Linked Lists

Towards Queues

How

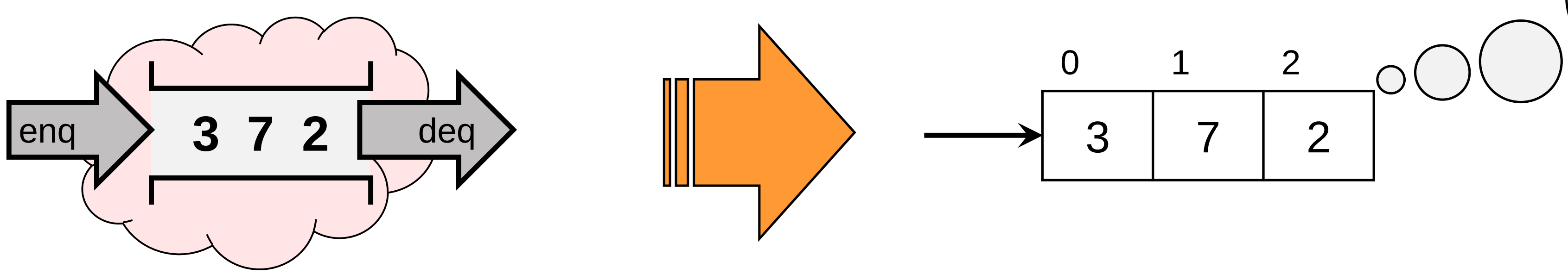
- We want to **implement** the queue library
  - So far we only wrote *client code* using its interface
- A queue stores a bunch of elements of the same type
  - Idea: represent a queue as an array

say int for a change

What

Queue Interface

```
// typedef ______* queue_t;

bool queue_empty(queue_t S)     // O(1)
  /*@requires S != NULL;           @*/ ;

queue_t  queue_new()             // O(1)
  /*@ensures \result != NULL;      @*/
  /*@ensures queue_empty(\result); @*/ ;

void enq(queue_t S, int x)       // O(1)
  /*@requires S != NULL;           @*/
  /*@ensures !queue_empty(S);      @*/ ;

int deq(queue_t S)               // O(1)
  /*@requires S != NULL;           @*/
  /*@requires !queue_empty(S);     @*/ ;
```




```
// Implementation-side type
struct queue_header {
  int[] data;
};
typedef struct queue_header queue;

// Client type
typedef queue* queue_t;
```

- But …
  - arrays have fixed length yet queues are unbounded
  - how would we add and remove elements?
  - can we achieve the complexity goals?

create new arrays each time?

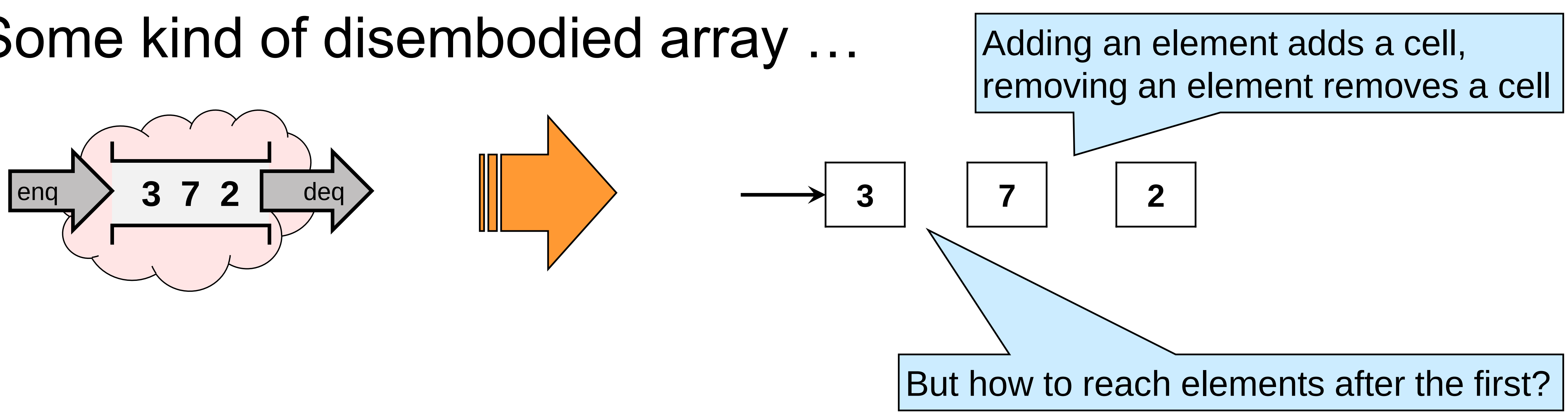
where is the front? the back?

move elements around?

Toward Queues

```
Queue Interface
// typedef ______* queue_t;

bool queue_empty(queue_t S)    // O(1)
  /*@requires S != NULL;          @*/ ;

queue_t  queue_new()            // O(1)
  /*@ensures \result != NULL;         @*/
  /*@ensures queue_empty(\result); @*/ ;

void enq(queue_t S, int x)     // O(1)
  /*@requires S != NULL;          @*/
  /*@ensures !queue_empty(S);     @*/ ;

int deq(queue_t S)             // O(1)
  /*@requires S != NULL;          @*/
  /*@requires !queue_empty(S);    @*/ ;
```

- *A queue stores a bunch of elements of the same type*
  - *Represent a queue as an array*
- We want something like an array but where
  - we can add/remove elements at the beginning and end
  - have it grow and shrink as needed
- Some kind of disembodied array …

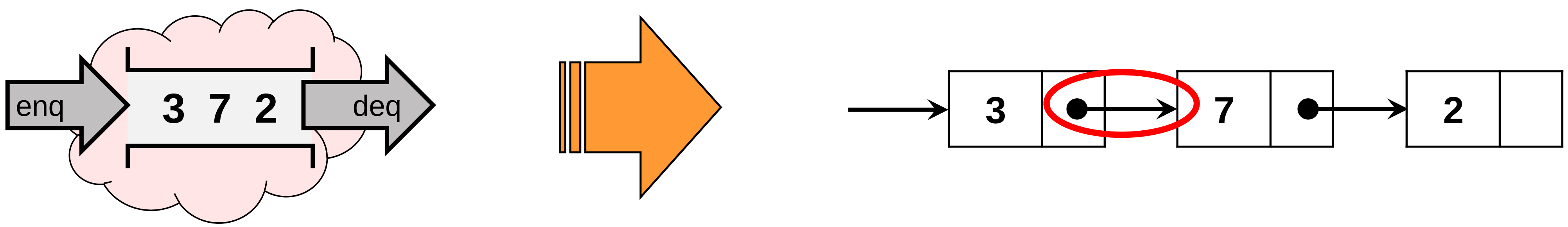


Queue Interface

```
// typedef ______* queue_t;

bool queue_empty(queue_t S)    // O(1)
  /*@requires S != NULL;             @*/ ;

queue_t  queue_new()              // O(1)
  /*@ensures \result != NULL;         @*/
  /*@ensures queue_empty(\result); @*/ ;

void enq(queue_t S, int x)        // O(1)
  /*@requires S != NULL;             @*/
  /*@ensures !queue_empty(S);       @*/ ;

int deq(queue_t S)                // O(1)
  /*@requires S != NULL;             @*/
  /*@requires !queue_empty(S);      @*/ ;
```

Toward Queues

- *A disembodied array*
  - *how to reach the elements after the first?*
- Use pointers to go to the next element

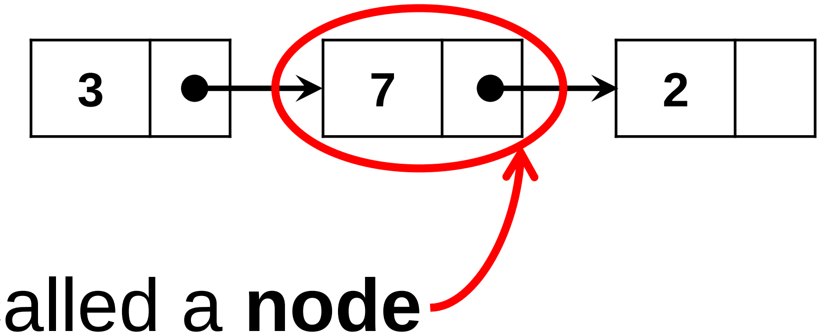


- This is called a **linked list**

Linked Lists

Lists of Nodes

- Linked lists use pointers to go to the next element



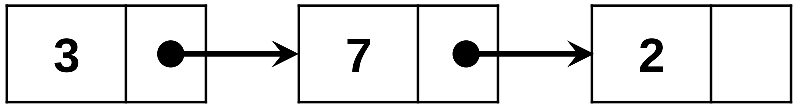
  - each block is called a **node**

Let's implement it:

- a node consists of
  - a data element
  - a pointer to the next node


```
struct list_node {
  int data;
 struct list_node* next;
};
```

- The whole list is a pointer to its first node

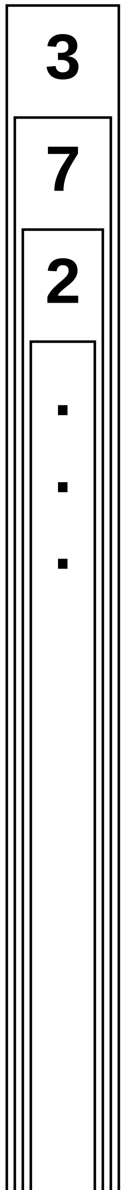


Lists of Nodes

```
struct list_node {
  int data;
 struct list_node* next;
};
```

- Linked lists are a **recursive type**
  - a struct list_node is defined in terms of itself
- What if we don’t have this pointer?

  a node that contains an int and
  a node that contains an int and
  a node that contains an int and
  …
  - It would take an *infinite amount of memory!*
  - The C0 compiler disallows this
    - recursion can only occur behind a pointer (or an array)



Lists of Nodes

```
struct list_node {
  int data;
 struct list_node* next;
};
```

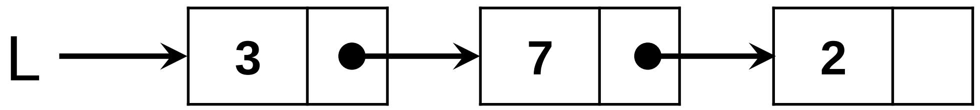
- Let’s make it more readable

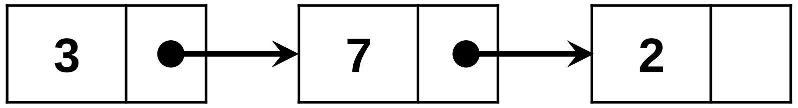
```
typedef struct list_node list;  // ADDED

struct list_node {
  int data;
  list* next;                    // MODIFIED
};
```

This must go before the struct

- Implementing this linked list

```
list* L = alloc(list);
L->data = 3;
L->next = alloc(list);
L->next->data = 7;
L->next->next = alloc(list);
L->next->next->data = 2;
```

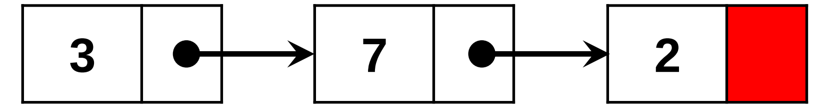




Lists of Nodes

- Does this help us implement queues?
  - Linked lists can be arbitrarily large or small
    - use just the nodes we need
    - size is not fixed like arrays
  - It's easy to insert an element at the beginning
    - allocate a new node and point its next field to the list
  - In fact, it's easy to insert an element between any two nodes
    - allocate a new node and move pointers around
- What about inserting an element at the end?
  - How do we indicate the end of a linked list?

