Unit 06: Abstract Data Types (ADTs)

Anthony Estey

CSC 115: Fundamentals of Programming II

University of Victoria

Data Structures

- A data structure is a container used to store data (objects).
- Typically, this is done in one of two ways:
  - Contiguous-based structure $\boxed{\text{Array}}$
  - Node-based structure $\boxed{\text{Linked List}}$
- Although their implementations are different, both data structures can store the same data, and perform the same operations on that data
  - For example: add and remove students from a course's waitlist
- There are tradeoffs associated with each implementation:
  - Arrays allow immediate access to the ith element in collection $(0 \leq i < n)$
  - A linked-list requires iteration from one end of the list to reach the ith position

Abstract Data Types (ADTs)

- An ADT is composed of:
  - A description of what data is stored (but not how the data is stored)
  - A set of operations on that data (but not how the operations are implemented)

- The separation of what something does (specification) and how it does it (implementation) is a fundamental concept in engineering!

Why the split?

- Think about it from a software development perspective:

- When we have specifications without implementation details:
  - This is the perfect medium for which clients and developers can communicate
  - Clients can request behaviours, developers can discuss these requests, until an agreement is made
  - Clients are not programmers, they are not interested in implementation details, but they are interested in *what* the program can do
  - Developers then provide an implementation based on the specification (the clients *use* the end product, but don't ever have to see the underlying code)

