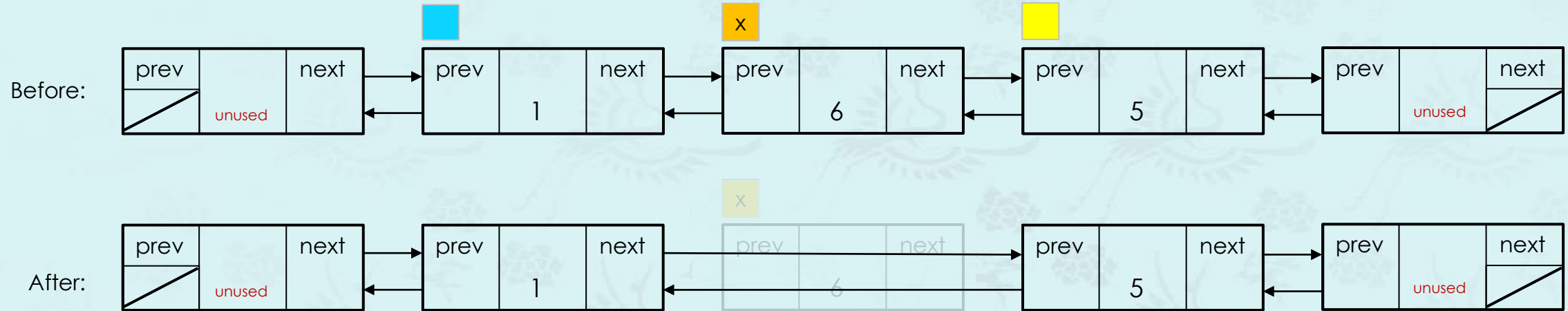


## Data Structures

### 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

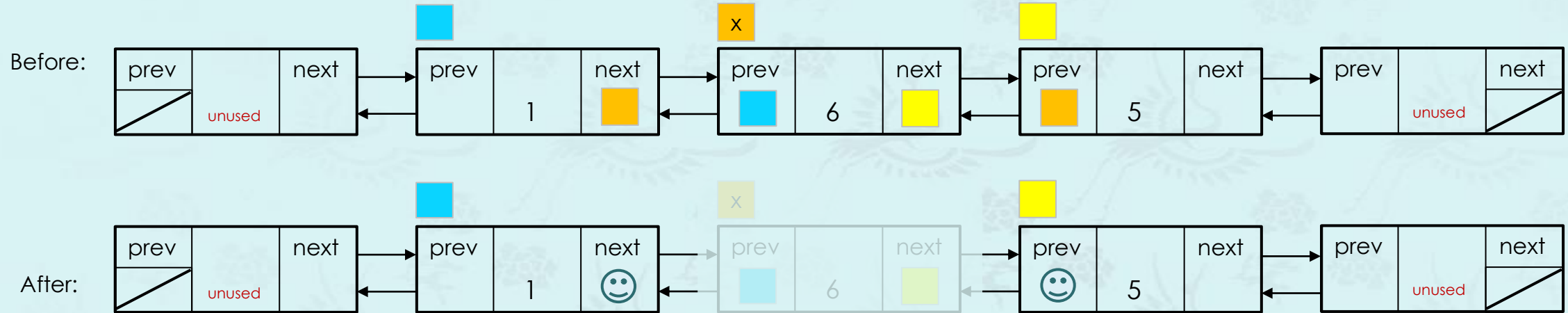
# Key Operations: Erase the node x



recall x alone is known  
recall x is not pList, but pNode

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

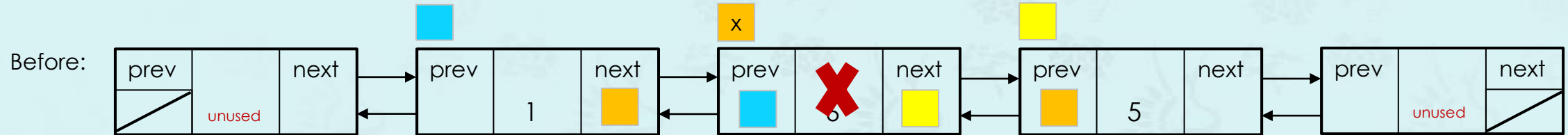
# Key Operations: Erase the node x



recall x alone is known  
recall x is not plist, but pNode

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

# Key Operations: Erase the node x



After:

Before:

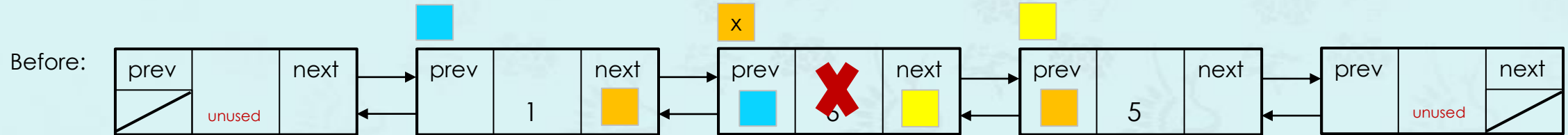
Express each color code in terms of x.	x ==	x
	==	
	==	
	==	
	==	

```
void erase(pNode x){
}

```






recall x alone is known  
recall x is not pList, but pNode

# Key Operations: Erase the node x



After:

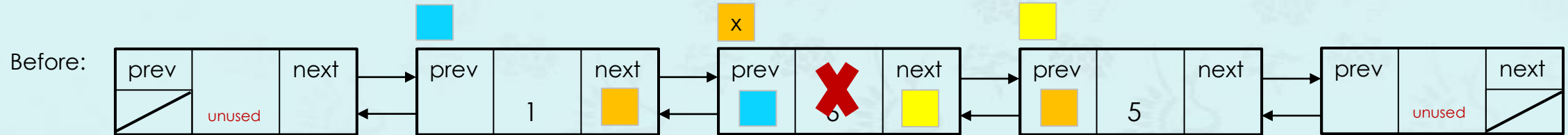
Before:

<code>x ==</code>	
<code>x-&gt;prev-&gt;next ==</code>	
<code>x-&gt;next-&gt;prev ==</code>	
	
	

```
void erase(pNode x){
    // recall x alone is known
    // recall x is not pList, but pNode
}






```

# Key Operations: Erase the node x



After:

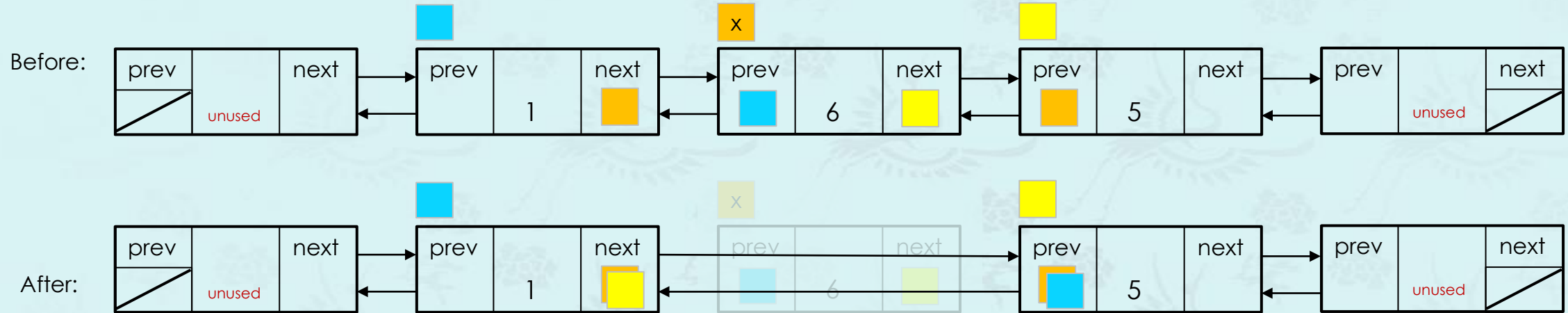
Before:

<code>x ==</code>	
<code>x-&gt;prev-&gt;next ==</code>	
<code>x-&gt;next-&gt;prev ==</code>	
<code>x-&gt;prev ==</code>	
<code>x-&gt;next ==</code>	

```
void erase(pNode x){
    // recall x alone is known
    // recall x is not pList, but pNode
}

```

# Key Operations: Erase the node x



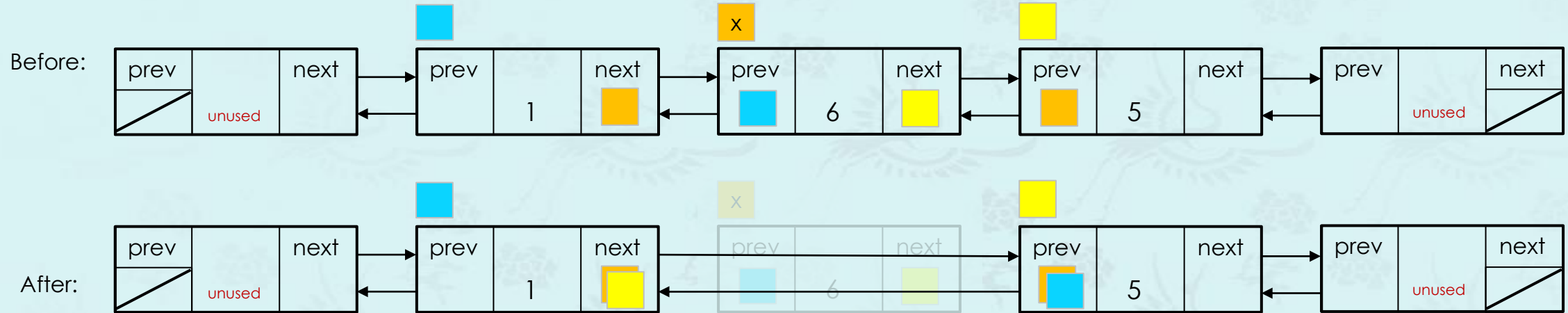
Before:

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






```
void erase(pNode x){
}

```


# Key Operations: Erase the node x



Before:

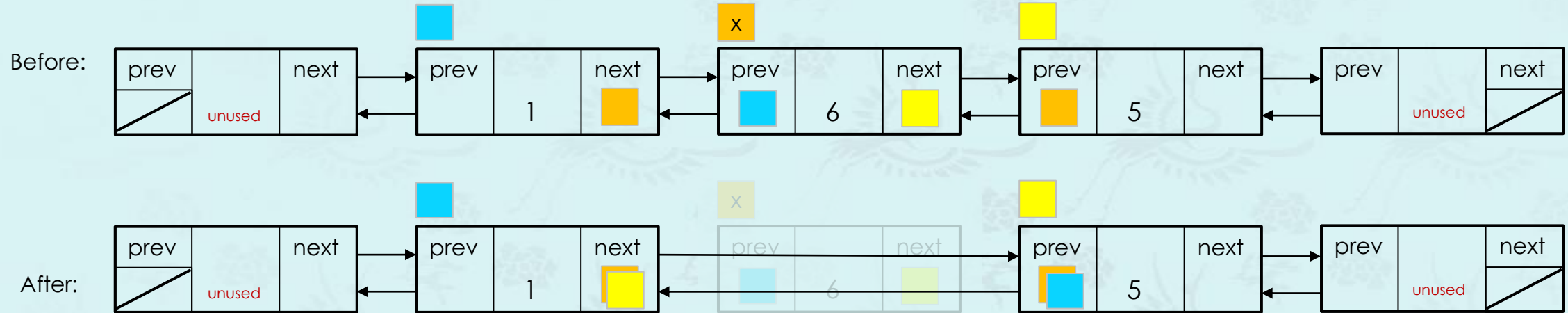
<code>x ==</code>	
<code>x-&gt;prev-&gt;next ==</code>	
<code>x-&gt;next-&gt;prev ==</code>	
<code>x-&gt;prev ==</code>	
<code>x-&gt;next ==</code>	

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





# Key Operations: Erase the node x



Before:

<code>x ==</code>	
<code>x-&gt;prev-&gt;next ==</code>	
<code>x-&gt;next-&gt;prev ==</code>	
<code>x-&gt;prev ==</code>	
<code>x-&gt;next ==</code>	

```
void erase(pNode x){
    x->prev->next = x->next;
    x->next->prev = x->prev;
    delete x;
} //stay tuned for the better
```

# Pop by value

- Implement pop() using erase() and find().

