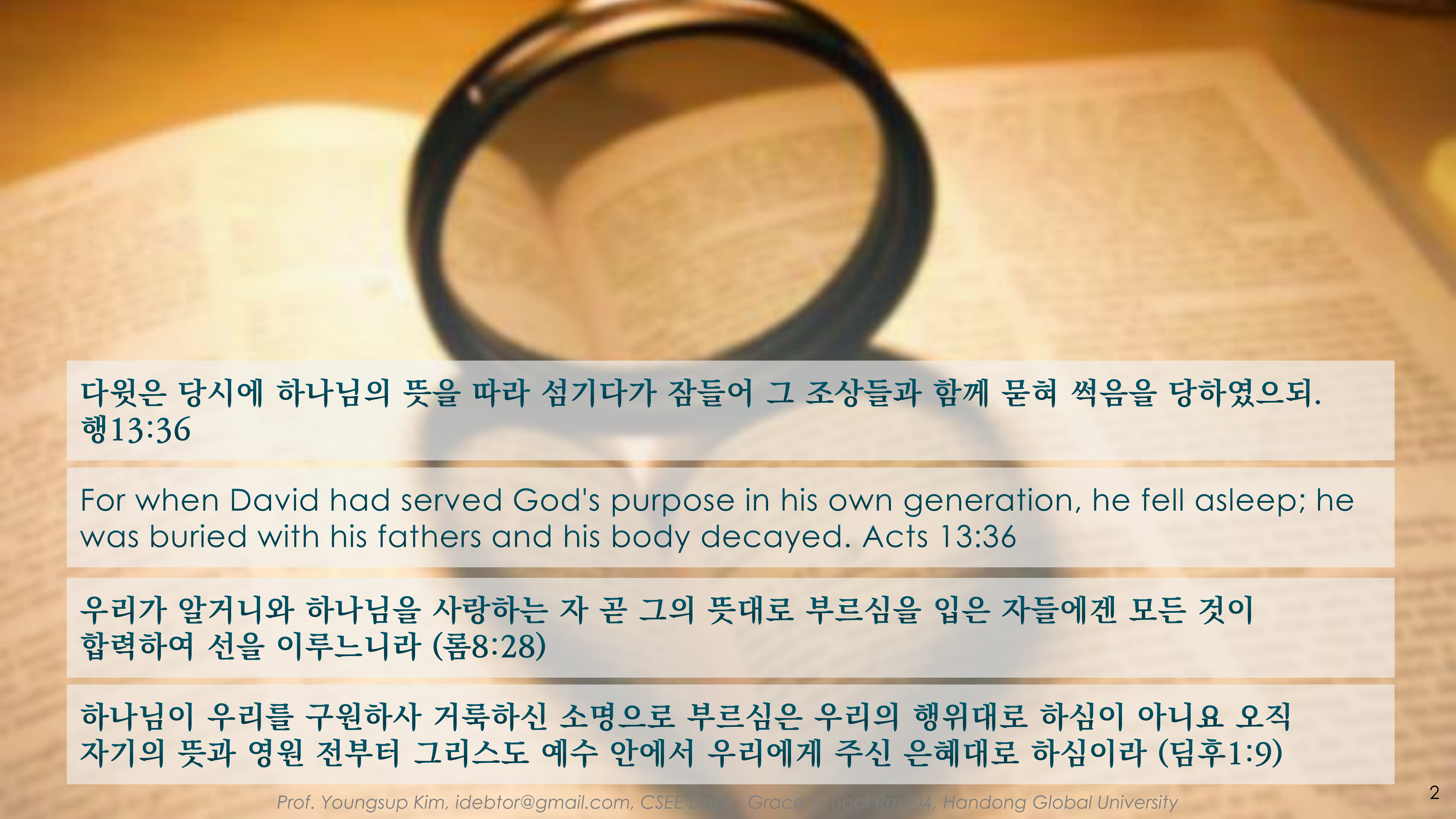
Data Structures
Chapter 4

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

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

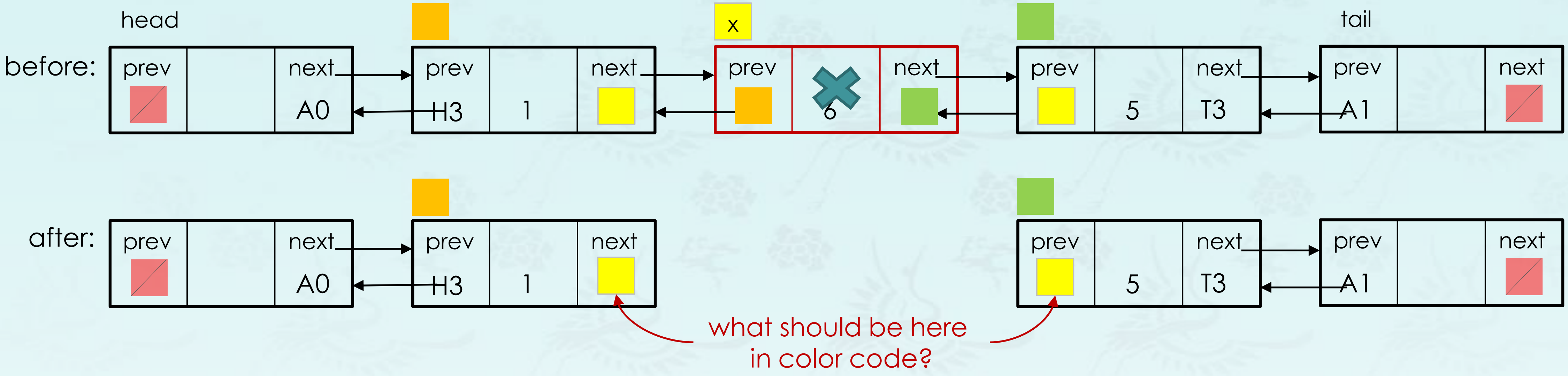
For when David had served God's purpose in his own generation, he fell asleep; he was buried with his fathers and his body decayed. Acts 13:36

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)

Revisit - erase()

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



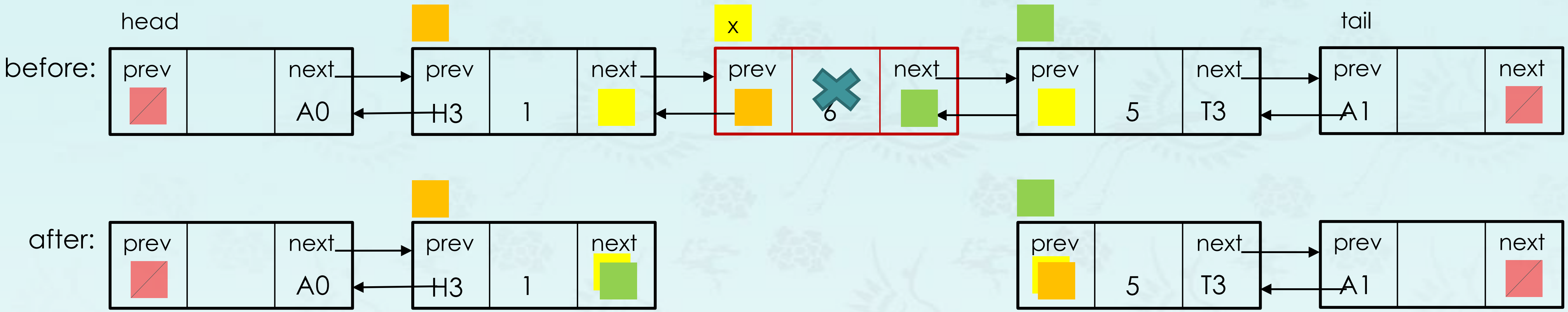
before:

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

```
void erase(pNode x) {

}
```

Revisit - erase()

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

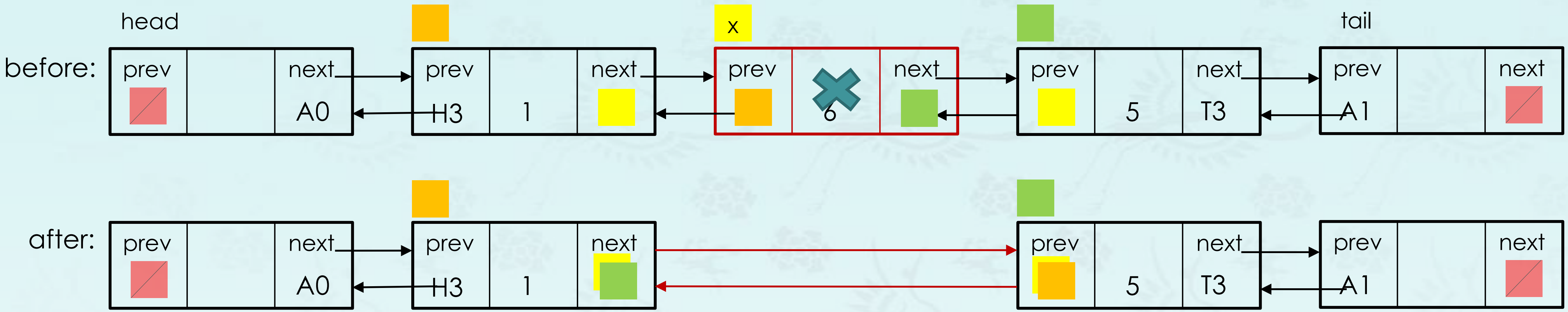


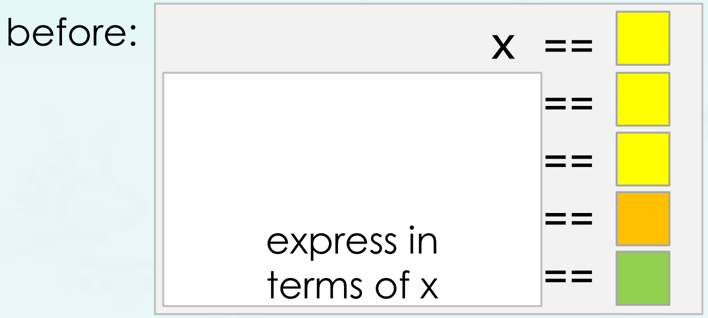
before:

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

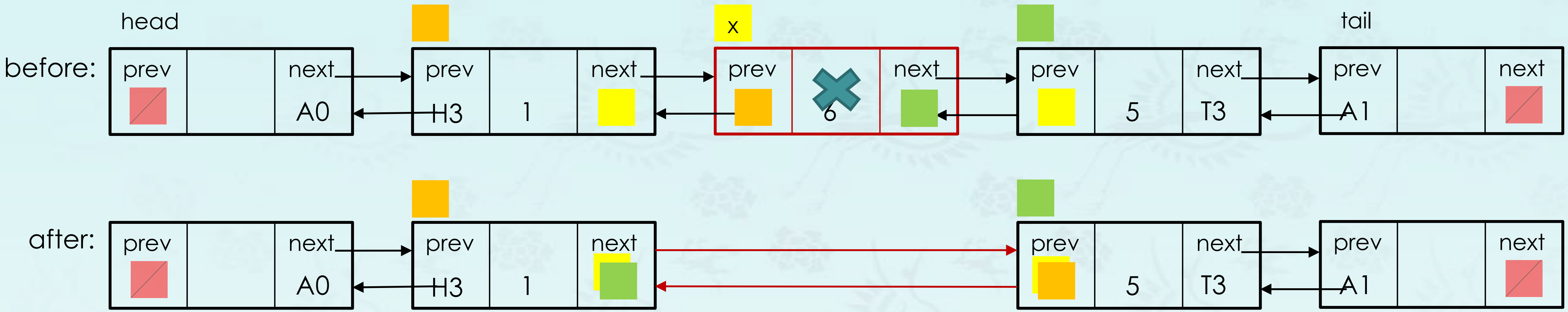
```
void erase(pNode x) {

}
```

Revisit - erase()





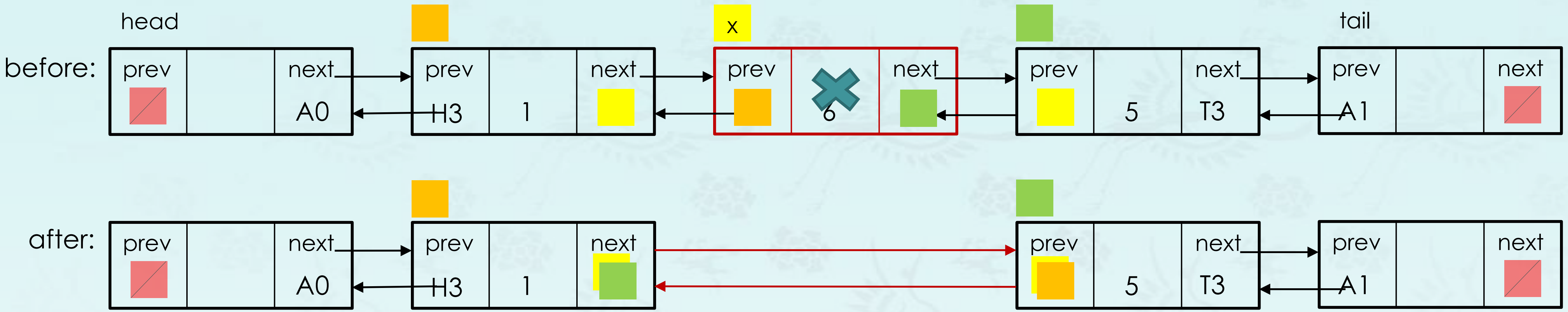
Revisit - erase()



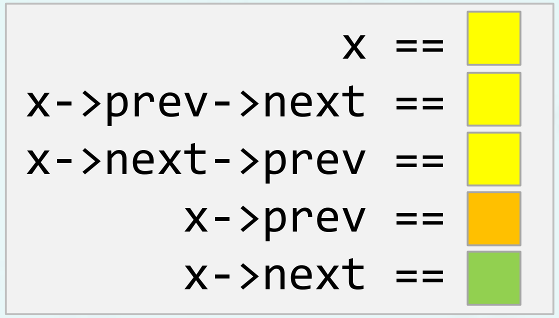
before:

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

Revisit - erase()



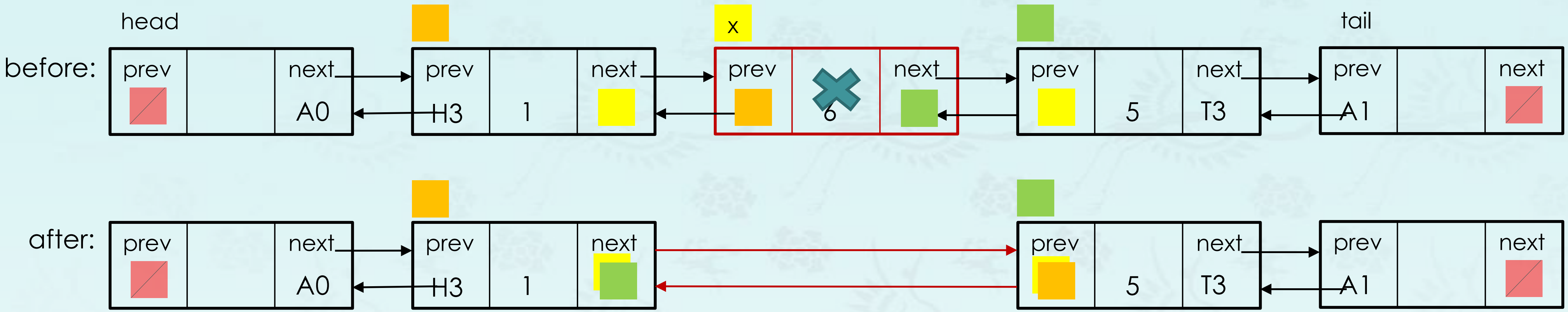
before:




```
void erase(pNode x) {

}
```

It should be coded using info in x only.

