3/4

Linked List

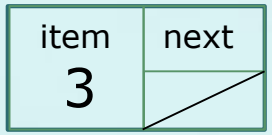
Data Structures
C++ for C Coders

한동대학교 김영섭 교수
idebtor@gmail.com

Why doubly linked list? – An introduction

a new node instantiation

A0

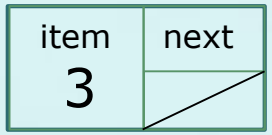



```
struct Node {
  int    item;
  Node*  prev;
  Node*  next;
};

struct List {
  Node*  head;
  Node*  tail;
};
using pNode = Node*;
using pList = List*;
```

unused in singly linked

a new node instantiation

A0



(1)
```
pNode n = new Node;
Node* n = new Node;
```

(2)
```
Node* n = new Node();
```

(3)
```
Node* n = new Node{};
```

(4)
```
Node* n = new Node(4);
```

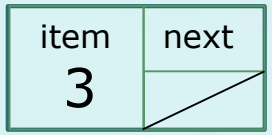
(5)
```
Node* n = new Node{5};
```

```
struct Node {
  int    item;
  Node*  prev;
  Node*  next;
};

struct List {
  Node*  head;
  Node*  tail;
};
using pNode = Node*;
using pList = List*;
```

unused in singly linked

a new node instantiation

A0



(1)
```
pNode n = new Node;
Node* n = new Node;
```
← no values set

(2)
```
Node* n = new Node();
```
← set to 0 or null

(3)
```
Node* n = new Node{};
```
← set to 0 or null

(4)
```
Node* n = new Node(4);
```
← Compiler error

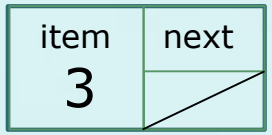
(5)
```
Node* n = new Node{5};
```
← set to 0 or null

```
struct Node {
  int    item;
  Node*  prev;
  Node*  next;
};

struct List {
  Node*  head;
  Node*  tail;
};
using pNode = Node*;
using pList = List*;
```

unused in singly linked

a new node instantiation

A0



(1)
```
pNode n = new Node(3);
Node* n = new Node(3);
```

{2}
```
Node* n = new Node{3};
```

{3}
```
Node* n = new Node{3, nullptr};
```

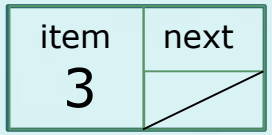
{4}
```
Node* n = new Node{3, nullptr, nullptr};
```

```
struct Node {
  int    item;
  Node*  prev;
  Node*  next;
};

struct List {
  Node*  head;
  Node*  tail;
};
using pNode = Node*;
using pList = List*;
```

unused in singly linked

a new node instantiation

A0

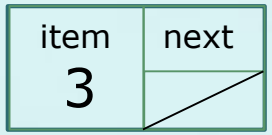



```
struct Node {
  int    item;
  Node*  prev;
  Node*  next;
};

struct List {
  Node*  head;
  Node*  tail;
};
using pNode = Node*;
using pList = List*;
```

```
struct Node{
  int item;
  Node* prev;
  Node* next;
  // constructor
  Node(int d=0, Node* p=nullptr, Node* n=nullptr) {
    item = d;    prev = p;        next = n;
  }
  // destructor
  ~Node() {}
};
```

a new node instantiation

A0




```
struct Node {
  int    item;
  Node*  prev;
  Node*  next;
};

struct List {
  Node*  head;
  Node*  tail;
};
using pNode = Node*;
using pList = List*;
```

```
pNode n = new Node(3);
```

← It is OK, now.

```
Node* n = new Node(3);
```

```
pNode n = new Node{3, nullptr, nullptr};
```

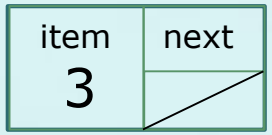
```
Node* n = new Node{3, nullptr, nullptr};
```

```
struct Node{
  int item;
  Node* prev;
  Node* next;
  // constructor
  Node(int d=0, Node* p=nullptr, Node* n=nullptr) {
    item = d;    prev = p;        next = n;
  }
  // destructor
  ~Node() {}
};
```

a new node instantiation

What is the meaning of mnemonic A0?

A0




```
struct Node {
  int    item;
  Node*  prev;
  Node*  next;
};

struct List {
  Node*  head;
  Node*  tail;
};
using pNode = Node*;
using pList = List*;
```

linking two nodes

Task: Link two nodes and set the first node as `head`.


```
pNode head = new Node{3};
pNode node = new Node{5};
```

```
struct Node {
  int    item;
  Node*  prev;
  Node*  next;
};

struct List {
  Node*  head;
  Node*  tail;
};
using pNode = Node*;
using pList = List*;
```

linking two nodes

Task: Link two nodes and set the first node as `head`.


```
pNode head = new Node{3};
pNode node = new Node{5};
head->next = node;
```

```
struct Node {
  int    item;
  Node*  prev;
  Node*  next;
};

struct List {
  Node*  head;
  Node*  tail;
};
using pNode = Node*;
using pList = List*;
```

linking two nodes

Task: Link two nodes and set the first node as `head`.


```
pNode head = new Node{3};
pNode node = new Node{5};
head->next = node;

pList list = new List{head, node};
```

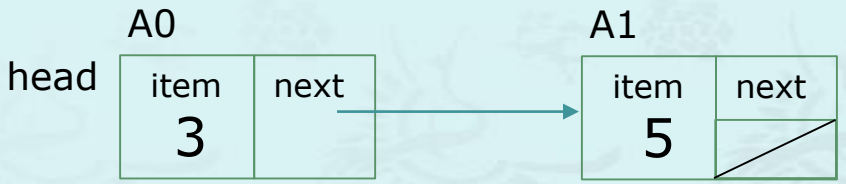
```
struct Node {
  int    item;
  Node*  prev;
  Node*  next;
};

struct List {
  Node*  head;
  Node*  tail;
};
using pNode = Node*;
using pList = List*;
```

Node structure initialization

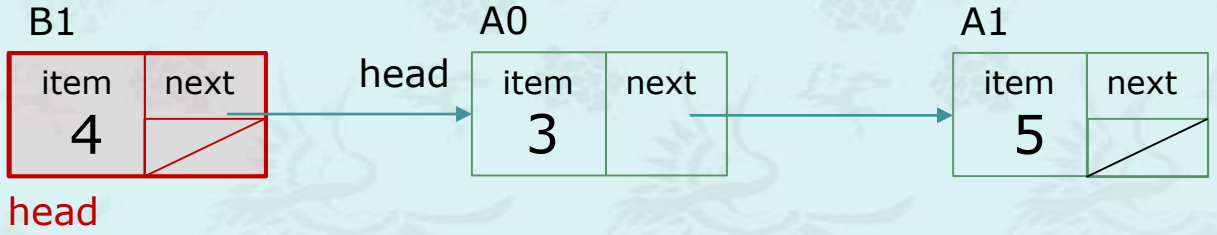
```
struct Node {
  int     item;
  Node*   prev;
  Node*   next;
  Node(const int d = 0, Node* p = nullptr, Node* n = nullptr) {
       item = d; prev = p; next = n;
  }
  ~Node() {}
};
using pNode = Node*;
```

Push a node: Three different cases

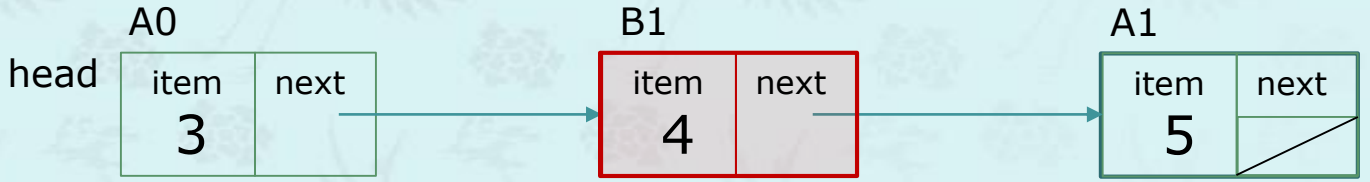
Given: an item(4) to insert – What was the most difficult part of this coding?