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

```
void erase(pNode x){  
    x->prev->next = x->next;  
    x->next->prev = x->prev;  
    delete x;  
} //stay tuned for the better
```

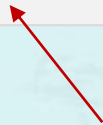
```
pNode find(pList p, int value)
```

# Pop by value

- Implement pop() using erase() and find().

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

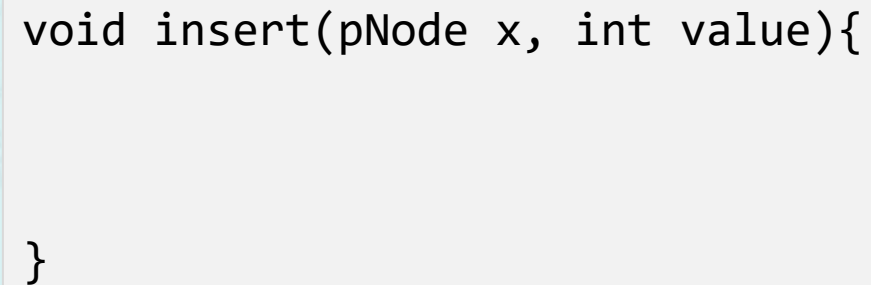
```
void pop(pList p, int value){  
    erase(find(p, value));  
} //stay tuned
```



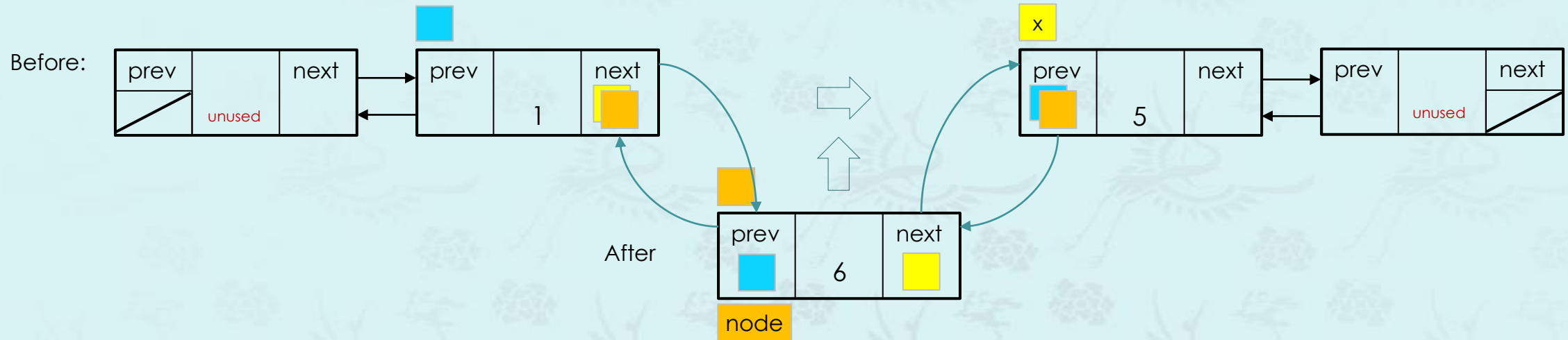
This code may not work some cases?  
How can you fix it?

```
void erase(pNode x){  
    x->prev->next = x->next;  
    x->next->prev = x->prev;  
    delete x;  
} //stay tuned for the better
```

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



## Key Operations: Insert a new node(value) in place of the **node x**



Before:

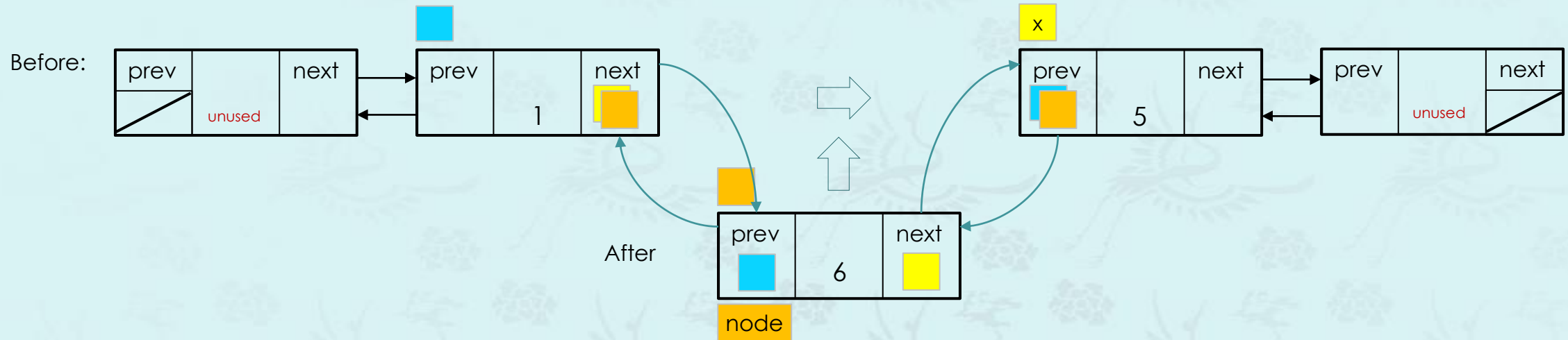
```
x == x
x->prev->next == 
x->prev == 
```

Express each color code  
in terms of x.

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

}
```

# Key Operations: Insert a new node(value) in place of the **node x**



Before:

```

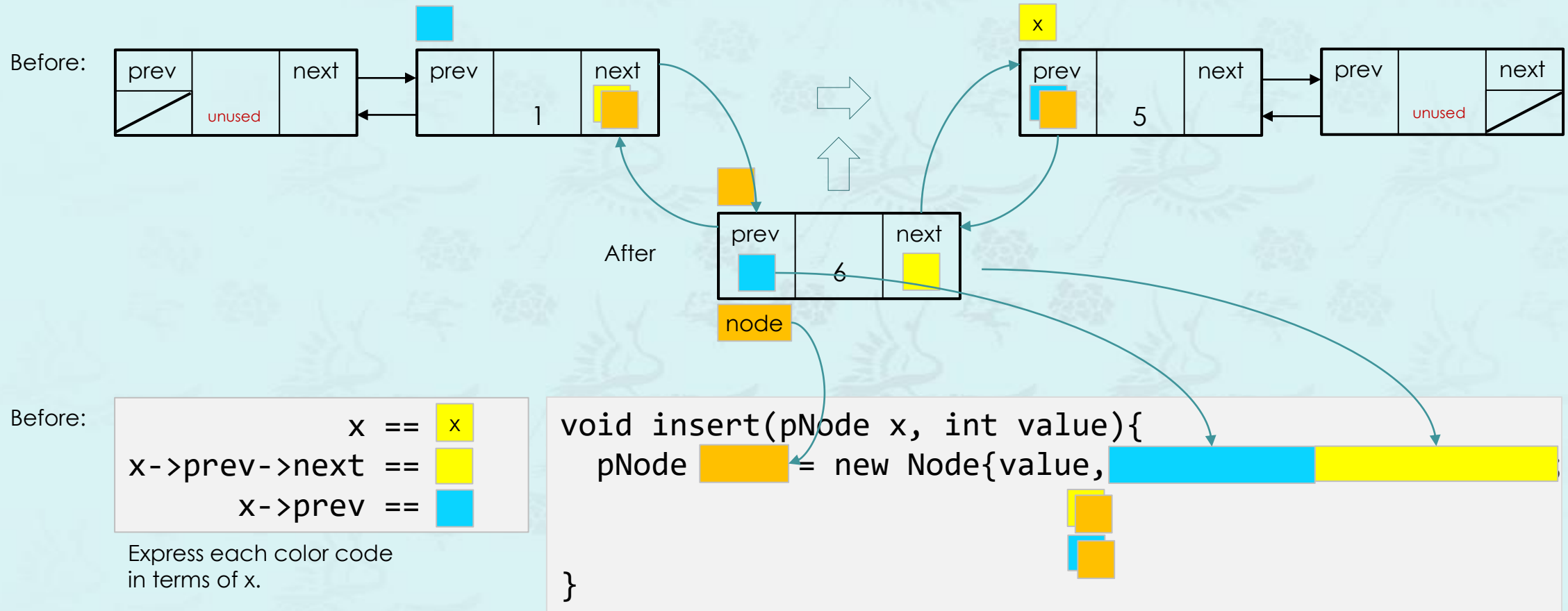
x == x
x->prev->next == 
x->prev == 
    
```

Express each color code  
in terms of x.

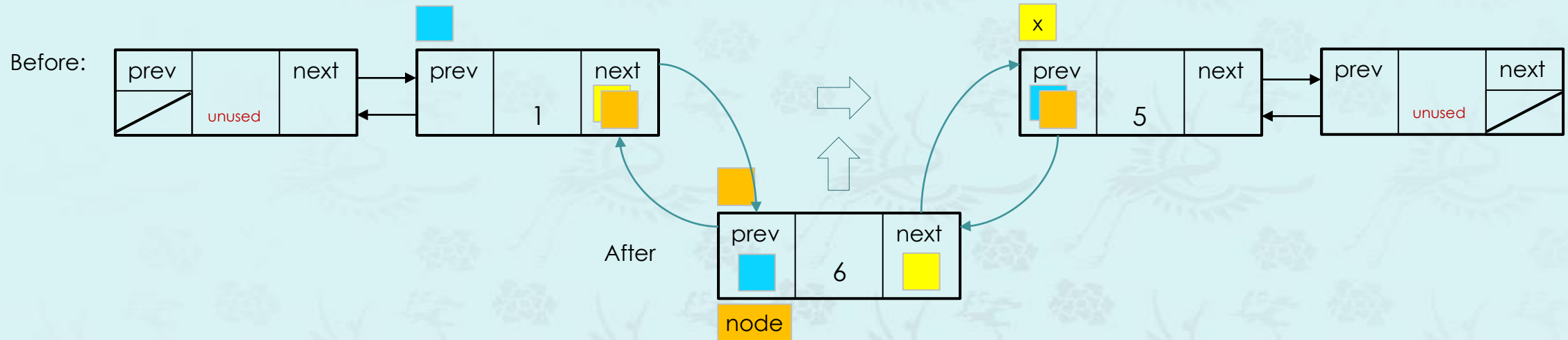
```

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

# Key Operations: Insert a new node(value) in place of the **node x**



# Key Operations: Insert a new node(value) in place of the **node x**



Before:

```

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

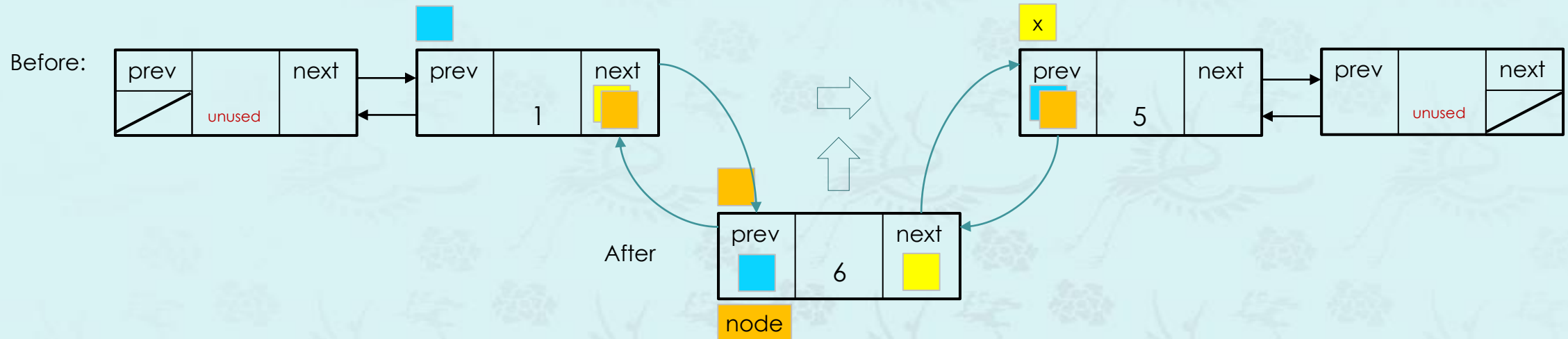
Express each color code  
in terms of x.