Revisit - erase()



before:

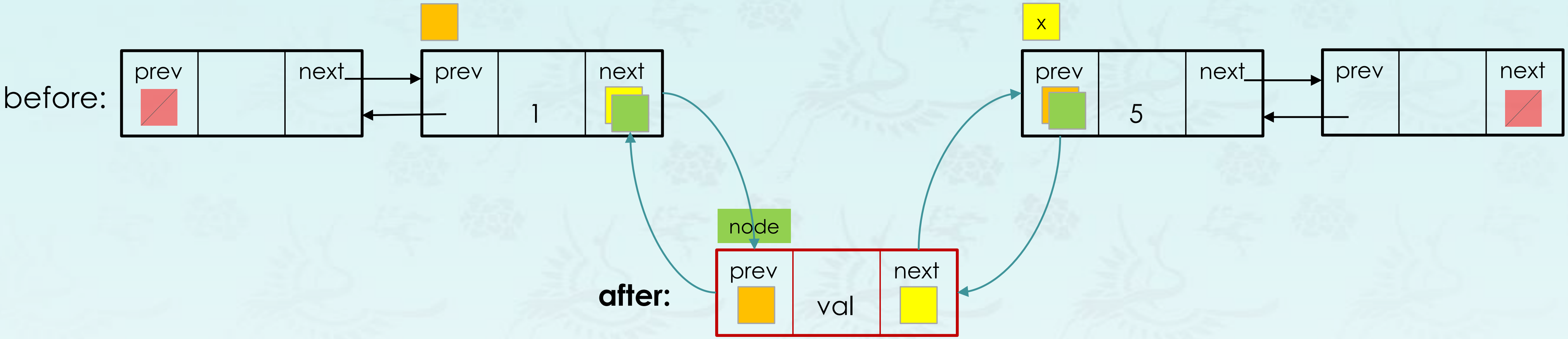
```
             x ==
x->prev->next ==
x->next->prev ==
      x->prev ==
      x->next ==
```

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

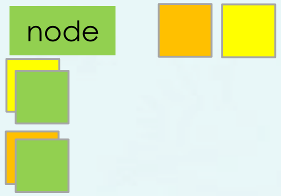
It should be coded using info in x only.

Revisit - insert()

Identify where are to be changed or set?

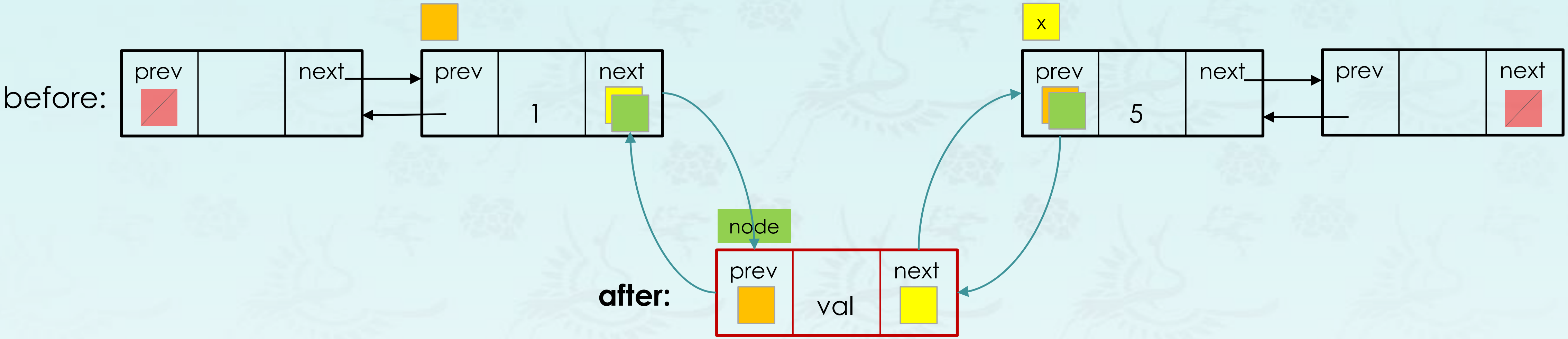



```
void insert(pNode x, int value) {

}
```

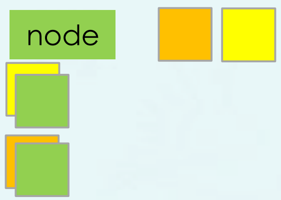


It should be coded using value and info in x only.

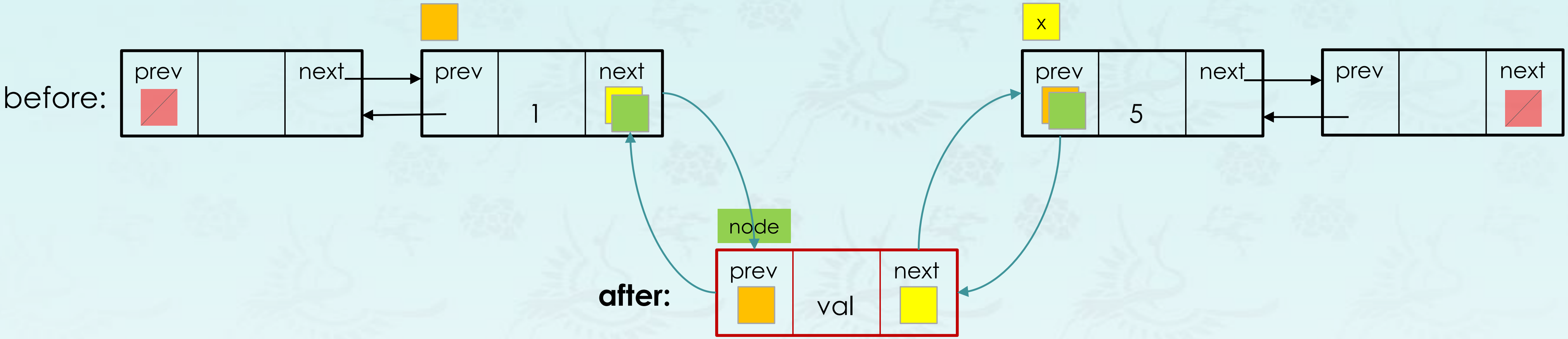
Revisit - insert()




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

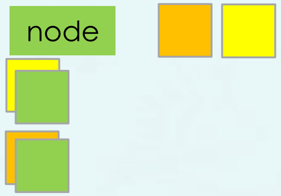


Revisit - insert()