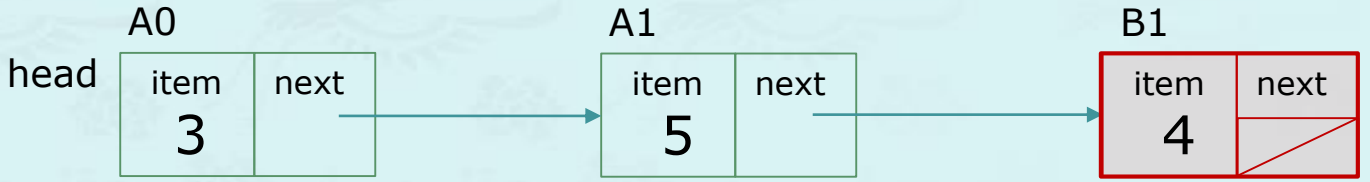
Case 1: in front



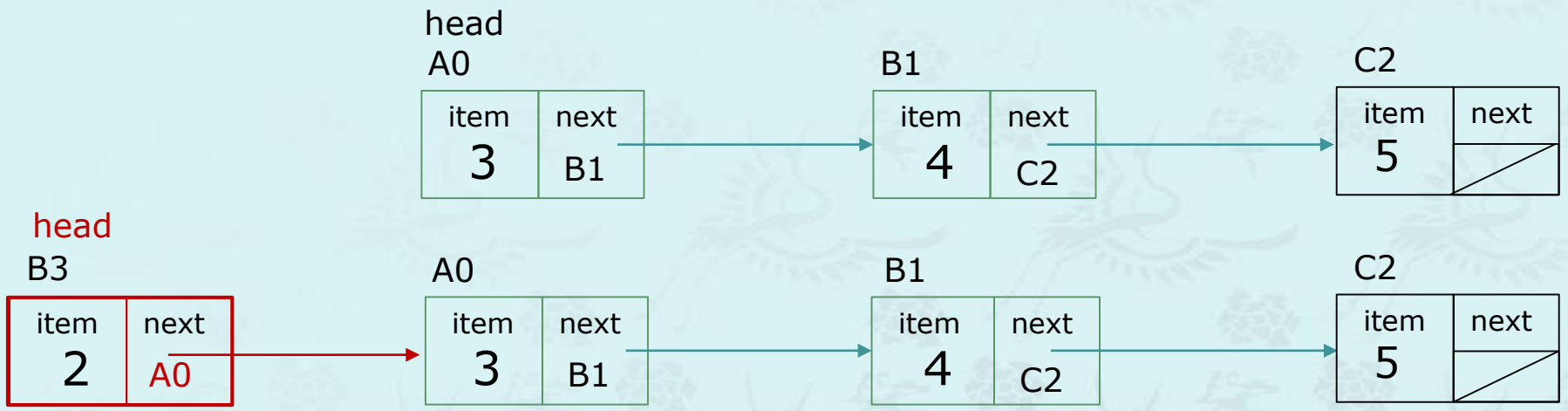
Case 2: in middle



Case 3: at end



push a node – Case 1; insert in front, head given

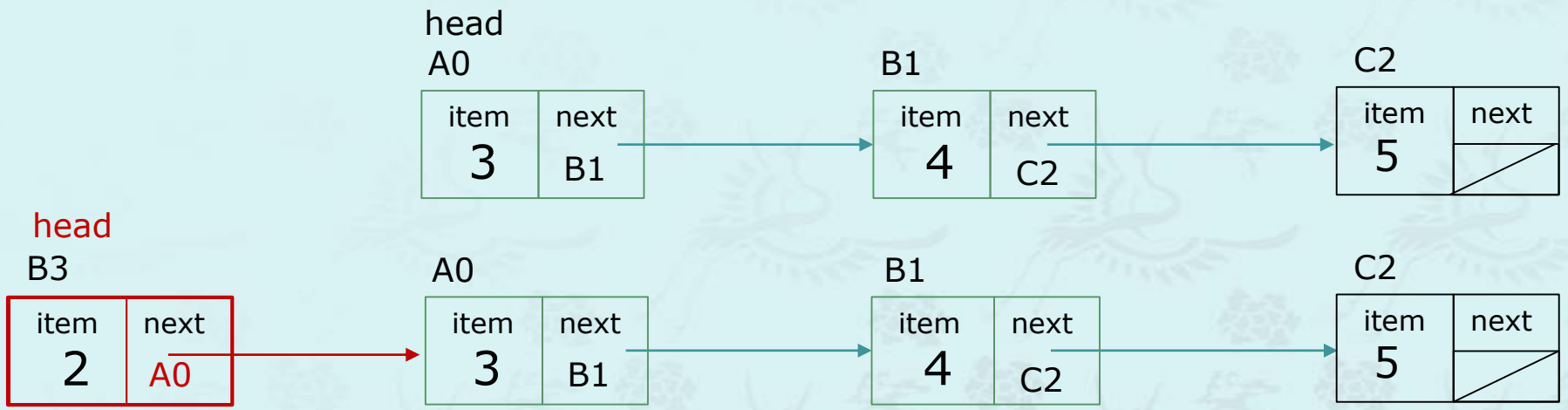



```
pNode node = new Node{2};
node->next = head;
head = node;
```

```
pNode node = new Node{2, nullptr, head};
head = node;
```

```
struct Node{
  int item;
  pNode prev;
  pNode next;
  // constructor
  Node(const int d=0, pNode p=nullptr, pNode n=nullptr){
    item = d;       prev = p; next = n;
  }
  // destructor
  ~Node() {}
} Node;
```

push a node – Case 1; insert in front, head given




```
pNode node = new Node{2};
node->next = head;
head = node;
```

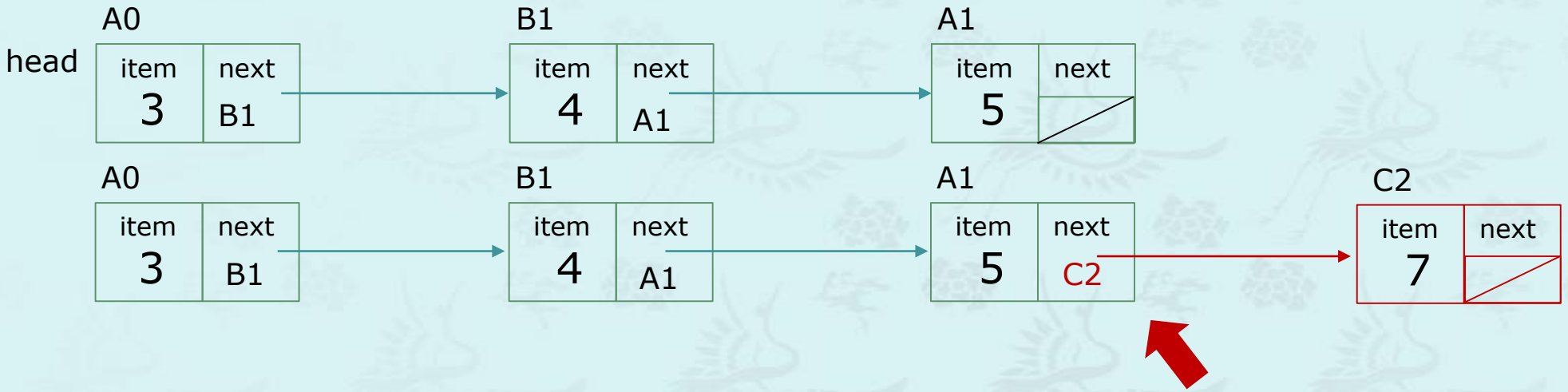
```
pNode node = new Node{2, nullptr, head};
head = node;
```

```
void push_front(pList p, int val) {

}
```

Complete push_front()

```
struct Node {
  int    item;
  Node*  prev;
  Node*  next;
};

struct List {
  Node*  head;
  Node*  tail;
};
using pNode = Node*;
using pList = List*;
```