```

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



# Key Operations: Insert a new node(value) in place of the **node x**



Before:

```

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

Express each color code in terms of x.

```

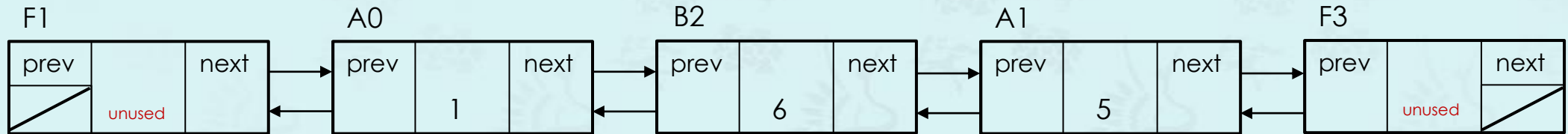
void insert(pNode x, int value){
    pNode node = new Node{value, x->prev, x};  x
    x->prev->next = node;
    x->prev = ;
}
    
```

Can we replace two lines above with \_\_\_\_\_

It is a matter of associativity!  
Then, what is the associativity of '='?  
**right to left**

- (1) x->prev = x->prev->next = node;
- (2) x->prev->next = x->prev = node;
- (3) either one

# push\_front(), push\_back(), push()



Case 1: in front

```
void push_front(pList p, int value){  
    insert(begin(p) , value);  
}
```

Case 3: at end

```
void push_back(pList p, int value){  
    insert(end(p) , value);  
}
```

Case 2: in middle

```
void push(pList p, int value, int z){  
    insert(find(p, z), value);  
}
```

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

pNode begin(pList p);	// returns the first node, not sentinel node
pNode end(pList p);	// returns the ending sentinel node
pNode half(pList p)	// returns the node in the middle of the list
pNode find(pList p, int value);	// returns the first node with value
void clear(pList p);	// free list of nodes
bool empty(pList p);	// true if empty, false if no empty
int size(pList p);	// returns size in the list
<b>void insert(pNode x, int value);</b>	// inserts a new node with value at the node x
<b>void erase (pNode x);</b>	// deletes a node and returns the previous node
	stay tuned for enhancement
void push(pList p, int value, int z);	// inserts a node with value at the node with x
void push_front(pList p, int value);	// inserts a node at front of the list
void push_back(pList p, int value);	// inserts a node with value at end of the list
void push_sorted(pList p, int value, bool ascending = true);	// inserts a node in sorted
void pop(pList p, int value);	// deletes the first node with value
void pop_front(pList p);	// deletes the first node in the list
void pop_back(pList p);	// deletes the last node in the list, O(1)
void pop_backN(pList p);	// deletes all the nodes O(n)
void pop_all(pList p, int value);	// deletes all the nodes with value
<b>pList</b> sort(pList p);	// returns a `new list` sorted
bool sorted(pList p);	// returns true if the list is sorted
void unique(pList p);	// returns list with no duplicates, sorted
void reverse(pList p);	// reverses the sequence
void shuffle(pList p);	// shuffles the list
void show(pList p);	// shows all data items in linked list

# 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 ? ?  
→