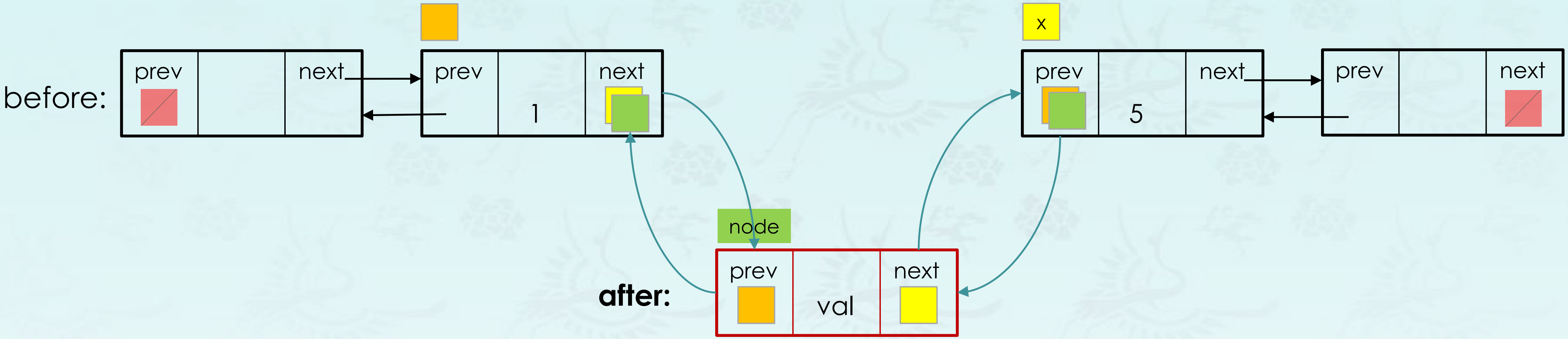



```
void insert(pNode x, int value) {
  pNode node = new Node{value, x->prev, x};
  x->prev->next = node;
  x->prev = node;
}
```

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



Revisit - insert()



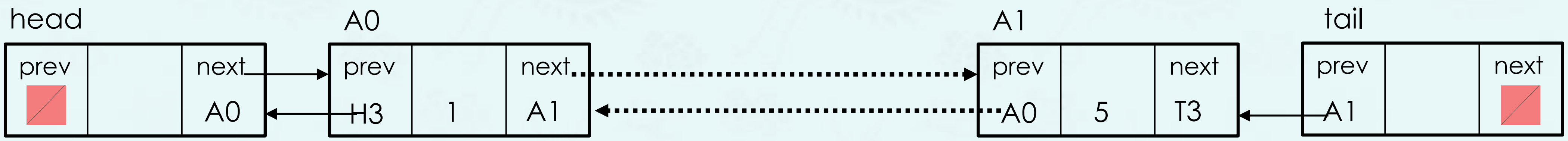

```
void insert(pNode x, int value) {
  pNode node = new Node{value, x->prev, x};
  x->prev->next = node;
  x->prev = node;
}
```

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

this is OK

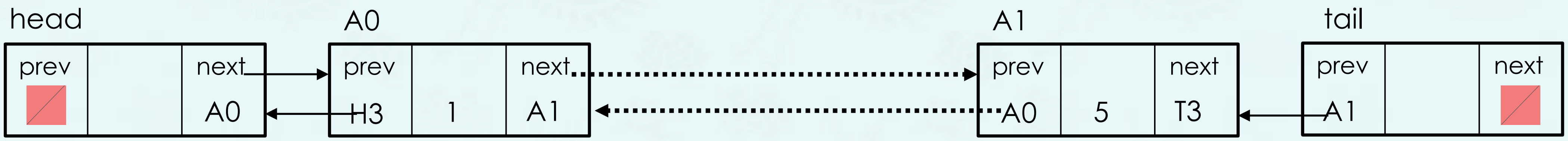
push_front()

```
// 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?

Is this good enough?

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

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).

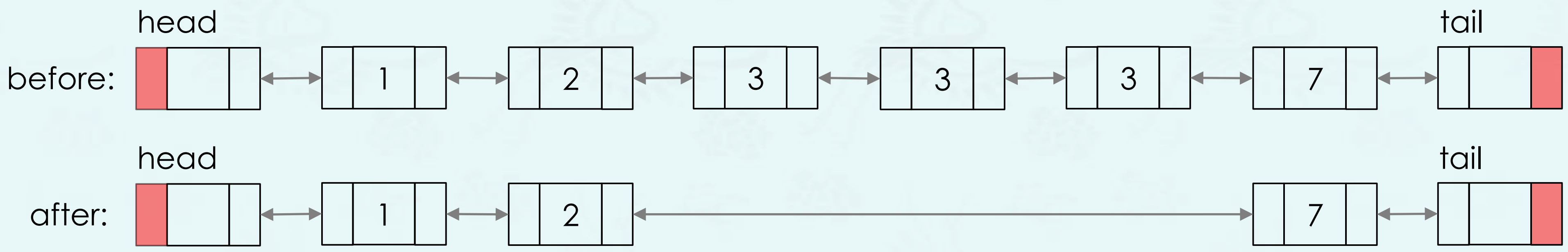
```
void pop_all(pList p, int value) { // value = 3 in this example
  while (find(p, value) != end(p)) {
    pop(p, value);
  }
} // version.1
```

O(n)

O(n)

O(n^2)

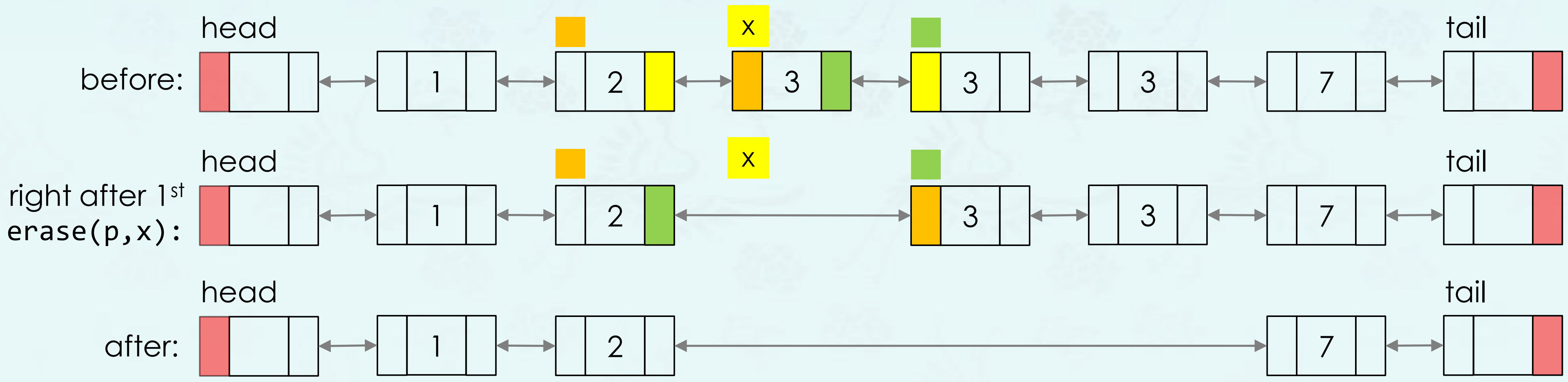
The code above works, then what is the problem?
What is the time complexity of each line and overall?



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
```