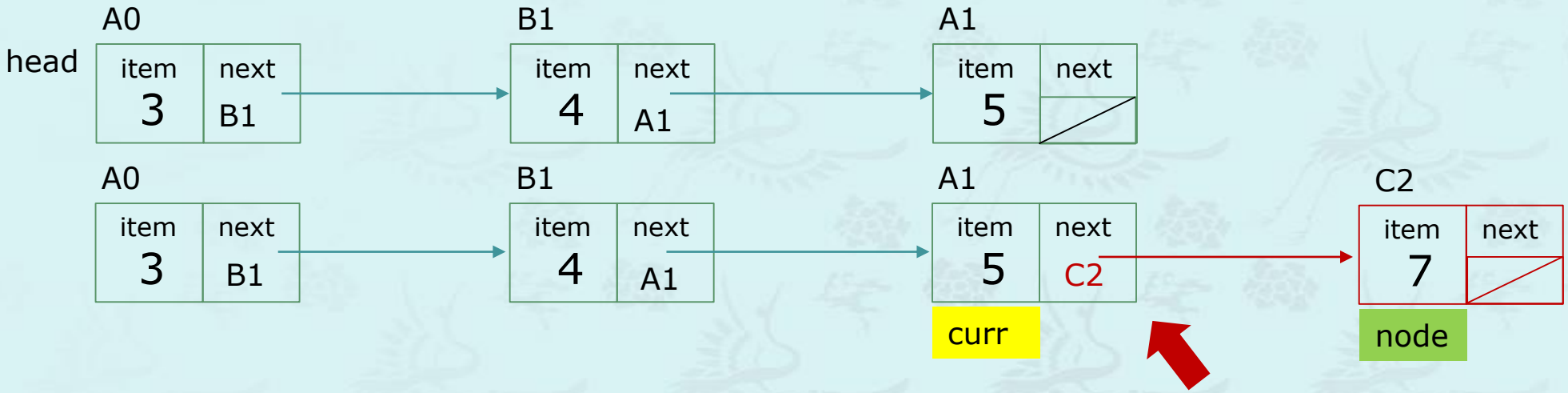
push a node – Case 3; insert at end, head given



- find the last node.
- append a new Node{7}.

```
pNode last(pNode list)
  pNode x = list;   ← need this?
  while
    x = x->next;
  return x;
```

push a node – Case 3; insert at end, head given




```
curr = last(list->head);
pNode node = new Node{7};
curr->next = node;
```

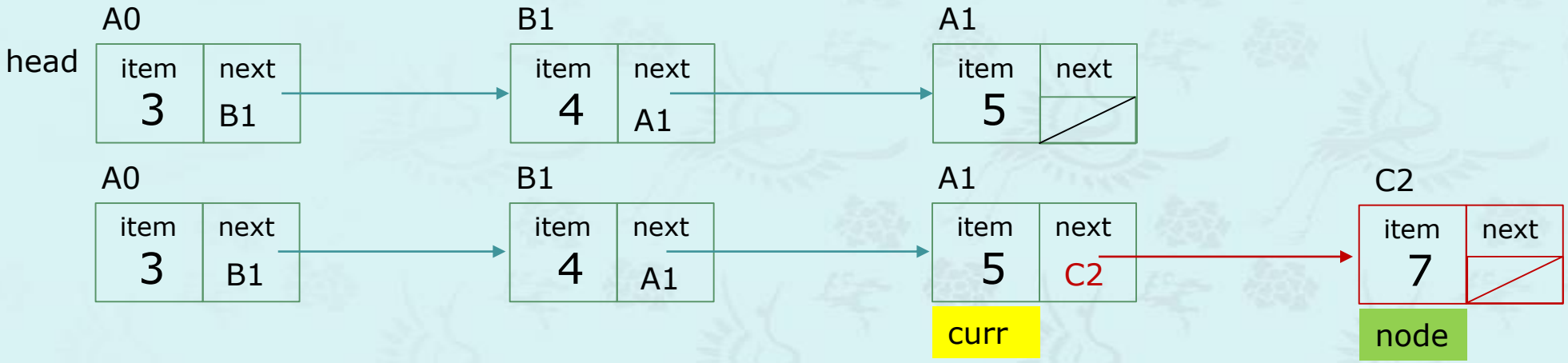
```
curr = last(list->head);
curr->next = new Node{7};
```

```
void push_back(pList p, int val) {
  curr = last(p->head);
  curr->next = new Node{val};
}
```

- find the last node.
- append a new Node{7}.

```
pNode last(pNode list)
  pNode x = list;
  while (x->next != nullptr)
    x = x->next;
  return x;
```

push a node – Case 3; insert at end, head given



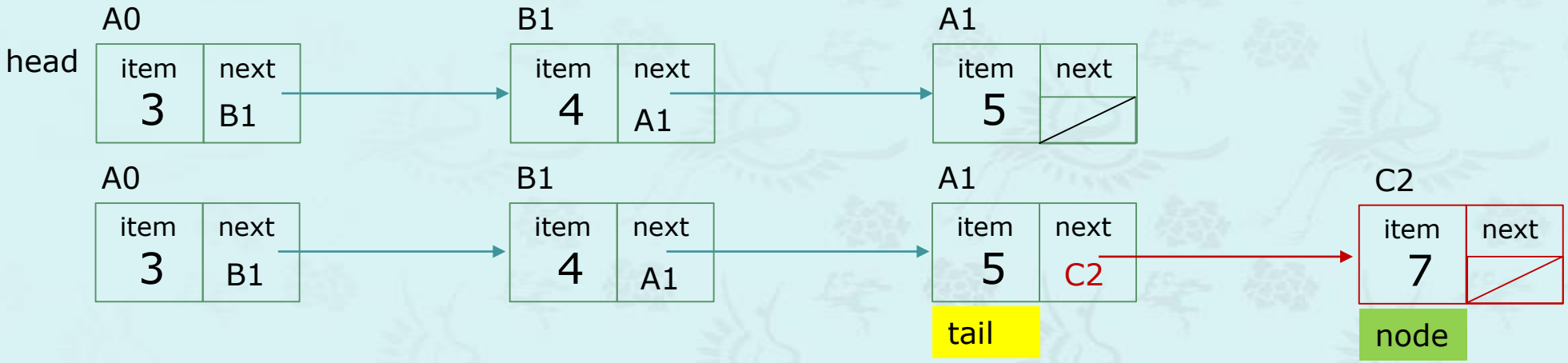

```
curr = last(list->head);
pNode node = new Node{7};
curr->next = node;
```

```
curr = last(list->head);
curr->next = new Node{7};
```

```
void push_back(pList p, int val) {
  curr = last(p->head);
  curr->next = new Node{val};
}
```

O(n)

push a node – Case 3; insert at end, head given




```
curr = last(list->head);
pNode node = new Node{7};
curr->next = node;
```

```
curr = last(list->head);
curr->next = new Node{7};
```

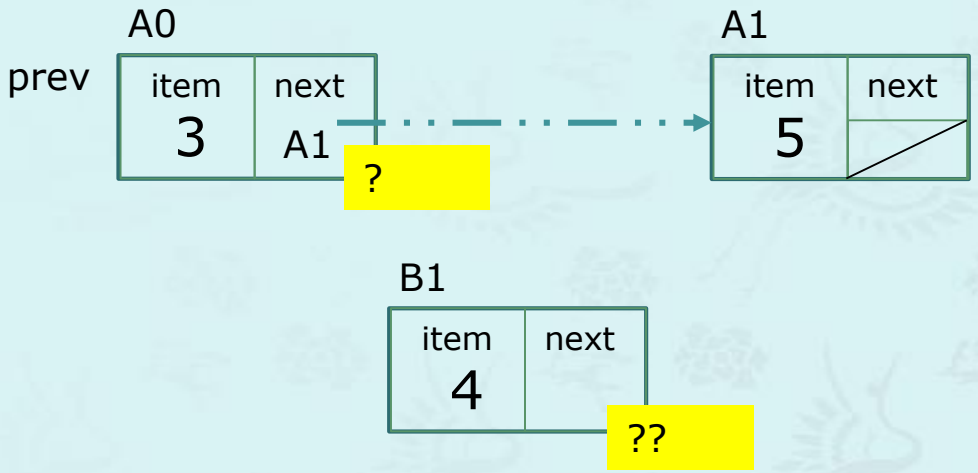
O(1)

```
void push_back(pList p, int val) {
  pNode node = new Node{val};
  p->tail->next = node;
  p->tail = node;
}
```