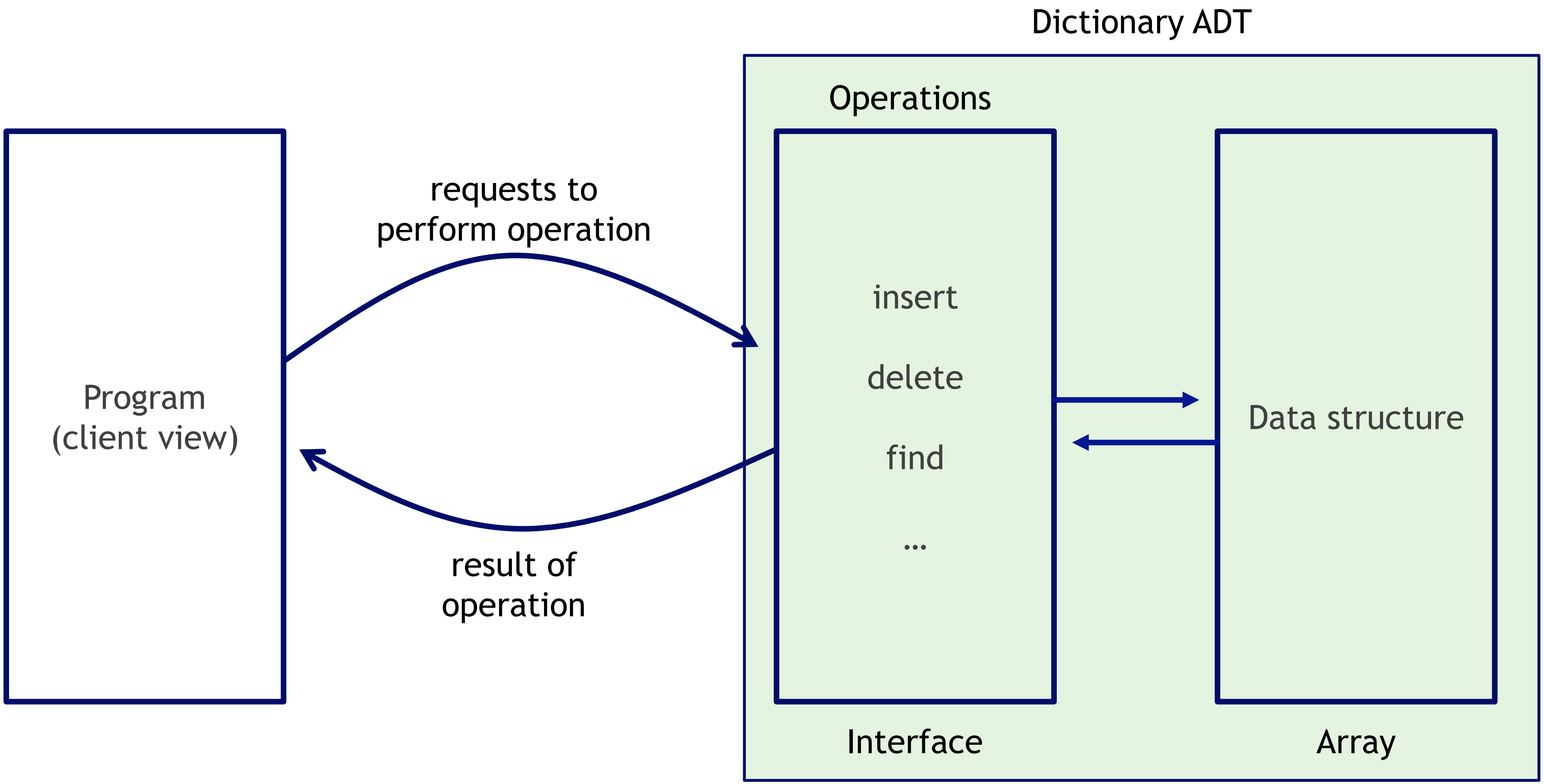
Example

- ADT Dictionary:
- ***What*** data is stored, and what operations must it perform?
  - Stores a pair of strings, representing the word and definition (data)
  - Operations:
    - insert(word, definition)
    - find(word)
    - delete(word)

We also know the effect operations have on the data.
If we ***delete*** a word from the dictionary, a subsequent ***find*** operation should fail.

- We use a data structure to implement an ADT
  - this is where the ***how*** comes in

Implementing the Dictionary ADT

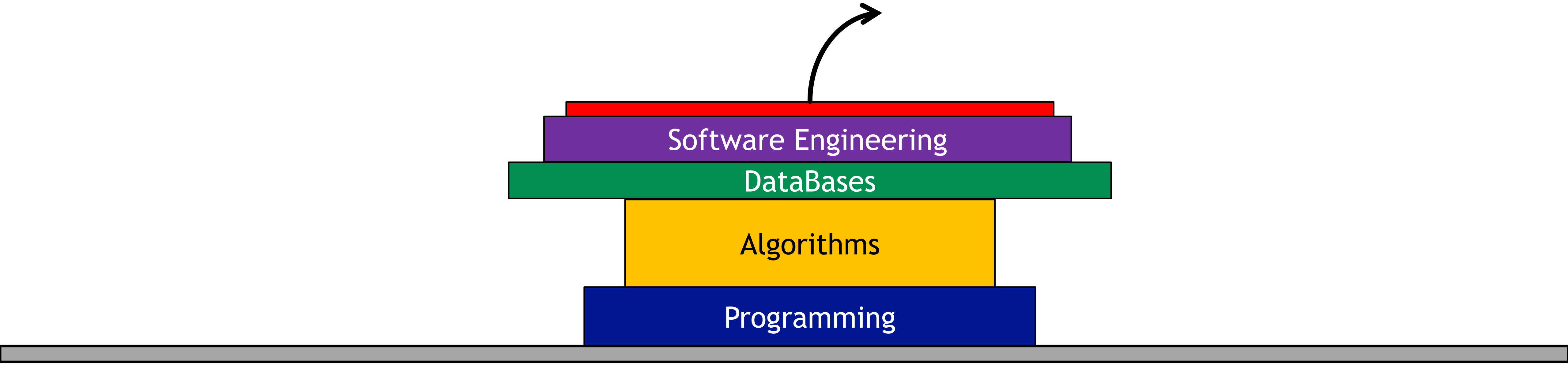


Client vs Programmer

- Clients know how to use something
  - ***What*** operations are available and ***what*** they do
- Programmers must decide ***how*** to implement the operations
- Their choices may be influenced by a number of things:
  - execution speed
  - memory requirements
  - maintenance (debugging, scalability, etc.)
- Dictionary example:
  - Clients/users: add new words to dictionary, look up words to see definitions
  - Programmer: determine ***how*** data is stored; ***how*** operations are implemented

