SFU

Linked List Operations III

CMPT 125
Mo Chen
SFU Computing Science
2/3/2020

Lecture 20

Today

- `Llcatenate(...)`
- `LLinsert(...)`

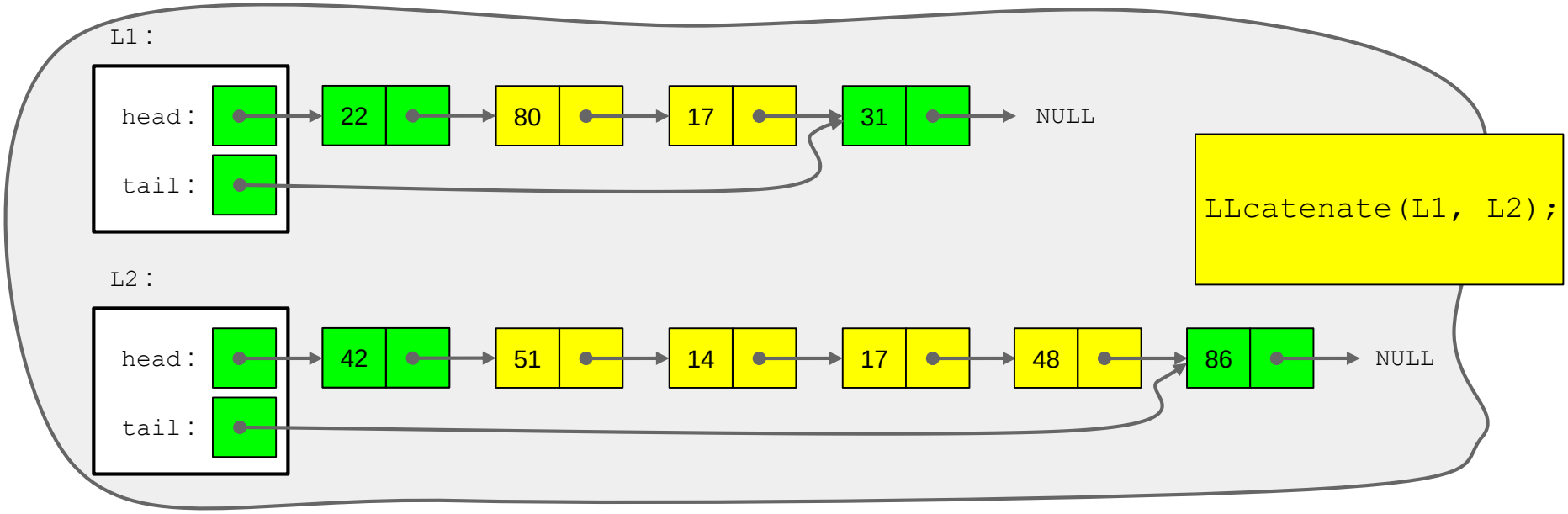
Linked List: `LLcatenate(L1, L2)`

Behaviour:

- append the elements of `L2` to the end of `L1`, i.e., `L1 <- L1 + L2`
- free `L2`, i.e., only `L1` survives

Q. What's the strategy?

Q. What's the running time?



Linked List: `LLcatenate(L1, L2)`

Behaviour:

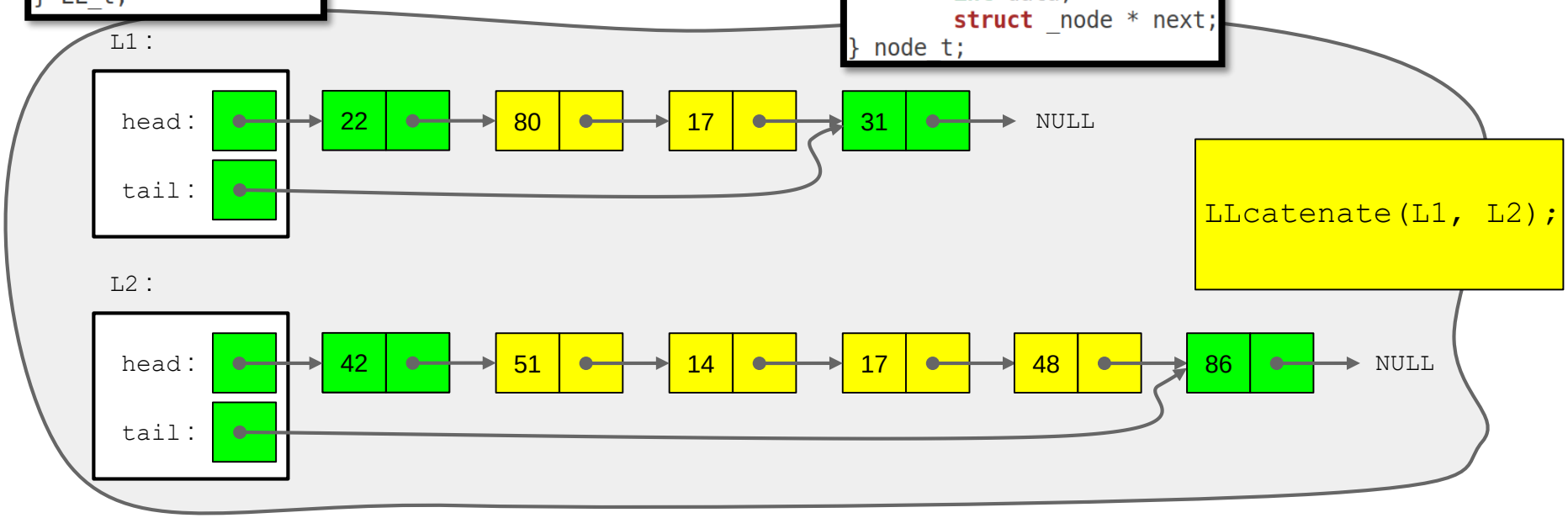
- append the elements of `L2` to the end of `L1`, i.e., `L1 <- L1 + L2`
- `L2`, i.e., only `L1` survives

s the strategy?

s the running time?

```
// LL.h
#include "LL-node.h"

typedef struct {
    node_t * head;
    node_t * tail;
} LL_t;
```

```
// LL-node.h
typedef struct _node {
        int data;
        struct _node * next;
} node_t;
```



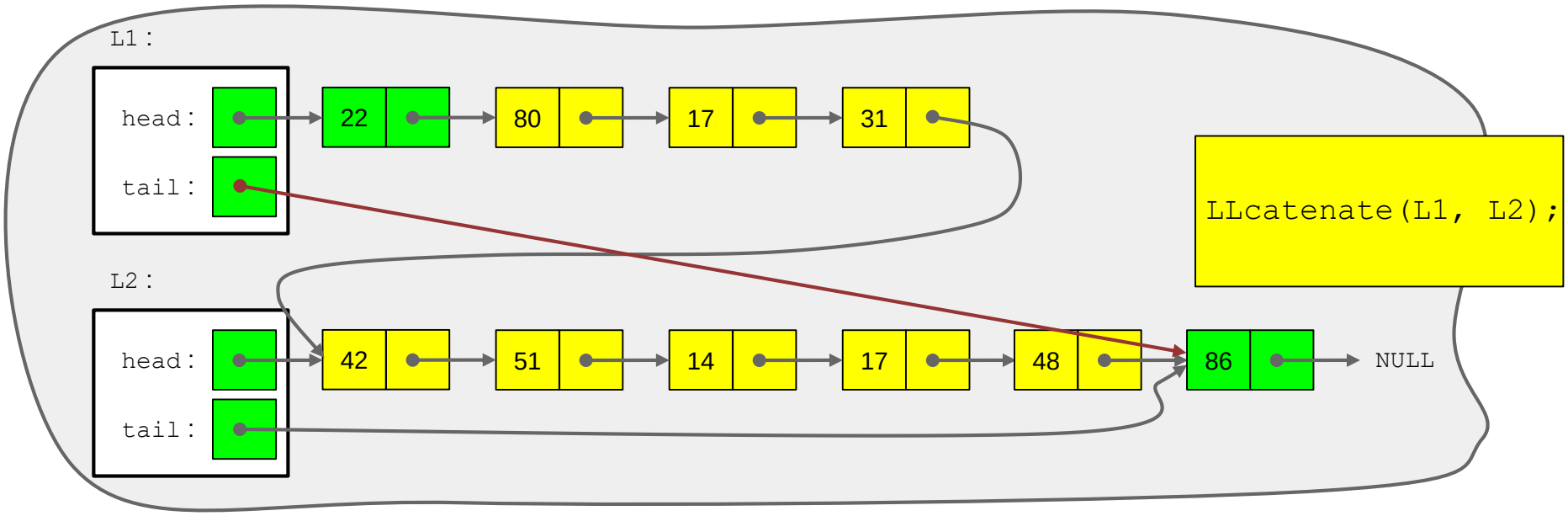
Linked List: `LLcatenate(L1, L2)`

Behaviour:

- append the elements of `L2` to the end of `L1`, i.e., `L1 <- L1 + L2`
- free `L2`, i.e., only `L1` survives

Q. What's the strategy?

Q. What's the running time?



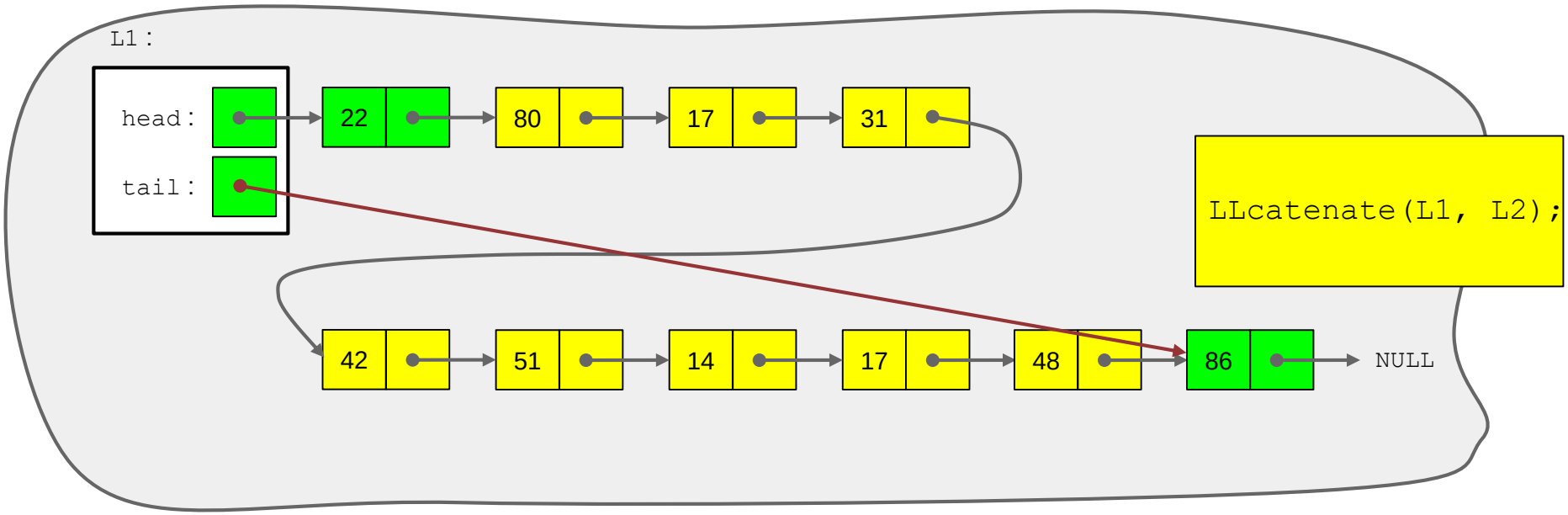
Linked List: `LLcatenate(L1, L2)`

Behaviour:

- append the elements of `L2` to the end of `L1`, i.e., `L1 <- L1 + L2`
- free `L2`, i.e., only `L1` survives

Q. What's the strategy?

Q. What's the running time?



Linked List: `LLcatenate(L1, L2)`

Behaviour:

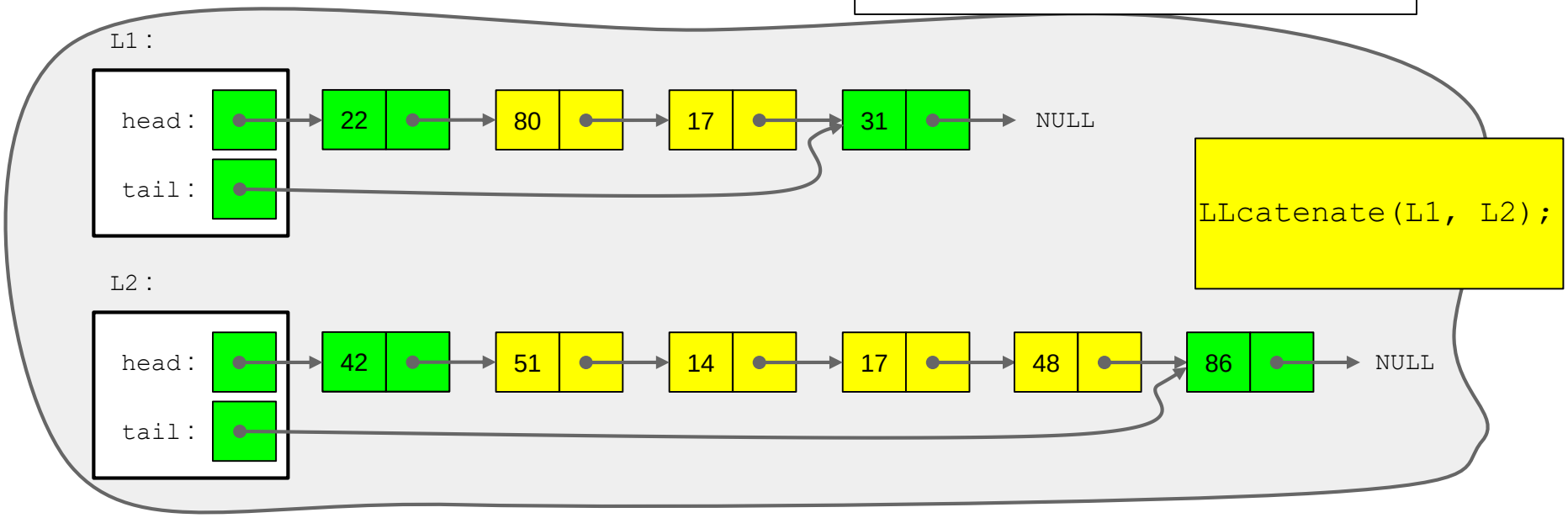
- append the elements of `L2` to the end of `L1`, i.e., `L1 <- L1 + L2`
- free `L2`, i.e., only `L1` survives

Q. What's the strategy?

Q. What's the running time?

All the steps:

```
L1->tail->next = L2->head;
```



Linked List: `LLcatenate(L1, L2)`

Behaviour:

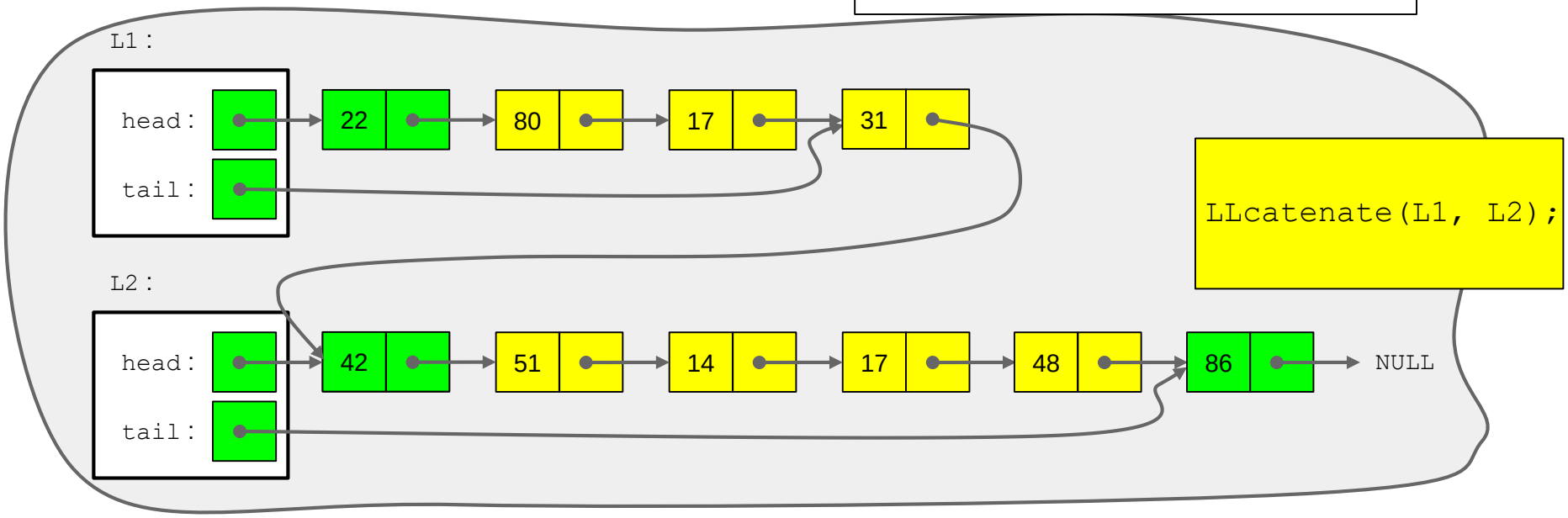
- append the elements of `L2` to the end of `L1`, i.e., `L1 <- L1 + L2`
- free `L2`, i.e., only `L1` survives

Q. What's the strategy?

Q. What's the running time?

All the steps:

```
L1->tail->next = L2->head;
```



Linked List: `LLcatenate(L1, L2)`

Behaviour:

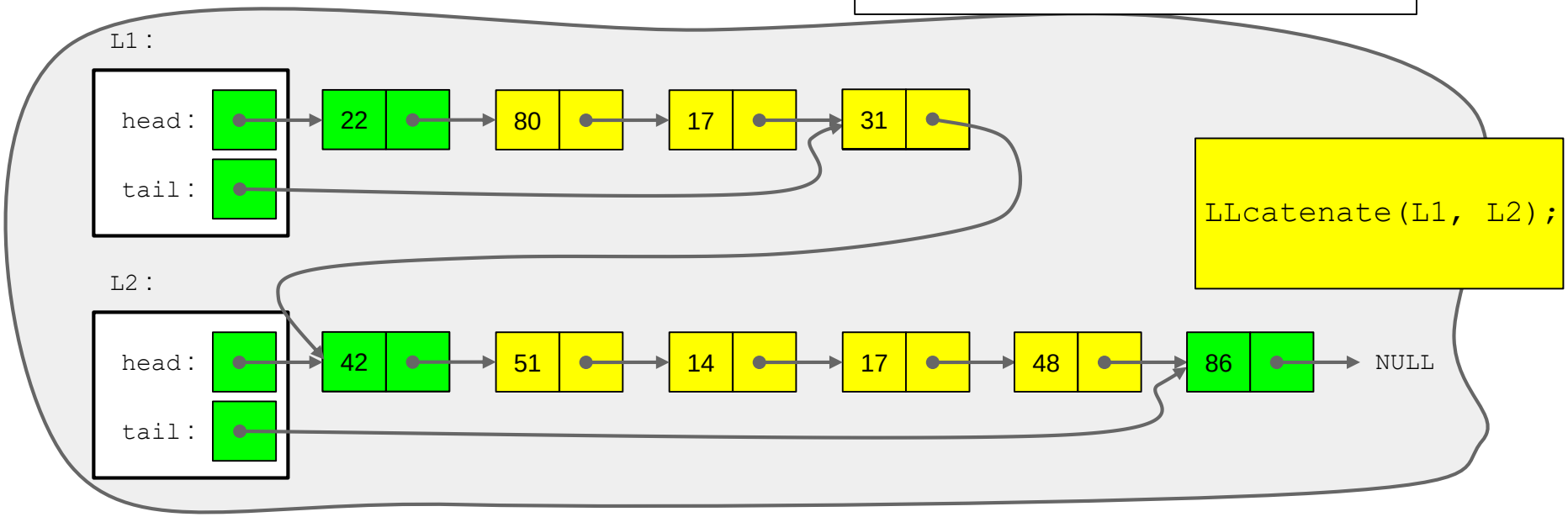
- append the elements of `L2` to the end of `L1`, i.e., `L1 <- L1 + L2`
- free `L2`, i.e., only `L1` survives

Q. What's the strategy?

Q. What's the running time?

All the steps:

```
L1->tail->next = L2->head;
L1->tail = L2->tail;
```



Linked List: `LLcatenate(L1, L2)`

Behaviour:

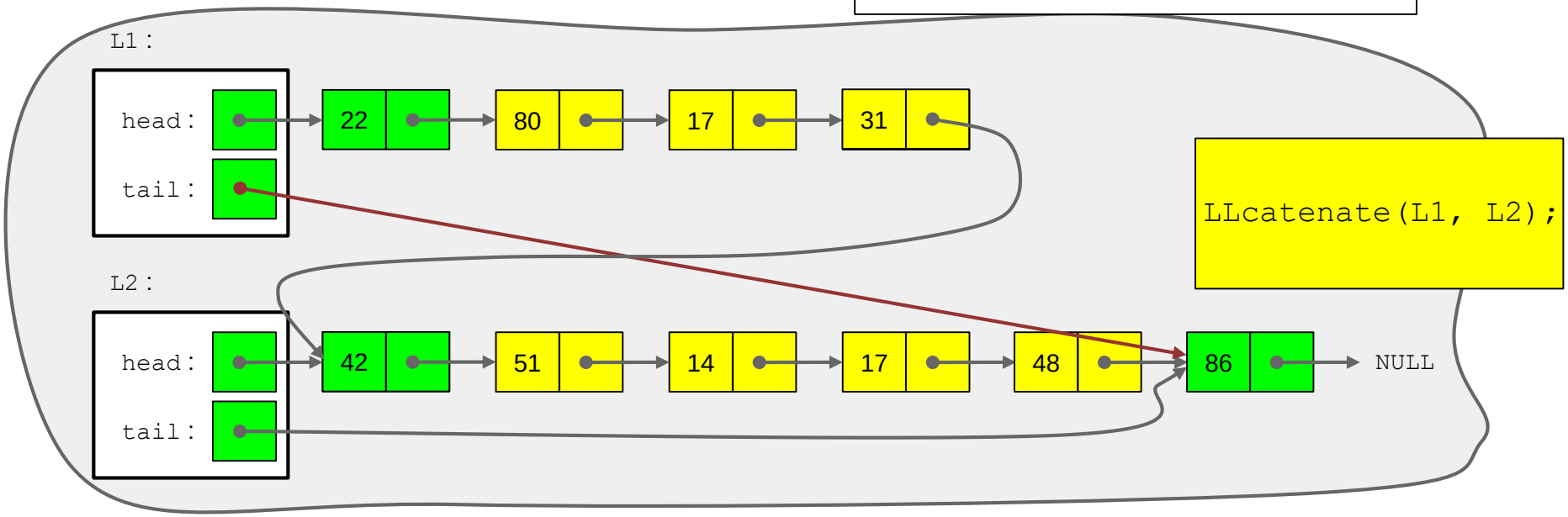
- append the elements of `L2` to the end of `L1`, i.e., `L1 <- L1 + L2`
- free `L2`, i.e., only `L1` survives

Q. What's the strategy?

Q. What's the running time?

All the steps:

```
L1->tail->next = L2->head;
L1->tail = L2->tail;
```



Linked List: `LLcatenate(L1, L2)`

Behaviour:

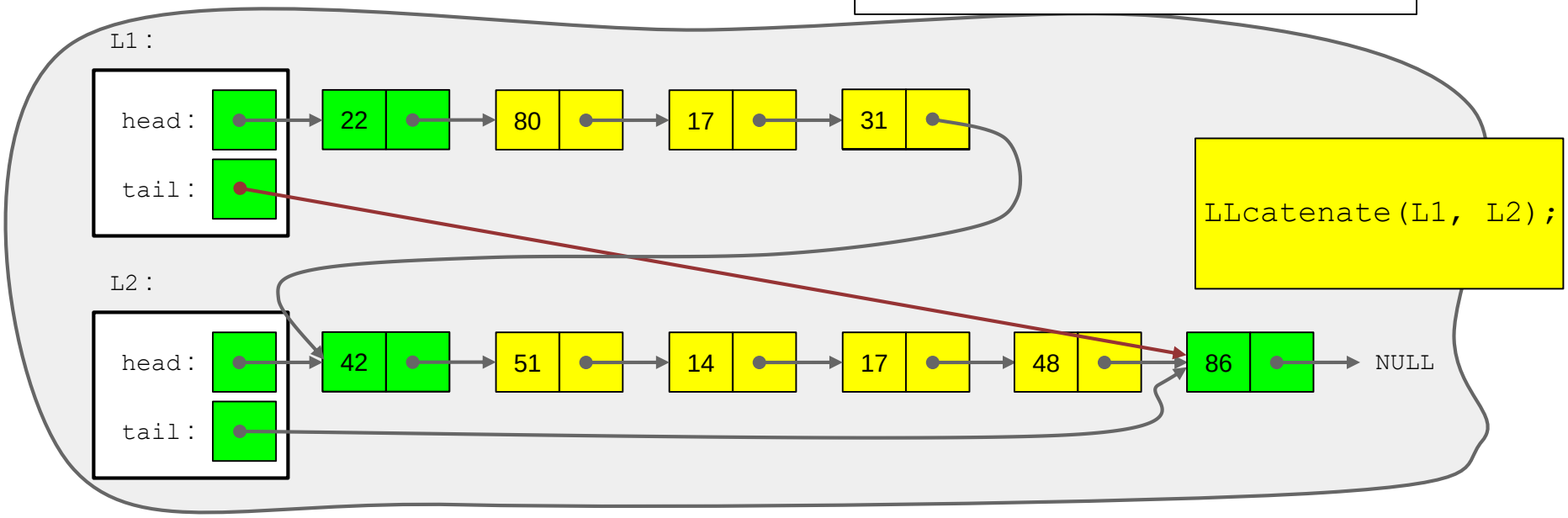
- append the elements of `L2` to the end of `L1`, i.e., `L1 <- L1 + L2`
- free `L2`, i.e., only `L1` survives

Q. What's the strategy?

Q. What's the running time?

All the steps:

```
L1->tail->next = L2->head;
L1->tail = L2->tail;
free(L2);
```



Linked List: `LLcatenate(L1, L2)`

Behaviour:

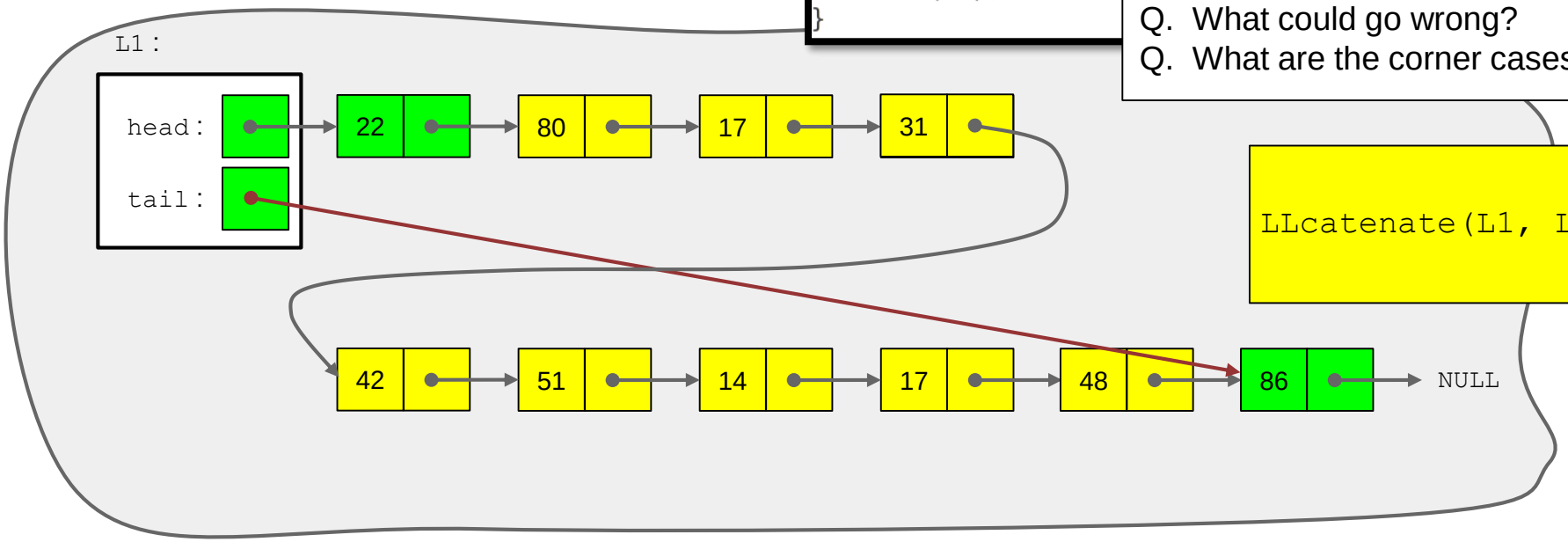
- append the elements of `L2` to
- free `L2`, i.e., only `L1` survives

Q. What's the strategy?

Q. What's the running time?

```
// joins different L1, L2 such that L1 <- L1 + L2, frees L2
void LLcatenate(LL_t * L1, LL_t * L2) {



        L1->tail->next = L2->head;
        L1->tail= L2->tail;

    free(L2);
}
```

Q. What could go wrong?
Q. What are the corner cases?



Linked Lists: Typical Corner Cases

When designing algorithms to modify a linked list, solve the “usual” case and then answer the following questions:

- Q. What should happen if list empty?
- Q. When would the head change?
- Q. When would the tail change?