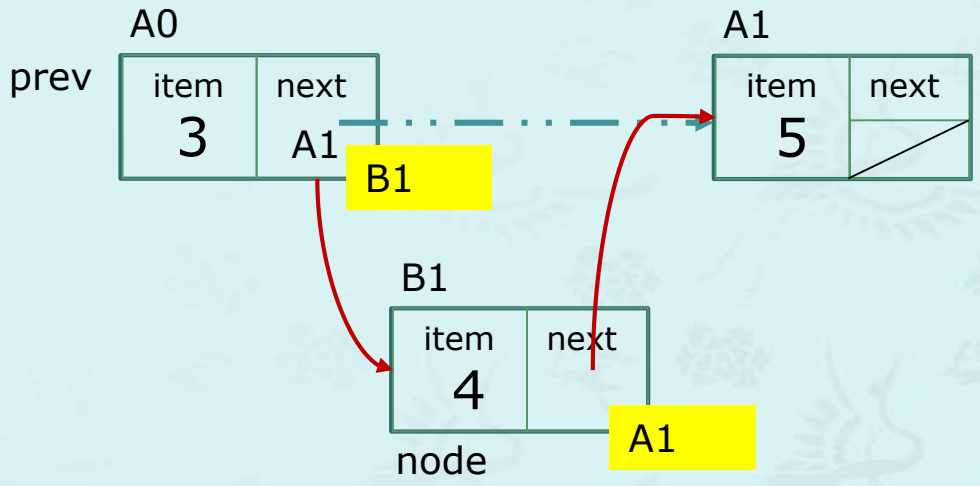
O(n)

```
void push_back(pList p, int val) {
  curr = last(p->head);
  curr->next = new Node{val};
}
```

push a node – Case 2; insert in middle, prev given



push a node – Case 2; insert in middle, prev given



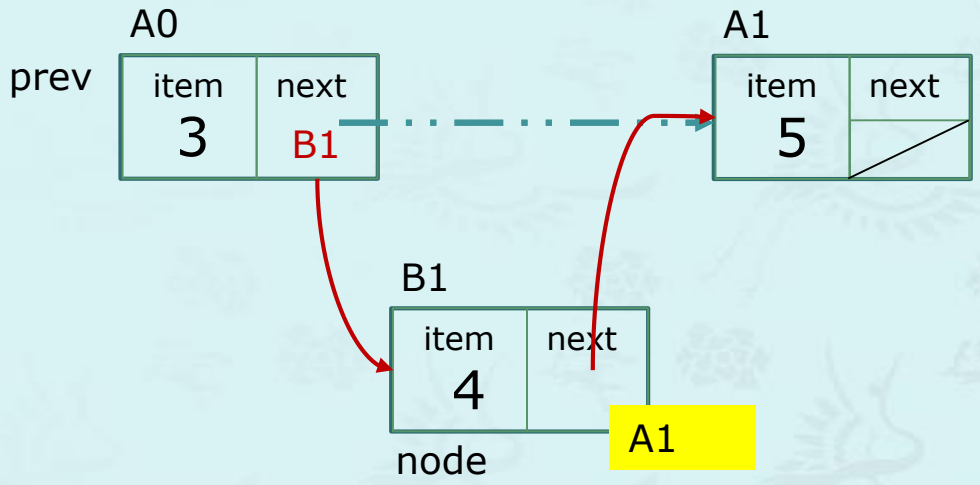

```
pNode node = new Node{4};
prev->next = node;
node->next = prev->next;
```

B1

A1

- We have a bug. Debug it!

push a node – Case 2; insert in middle, prev given




```
pNode node = new Node{4};
prev->next = node;
node->next = prev->next;
```

B1

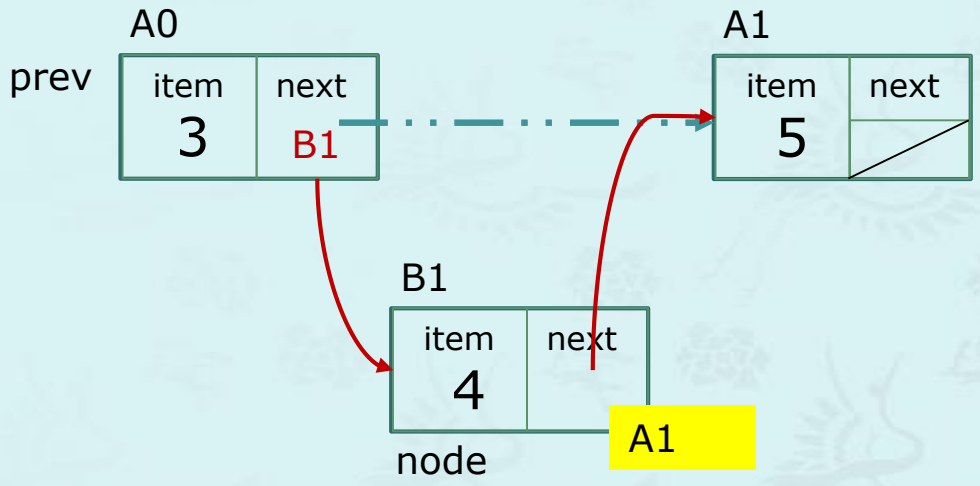
A1

```
pNode node = new Node{4};
node->next = prev->next;
prev->next = node;
```

A1

B1

push a node – Case 2; insert in middle, prev given




```
pNode node = new Node{4};
prev->next = node;
node->next = prev->next;
```

B1
A1

```
pNode node = new Node{4};
node->next = prev->next;
prev->next = node;
```

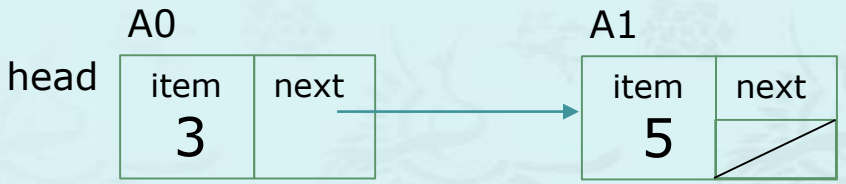
A1
B1

```
prev->next = new Node(4, nullptr, prev->next);
```

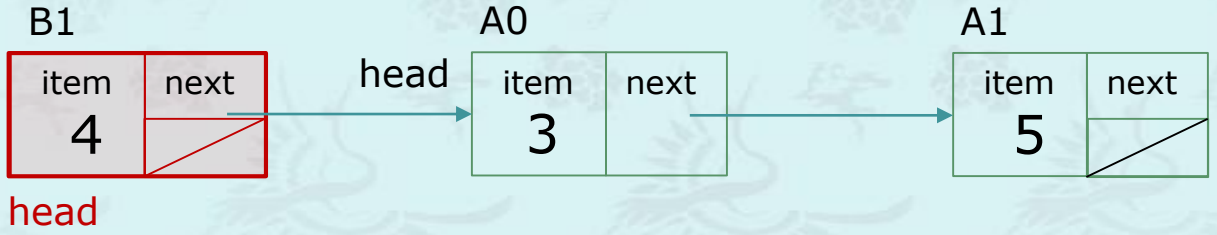
```
// inserts a node val at node x
void push_at(pList p, int val, int x) {
  if (empty(p)) return push_front(p, val);
  // if the first node is x;
  if (p->head->item == x)
    return push_front(p, val);

  pNode curr = p->head;
  pNode prev = nullptr;
  while (curr != nullptr) {
    if (curr->item == x) {
      prev->next = new Node{val, prev->next};
      return;
    }
    prev = curr;
    curr = curr->next;
  }
}
```