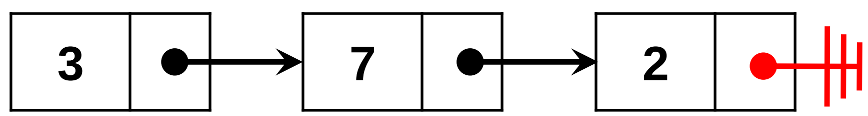
> So far we just drew an empty box …



The End of a List

We need to make the pointer in the last node **special**

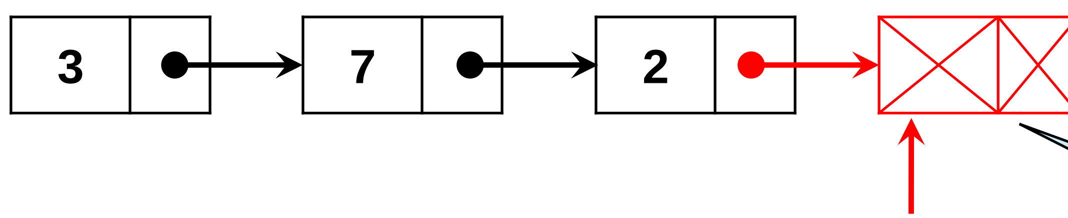
- Use the NULL pointer



This is a great idea if we don't need direct access to the end of the list

  - This is a **NULL-terminated list**

- Point it to a special node we keep track of somewhere

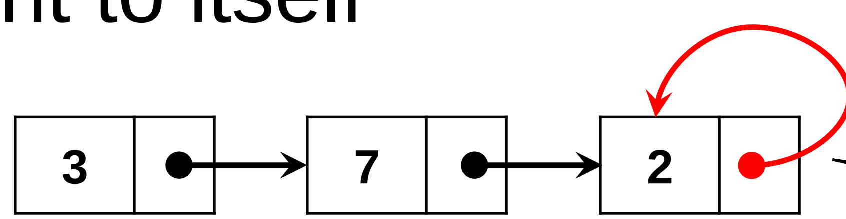


This is a great idea if we **do** need direct access to the end of the list

  - We know we reached the end of the list if its next field is equal to the address of the dummy node

This node is called the **dummy node** or the **sentinel**

- Have it point to itself

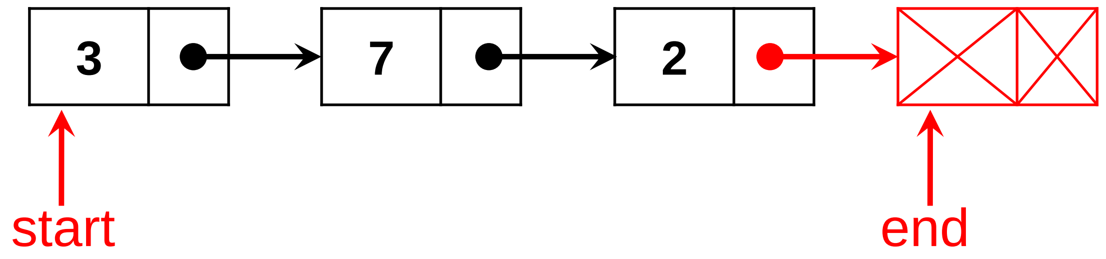


This works too, but nobody does that

List Segments

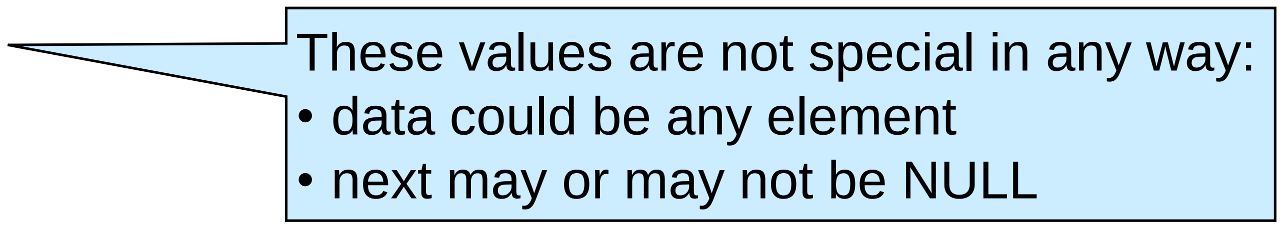
Lists with a Dummy Node

- We need to keep track of *two* pointers



  - **start**: where the first node is
  - **end**: the address in the next field of the last node
    - the address of the dummy node

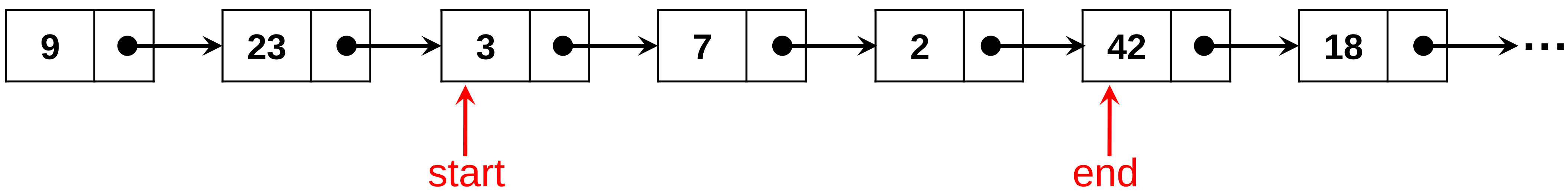
- What's in the dummy node?
  - some values that are unimportant to us
    - some number and some pointer



- A dummy value is a value we **don't care** what it is

List Segments

- There may be more nodes before and after



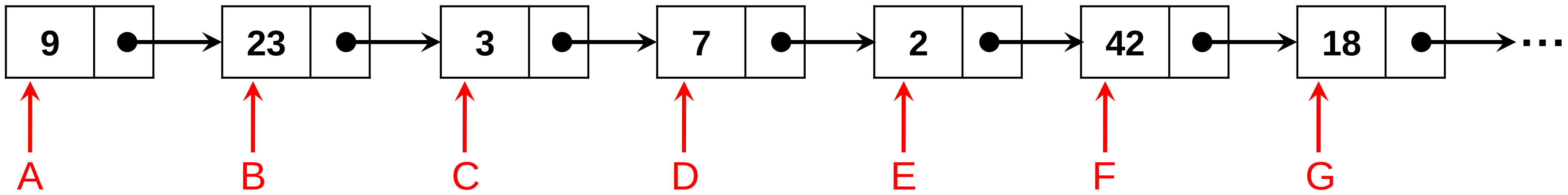
  - The pair of pointers start and end identify our list exactly
    - start is **inclusive** (the first node of the list)
    - end is **exclusive** (one past the last node of the list)

points to the dummy node

  - They identify the **list segment** [start, end)
    - here it contain values 3, 7 and 2
    - similar to array segments A[lo, hi)

List Segments

- There are many list segments in a list



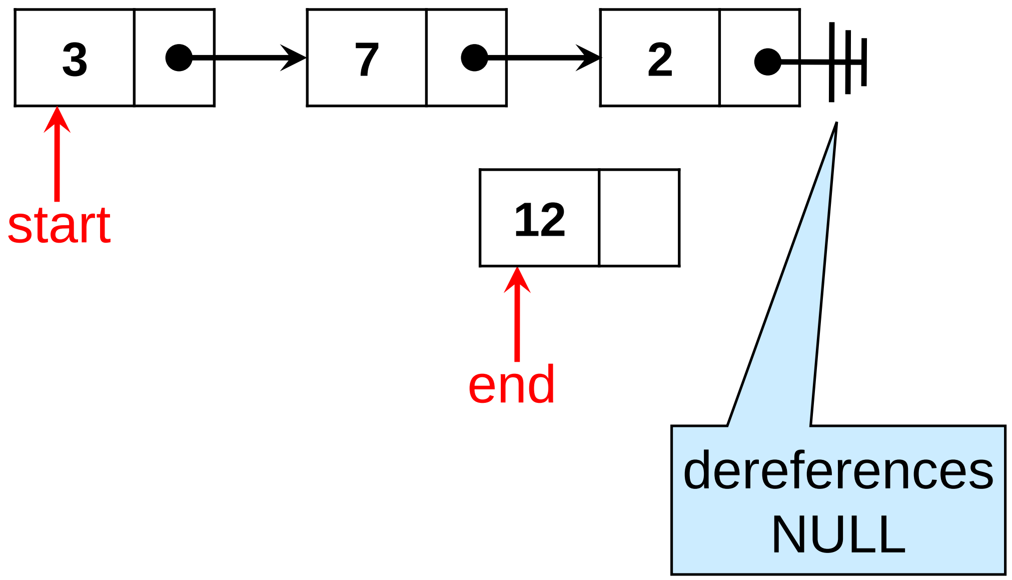
  - The list segment [C, F) contains elements 3, 7, 2
    - its dummy node contains 42 and the pointer G
  - The list segment [A, G) contains 9, 23, 3, 7, 2, 42
    - its dummy node contains 18 and the some pointer
  - The list segment [B, D) contains 23, 3
    - its dummy node contains 7 and the pointer E
  - The list segment [C, C) contains no elements
    - its dummy node contains 3 and the pointer D
    - this is the **empty segment**
    - any segment where start is the same as end
      - [A, A), [B, B), …

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

Checking for List Segments

- We want to write a specification function that checks that two pointers start and end form a list segment
  - Follow the next pointer from start until we reach end

```
bool is_segment(list* start, list* end) {
  list* l = start;
  while (l != end) {
    l = l->next;
  }
  return true;
}
```



- Does this work?
  - the dereference l->next may not be safe
    - we need NULL-checks!
  - we never return false

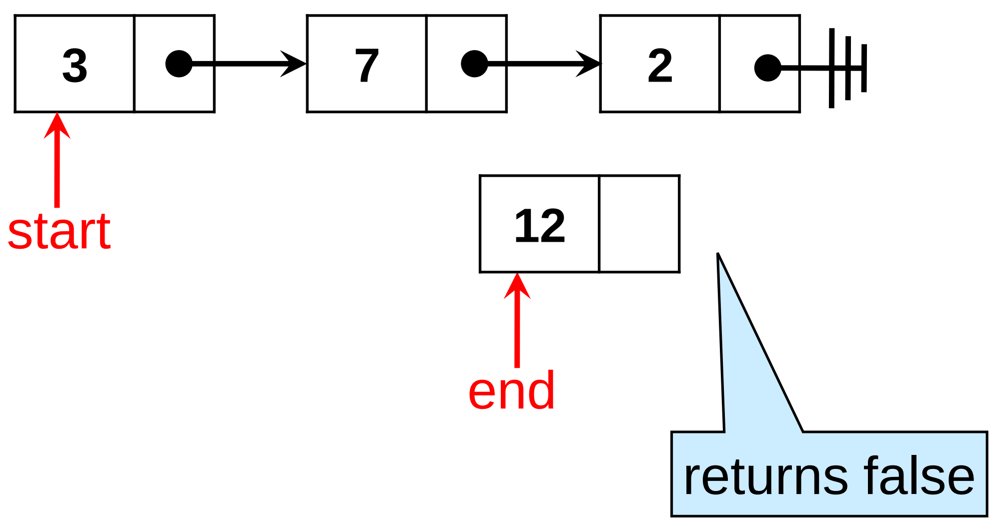
✘

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

Checking for List Segments

- *We want to write a specification function that checks that two pointers start and end form a list segment*
  - *Follow the next pointer from start until we reach end*

```
bool is_segment(list* start, list* end) {
  list* l = start;
  while (l != NULL) {            // MODIFIED
    if (l == end) return true;   // ADDED
    l = l->next;
  }
  return false;                  // MODIFIED
}
```



- Does this work?
  - ➢ if there is a list segment from start to end, it will return true
  - ➢ if it returns false, there is no list segment from start to end
- It works then …

Checking for List Segments

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

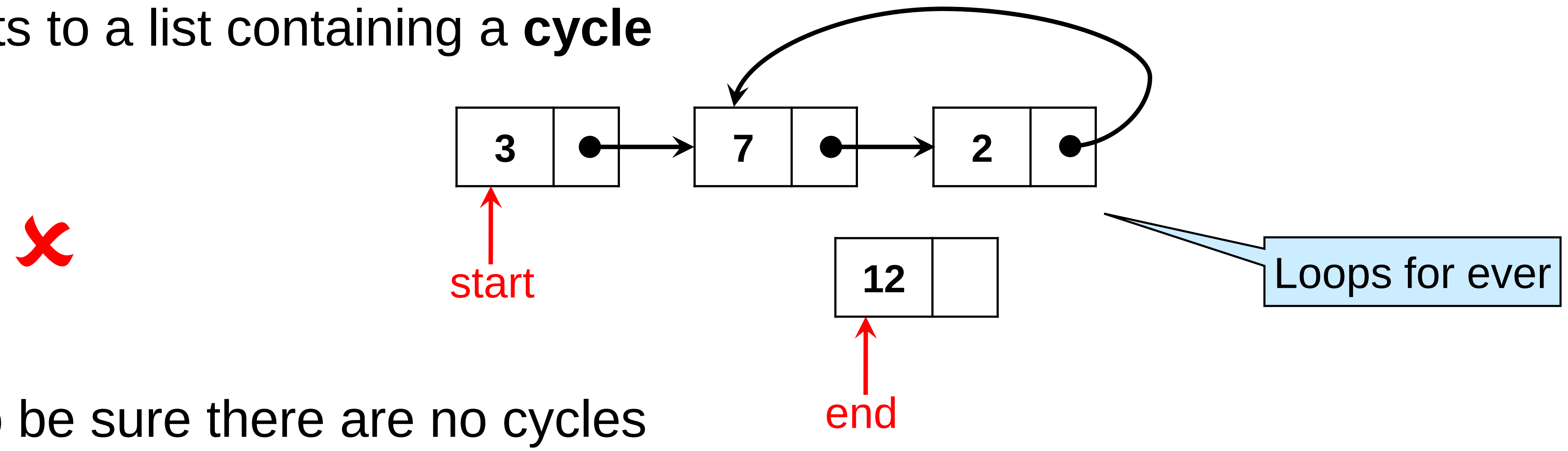
- A function that checks that start and end form a list segment

```
bool is_segment(list* start, list* end) {
  list* l = start;
  while (l != NULL) {
    if (l == end) return true;
    l = l->next;
  }
  return false;
}
```

- *if there is a list segment from start to end, it will return true*
- *if it returns false, there is no list segment from start to end*