Linked List: `LLcatenate(L1, L2)`

Behaviour:

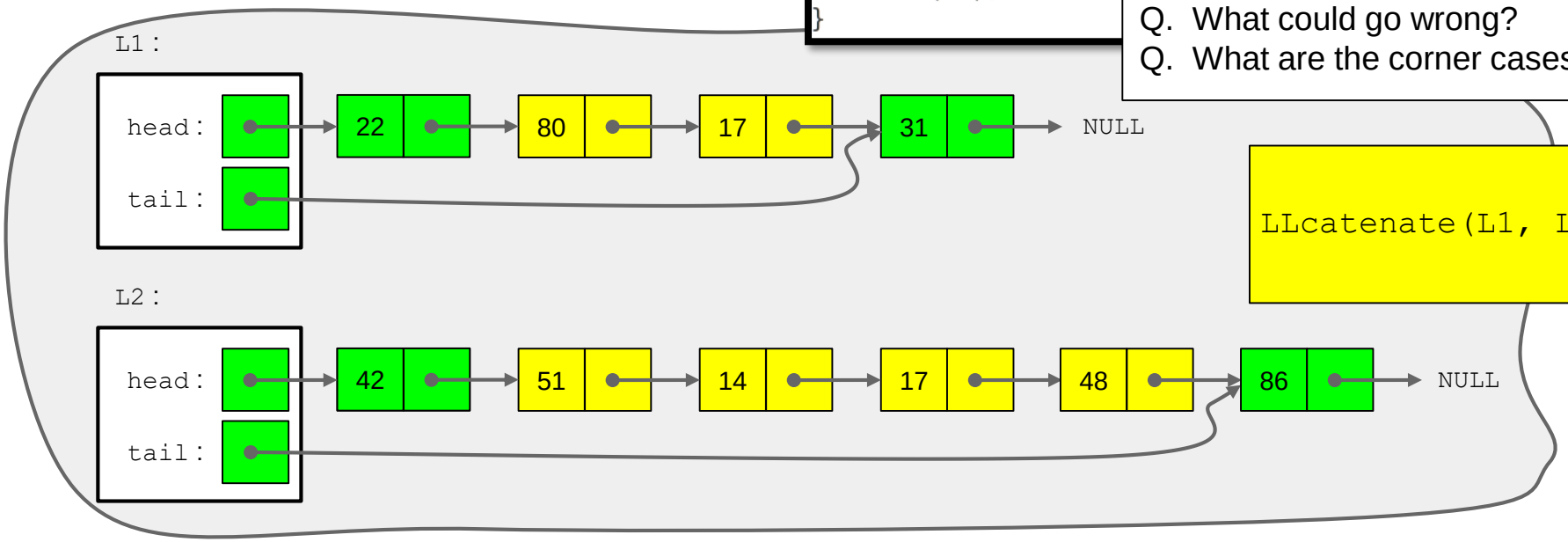
- append the elements of `L2` to
- free `L2`, i.e., only `L1` survives

Q. What's the strategy?

Q. What's the running time?

```
// joins different L1, L2 such that L1 <- L1 + L2, frees L2
void LLcatenate(LL_t * L1, LL_t * L2) {



        L1->tail->next = L2->head;
        L1->tail= L2->tail;

    free(L2);
}
```

Q. What could go wrong?
Q. What are the corner cases?




```
LLcatenate(L1, L2);
```

Linked List: `LLcatenate(L1, L2)`

Behaviour:

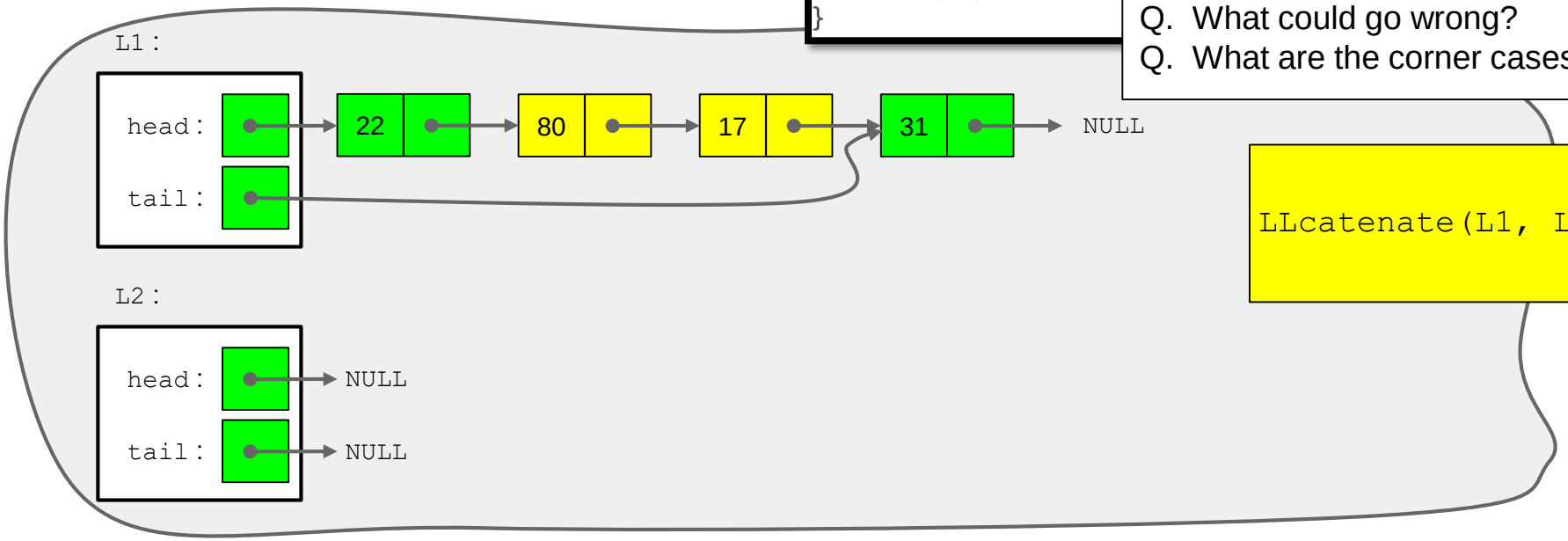
- append the elements of `L2` to
- free `L2`, i.e., only `L1` survives

Q. What's the strategy?

Q. What's the running time?

```
// joins different L1, L2 such that L1 <- L1 + L2, frees L2
void LLcatenate(LL_t * L1, LL_t * L2) {



        L1->tail->next = L2->head;
        L1->tail= L2->tail;

    free(L2);
}
```

Q. What could go wrong?
Q. What are the corner cases?




```
LLcatenate(L1, L2);
```

Linked List: `LLcatenate(L1, L2)`

Behaviour:

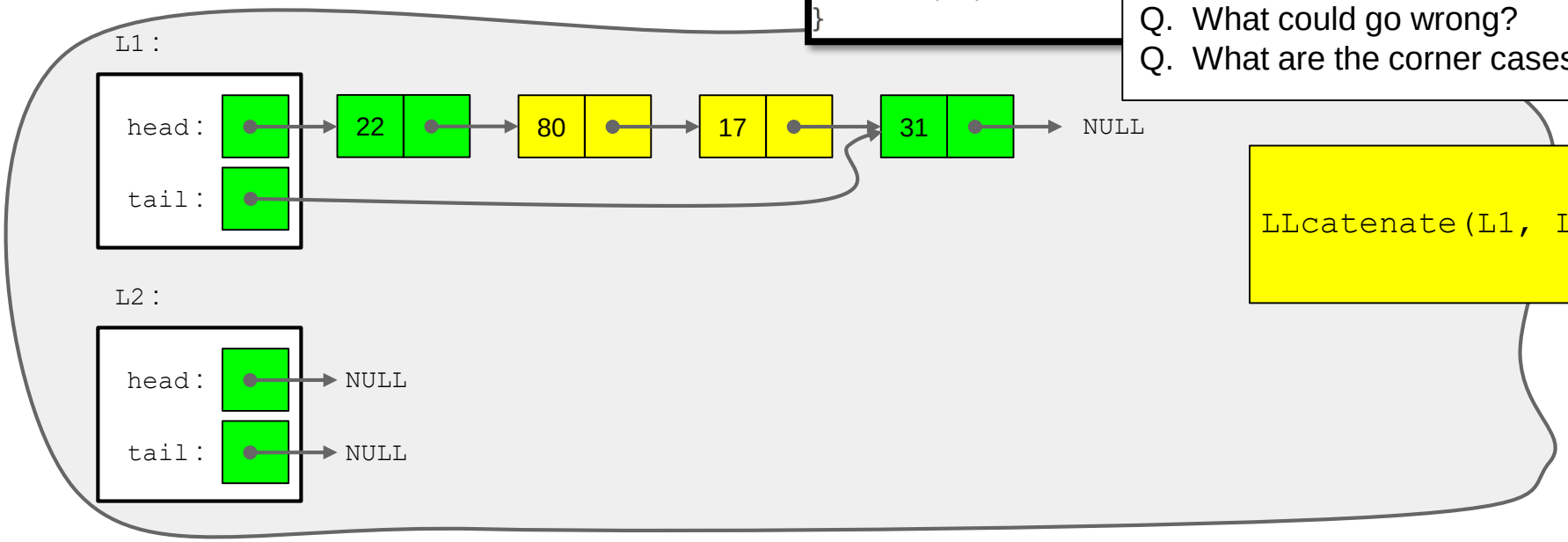
- append the elements of `L2` to
- free `L2`, i.e., only `L1` survives

Q. What's the strategy?

Q. What's the running time?

```
// joins different L1, L2 such that L1 <- L1 + L2, frees L2
void LLcatenate(LL_t * L1, LL_t * L2) {
    if (L2->head == NULL) {
        assert(L2->tail == NULL);



    } else {
        L1->tail->next = L2->head;
        L1->tail= L2->tail;

    free(L2);
}
```

Q. What could go wrong?
Q. What are the corner cases?



`LLcatenate(L1, L2);`

Linked List: `LLcatenate(L1, L2)`

Behaviour:

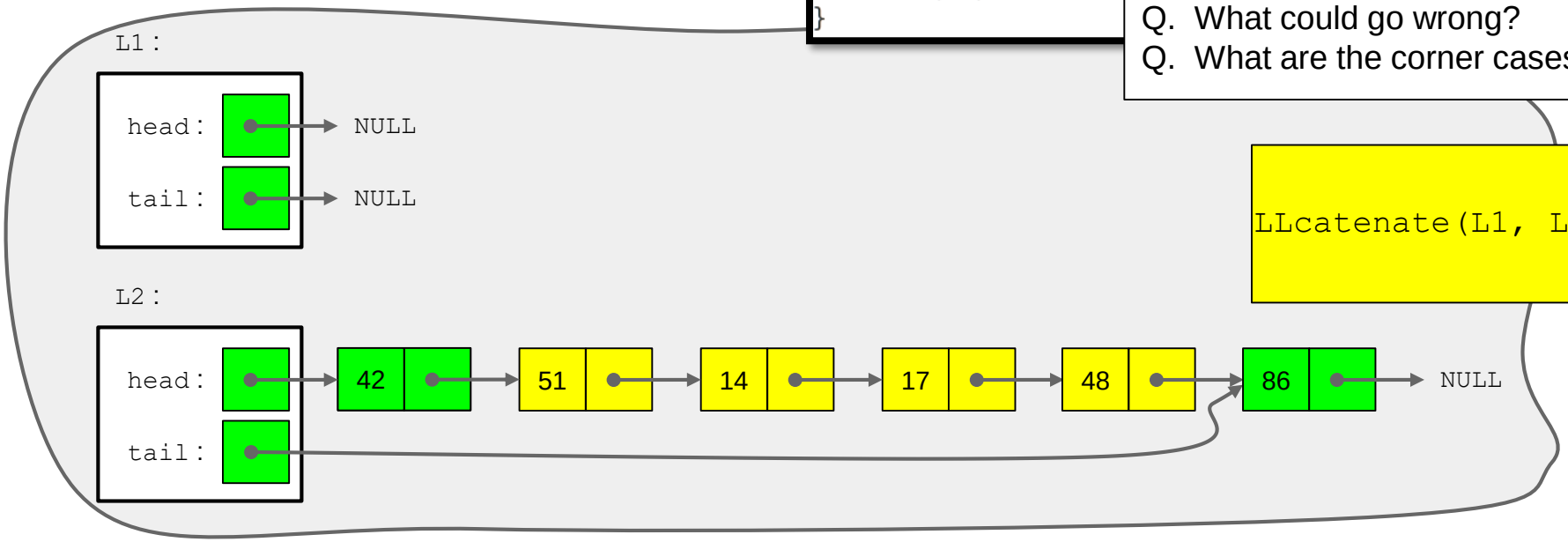
- append the elements of `L2` to
- free `L2`, i.e., only `L1` survives

Q. What's the strategy?

Q. What's the running time?

```
// joins different L1, L2 such that L1 <- L1 + L2, frees L2
void LLcatenate(LL_t * L1, LL_t * L2) {
    if (L2->head == NULL) {
        assert(L2->tail == NULL);



    } else {
        L1->tail->next = L2->head;
        L1->tail= L2->tail;

    free(L2);
}
```

Q. What could go wrong?
Q. What are the corner cases?



Linked List: `LLcatenate(L1, L2)`

Behaviour:

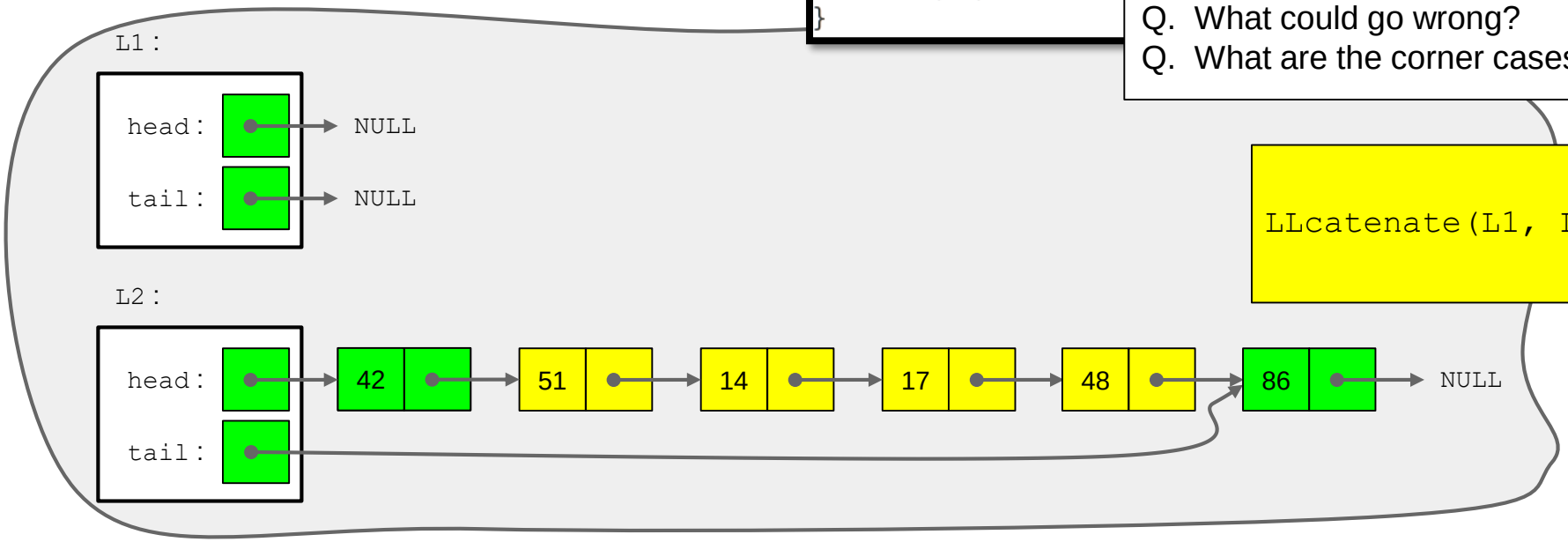
- append the elements of `L2` to
- free `L2`, i.e., only `L1` survives

Q. What's the strategy?

Q. What's the running time?

```
// joins different L1, L2 such that L1 <- L1 + L2, frees L2
void LLcatenate(LL_t * L1, LL_t * L2) {
    if (L2->head == NULL) {
        assert(L2->tail == NULL);
    } else if (L1->head == NULL) {
        assert(L1->tail == NULL);
        //L1->head = L2->head;
        //L1->tail = L2->tail;
        *L1 = *L2;
    } else {
        L1->tail->next = L2->head;
        L1->tail= L2->tail;

    free(L2);
}
```

Q. What could go wrong?
Q. What are the corner cases?



`LLcatenate(L1, L2);`

Linked List: `LLcatenate(L1, L2)`

Behaviour:

- append the elements of `L2` to
- free `L2`, i.e., only `L1` survives

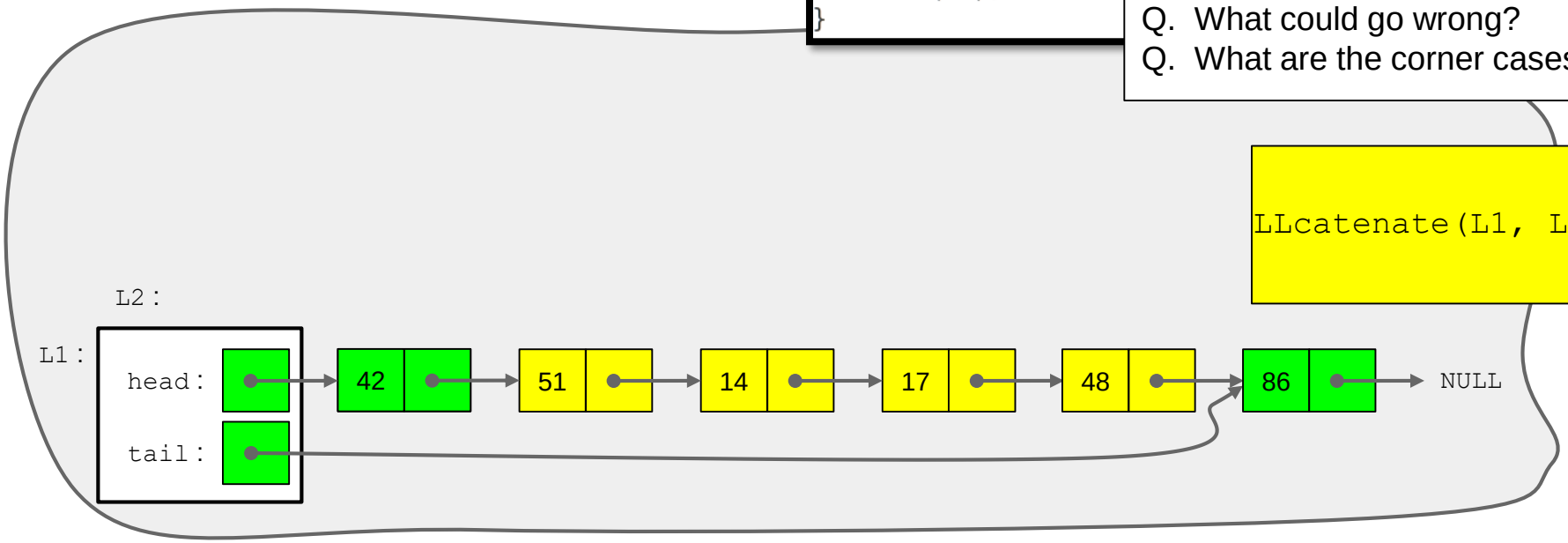
Q. What's the strategy?

Q. What's the running time?

```
// joins different L1, L2 such that L1 <- L1 + L2, frees L2
void LLcatenate(LL_t * L1, LL_t * L2) {
    if (L2->head == NULL) {
        assert(L2->tail == NULL);
    } else if (L1->head == NULL) {
        assert(L1->tail == NULL);
        //L1->head = L2->head;
        //L1->tail = L2->tail;
        *L1 = *L2;
    } else {
        L1->tail->next = L2->head;
        L1->tail= L2->tail;

    free(L2);
}
```

Q. What could go wrong?
Q. What are the corner cases?

`LLcatenate(L1, L2);`



Linked List: `LLcatenate(L1, L2)`

Behaviour:

- append the elements of `L2` to
- free `L2`, i.e., only `L1` survives

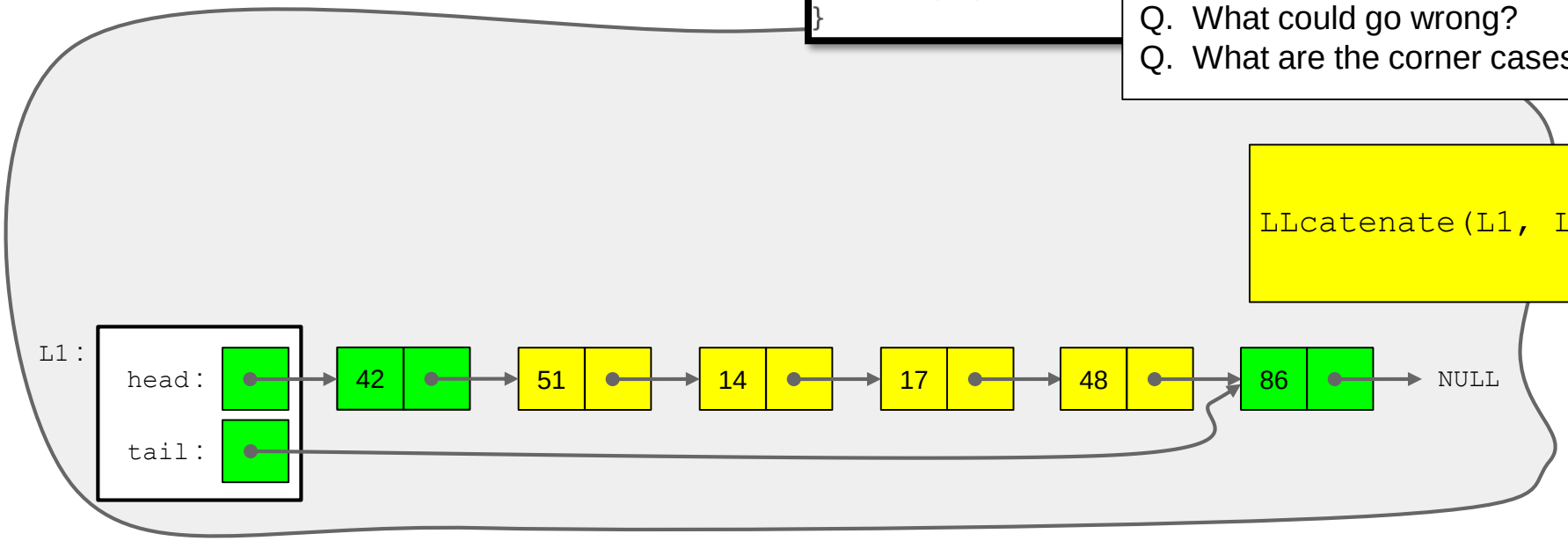
Q. What's the strategy?