Push a node: Three different cases

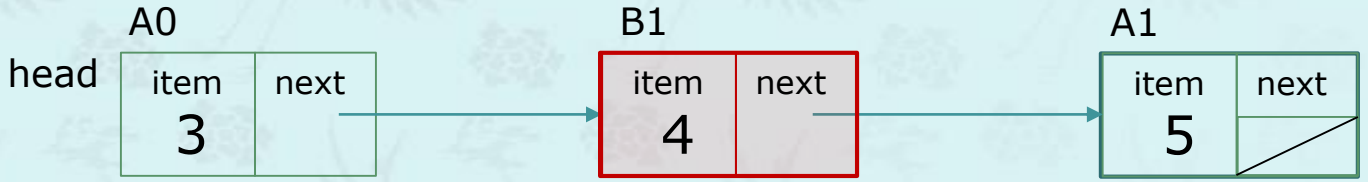
Given: an item(4) to insert – **What was the most difficult part of this coding?**



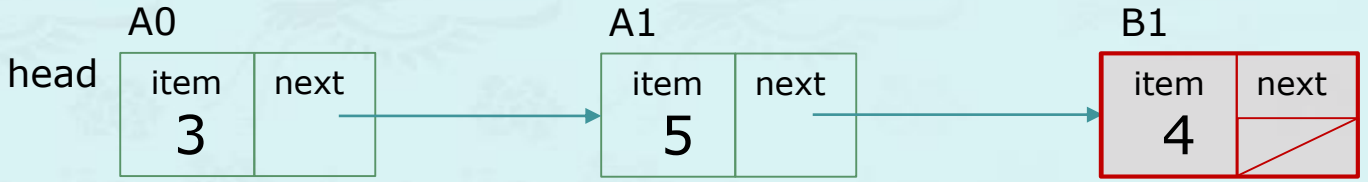
Case 1: in front



Case 2: in middle

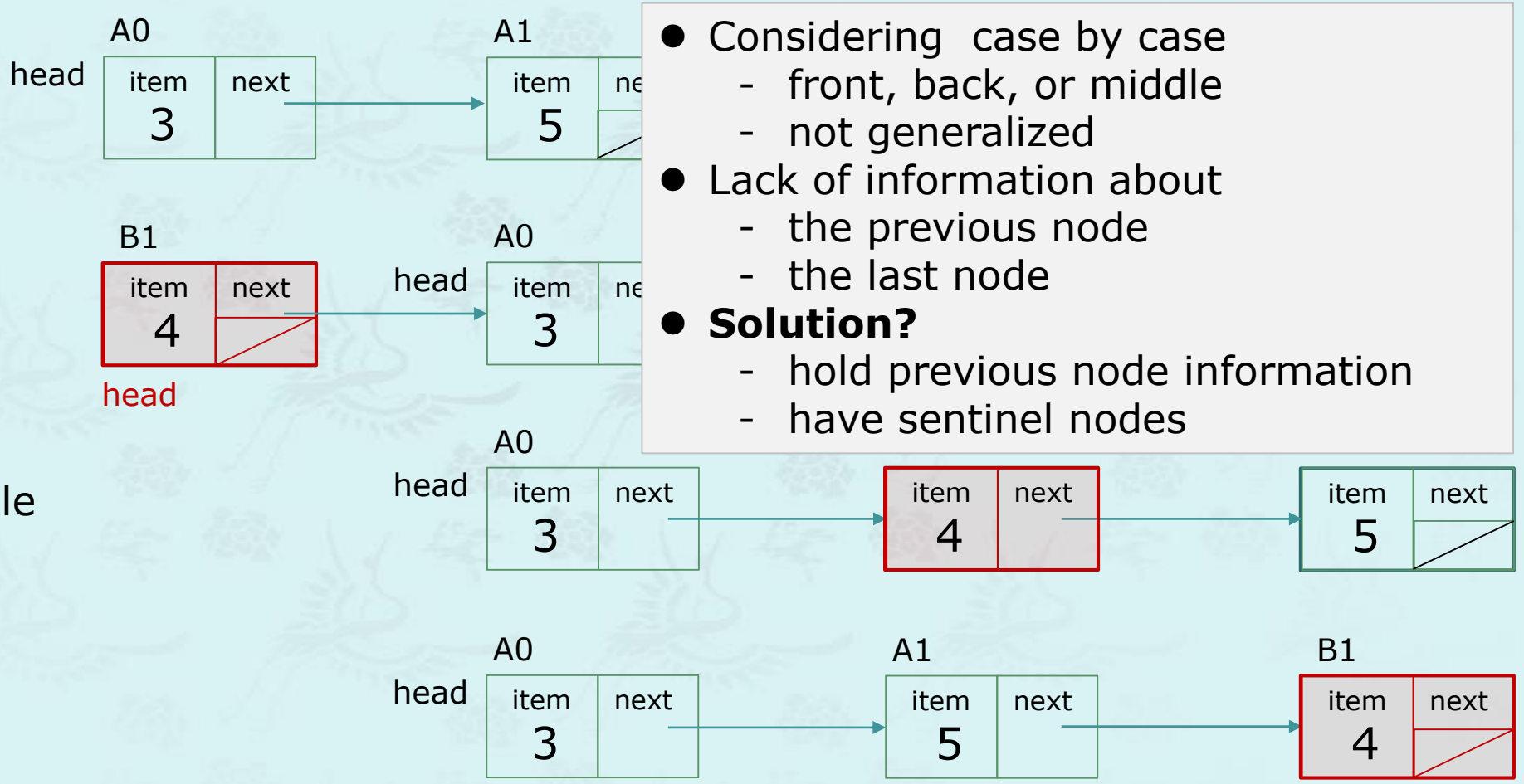


Case 3: at end



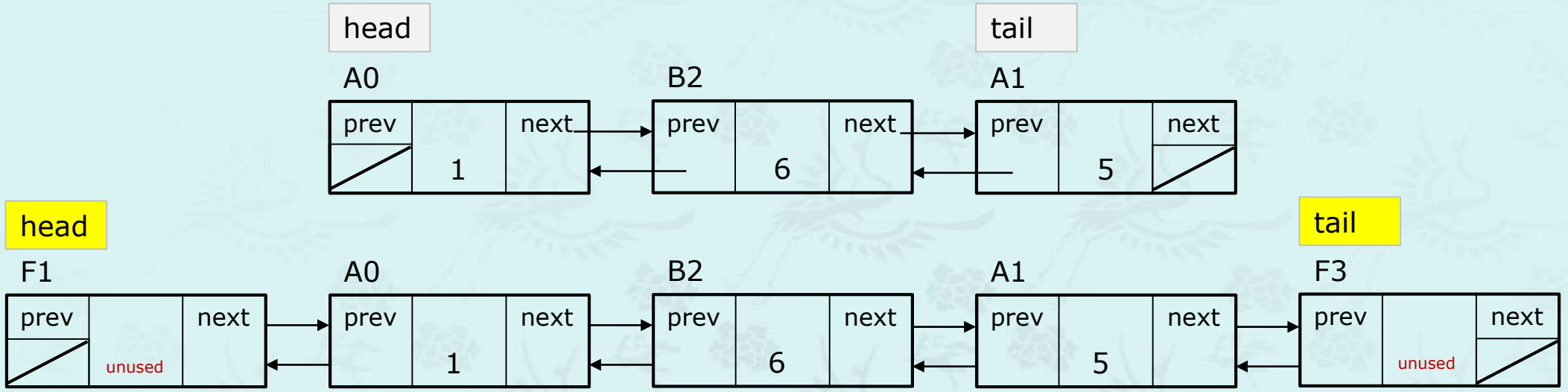
Push a node: Three different cases

Given: an item(4) to insert – **What was the most difficult part of this coding?**