The Notion of a Stack

- Collection of items
  - Items are returned in the ***reverse*** order they were added
  - This is behavior is often abbreviated LIFO (Last In, First Out)



The Stack ADT

- The Stack ADT specifies the following operations:
  - push(*o*): Insert object *o* onto the top of the stack
  - pop(): Access and remove the object from the top of the stack; an error occurs if the stack is empty
  - isEmpty(): Determines whether the stack is currently empty
  - top(): Accesses the object on top of the stack without removing it; an error occurs if the stack is empty
  - size(): Gets the current number of objects in the stack

Stack Examples

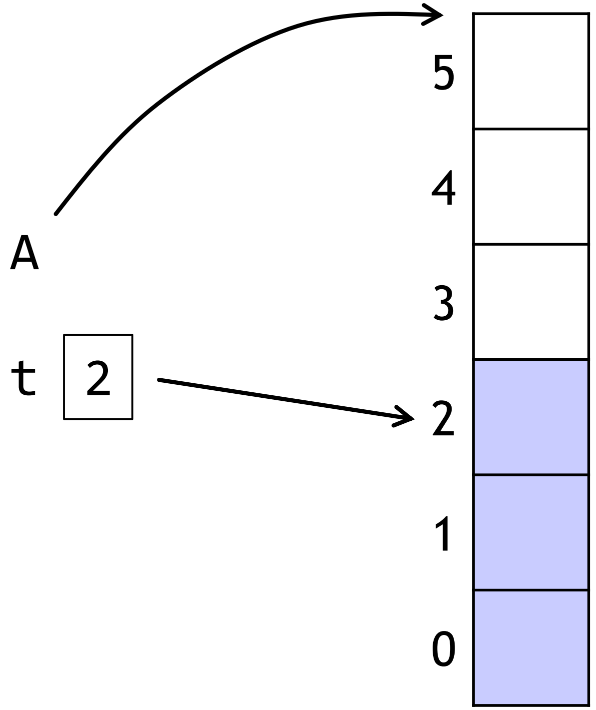
- Things we use all the time:
  - “Undo” function found in most applications
  - Back button when browsing the web

- Programming:
  - The runtime environment’s handling of nested method calls
  - Recursion

- Problem solving:
  - Approaches where a problem is solved by breaking the problem up into smaller version of the same problem. (Divide-and-conquer)

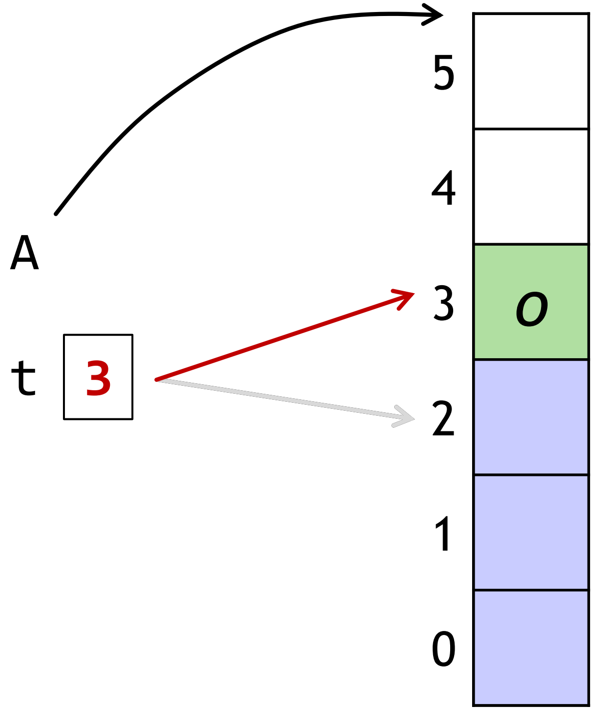
Stack Implementation

- A stack can be implemented in multiple ways:
- In an array implementation, we typically have the following:
  - A: an n-element array
  - t: an integer to keep track of the index of the top element in the array
- Example:



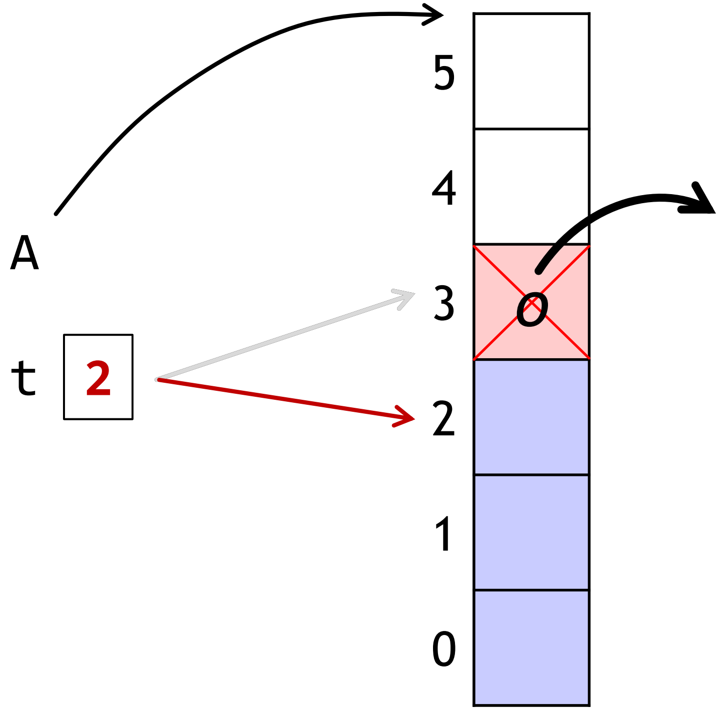
Stack Implementation

- A stack can be implemented in multiple ways:
- In an array implementation, we typically have the following:
  - A: an n-element array
  - t: an integer to keep track of the index of the top element in the array
- Example:
  - **push(*o*)**



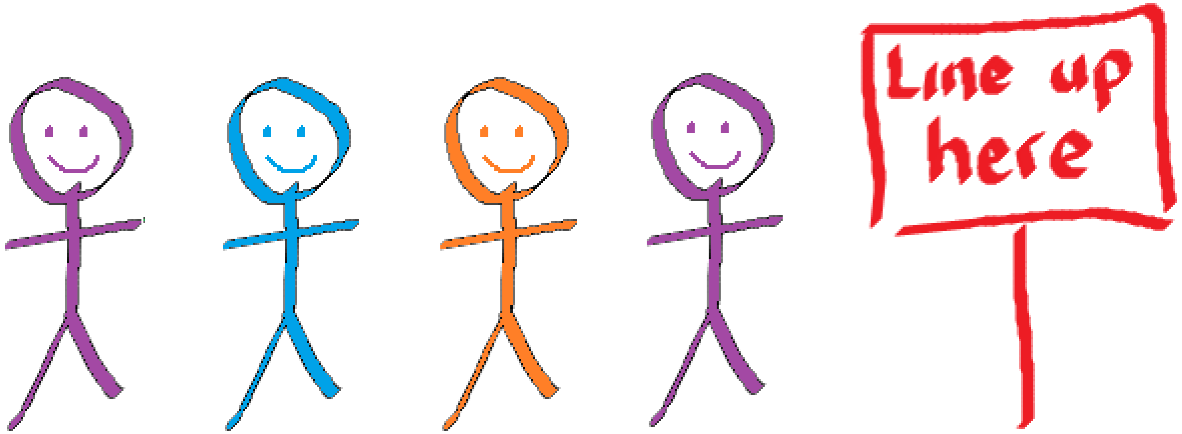
Stack Implementation

- A stack can be implemented in multiple ways:
- In an array implementation, we typically have the following:
  - A: an n-element array
  - t: an integer to keep track of the index of the top element in the array
- Example:
  - push(o)
  - **pop()**



The Notion of a Queue

- Collection of items
  - Items are returned in the ***same*** order they were added
  - This is behavior is often abbreviated FIFO (First In, First Out)



The Queue ADT

- The Queue ADT specifies the following operations:
  - enqueue(*o*): Insert object *o* at the rear (back) of the queue
  - dequeue(): Access and remove the object from the front of the queue; an error occurs if the queue is empty
  - isEmpty(): Determines whether the queue is currently empty
  - front(): Accesses the object at the front of the queue without removing it; an error occurs if the queue is empty
  - size(): Gets the current number of objects in the queue

Queue Examples

- Any time people wait in line for something
  - the bank, the cafeteria, etc.