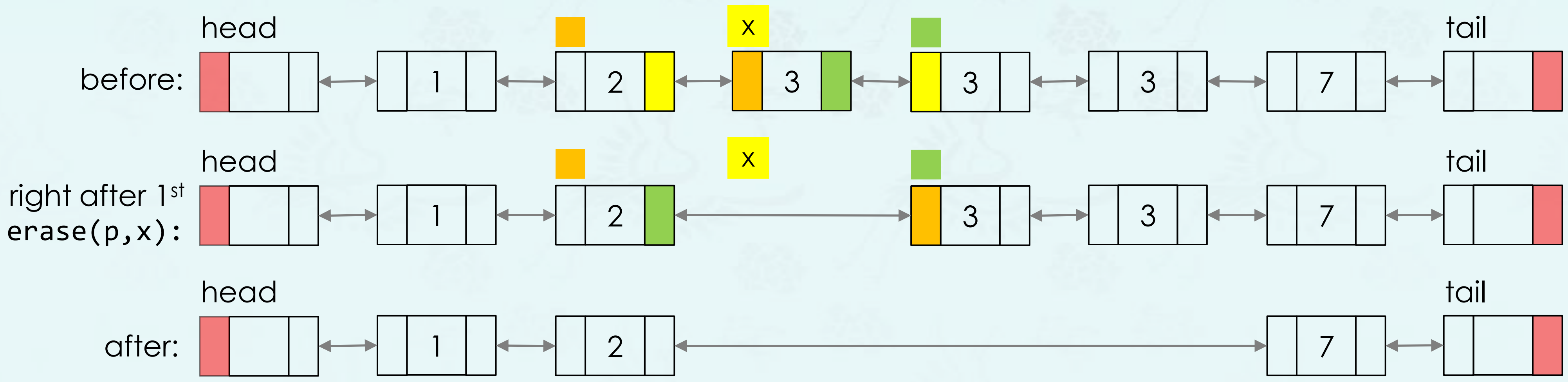


doubly linked list – **pop_all()***

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.

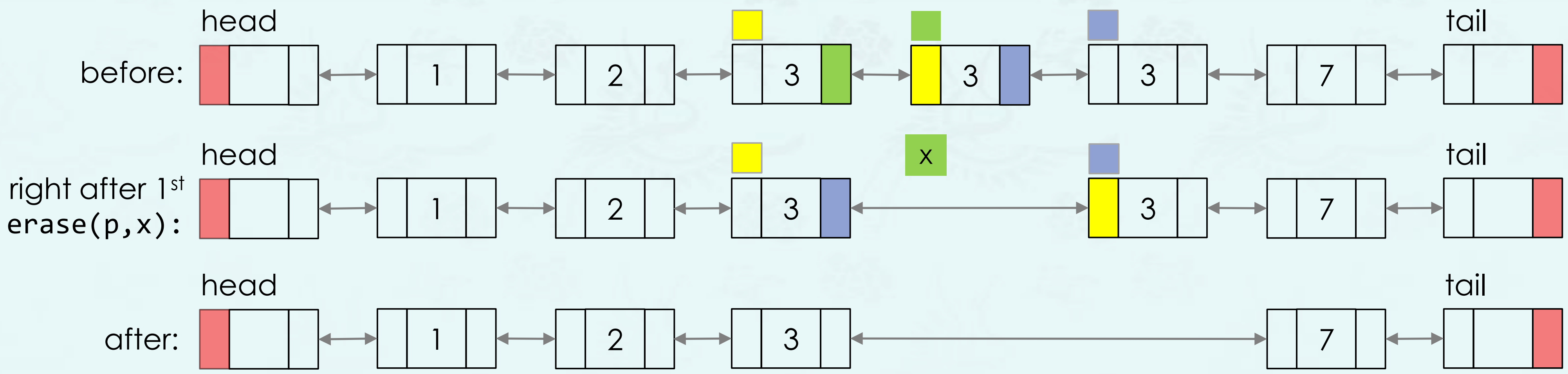


doubly linked list – **unique()***

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.



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;
}
```

- 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;
}
```

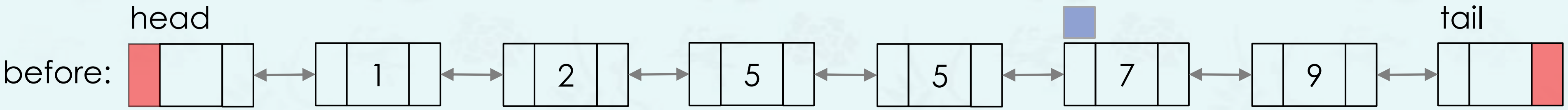
```
pNode find(pList p, int value){
  pNode x = begin(p);
  while(x != end(p)) {
    if (x->data == value) return x;
    x = x->next;
  }
  return x;
}
```

doubly linked list – **push_sorted()**

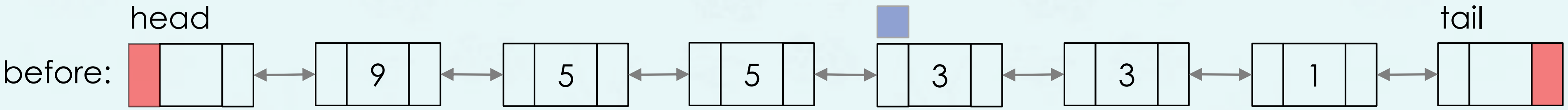
Write push_sorted function that inserts a node with value into a "sorted" list.

```
// 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);
}
```

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



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

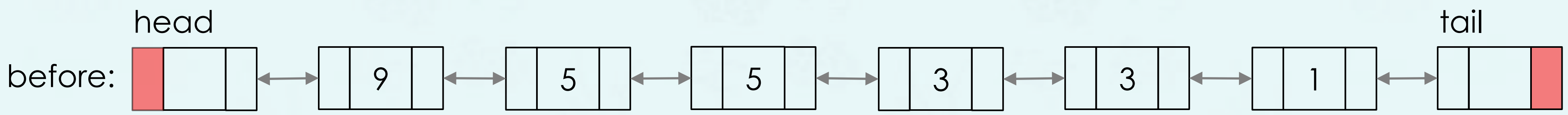


doubly linked list – **push_sortedNlog(pList p, int N)***

Write `push_sortedNlog` function that merges two sequences of numbers. One list already exists as a sorted linked list, the other will be created as an array of randomly generated numbers.
For example, if you have a list of 6 nodes and N = 5, then randomly generated numbers are in the range of `[0..(N + size(p))]` → `[0..(6+5)]` → `[0..11]`.

```
void push_sortedNlog(pList p, int N) {

  // your code here

}
```

```
int values[] = {15, 10, 5, 4, 2}
```



doubly linked list – **bubbleSort()**, **selectionSort()**, **insertionSort()** and **quickSort()**

```
void bubbleSort(pList p, int(*comp)(int, int)) {
  if (sorted(p)) return reverse(p);

  pNode x;
  for (pNode i = begin(p); i != end(p); i = i->next) {
    for (x = begin(p); x->next != end(p); x = x->next) {
      if (comp(x->data, x->next->data) > 0)
          swap(x->data, x->next->data);
    }
    tail = x;
  }
}
```

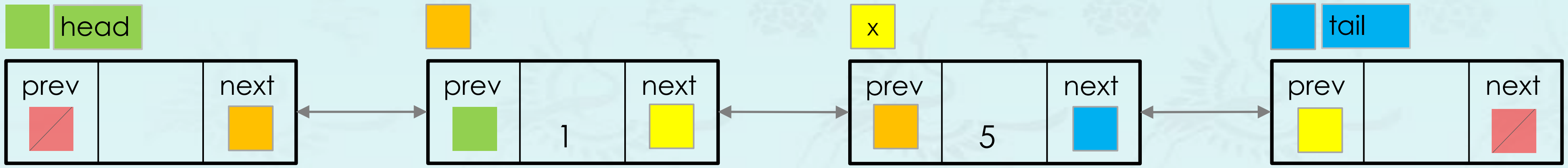
doubly linked list – **reverse()**

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).