- Considering case by case
  - front, back, or middle
  - not generalized
- Lack of information about
  - the previous node
  - the last node
- **Solution?**
  - hold previous node information
  - have sentinel nodes

doubly linked list with sentinel nodes

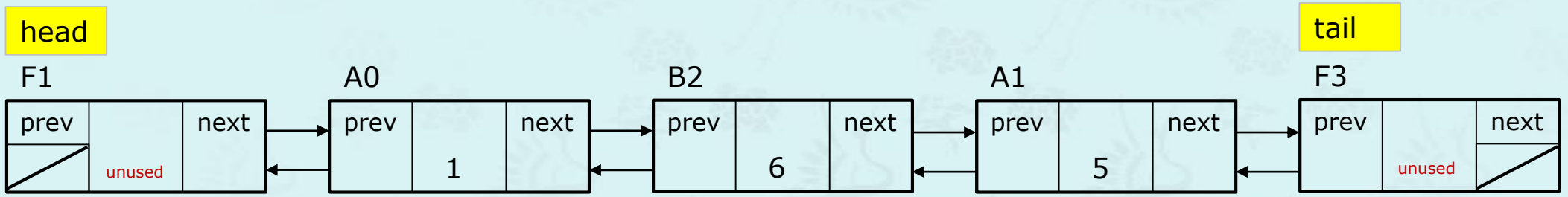


- **Solution**
  - **doubly linked list with sentinel nodes**
  - Each node carries the pointer to the previous node.
  - There is only one case (middle) with two sentinel nodes.

doubly linked list with sentinel nodes

```
struct Node {
  int     item;
  Node*   prev;
  Node*   next;
  Node(const int d = 0, Node* p = nullptr, Node* n = nullptr) {
    item = d; prev = p; next = n;
  }
  ~Node() {}
};

struct List {
  Node* head;
  Node* tail;
  List() { head = new Node;      tail = new Node;
           head->next = tail;    tail->prev = head;
           head->prev = nullptr; tail->next = nullptr;
  }
  ~List() {}
};

using pNode = Node*;
using pList = List*;
```

doubly linked list with sentinel nodes - **Exercise**

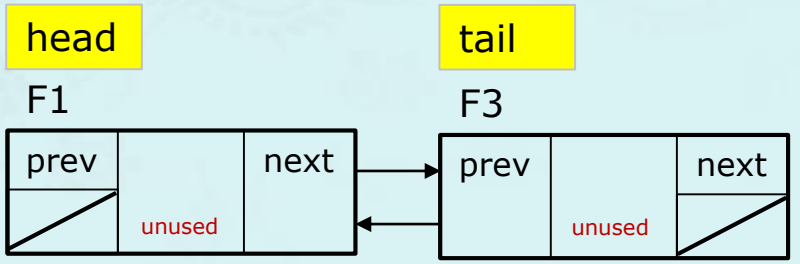



```
pNode begin(pList p) {
  return p->head->next;
}
```

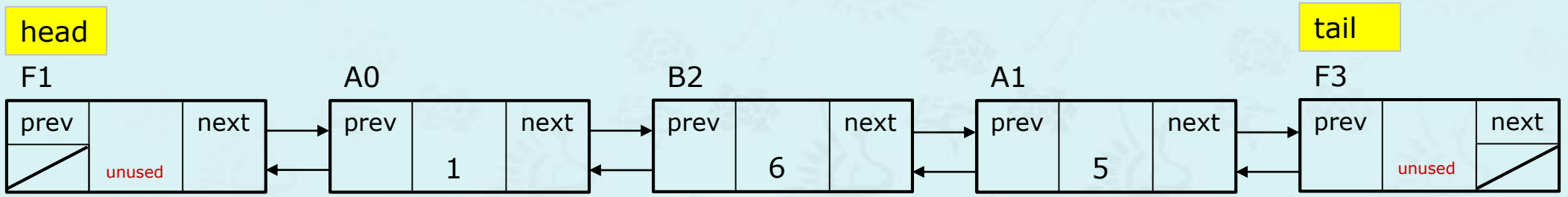
```
pNode empty(pList p) {
  return begin(p) == end(p);
}
```

```
pNode end(pList p) {
  return p->tail;
}
```

```
pNode last(pList p) {
  return end(p)->prev;
}
```



doubly linked list with sentinel nodes - **Exercise**

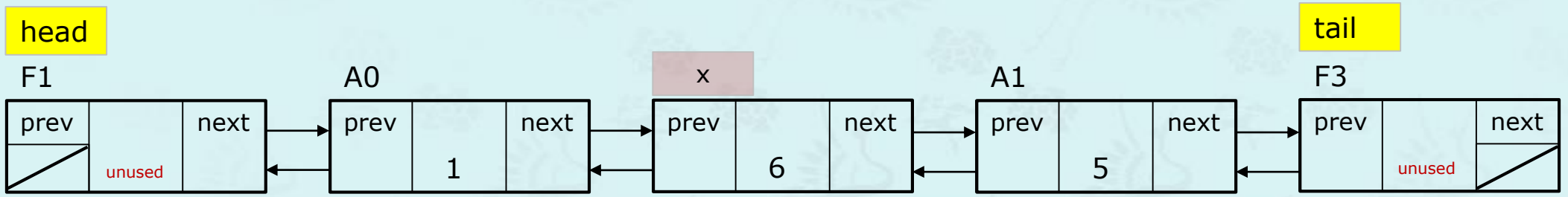



```
pNode begin(pList p) {
  return p->head->next;
}
```

```
pNode end(pList p) {
  return p->tail;
}
```

```
int [    ](pList p) {
  int count = 0;
  pNode c = begin(p);
  while(c != end(p)) {
    count++;
    c = c->next;
  }
  return count;
}
```

doubly linked list with sentinel nodes

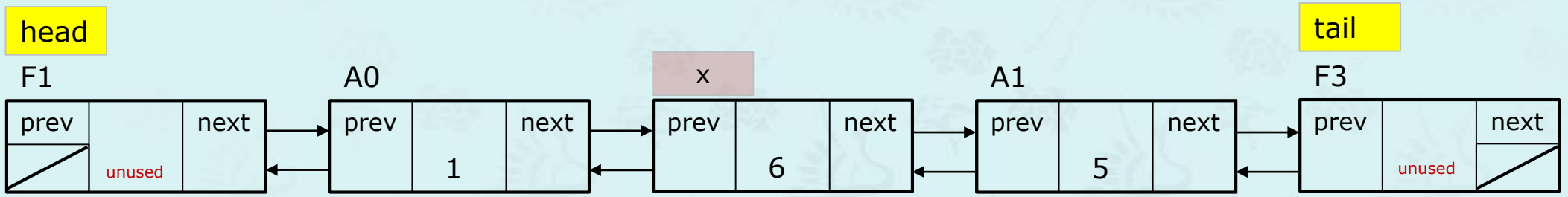



```
pNode begin(pList p) {
  return p->head->next;
}
```

```
pNode end(pList p) {
  return p->tail;
}
```

```
pNode [          ](pList p, int val){
  pNode curr = begin(p);
  while(curr != end(p)) {
    if (curr->item == val)
      return curr;
    curr = curr->next;
  }
  return curr;
}
```

doubly linked list with sentinel nodes



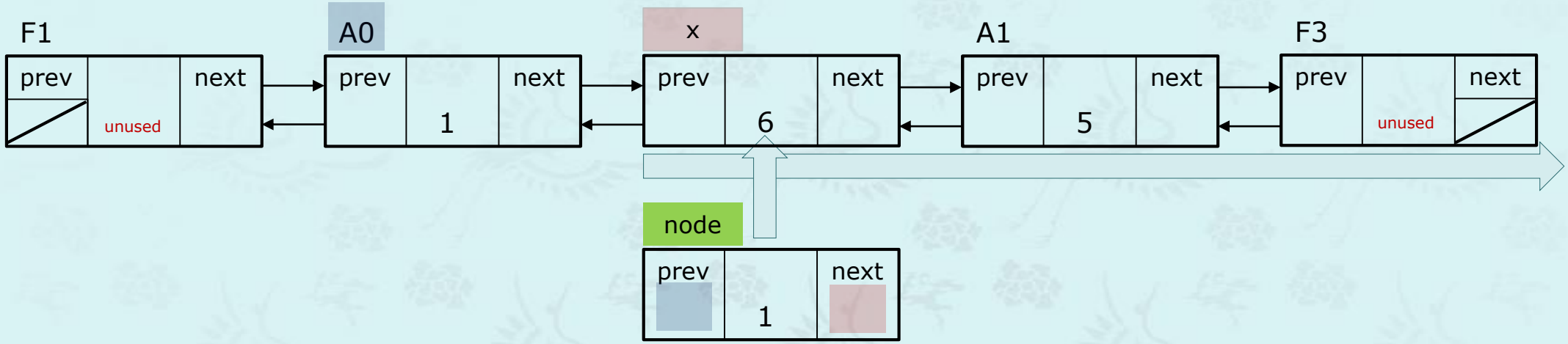

```
pNode begin(pList p) {
  return p->head->next;
}
```

```
pNode end(pList p) {
  return p->tail;
}
```

```
pNode find(pList p, int val){
  pNode curr = begin(p);
  while(curr != end(p)) {
    if (curr->item == val)
      return curr;
    curr = curr->next;
  }
  return curr;
}
```