○ Can there be no list segment but it does not return false

- if start points to a list containing a **cycle**



- We need to be sure there are no cycles

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

Checking for List Segments

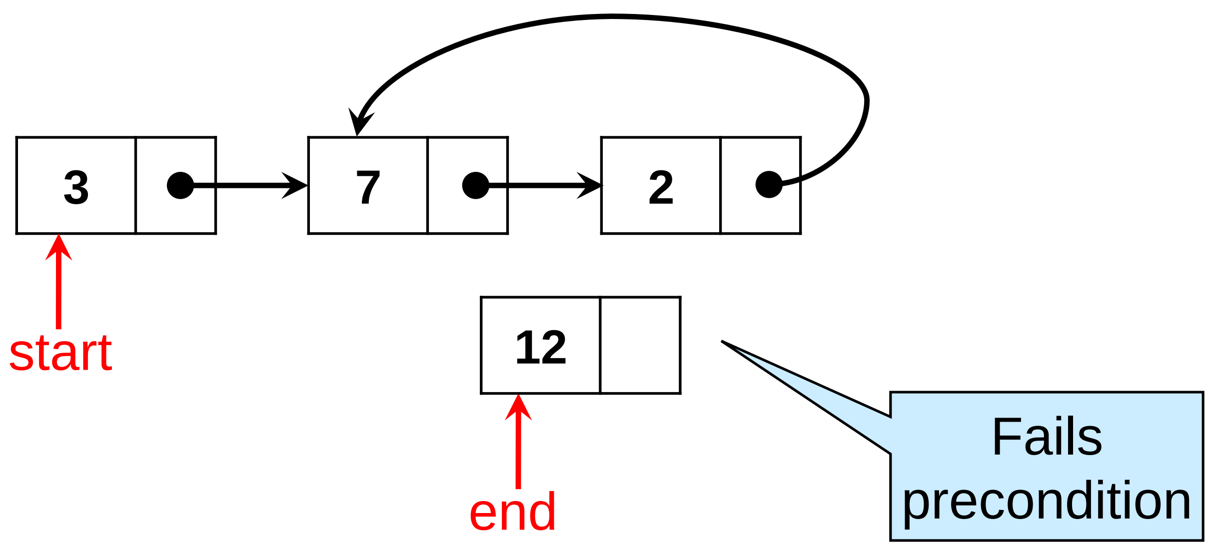
- A function that checks that start and end form a list segment
  - We need to be sure there are no cycles

```
bool is_segment(list* start, list* end)
//@requires is_acyclic(start);     // ADDED
{
  list* l = start;
  while (l != NULL) {
    if (l == end) return true;
    l = l->next;
  }
  return false;
}
```

We will implement it later

  - Does this work?
    - Yes!

✓



```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

Checking for List Segments

- A function that checks that start and end form a list segment

```
bool is_segment(list* start, list* end)
//@requires is_acyclic(start);
{
  list* l = start;
  while (l != NULL) {
    if (l == end) return true;
    l = l->next;
  }
  return false;
}
```

  - Notes:
    - returns false if start == NULL
    - or if end == NULL
      - NULL is not a pointer to a list node
      - subsumes NULL-check for both start and end

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

Checking for List Segments

- We can also write it more succinctly
  - using a for loop

```
bool is_segment(list* start, list* end)
//@requires is_acyclic(start);
{
  for (list* l = start; l != NULL; l = l->next) {
    if (l == end) return true;
  }
  return false;
}
```

  - recursively

```
bool is_segment(list* start, list* end)
//@requires is_acyclic(start);
{
  if (start == NULL) return false;
  return start == end
         || is_segment(start->next, end);
}
```

All 3 versions are equivalent

Detecting Cycles

- How to check if a list is cyclic?
  - Use a counter and look for overflows
    - very inefficient!
    - also, C0 pointers are 64 bits but ints are 32 bits

      In C0, there are more pointers than integers!
  - Keep track of visited nodes somewhere
    - in an array?

      how big to make it?

      array indices are 32 bits
    - in another list?

      how do we check it has no cycles?
  - Add a “visited” field to the nodes (a boolean)
    - we need to know the list is acyclic to initialize it to false!
  - What then?

Detecting Cycles

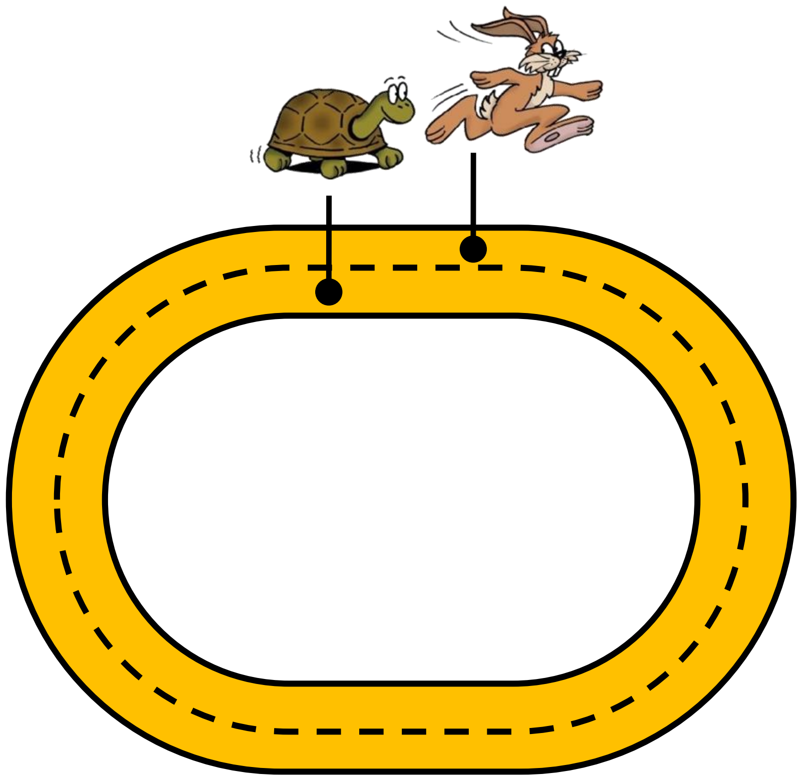
- The tortoise and hare algorithm

by this dude

Robert W. Floyd

  - Traverse the list using two pointers
    - the tortoise starts at the beginning and moves by 1 step
    - the hare starts just ahead of the tortoise and moves by 2 steps
  - If the hare ever overtakes the tortoise, there is a cycle

```
bool is_acyclic(list* start) {
  if (start == NULL) return true;
  list* t = start;              // tortoise
  list* h = start->next;        // hare
  while (h != t) {
    if (h == NULL || h->next == NULL) return true;
    //@assert t != NULL;  // hare hits NULL quicker
    t = t->next;              // tortoise moves by 1 step
    h = h->next->next;        // hare moves by 2 steps
  }
  //@assert h == t;           // hare has overtaken tortoise
  return false;
}
```



Detecting Cycles

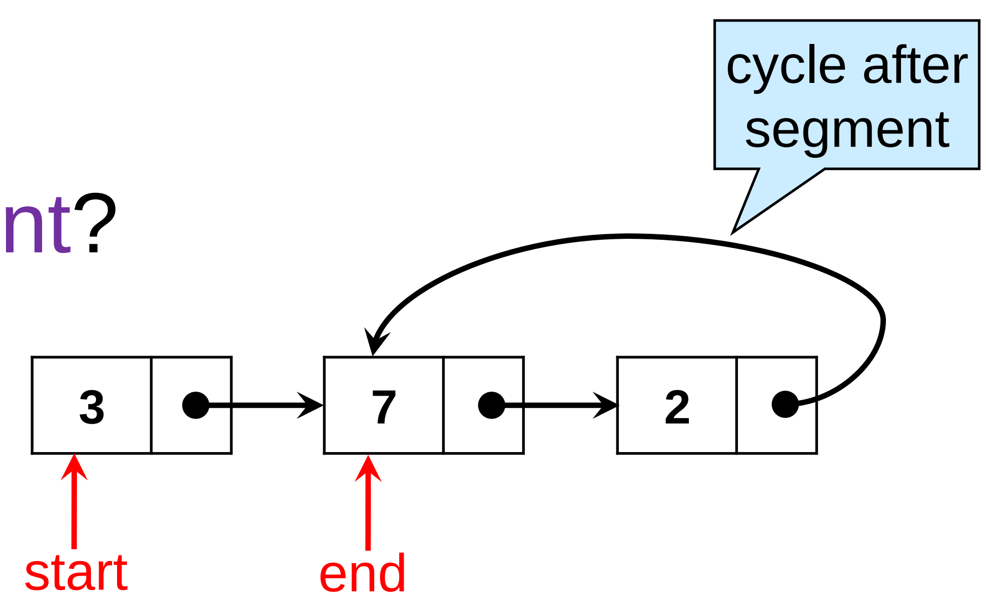
- The tortoise and hare algorithm