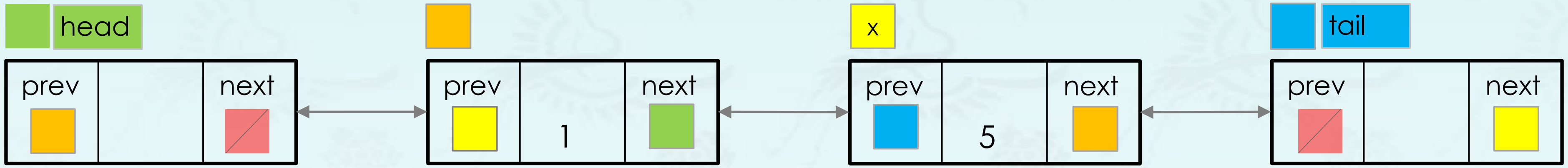
```
// reverses the order of the nodes in the list container. O(n)
void reverse(pList p) {
  if (size(p) <= 1) return;
  // hint: swap prev and next in every node including two sentinel nodes.
  // then, swap head and tail.
  // your code here

}
```

doubly linked list – **reverse()**

before:

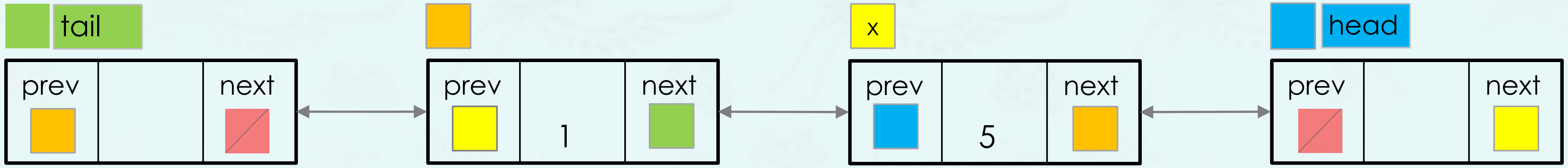


Step 1: swap prev and next in every node **including two sentinel nodes.**



Step 2: swap head and tail node.

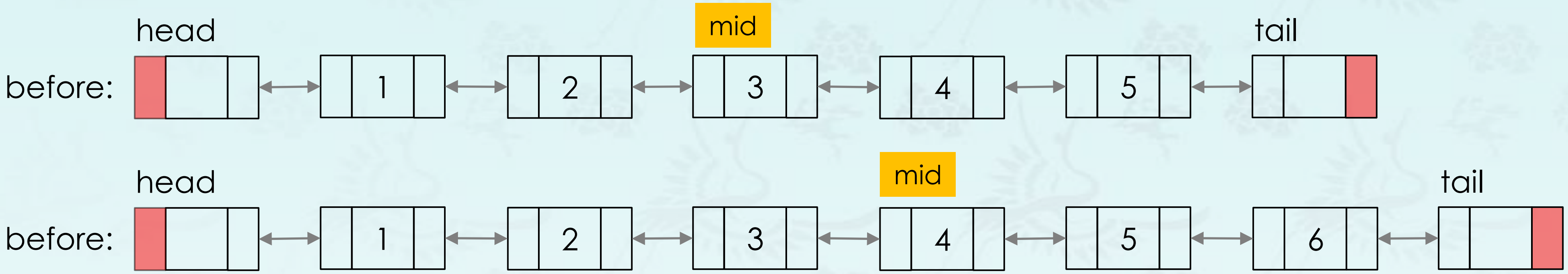
after:



doubly linked list – **half()**

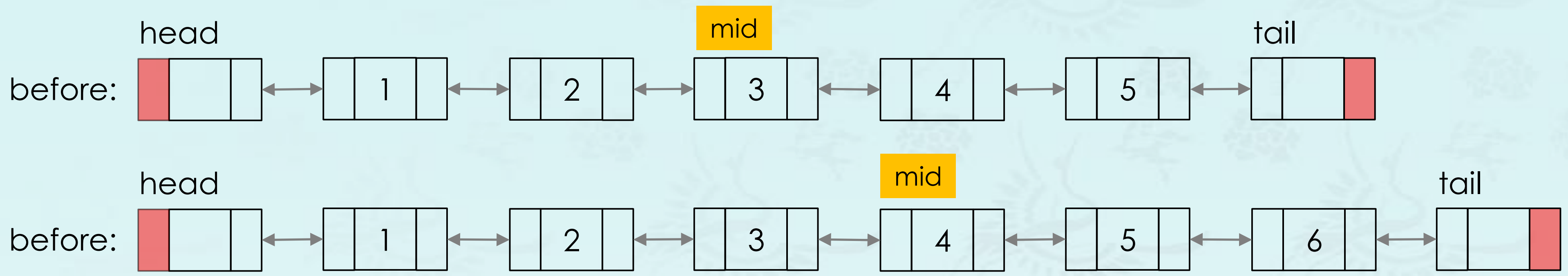
Write `half()` function that returns the mid node of the list.

- 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.