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

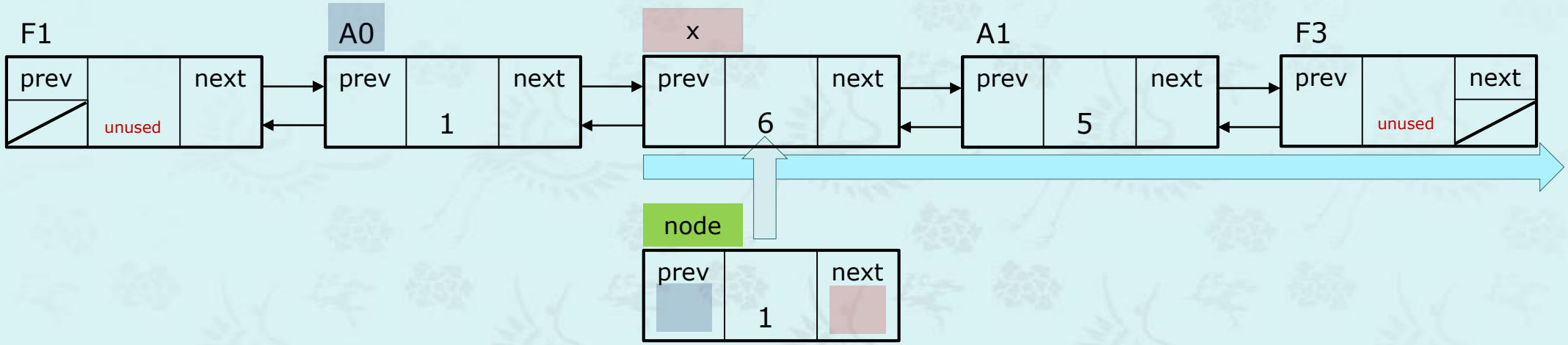
doubly linked list with sentinel nodes




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


}
```

```
struct Node{
  int item;
  Node* prev;
  Node* next;
  // constructor
  Node(int d=0, Node* p=nullptr, Node* n=nullptr) {
    item = d;          prev = p;  next = n;
  }
  // destructor
  ~Node() {}
};
```

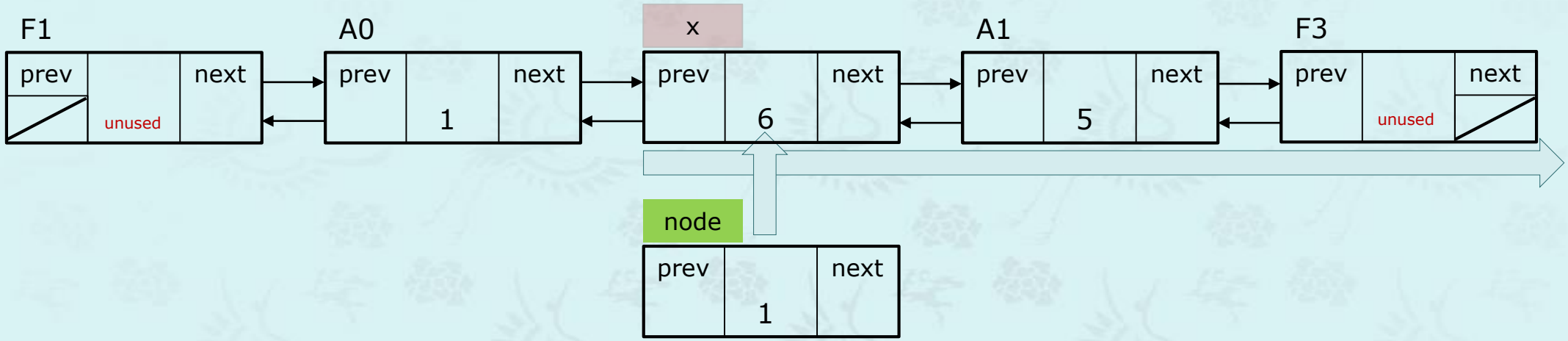
doubly linked list with sentinel nodes




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

}
```

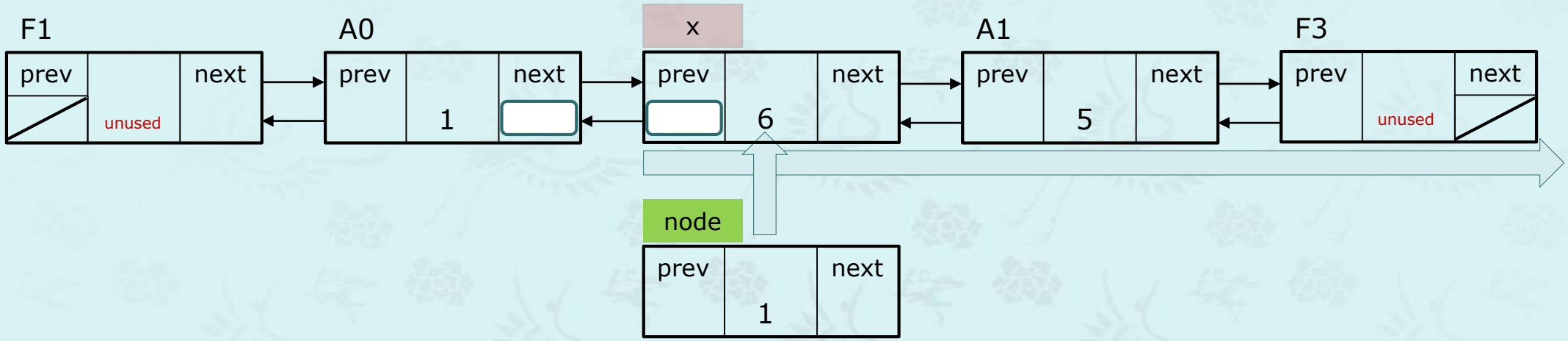
```
struct Node{
  int item;
  Node* prev;
  Node* next;
  // constructor
  Node(int d=0, Node* p=nullptr, Node* n=nullptr) {
    item = d;          prev = p;  next = n;
  }
  // destructor
  ~Node() {}
};
```

doubly linked list with sentinel nodes




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

doubly linked list with sentinel nodes




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

doubly linked list with sentinel nodes

Case 1: in front

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

Case 3: at end

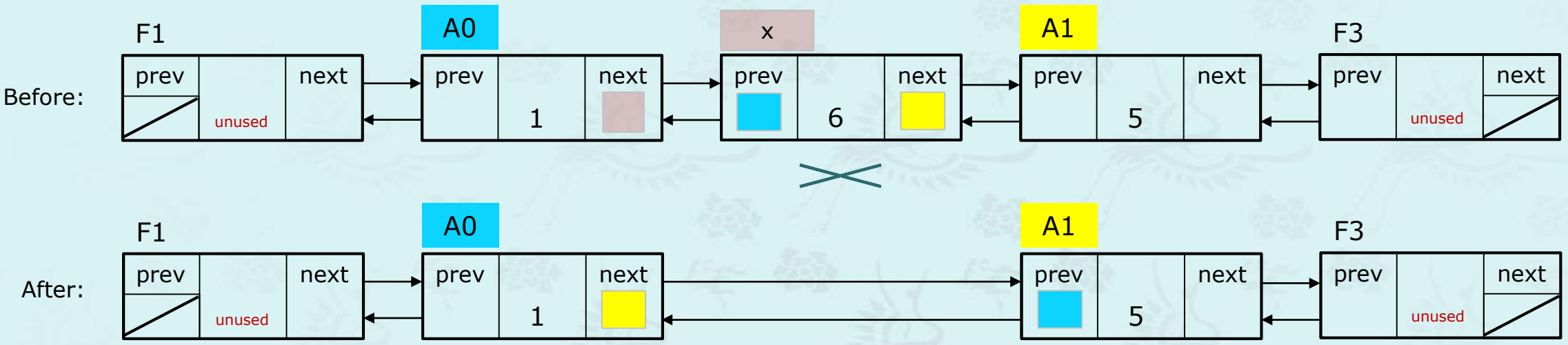
```
void push_back(pList p, int val){
  insert(end(p), val);
}
```