```
bool is_acyclic(list* start) {
  if (start == NULL) return true;
  list* t = start;            // tortoise
  list* h = start->next;      // hare
  while (h != t) {
    if (h == NULL || h->next == NULL) return true;
    //@assert t != NULL;  // hare hits NULL quicker
    t = t->next;              // tortoise moves by 1 step
    h = h->next->next;        // hare moves by 2 steps
  }
  //@assert h == t;           // hare has overtaken tortoise
  return false;
}
```

  - Returns
    - true if there is no cycle
    - false if there is a cycle

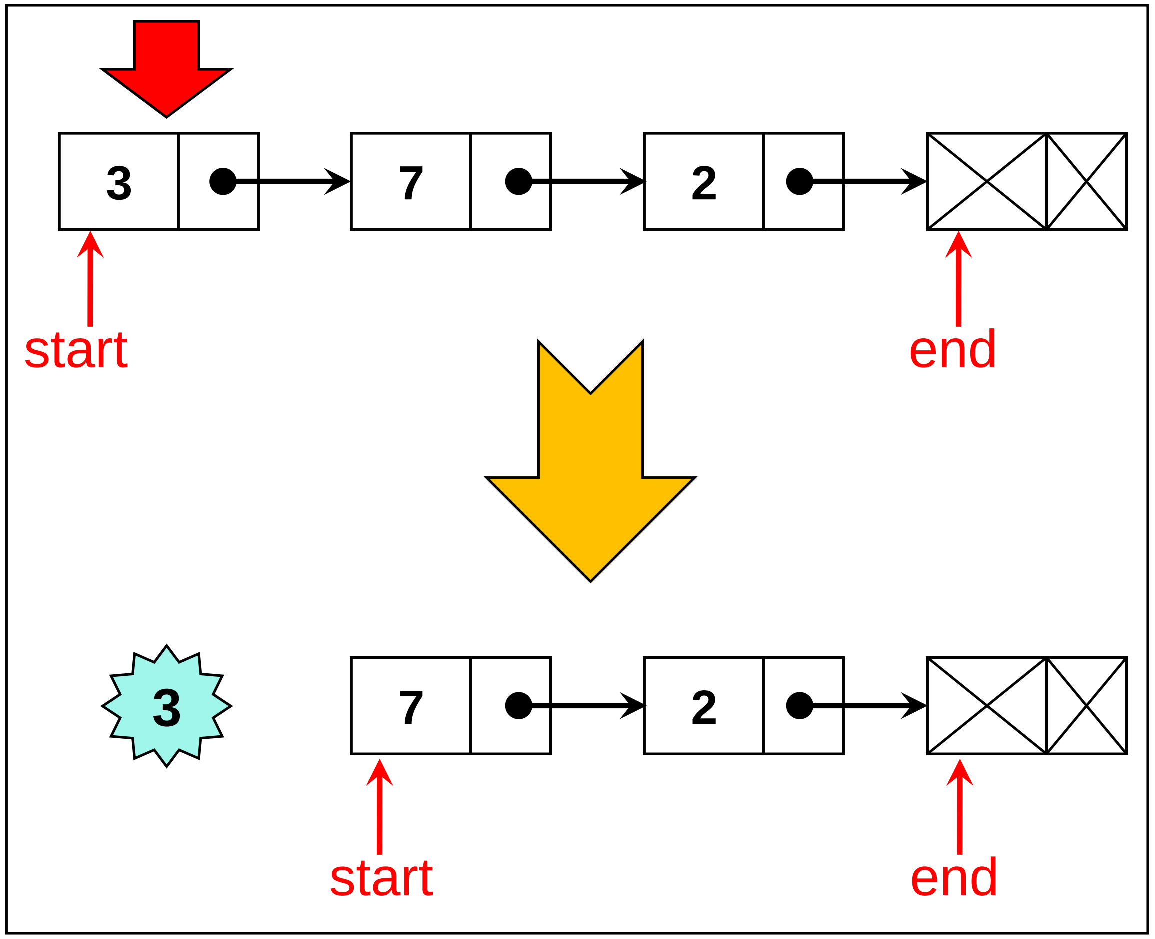
- Does it fix our problem with is_segment?
  - Too aggressive
  - *Exercise: fix it!*

Hint*: you need to account for end*



Manipulating List Segments

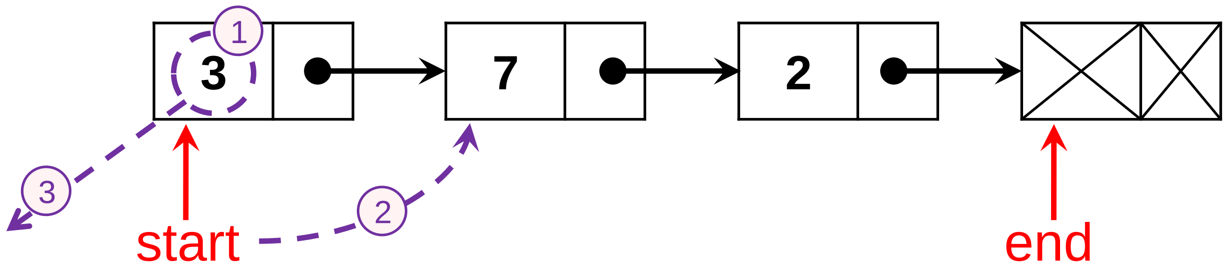
Deleting an Element



- How do we remove the node **at the beginning** of a non-empty list segment [start, end)?
  - and return the value in there

1. grab the value in the start node
2. move start to point to the next node
3. return the value

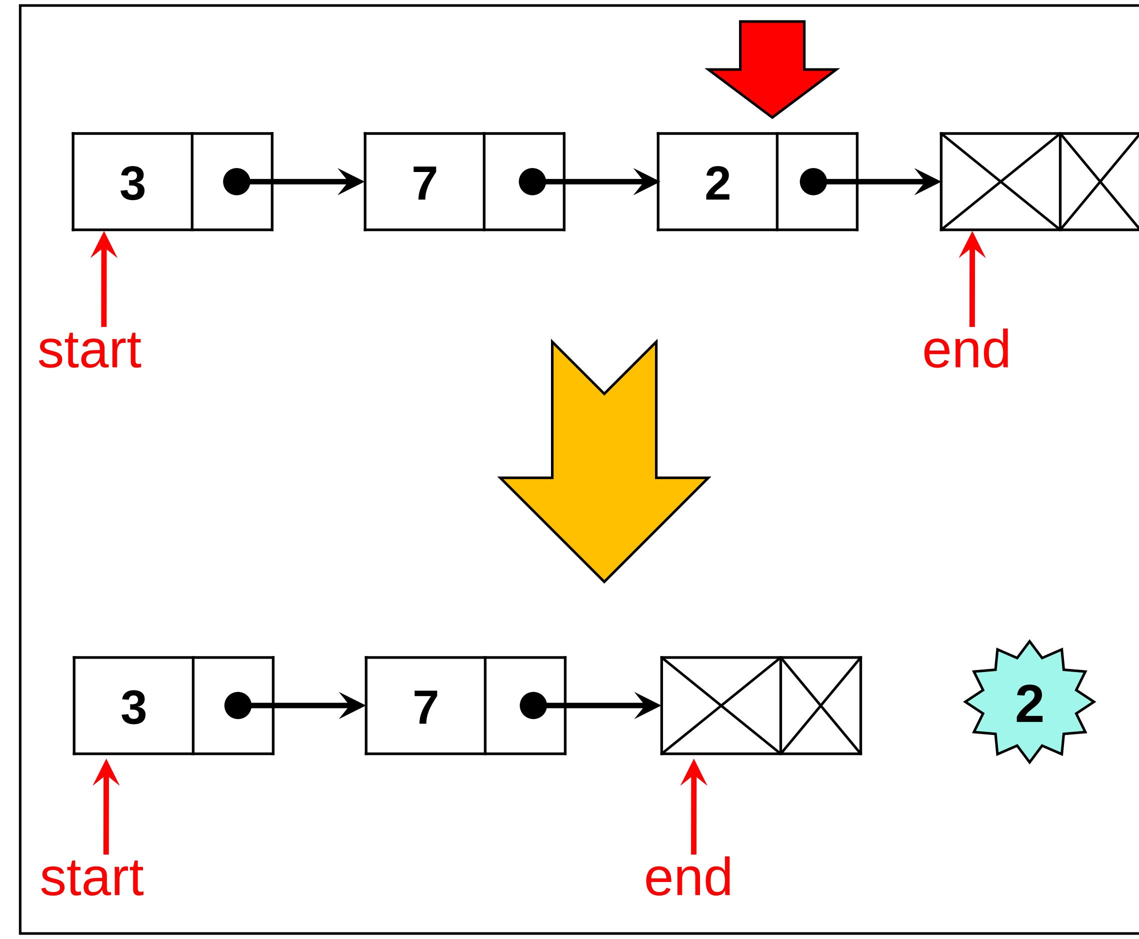
```
① int x = start->data;
② start = start->next;
③ return x;
```



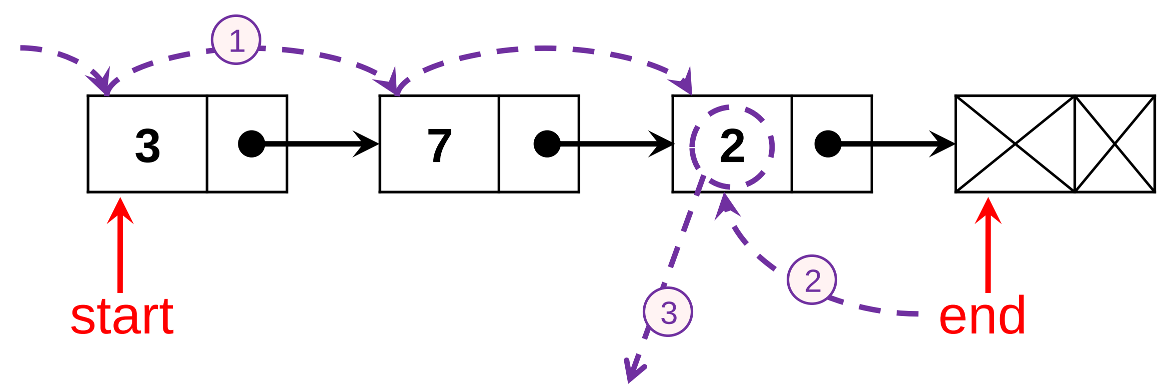
- **Complexity**: O(1)

Note: we are not "deleting" the node, just making the segment shorter

Deleting an Element



- How do we remove the **last** node of a non-empty list segment [start, end)?
  - and return the value in there
  - we must go from start
    - end is one node too far
  1. follow next until just before end
  2. move end to that node
  3. return its value



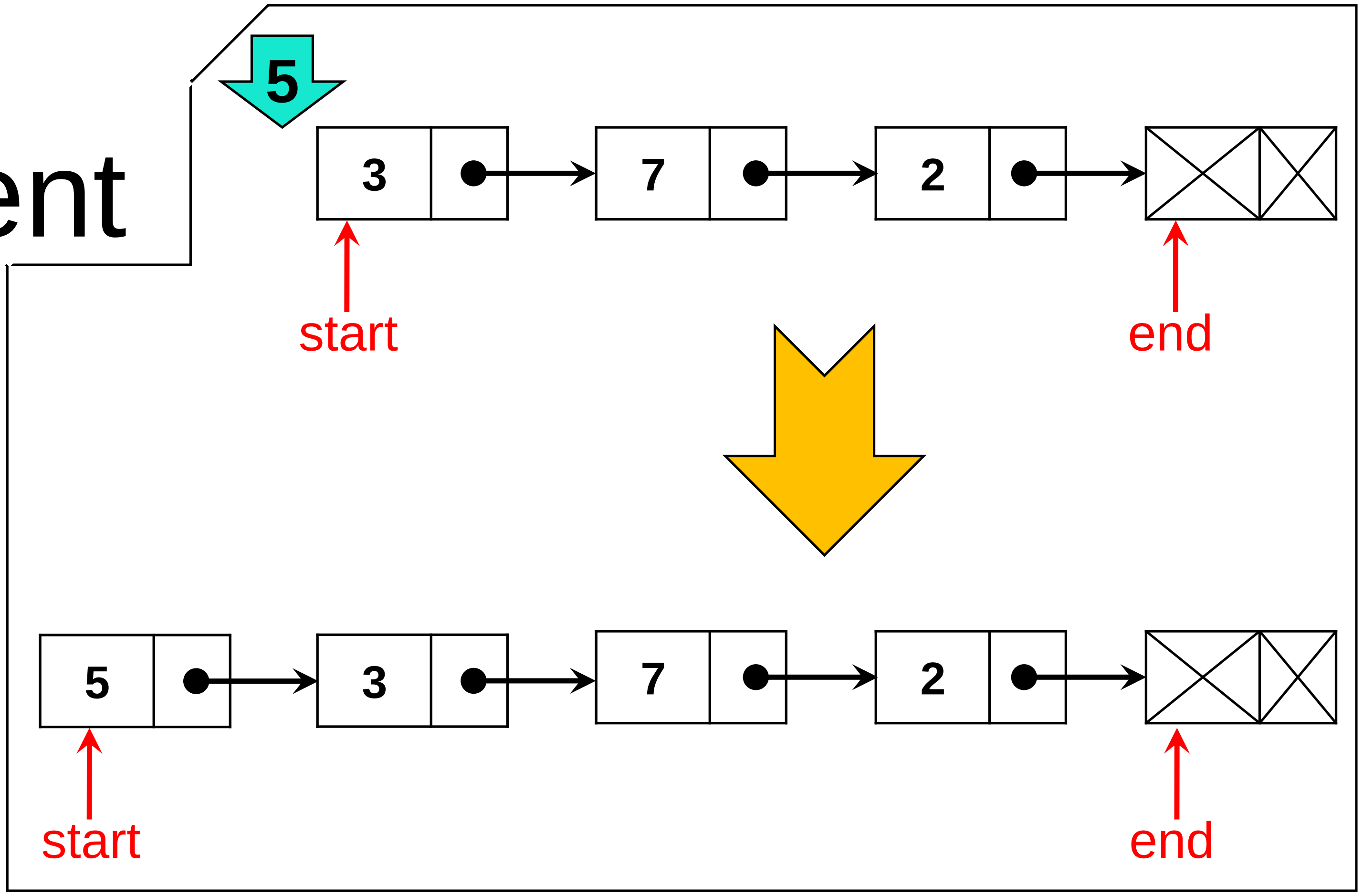

```
(1) list* l = start;
    while (l->next != end)
      l = l->next;
(2) end = l;
(3) return l->data;
```

- **Complexity**: O(n)

Expensive!

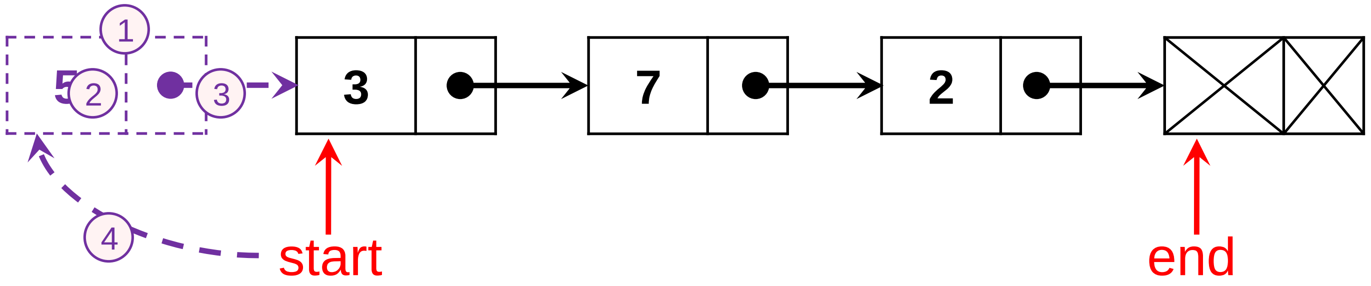
Notes:
- The old last node becomes the new dummy node
- We are not “deleting” anything, just making the segment shorter

Inserting an Element



- How do we add a node **at the beginning** of a list segment [start, end)?

  1. create a new node
  2. set its data field to the value to add
  3. set its next field to start
  4. set start to it

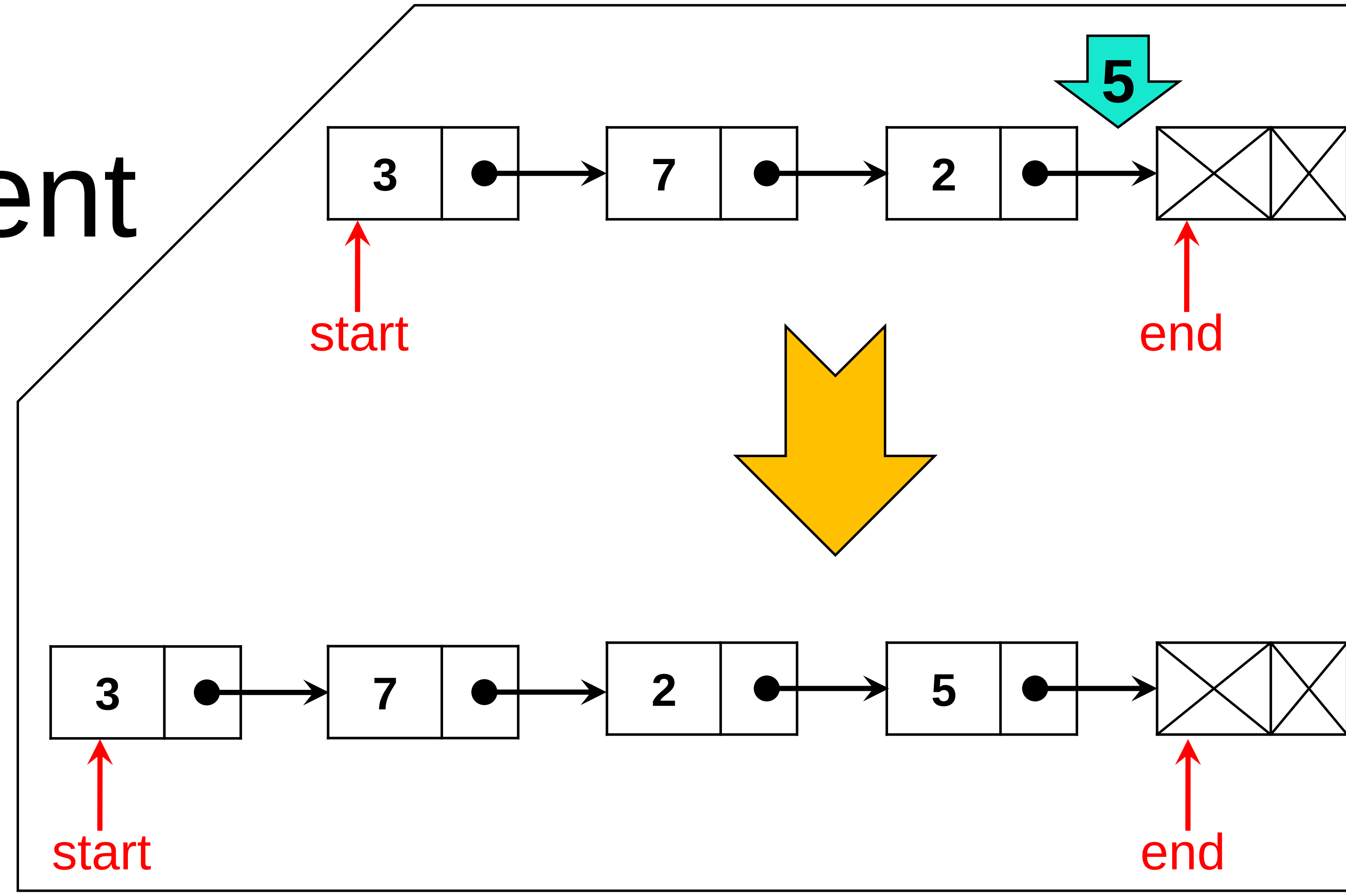