- Waitlists for classes here at Uvic!

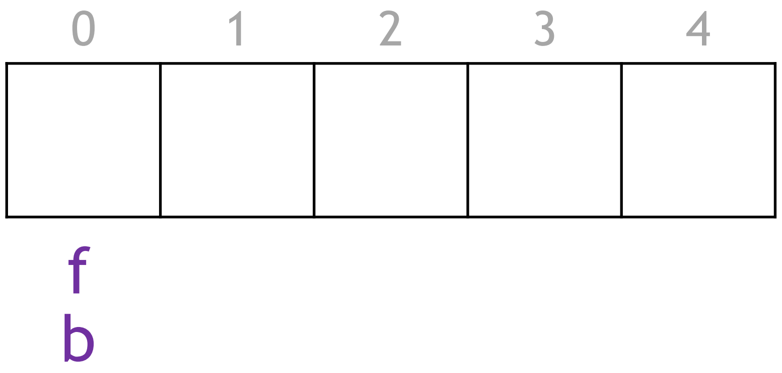
Queue Implementation

- We will explore an array-based implementation of a queue during lecture

Queue implementation with a circular array

Algorithm enqueue(obj):
  if isFull() **then**
    return an error (queue is full)
  $A[b] \leftarrow obj$
  $b \leftarrow (b + 1)$ mod N

Algorithm dequeue():
  if isEmpty() **then**
    return an error (queue is empty)
  $e \leftarrow A[f]$
  $f \leftarrow (f + 1)$ mod N
  return e

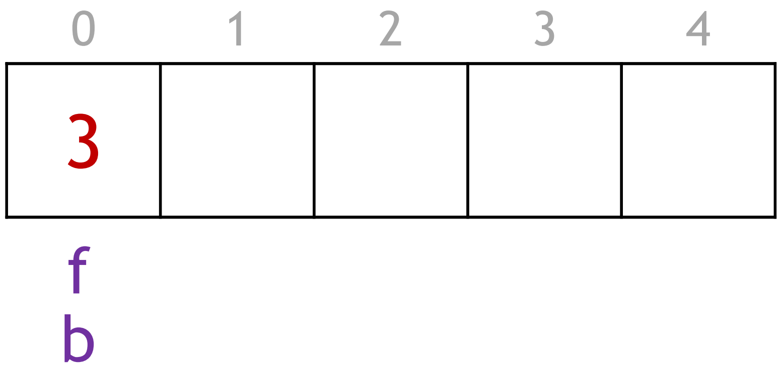


Queue implementation with a circular array

Algorithm enqueue(obj):
  if isFull() **then**
    return an error (queue is full)
  $A[b] \leftarrow obj$
  $b \leftarrow (b + 1)$ mod N

Algorithm dequeue():
  if isEmpty() **then**
    return an error (queue is empty)
  $e \leftarrow A[f]$
  $f \leftarrow (f + 1)$ mod N
  return e

enqueue(3);

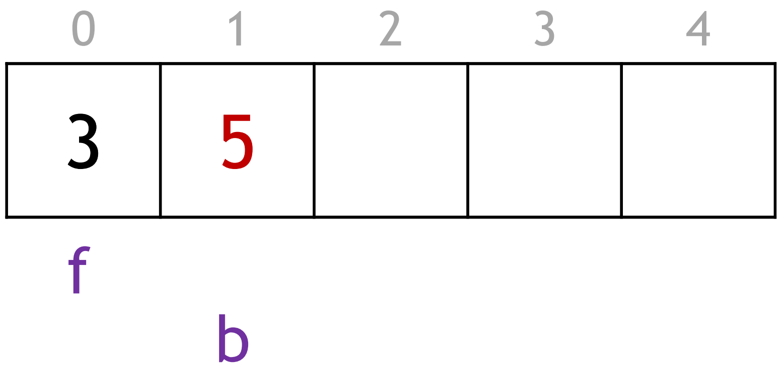


Queue implementation with a circular array

Algorithm enqueue(obj):
  if isFull() **then**
    return an error (queue is full)
  $A[b] \leftarrow obj$
  $b \leftarrow (b + 1)$ mod N