Q. What's the running time?

```
// joins different L1, L2 such that L1 <- L1 + L2, frees L2
void LLcatenate(LL_t * L1, LL_t * L2) {
    if (L2->head == NULL) {
        assert(L2->tail == NULL);
    } else if (L1->head == NULL) {
        assert(L1->tail == NULL);
        //L1->head = L2->head;
        //L1->tail = L2->tail;
        *L1 = *L2;
    } else {
        L1->tail->next = L2->head;
        L1->tail= L2->tail;

    free(L2);
}
```

Q. What could go wrong?
Q. What are the corner cases?

`LLcatenate(L1, L2);`



Linked List: `LLinsert(x)`

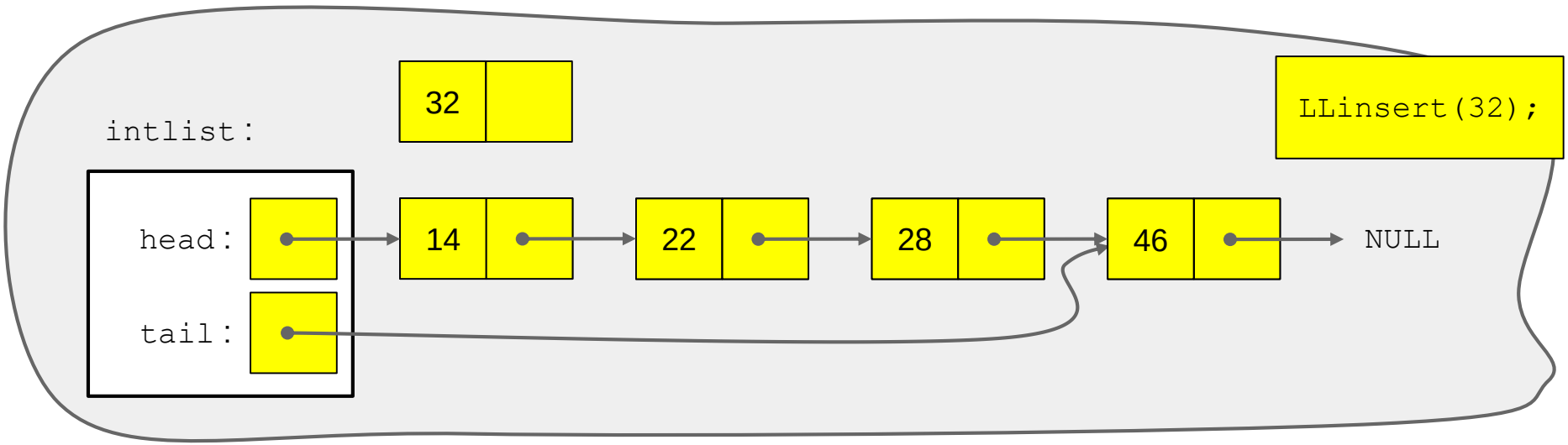
Motivation: Keep data in sorted order

- `LLinsert(x)` maintains that invariant

Strategy:

- search for the insertion point

```
curr = head->next;
while (curr->data < x) {
    curr = curr->next;

}
```



Linked List: `LLinsert(x)`

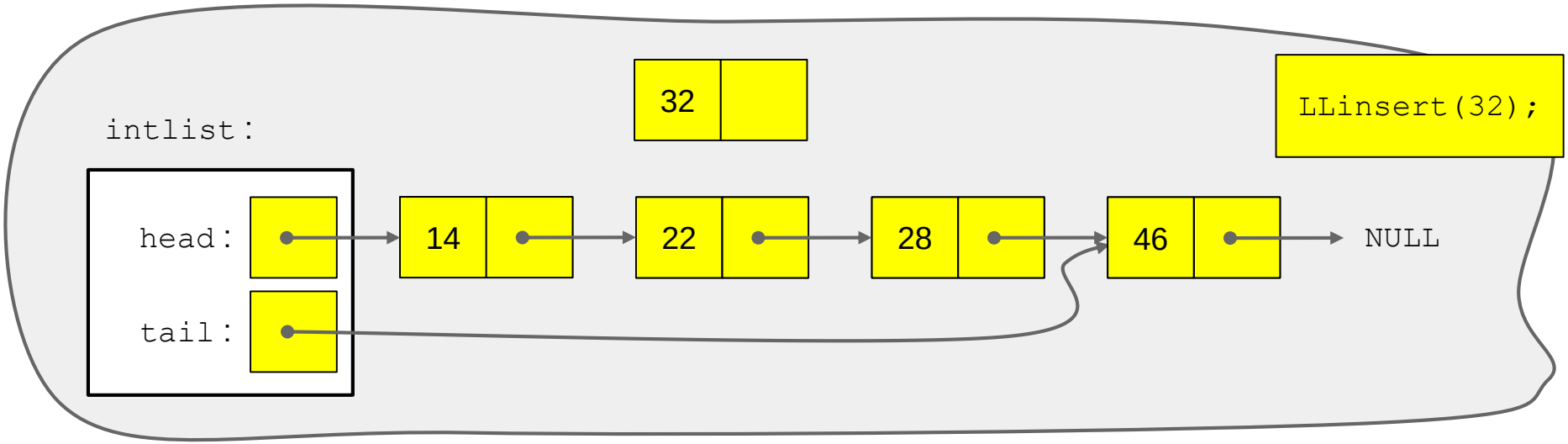
Motivation: Keep data in sorted order

- `LLinsert(x)` maintains that invariant

Strategy:

- search for the insertion point

```
curr = head->next;
while (curr->data < x) {
    curr = curr->next;

}
```



Linked List: `LLinsert(x)`

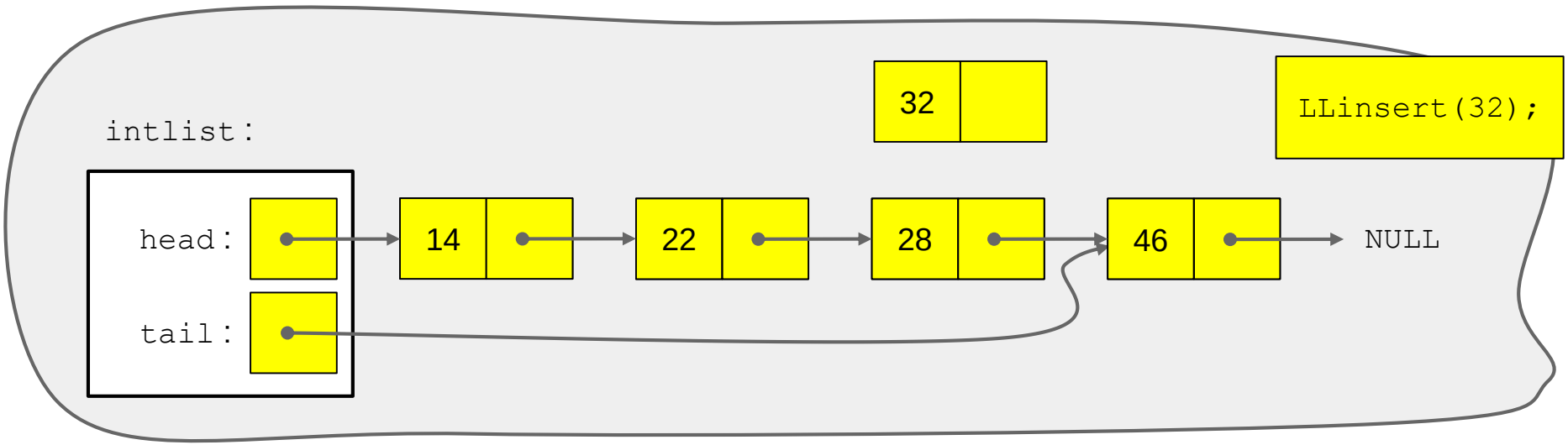
Motivation: Keep data in sorted order

- `LLinsert(x)` maintains that invariant

Strategy:

- search for the insertion point

```
curr = head->next;
while (curr->data < x) {
    curr = curr->next;

}
```