Case 2: in middle

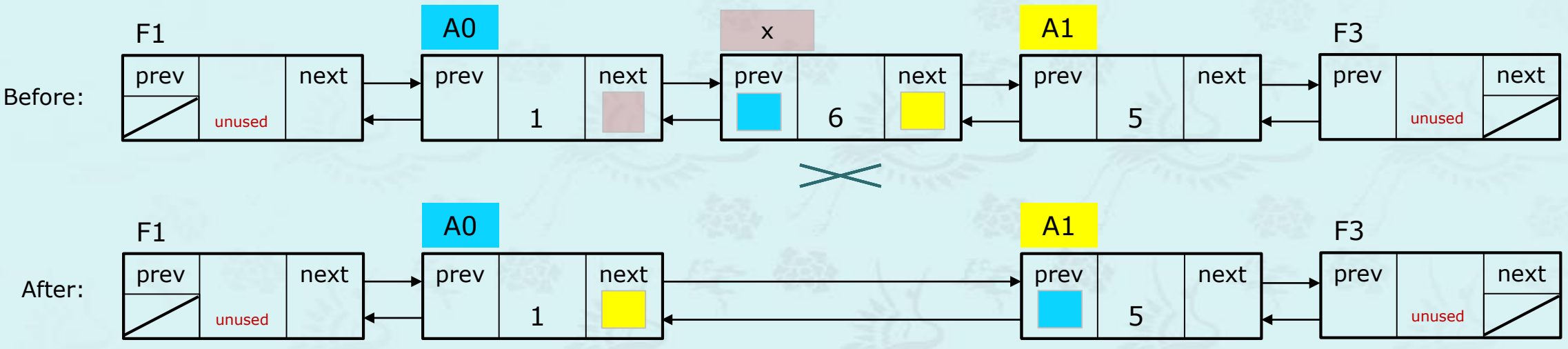
```
void push(pList p, int val, int x){
  insert(find(p, x), val);
}
```

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

doubly linked list with sentinel nodes

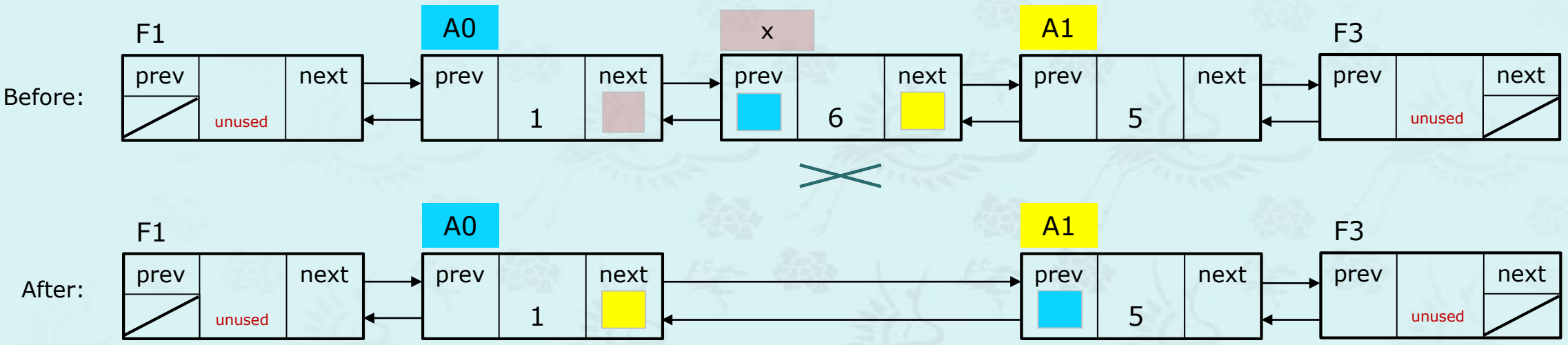


doubly linked list with sentinel nodes




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

doubly linked list with sentinel nodes



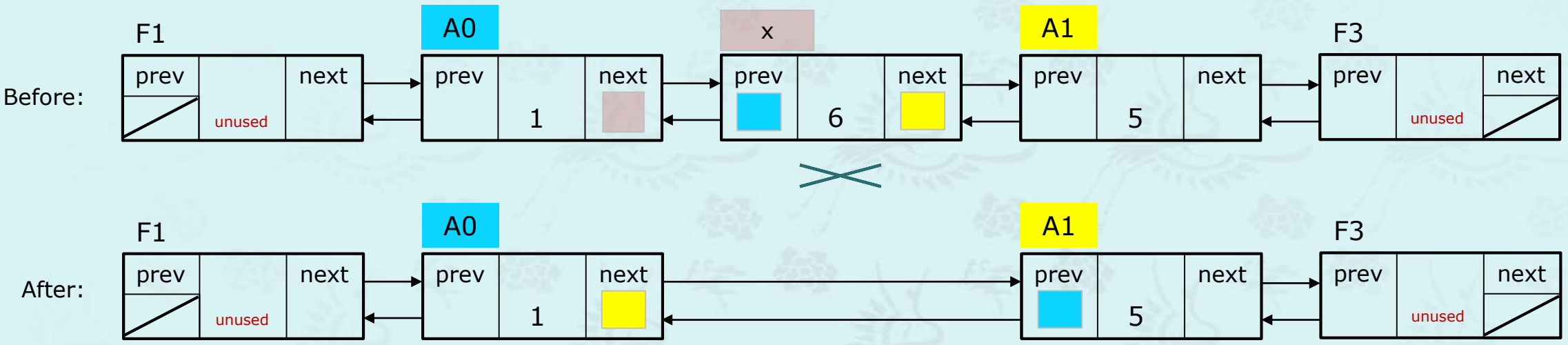

```
void erase(pNode x){
   x->prev->next = x->next;
   x->next->prev = x->prev;
   delete x;
}
```

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

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

```
void pop(pList p, int val){

}
```

doubly linked list with sentinel nodes




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

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

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

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

doubly linked list with sentinel nodes

```
struct Node {
  int     item;
  Node*   prev;
  Node*   next;
  Node(const int d = 0, Node* p = nullptr, Node* n = nullptr) {
    item = d; prev = p; next = n;
  }
  ~Node() {}
};

struct List {
  Node* head;
  Node* tail;
  List() { head = new Node;      tail = new Node;
           head->next = tail;    tail->prev = head;
           head->prev = nullptr; tail->next = nullptr;
  }
  ~List() {}
};
using pNode = Node*;
using pList = List*;
```

```
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 val);                  // returns the first node with val
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
void insert(pNode x, int val);                 // inserts a new node with val at the node x
void erase(pNode x);                           // deletes a node and returns the previous node

void push(pList p, int val, int x);            // inserts a node with val at the node with x
void push_front(pList p, int val);             // inserts a node at front of the list
void push_back(pList p, int val);              // inserts a node with val at end of the list
void push_sorted(pList p, int val, bool ascending = true); // inserts a node in sorted
void pop(pList p, int val);                    // deletes the first node with val
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 val);                // deletes all the nodes with val

pList 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 items in linked list
```


quaestio quaestio qo