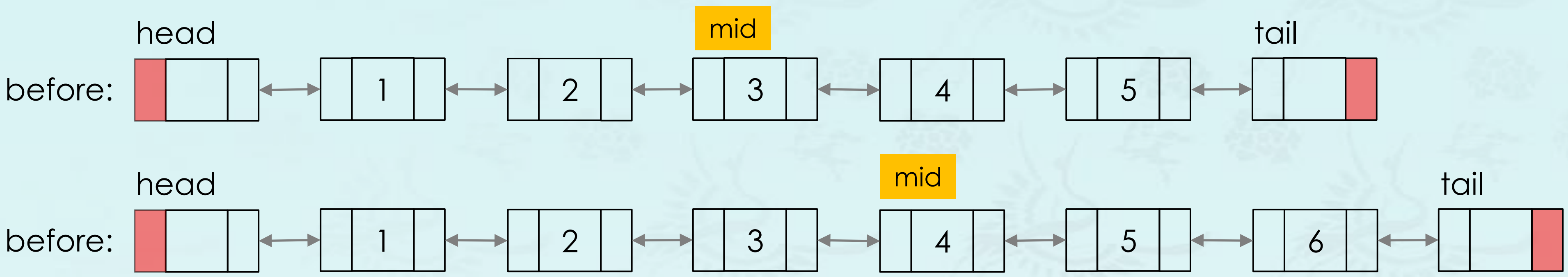
doubly linked list – **half()**



- **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()**



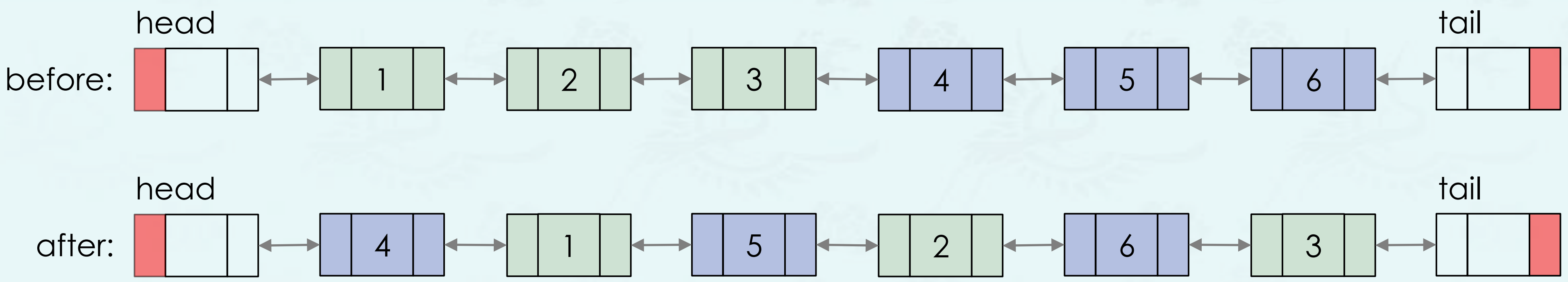
- **Method 2:** It works by sending rabbit and turtle down the list: turtle moving at speed one, and rabbit moving at speed two. As soon as the rabbit hits the end, you know that the turtle is at the halfway point as long as the rabbit gets asleep at the halfway. O(n)

```
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 – **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.

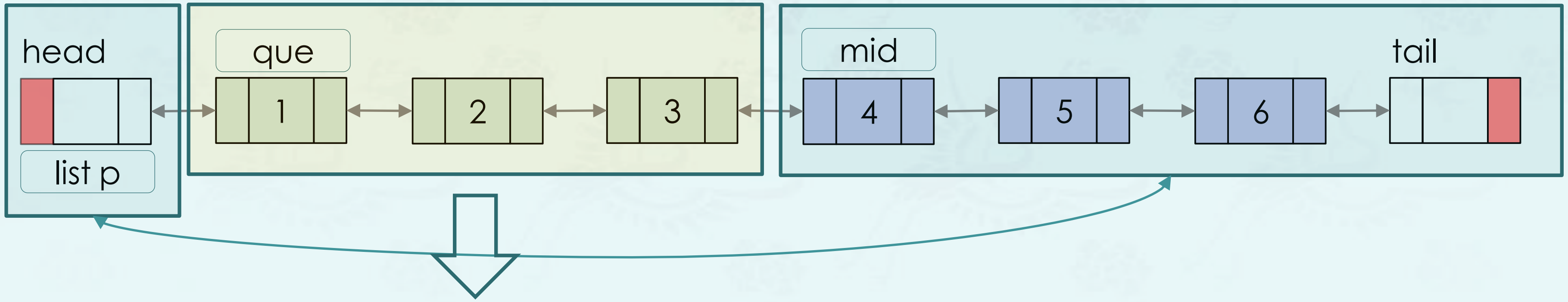


doubly linked list – **shuffle()****

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.

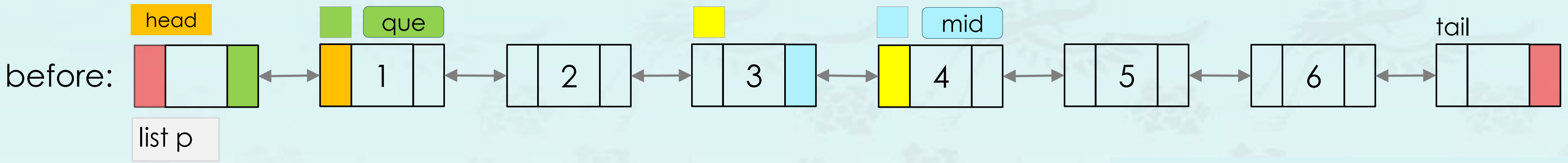
Algorithm:
1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.
4) keep on interleaving nodes until the "que" is exhausted.
   - save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



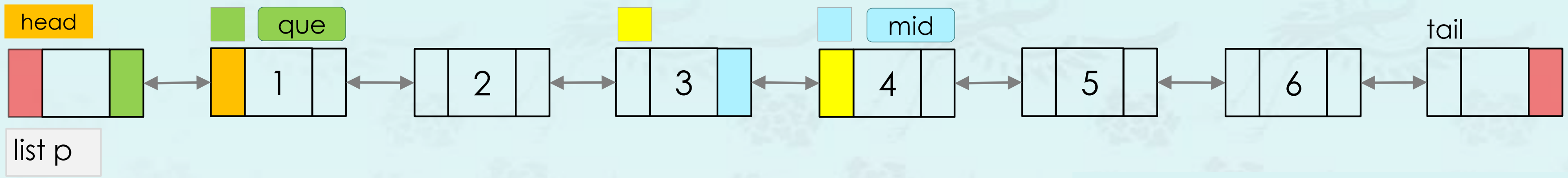
(1) Name functions to get **que** and **mid** nodes?

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

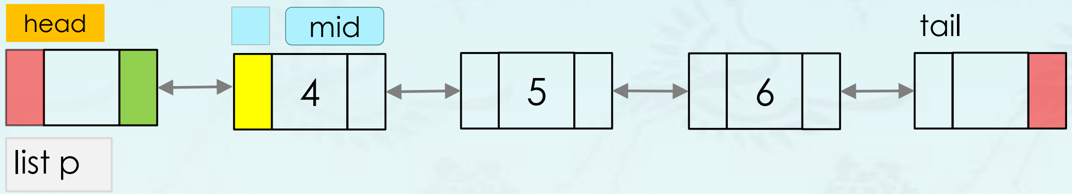
doubly linked list – **shuffle()****

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:



before:



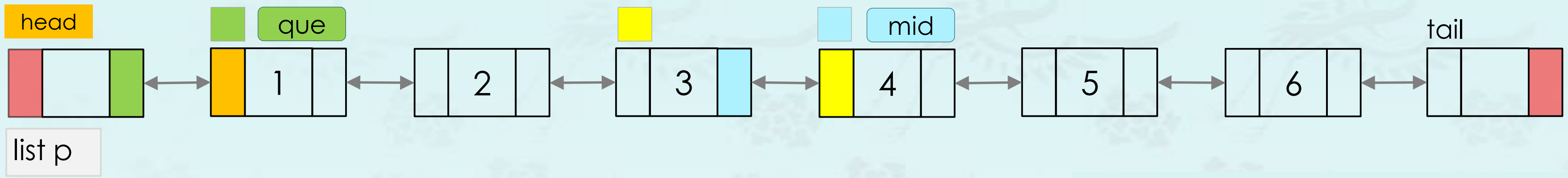
(1) Name functions to get **que** and **mid** nodes?
(2) Identify where the color should be updated.


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

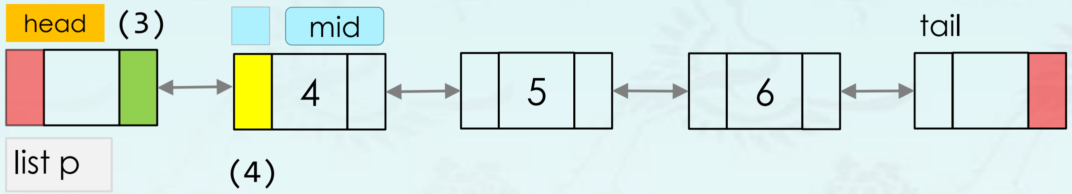
doubly linked list – **shuffle()****

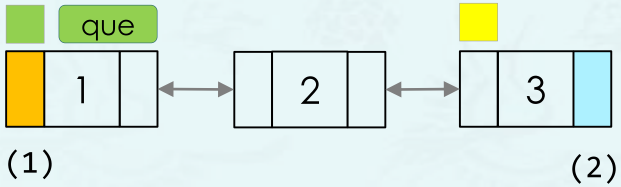