Algorithm dequeue():
  if isEmpty() **then**
    return an error (queue is empty)
  $e \leftarrow A[f]$
  $f \leftarrow (f + 1)$ mod N
  return e

enqueue(3);
enqueue(5);

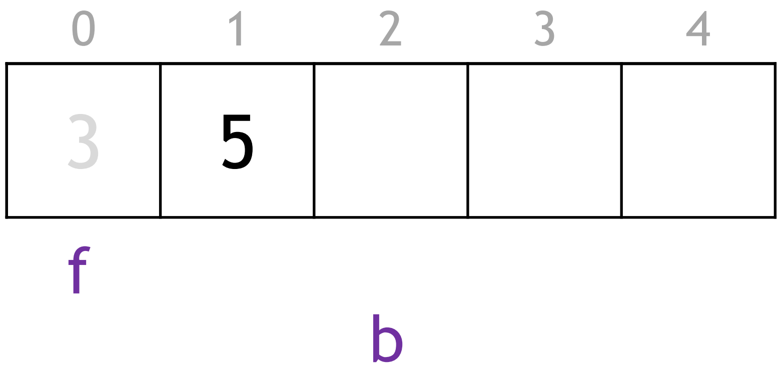


Queue implementation with a circular array

Algorithm enqueue(obj):
  if isFull() **then**
    return an error (queue is full)
  $A[b] \leftarrow obj$
  $b \leftarrow (b + 1)$ mod N

Algorithm dequeue():
  if isEmpty() **then**
    return an error (queue is empty)
  $e \leftarrow A[f]$
  $f \leftarrow (f + 1)$ mod N
  return e

enqueue(3);
enqueue(5);
dequeue();



Queue implementation with a circular array

Algorithm enqueue(obj):
  if isFull() **then**
    return an error (queue is full)
  $A[b] \leftarrow obj$
  $b \leftarrow (b + 1)$ mod N

Algorithm dequeue():
  if isEmpty() **then**
    return an error (queue is empty)
  $e \leftarrow A[f]$
  $f \leftarrow (f + 1)$ mod N
  return e

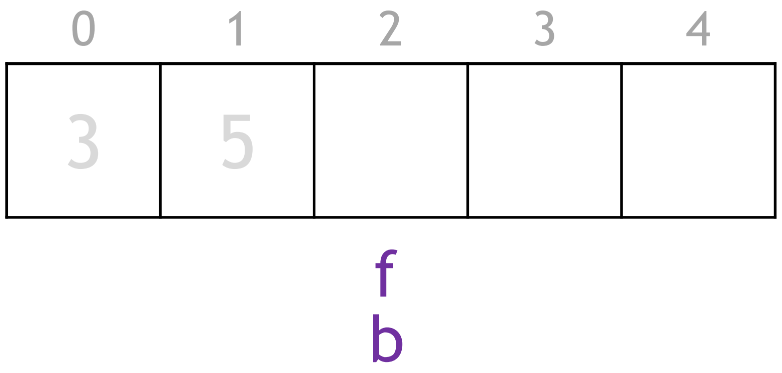
0	1	2	3	4
3	5			
	f			
		b		

enqueue(3);
enqueue(5);
dequeue();
dequeue();

Queue implementation with a circular array

Algorithm enqueue(obj):
  if isFull() **then**
    return an error (queue is full)
  $A[b] \leftarrow obj$
  $b \leftarrow (b+1)$ mod N

Algorithm dequeue():
  if isEmpty() **then**
    return an error (queue is empty)
  $e \leftarrow A[f]$
  $f \leftarrow (f+1)$ mod N
  return e



enqueue(3);
enqueue(5);
dequeue();
dequeue();