Linked List: `LLinsert(x)`

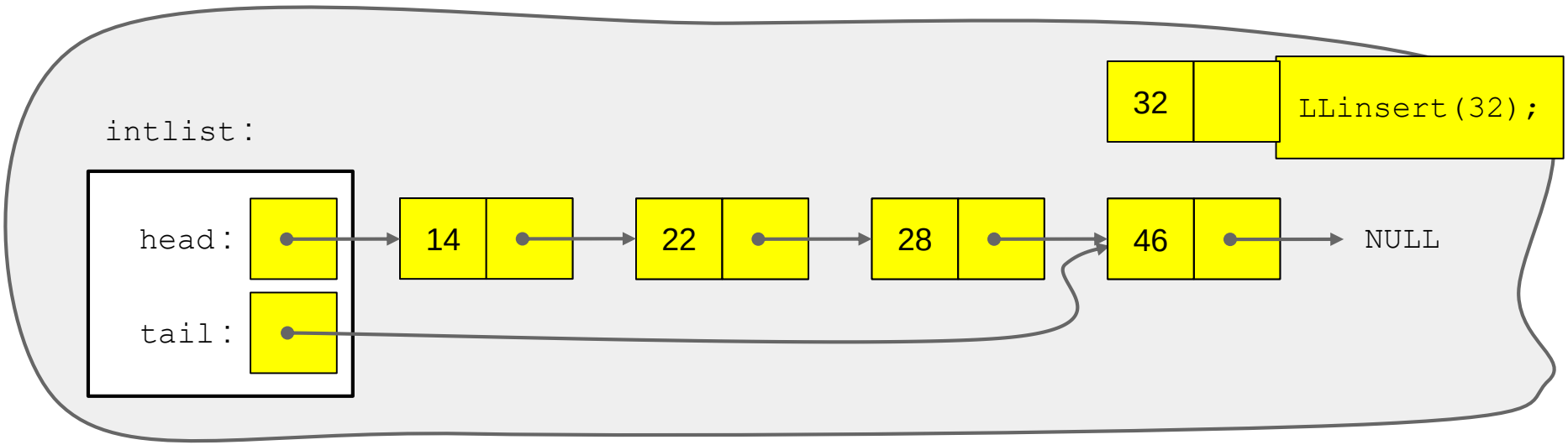
Motivation: Keep data in sorted order

- `LLinsert(x)` maintains that invariant

Strategy:

- search for the insertion point
- splice the list

```
curr = head->next;
while (curr->data < x) {
    curr = curr->next;

}
```



Linked List: `LLinsert(x)`

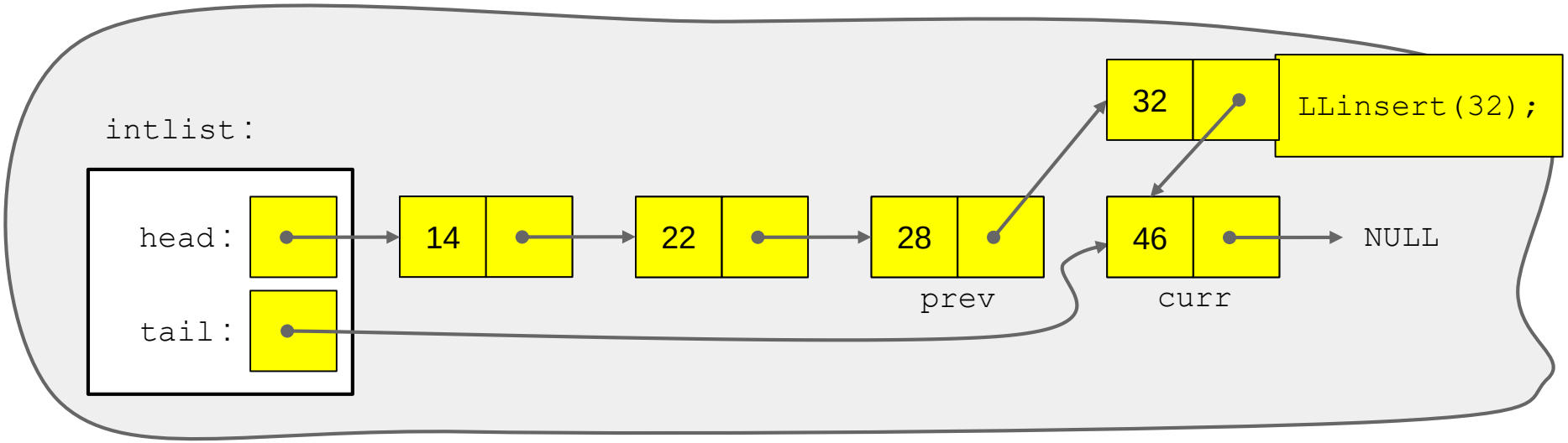
Motivation: Keep data in sorted order

- `LLinsert(x)` maintains that invariant

Strategy:

- search for the insertion point
- splice the list
  - requires the previous node

```
prev = head;
curr = head->next;
while (curr->data < x) {
    curr = curr->next;
    prev = prev->next;
}
newNode->next = curr;
prev->next = newNode;
```



Linked List: `LLinser`

Motivation: Keep data in sorted orc

- `LLinsert(x)` maintains that invarian

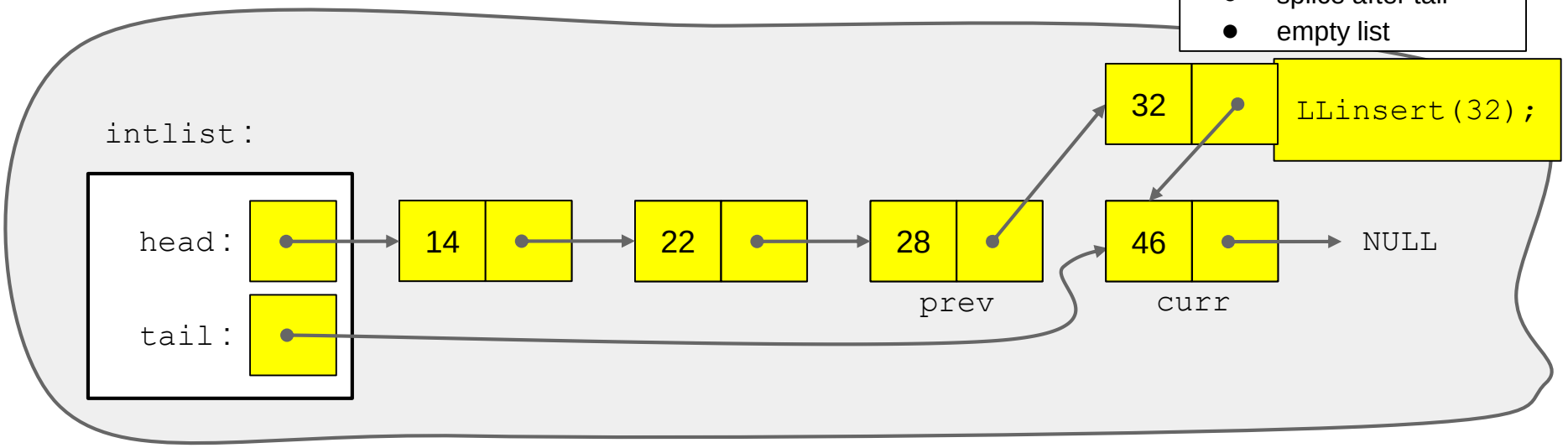
Strategy:

- search for the insertion point
- splice the list
  - requires the previous node

```
// Pre/Post:  intlist is sorted
void LLinsert(LL_t * intlist, int value) {
    node_t *newNode = malloc(sizeof(node_t));
    newNode->data = value;

    node_t * prev = intlist->head;
    node_t * curr = intlist->head->next;
    while (curr != NULL && curr->data < newNode->data) {
        curr = curr->next;
        prev = prev->next;
    }
    newNode->next = curr;
    prev->next = newNode;
    if (curr == NULL) {
        // new tail
        intlist->tail = newNode;
    }
}
```

Corner cases?

- splice before head
- splice after tail
- empty list



Linked List: `LLinser`

Motivation: Keep data in sorted orc

- `LLinsert(x)` maintains that invarian

Strategy:

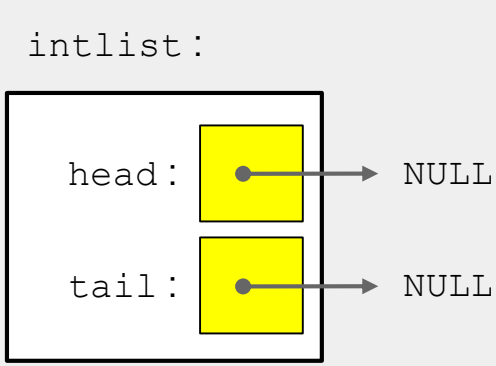
- search for the insertion point
- splice the list
  - requires the previous node

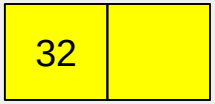
```
// Pre/Post:  intlist is sorted
void LLinsert(LL_t * intlist, int value) {
    node_t *newNode = malloc(sizeof(node_t));
    newNode->data = value;

    node_t * prev = intlist->head;
    node_t * curr = intlist->head->next;
    while (curr != NULL && curr->data < newNode->data) {
        curr = curr->next;
        prev = prev->next;
    }
    newNode->next = curr;
    prev->next = newNode;
    if (curr == NULL) {
        // new tail
        intlist->tail = newNode;
    }
}
```

Corner cases?

- splice before head
- splice after tail
- empty list

```
LLinsert(32);
```





Linked List: `LLinser`

Motivation: Keep data in sorted orc

- `LLinsert(x)` maintains that invarian

Strategy:

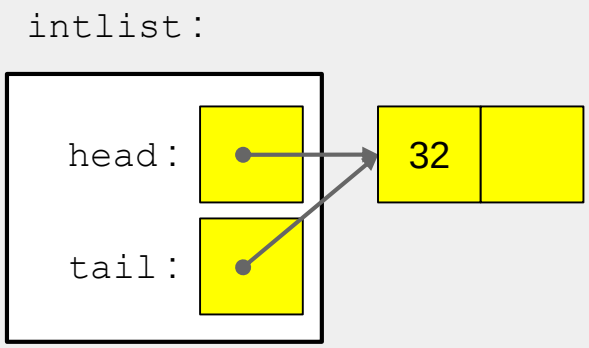
- search for the insertion point
- splice the list
  - requires the previous node

```
// Pre/Post:  intlist is sorted
void LLinsert(LL_t * intlist, int value) {
    node_t *newNode = malloc(sizeof(node_t));
    newNode->data = value;

    node_t * prev = intlist->head;
    node_t * curr = intlist->head->next;
    while (curr != NULL && curr->data < newNode->data) {
        curr = curr->next;
        prev = prev->next;
    }
    newNode->next = curr;
    prev->next = newNode;
    if (curr == NULL) {
        // new tail
        intlist->tail = newNode;
    }
}
```

Corner cases?