```
① list* l = alloc(list);
② l->data = x;
③ l->next = start;
④ start = l;
```

Note: we are adding a brand new node

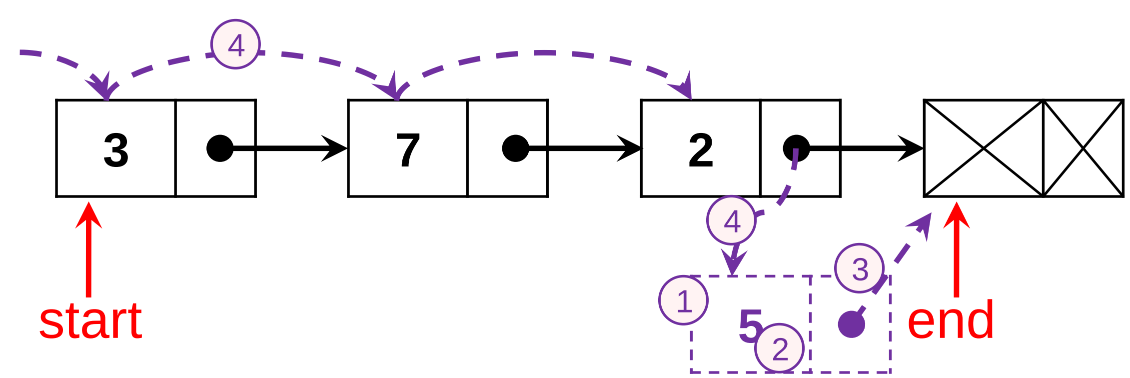
- **Complexity**: O(1)

Inserting an Element



- How do we add a node **as the last** node of a list segment [start, end)?

  1. create a new node
  2. set its data field to the value to add
  3. set its next field to end
  4. point the old last node to it



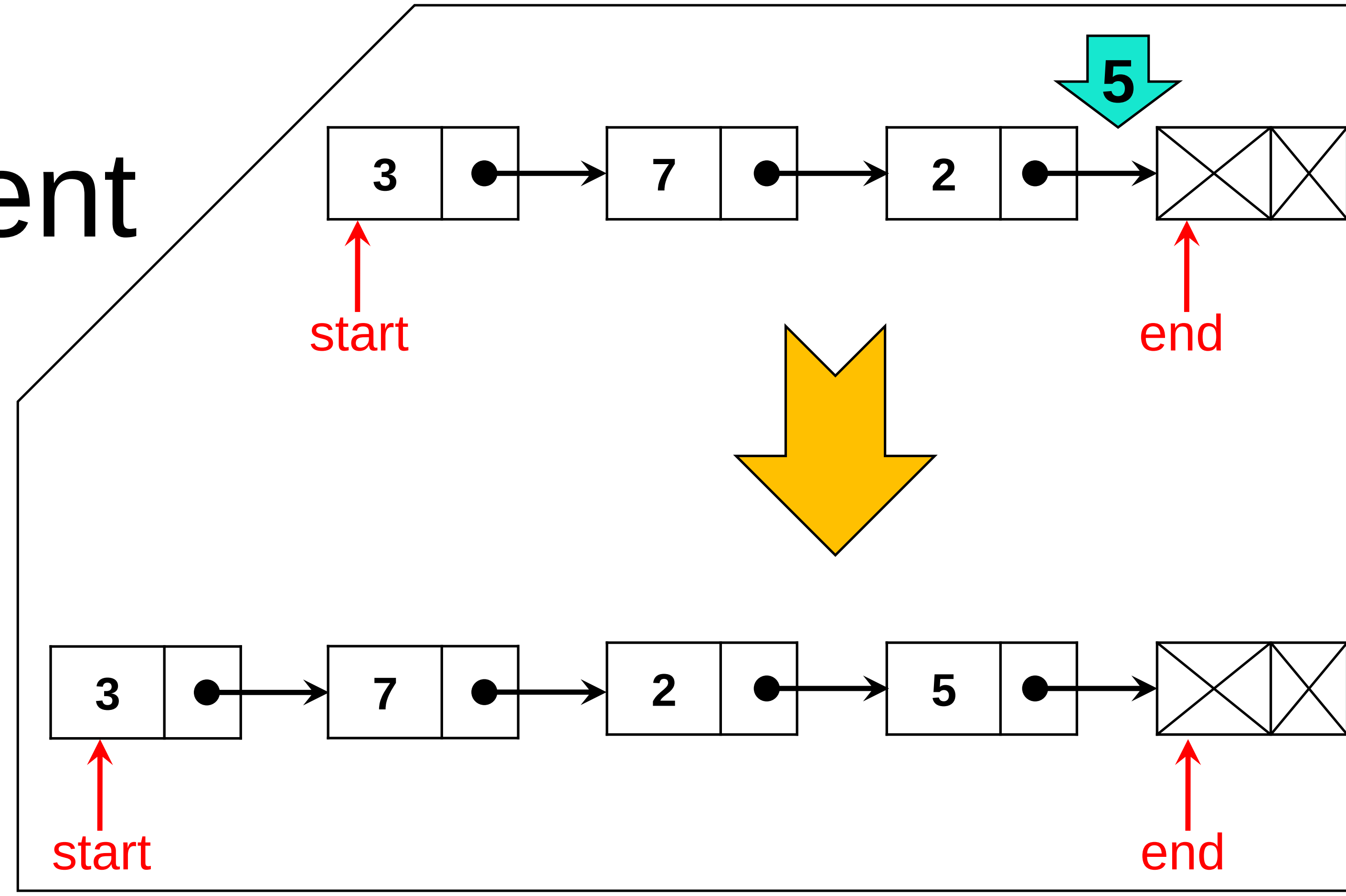

```
① list* new_last = alloc(list);
② new_last->data = x;
③ new_last->next = end;
④ list* l = start;
   while (l->next != end)
     l = l->next;
   l->next = new_last;
```

Note: we are adding a new last node, but we modify the next pointer of the old last node

- **Complexity**: O(n)

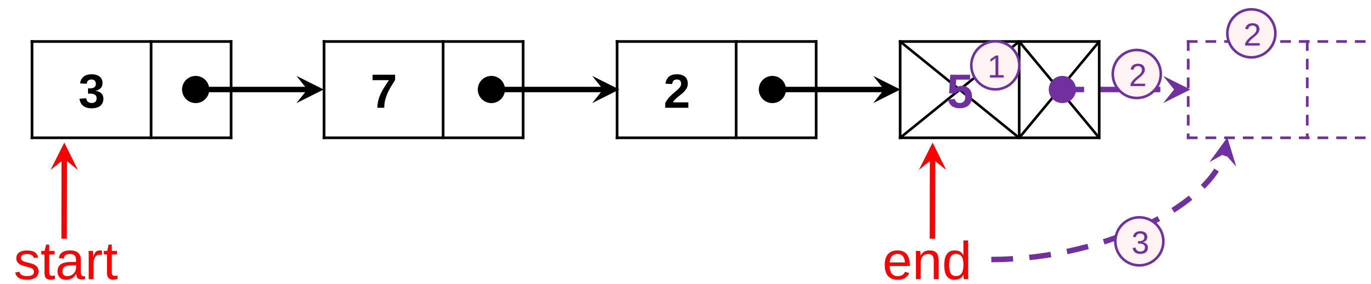
Expensive!

Inserting an Element



- How do we add a node **as the last** node of a list segment [start, end)?
  - ***Can we do better?***
    1. set the data field of end to the value to add
    2. set its next field to a new dummy node
    3. set end to it