1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.
4) keep on interleaving nodes until the "que" is exhausted

before:



before:





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

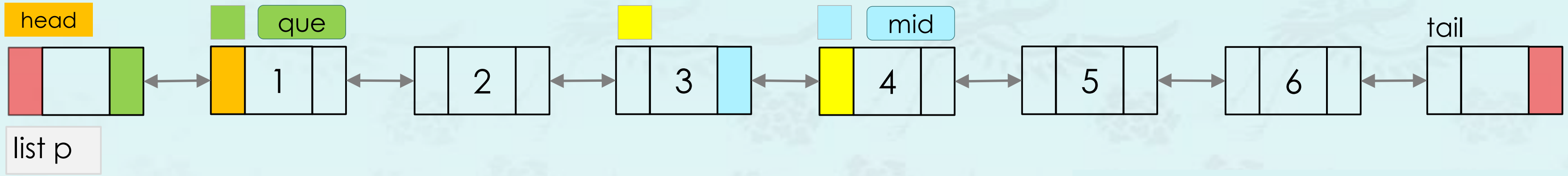
```
pNode mid = half(p);
pNode que = begin(p);
```

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

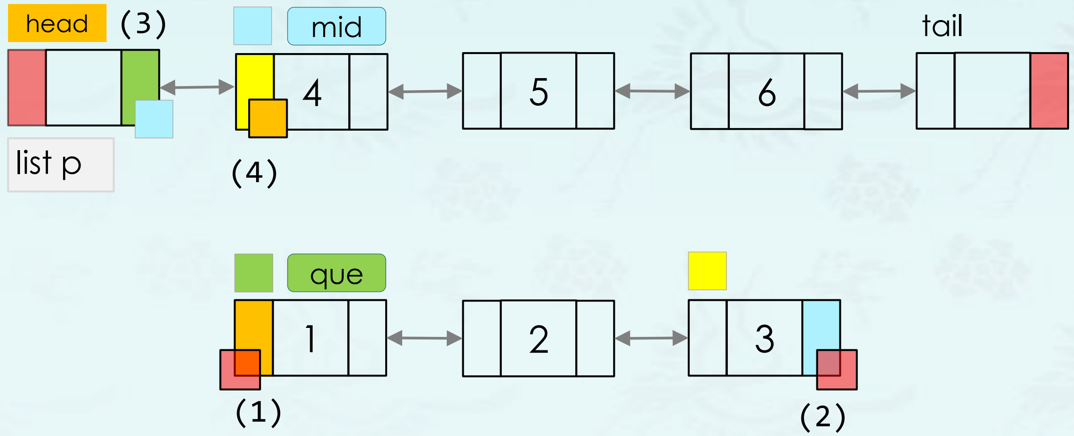
doubly linked list – **shuffle()****

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:



before:



(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?

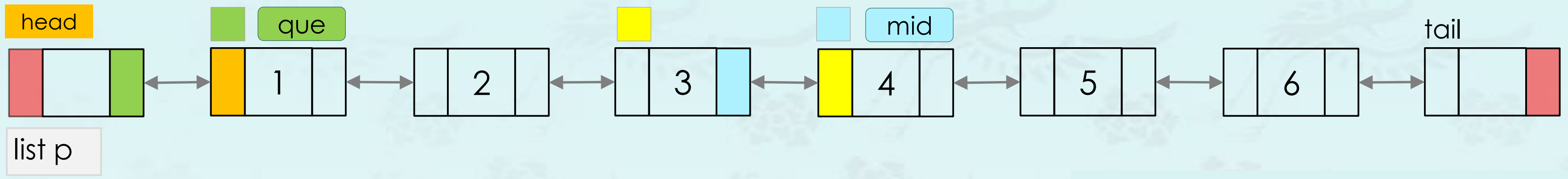
```
pNode mid = half(p);
pNode que = begin(p);
```

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

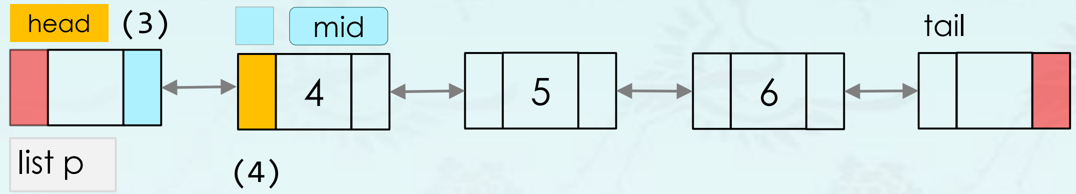
doubly linked list – **shuffle()****

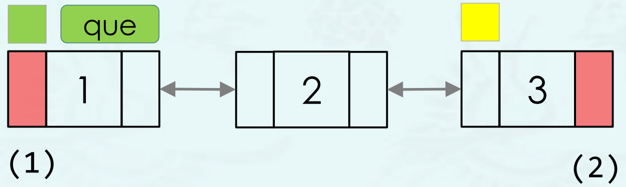
1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.
4) keep on interleaving nodes until the "que" is exhausted

before:



before:





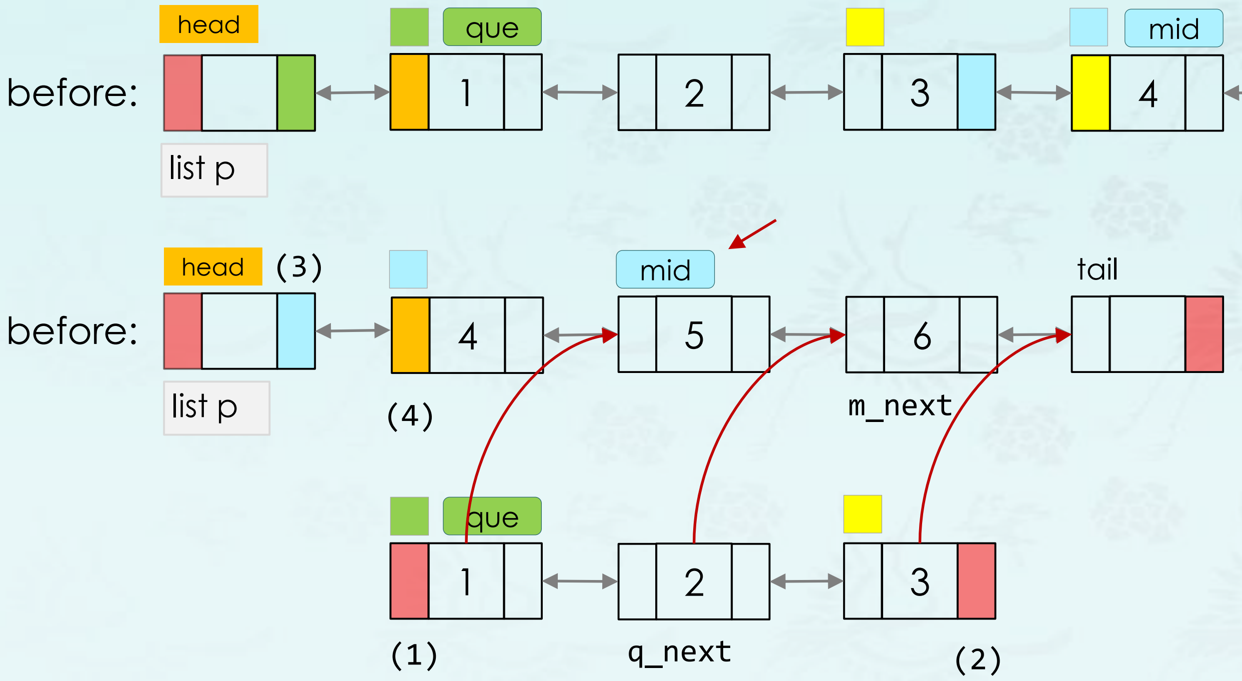
(1) Name functions to get **que** and **mid** nodes?
(2) Identify where the color should be updated. How about them in code?
(3) What color code must be there? How about them in code?

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

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

doubly linked list – **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.




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

   // que is inserted at mid.
   ...

   mid = m_next;
   que = q_next;
}
```

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 ? ? ? ?