- splice before head
- splice after tail
- empty list

```
LLinsert(32);
```



Linked List: `LLinser`

Motivation: Keep data in sorted orc

- `LLinsert(x)` maintains that invarian

Strategy:

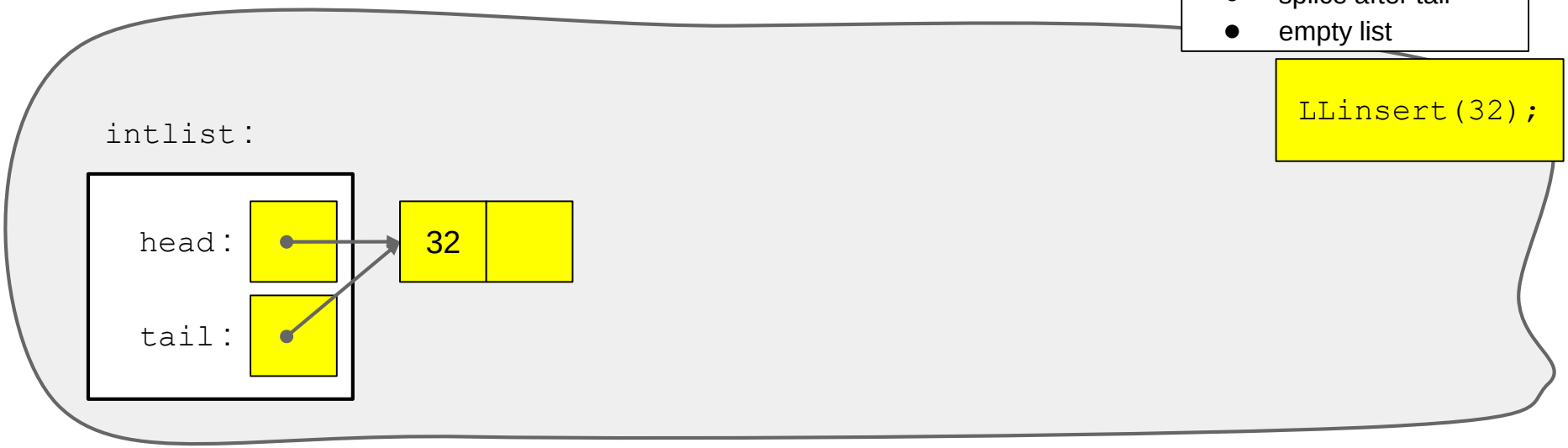
- search for the insertion point
- splice the list
  - requires the previous node

```c
// Pre/Post:  intlist is sorted
void LLinsert(LL_t * intlist, int value) {
    node_t *newNode = malloc(sizeof(node_t));
    newNode->data = value;
    // list empty
    if (intlist->head == NULL) {
        assert(intlist->tail == NULL);
        intlist->head = newNode;
        intlist->tail = newNode;
        newNode->next = NULL;



    } else {
        // typical case
        node_t * prev = intlist->head;
        node_t * curr = intlist->head->next;
        while (curr != NULL && curr->data < newNode->data) {
            curr = curr->next;
            prev = prev->next;
        }
        newNode->next = curr;
        prev->next = newNode;
        if (curr == NULL) {
            // new tail
            intlist->tail = newNode;
        }
    }
}
```

Corner cases?

- splice before head
- splice after tail
- empty list

```
LLinsert(32);
```



Linked List: `LLinser`

Motivation: Keep data in sorted orc

- `LLinsert(x)` maintains that invarian

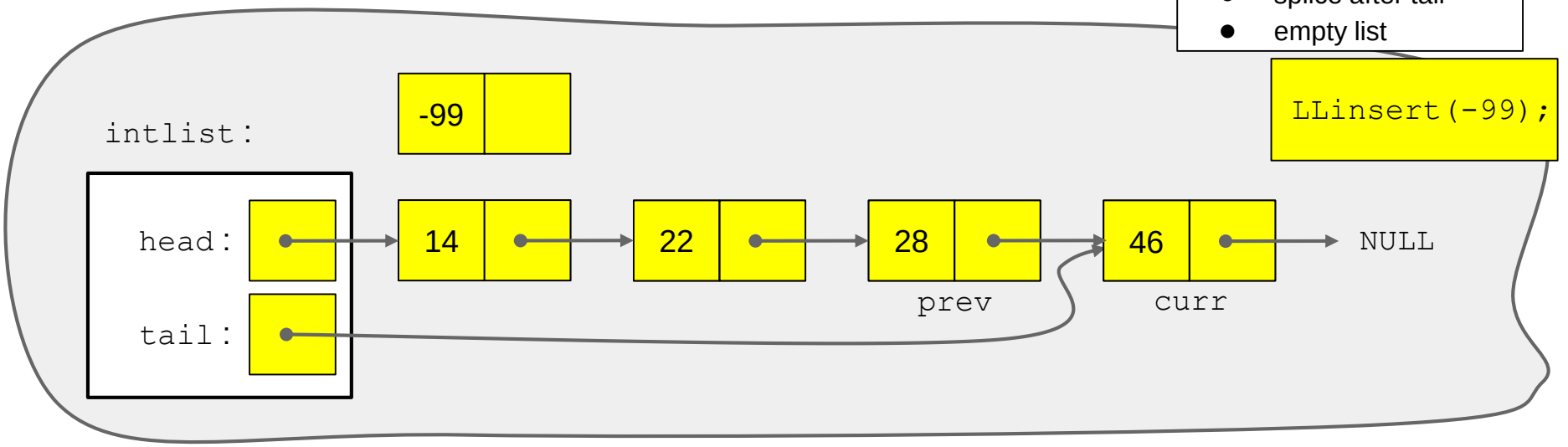
Strategy:

- search for the insertion point
- splice the list
  - requires the previous node

```
// Pre/Post:  intlist is sorted
void LLinsert(LL_t * intlist, int value) {
    node_t *newNode = malloc(sizeof(node_t));
    newNode->data = value;
    // list empty
    if (intlist->head == NULL) {
        assert(intlist->tail == NULL);
        intlist->head = newNode;
        intlist->tail = newNode;
        newNode->next = NULL;



    } else {
        // typical case
        node_t * prev = intlist->head;
        node_t * curr = intlist->head->next;
        while (curr != NULL && curr->data < newNode->data) {
            curr = curr->next;
            prev = prev->next;
        }
        newNode->next = curr;
        prev->next = newNode;
        if (curr == NULL) {
            // new tail
            intlist->tail = newNode;
        }
    }
}
```

Corner cases?

- splice before head
- splice after tail
- empty list



Linked List: `LLinser`

Motivation: Keep data in sorted orc

- `LLinsert(x)` maintains that invarian

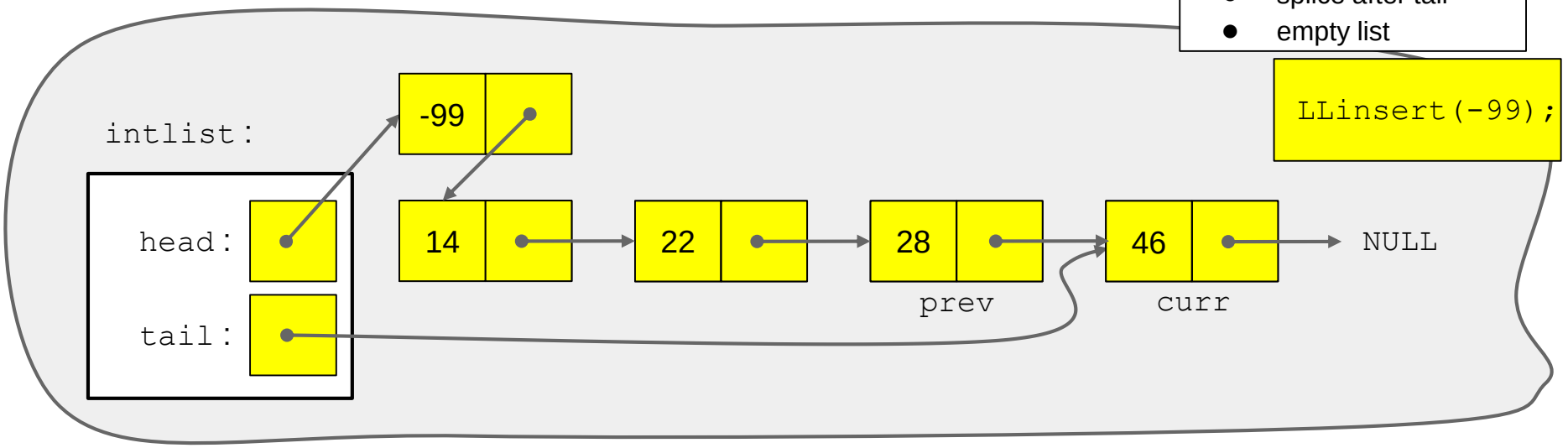
Strategy:

- search for the insertion point
- splice the list
  - requires the previous node

```
// Pre/Post:  intlist is sorted
void LLinsert(LL_t * intlist, int value) {
    node_t *newNode = malloc(sizeof(node_t));
    newNode->data = value;
    // list empty
    if (intlist->head == NULL) {
        assert(intlist->tail == NULL);
        intlist->head = newNode;
        intlist->tail = newNode;
        newNode->next = NULL;



    } else {
        // typical case
        node_t * prev = intlist->head;
        node_t * curr = intlist->head->next;
        while (curr != NULL && curr->data < newNode->data) {
            curr = curr->next;
            prev = prev->next;
        }
        newNode->next = curr;
        prev->next = newNode;
        if (curr == NULL) {
            // new tail
            intlist->tail = newNode;
        }
    }
}
```

Corner cases?

- splice before head
- splice after tail
- empty list



Linked List: `LLinser`

Motivation: Keep data in sorted orc

- `LLinsert(x)` maintains that invarian

Strategy:

- search for the insertion point
- splice the list
  - requires the previous node

```
// Pre/Post:  intlist is sorted
void LLinsert(LL_t * intlist, int value) {
    node_t *newNode = malloc(sizeof(node_t));
    newNode->data = value;
    // list empty
    if (intlist->head == NULL) {
        assert(intlist->tail == NULL);
        intlist->head = newNode;
        intlist->tail = newNode;
        newNode->next = NULL;
    } else if (intlist->head->data >= newNode->data) {
        // new head
        newNode->next = intlist->head;
        intlist->head = newNode;
    } else {
        // typical case
        node_t * prev = intlist->head;
        node_t * curr = intlist->head->next;
        while (curr != NULL && curr->data < newNode->data) {
            curr = curr->next;
            prev = prev->next;
        }
        newNode->next = curr;
        prev->next = newNode;
        if (curr == NULL) {
            // new tail
            intlist->tail = newNode;
        }
    }
}
```

Corner cases?

- splice before head
- splice after tail
- empty list