```
① end->data = x;
② end->next = alloc(list);
③ end = end->next;
```



Note: we are using the old dummy node as the new last node, and creating a new dummy

- **Complexity**: O(1)

Much better!

Summary

	at the beginning	*at the end*
Inserting	O(1)	O(1)
Deleting	O(1)	O(n)

Good

Bad

- We will use this as a guide when implementing queues (and stacks) to achieve their complexity goals

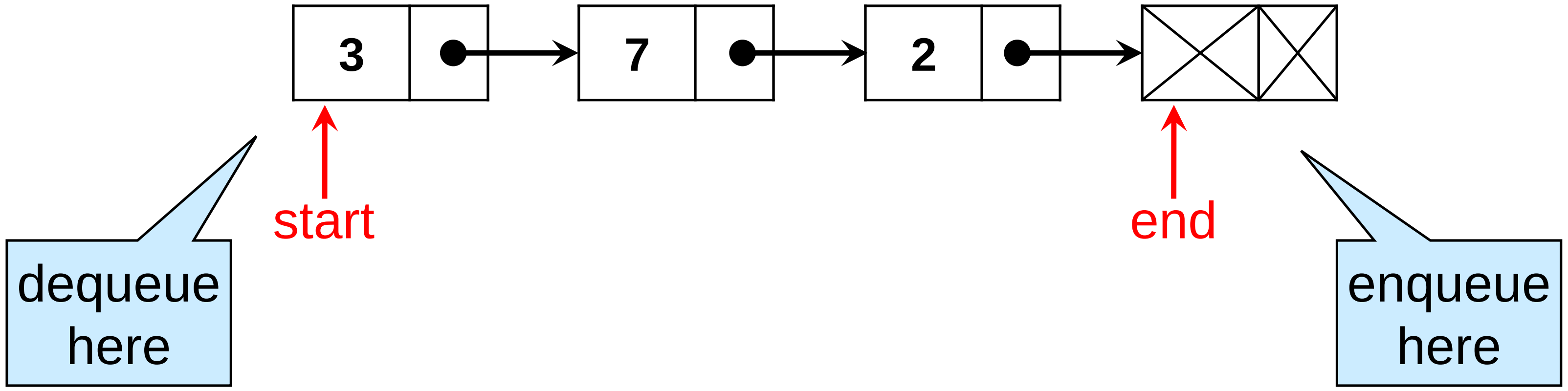
Implementing Queues

Queues as List Segments

- Implementing queues
  - We add and remove from *opposite ends*
  - Cost must be O(1)

	at the beginning	*at the end*
Inserting	O(1)	O(1)
Deleting	O(1)	O(n)

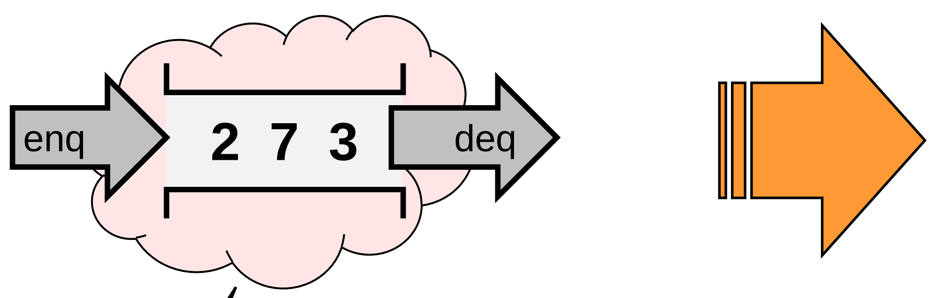
- The **front** of the queue is the start of the segment
  - because that's where we remove elements from
    - choosing the end would give deq cost O(n)
- The **back** of the queue is the end of the segment
    - the dummy node

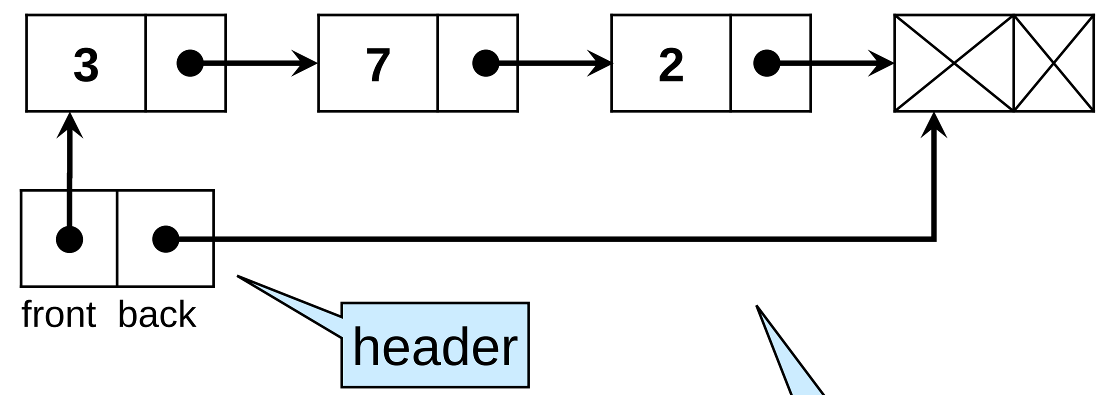


Queues as List Segments

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

- The **front** of the queue is the start of the segment
- The **back** of the queue is the end of the segment





Client view

Implementation view

Notice the order

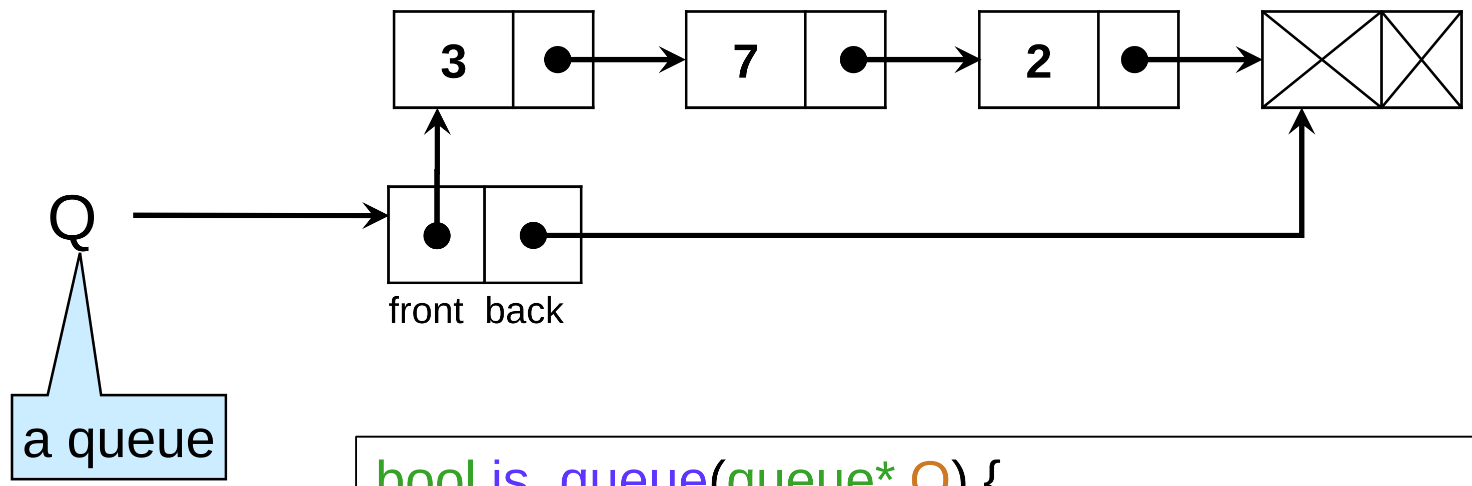
```
// Implementation-side type
struct queue_header {                    // Concrete type
  list* front;    // start of segment, where we deq
  list* back;    // end of segment, where we enq
};
typedef struct queue_header queue;   // Internal name

// … rest of implementation …

// Client-side type (abstract)
typedef queue* queue_t;
```

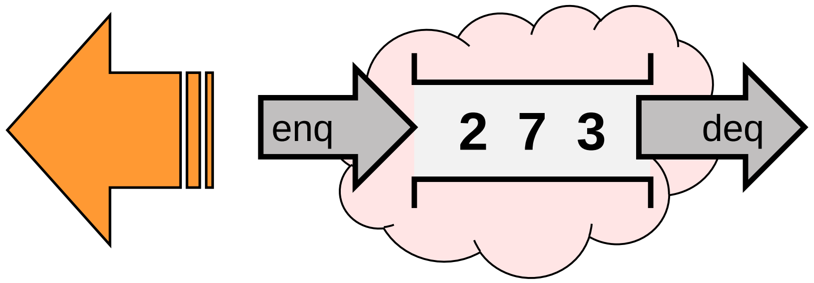
Queues as List Segments

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

```
struct queue_header {
  list* front;
  list* back;
};
typedef struct queue_header queue;
```

- Internally, queues are values of type queue*
  - must be non-NULL
  - front and back fields must bracket a valid list segment






```
bool is_queue(queue* Q) {
  return Q != NULL
      && is_acyclic(Q->front)
      && is_segment(Q->front, Q->back);
}
```

Queues as List Segments

- Next we implement the operations exported by the interface

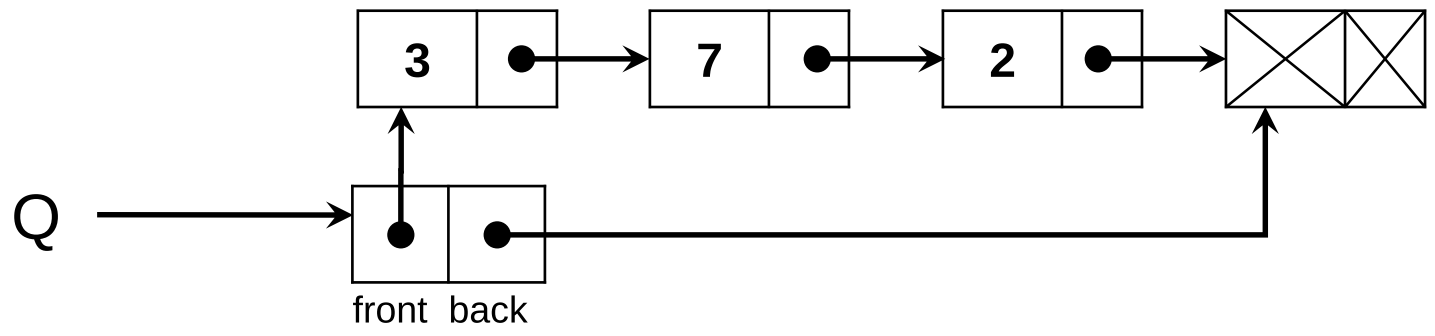
Queue Interface

```
// typedef ______* queue_t;

bool queue_empty(queue_t S)    // O(1)
  /*@requires S != NULL;          @*/ ;

queue_t  queue_new()             // O(1)
  /*@ensures \result != NULL;       @*/
  /*@ensures queue_empty(\result); @*/ ;

void enq(queue_t S, int x)       // O(1)
  /*@requires S != NULL;            @*/
  /*@ensures !queue_empty(S);       @*/ ;

int deq(queue_t S)                // O(1)
  /*@requires S != NULL;            @*/
  /*@requires !queue_empty(S);      @*/ ;
```

Queues as List Segments

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

```
struct queue_header {
  list* front;
  list* back;
};
typedef struct queue_header queue;
```



- Enqueuing
  - add at the back

```
void enq(queue* Q, int x)
//@requires is_queue(Q);
//@ensures is_queue(Q);
//@ensures !queue_empty(Q);
{
  Q->back->data = x;
  Q->back->next = alloc(list);
  Q->back = Q->back->next;
}
```

- Dequeueing
  - remove from the front

```
int deq(queue* Q)
//@requires is_queue(Q);
//@requires !queue_empty(Q);
//@ensures is_queue(Q);
{
  int x = Q->front->data;
  Q->front = Q->front->next;
  return x;
}
```

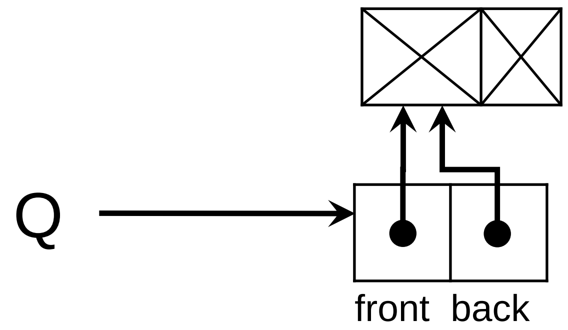
Cost is **O(1)**

- This is the code we wrote earlier with
  - start changed to Q->front
  - end changed to Q->back

Queues as List Segments

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

```
struct queue_header {
  list* front;
  list* back;
};
typedef struct queue_header queue;
```



- The empty queue
  - empty segment has start equal to end

```
bool queue_empty(queue* Q)
//@requires is_queue(Q);
{
  return Q->front == Q->back;
}
```

- Creating a queue
  - we create an empty queue

```
queue* queue_new()
//@ensures is_queue(\result);
//@ensures queue_empty(\result);
{
  queue* Q = alloc(queue);
  Q->front = alloc(list);
  Q->back = Q->front;
  return Q;
}
```

Cost is **O(1)**

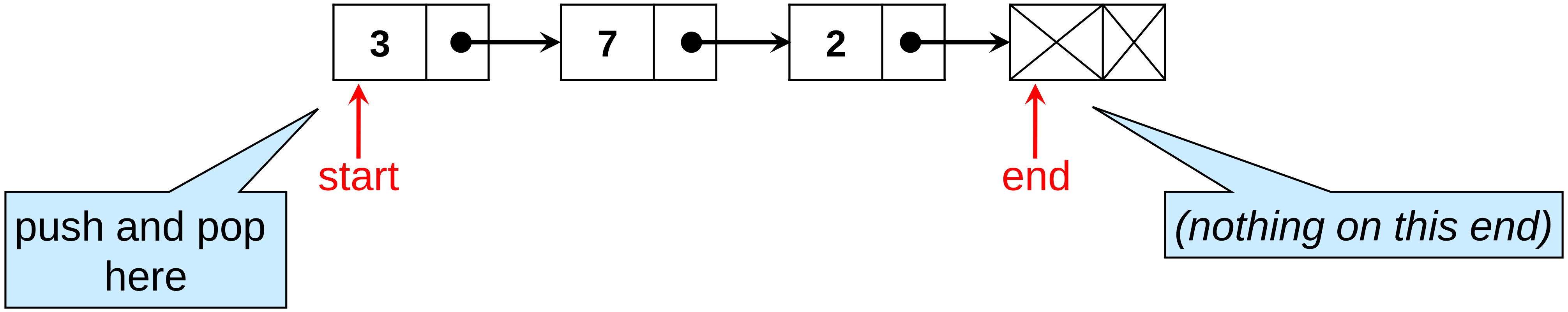
Implementing Stacks

Stacks as List Segments

- Implementing stacks
  - We add and remove from *the same end*
  - Cost must be O(1)

	at the beginning	*at the end*
Inserting	O(1)	O(1)
Deleting	O(1)	O(n)

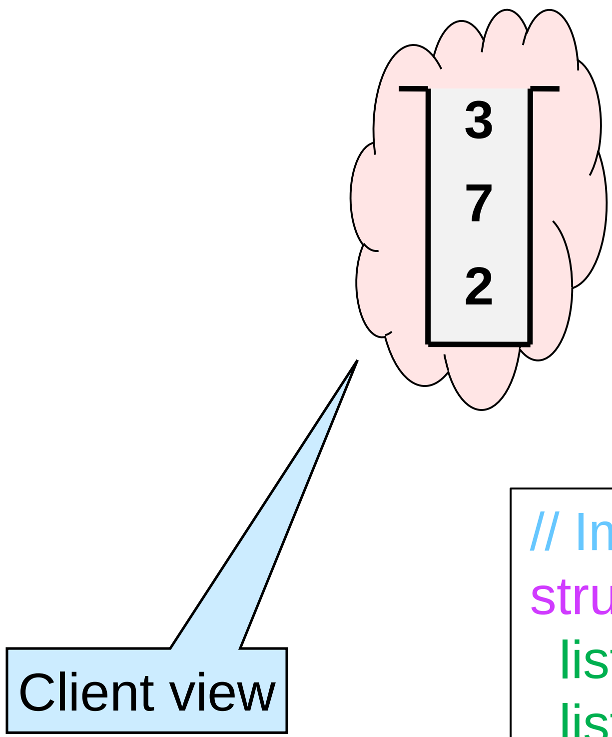
- The **top** of the stack is the start of the segment
  - because that's where we add and remove elements
    - choosing the end would give pop cost O(n)
- The **floor** of the stack is the end of the segment
    - the dummy node

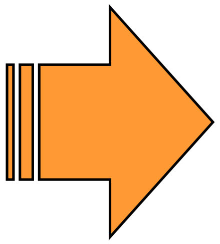


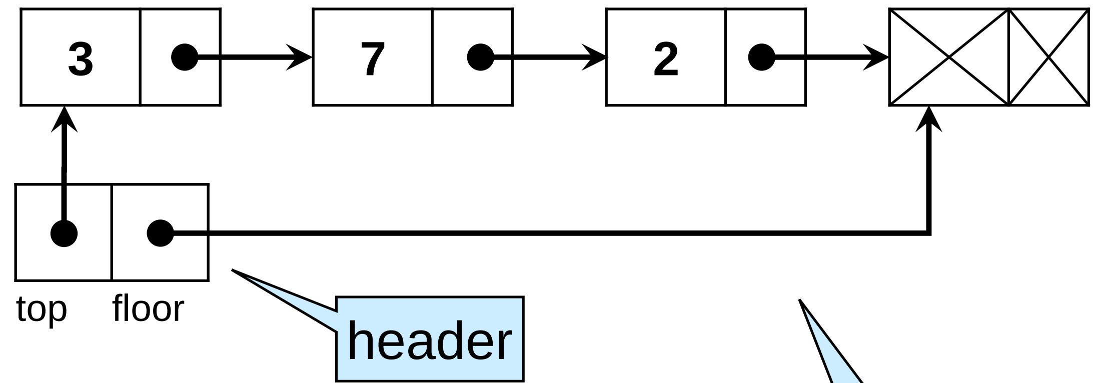
```c
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

Stack as List Segments

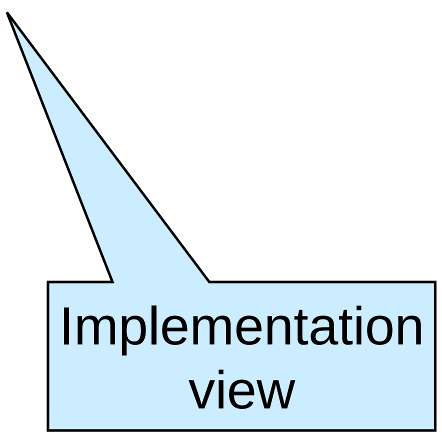
- The **top** of the stack is the start of the segment








```c
// Implementation-side type
struct stack_header {                   // Concrete type
  list* top;    // start of segment, where we push and pop
  list* floor;
};
typedef struct stack_header stack;      // Internal name

// … rest of implementation …

// Client-side type (abstract)
typedef stack* stack_t;
```



  - The representation invariant is_stack is just like is_queue

Stacks as List Segments

- Next we implement the operations exported by the interface

Stack Interface

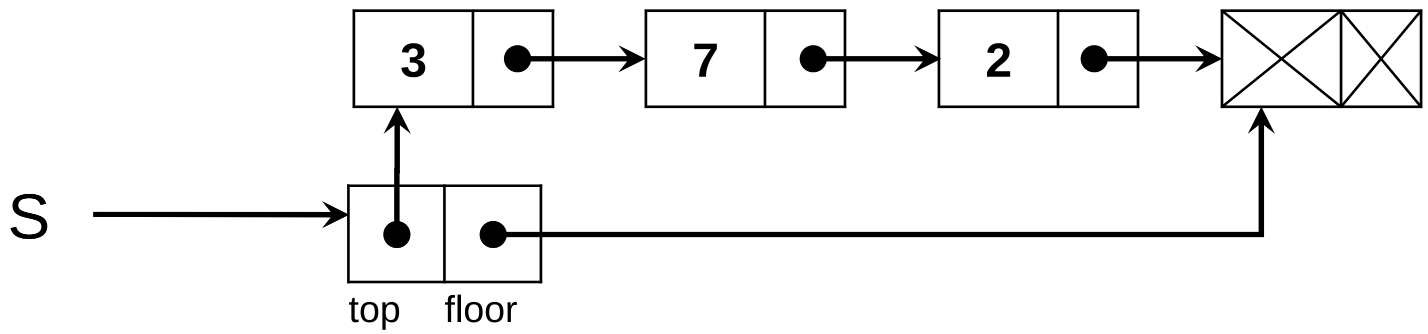
```
// typedef ______* stack_t;

bool stack_empty(stack_t S)     // O(1)
  /*@requires S != NULL;             @*/ ;

stack_t  stack_new()               // O(1)
  /*@ensures \result != NULL;        @*/
  /*@ensures stack_empty(\result);  @*/ ;

void push(stack_t S, int x)        // O(1)
  /*@requires S != NULL;             @*/
  /*@ensures !stack_empty(S);        @*/ ;

int pop(stack_t S)                 // O(1)
  /*@requires S != NULL;             @*/
  /*@requires !stack_empty(S);       @*/ ;
```

Also updated to int elements

Stacks as List Segments

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

```
struct stack_header {
  list* top;
  list* floor;
};
typedef struct stack_header stack;
```




```
bool stack_empty(stack* S)
//@requires is_stack(S);
{
  return S->top == S->floor;
}
```

```
stack* stack_new()
//@ensures is_stack(\result);
//@ensures stack_empty(\result);
{
  stack* S = alloc(stack);
  S->top = alloc(list);
  S->floor = S->top;
  return S;
}
```

All **O(1)**

Same code we wrote for queues
with front/back replaced with top/floor

```
int pop(stack* S)
//@requires is_stack(S);
//@requires !stack_empty(S);
//@ensures is_stack(S);
{
  int x = S->top->data;
  S->top = S->top->next;
  return x;
}
```

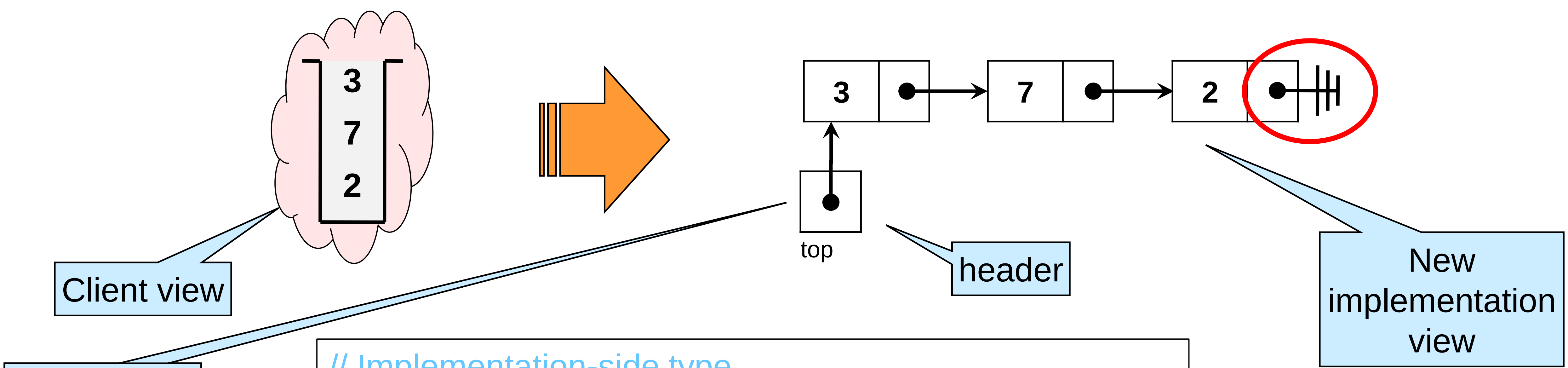
```
void push(stack* S, int x)
//@requires is_stack(S);
//@ensures is_stack(S);
//@ensures !stack_empty(S);
{
  list* l = alloc(list);
  l->data = x;
  l->next = S->top
  S->top = l;
}
```

Code we wrote earlier
with start replaced with S->top

Another Implementation of Stacks

- The floor field goes mostly unused
  - only to check that a stack is empty
- We can get rid of it …
  - … if we represent stacks as **NULL-terminated** lists

This is a great idea if we don’t need direct access to the end of the list



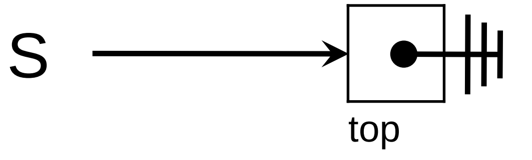

```
// Implementation-side type
struct stack_header {                         // Concrete type
  list* top;     // start of segment, where we push and pop
};
typedef struct stack_header stack;     // Internal name
```

Another Implementation of Stacks

- Valid stacks are
  - non-NULL and
  - the top field is a NULL-terminated list
    - i.e., is acyclic

```
bool is_stack(stack* S) {
  return S != NULL
      && is_acyclic(S->top);
}
```

- The empty stack has NULL in the top field




```
bool stack_empty(stack* S)
//@requires is_stack(S);
{
   return S->top == NULL;
}
```

```
stack* stack_new()
//@ensures is_stack(\result);
//@ensures stack_empty(\result);
{
  stack* S = alloc(stack);
  S->top = NULL;
  return S;
}
```

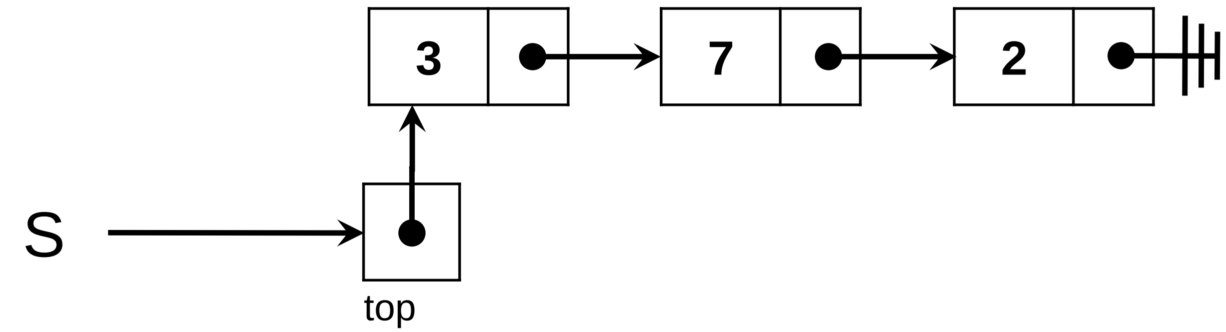
- Nothing else changes!

Sharing

Stacks without Headers

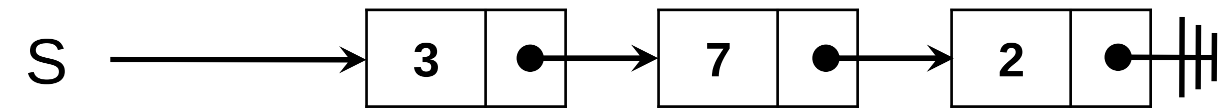
- Since the header contains just one field,

```
struct stack_header {
  list* top;
};
typedef struct stack_header stack;
```



  - why not get rid of it?

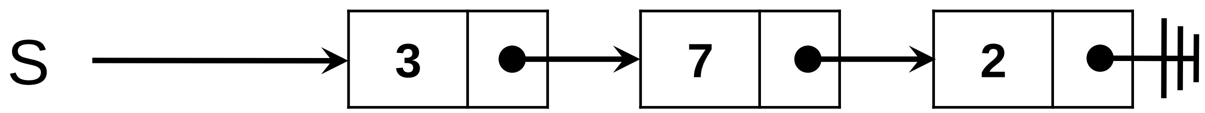
```
typedef list* stack;
```



  - push and pop are now incorrect
    - they modify the local stack variable but not the caller’s
    - aliasing!
  - it breaks the interface: NULL is now the empty stack

Stacks without Headers

```
typedef list* stack;
```



- But we're fine if we always *return* the updated stack

***Functional* stack Interface**

```
// typedef ______* stack_t;

bool stack_empty(stack_t S) ;   // O(1)

stack_t  stack_new()              // O(1)
/*@ensures stack_empty(\result);    @*/ ;

stack_t push(stack_t S, int x)     // O(1)
/*@ensures !stack_empty(\result);   @*/ ;

stack_t pop(stack_t S, int* res) // O(1)
/*@requires !stack_empty(S);         @*/ ;
```

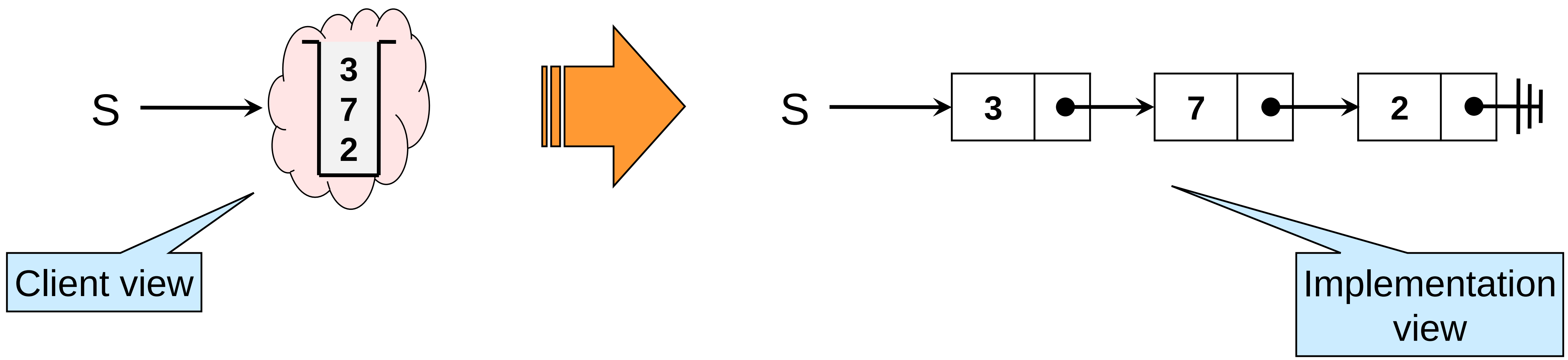
No more NULL checks

Our trick to return two outputs

  - Functions transform an input stack into an output stack
    - this is a **functional interface**

Functional Stacks

- How to create this stack?



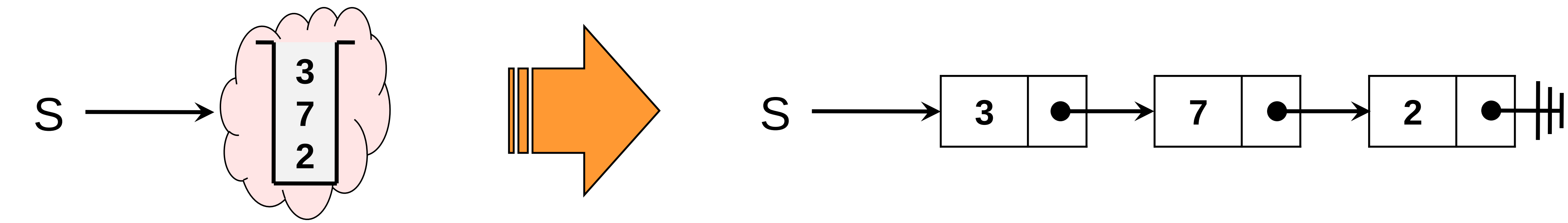

```
stack_t S = stack_new();
S = push(S, 2);
S = push(S, 7);
S = push(S, 3);
```

- equivalently

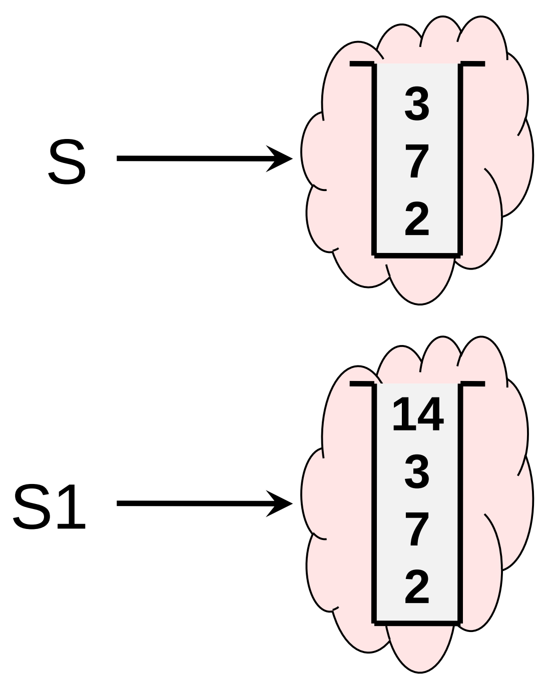
```
stack_t S = push(push(push(stack_new(), 2), 7), 3);
```

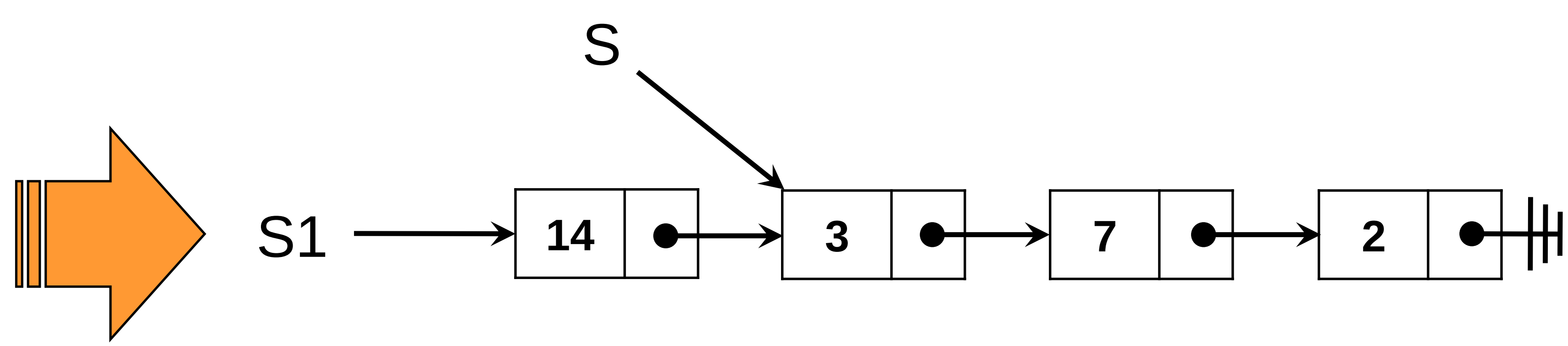
  - *but harder to read*

Functional Stacks



○ What if now we do `stack_t S1 = push(S, 14);` ?





- The client has two stacks
  - S with 3, 7, 2
  - S1 with 14, 3, 7, 2
- In the implementation, they **share** a suffix
  - the linked list 3, 7, 2 is shared

Sharing

- A functional stack library supports sharing list suffixes
  - This takes up much less space than our earlier implementation!
  - The client has no idea

- What if we now do this?

```
stack_t S2 = push(S, 42);
stack_t S3 = pop(S, x_ptr);
```

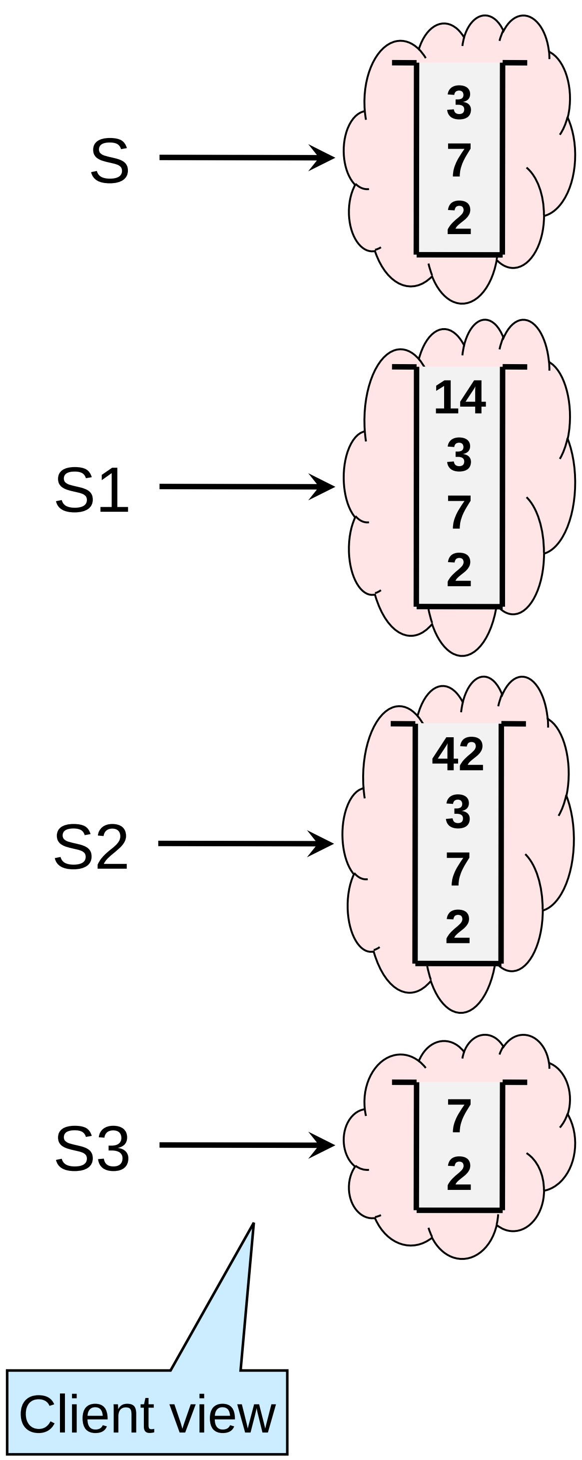
The variable S is still around

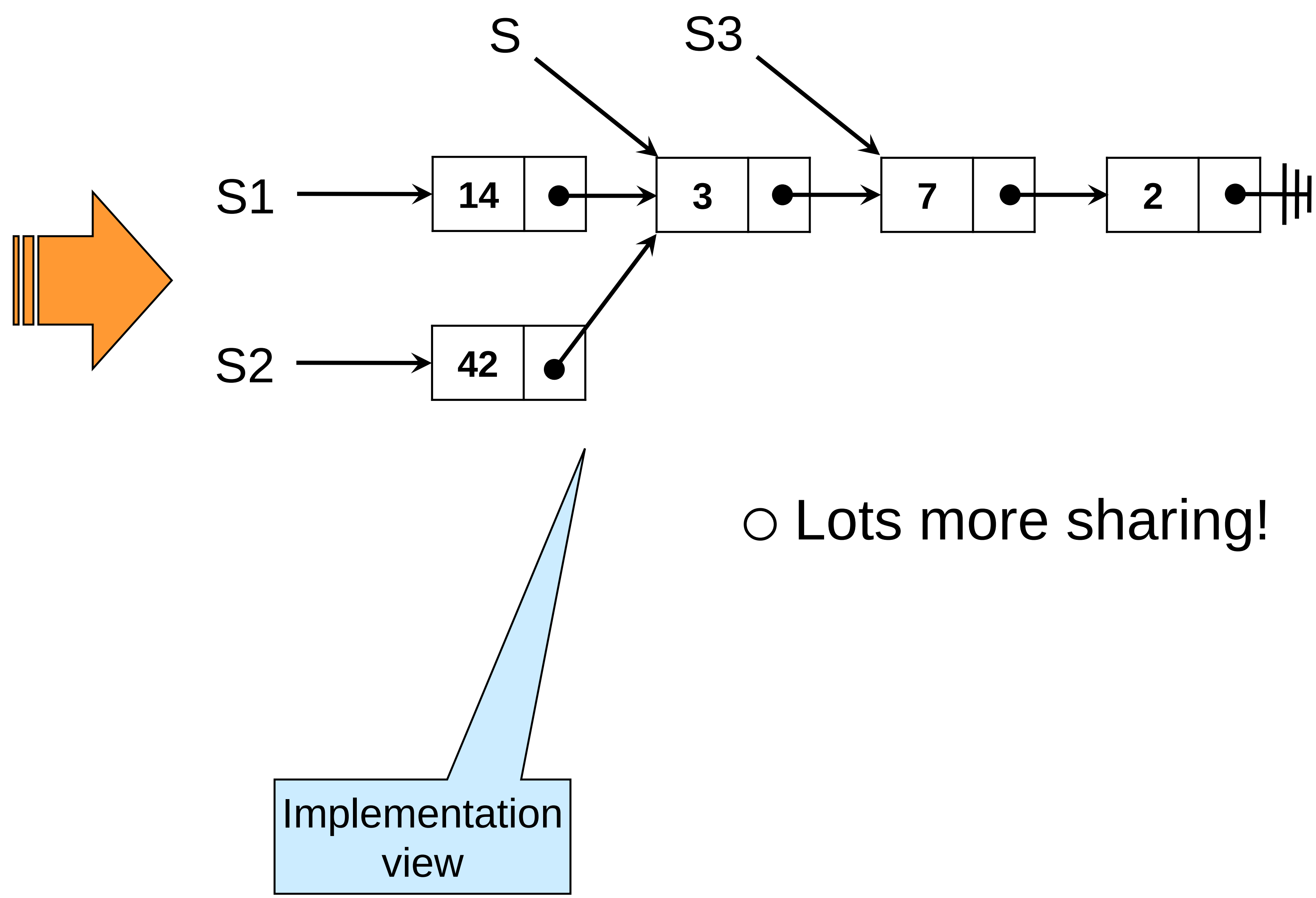
Sharing

- What if we now do

```
stack_t S2 = push(S, 42);
stack_t S3 = pop(S, x_ptr);
```

?





- ○ Lots more sharing!

Sharing

- If sharing is so great, why don't our libraries always use it?
  - It takes a change of mindset
    - using functions that don't modify data structures in place
  - A lot of code we write uses one instance of a data structure
    - So what? Sharing wouldn't hurt anyway
      - Good point
  - It doesn't work for all data structures
    - Try it on queues!

- Functional programming languages rely heavily on sharing

Wrap Up

What have we done?

- We introduced **linked lists** and two common ways to use them
  - NULL-terminated linked lists
  - list segments
- We learned about list manipulations and their complexity
- We used them to implement stacks and queues
- We talked about sharing

Linked Lists vs. Arrays

- How do they compare?

	Arrays (unsorted)	***Linked lists***
Pros	○ O(1) access ○ built-in	○ self-resizing ○ O(1) insertion* ○ O(1) deletion* * *Given the right pointers*
Cons	○ fixed size ○ O(n) insertion	○ O(n) access ○ no special syntax

- Question to help decide which one to use:
  - Can we anticipate the size we need?
  - Do they allow us to achieve our target complexity?