Queue implementation with a circular array

Algorithm enqueue(obj):
  if isFull() **then**
    return an error (queue is full)
  $A[b] \leftarrow obj$
  $b \leftarrow (b + 1)$ mod N

Algorithm dequeue():
  if isEmpty() **then**
    return an error (queue is empty)
  $e \leftarrow A[f]$
  $f \leftarrow (f + 1)$ mod N
  return e

0	1	2	3	4
4	5	7	2	1
		f b		

enqueue(3);
enqueue(5);
dequeue();
dequeue();
enqueue(7);
enqueue(2);
enqueue(1);
enqueue(4);
enqueue(5);

Queue implementation with a circular array

Algorithm enqueue(obj):
  if isFull() **then**
    return an error (queue is full)
  $A[b] \leftarrow obj$
  $b \leftarrow (b + 1)$ mod N

Algorithm dequeue():
  if isEmpty() **then**
    return an error (queue is empty)
  $e \leftarrow A[f]$
  $f \leftarrow (f + 1)$ mod N
  return e

0	1	2	3	4
4	5	7	2	1

f
b

enqueue(3);
enqueue(5);
dequeue();
dequeue();
enqueue(7);
enqueue(2);
enqueue(1);
enqueue(4);
enqueue(5);

Queue implementation with a circular array

Algorithm enqueue(obj):
  if isFull() **then**
    return an error (queue is full)
  $A[b] \leftarrow obj$
  $b \leftarrow (b+1)$ mod N
  $count \leftarrow count + 1$

Algorithm dequeue():
  if isEmpty() **then**
    return an error (queue is empty)
  $e \leftarrow A[f]$
  $f \leftarrow (f+1)$ mod N
  $count \leftarrow count - 1$
  return e

Algorithm isFull():
  return $count = N$

Algorithm isEmpty():
  return $count = 0$

0	1	2	3	4
4	5	7	2	1
		f b		

enqueue(3);
enqueue(5);
dequeue();
dequeue();
enqueue(7);
enqueue(2);
enqueue(1);
enqueue(4);
enqueue(5);

Method 1: Use a count variable

Pros:
- simplicity. Both isFull and isEmpty are easy to implement

Cons:
- Memory (allocated another variable)
- additional operations to update the variable every enqueue and dequeue

Queue implementation with a circular array

Algorithm enqueue(obj):
  if isFull() **then**
    return an error (queue is full)
  $A[b] \leftarrow obj$
  $b \leftarrow (b+1)$ mod N

Algorithm dequeue():
  if isEmpty() **then**
    return an error (queue is empty)
  $e \leftarrow A[f]$
  $f \leftarrow (f+1)$ mod N
return e

Algorithm isFull():
  return $(b+1)$ mod $N = f$

Algorithm isEmpty():
  return $f = b$

0	1	2	3	4
4		7	2	1
	b	f		

0	1	2	3	4
3	5	2	6	
f				b

enqueue(3);
enqueue(5);
dequeue();
dequeue();
enqueue(7);
enqueue(2);
enqueue(1);
enqueue(4);
~~enqueue(5);~~

Method 2: Full when $size = N - 1$

Pros:
- Speed. Fewest operations required

Cons:
- Array memory is never fully utilized (always at least 1 unused spot)

Queue implementation with a circular array

Algorithm enqueue(obj):
  if isFull() **then**
    return an error (queue is full)
  else if isEmpty() **then**
    $f \leftarrow 0$
    $b \leftarrow 0$
  $A[b] \leftarrow obj$
  $b \leftarrow (b + 1)$ mod N

Algorithm dequeue():
  if isEmpty() **then**
    return an error (queue is empty)
  $e \leftarrow A[f]$
  $f \leftarrow (f + 1)$ mod N
  if $f = b$ **then**
    $f \leftarrow -1$
return e

Algorithm isEmpty():
  return $f = -1$

Algorithm isFull():
  return $f = b$

-1	0	1	2	3	4
	9	8	2	5	4

f
b

enqueue(3);
enqueue(8);
enqueue(2);
dequeue();
enqueue(5);
enqueue(4);
enqueue(9);

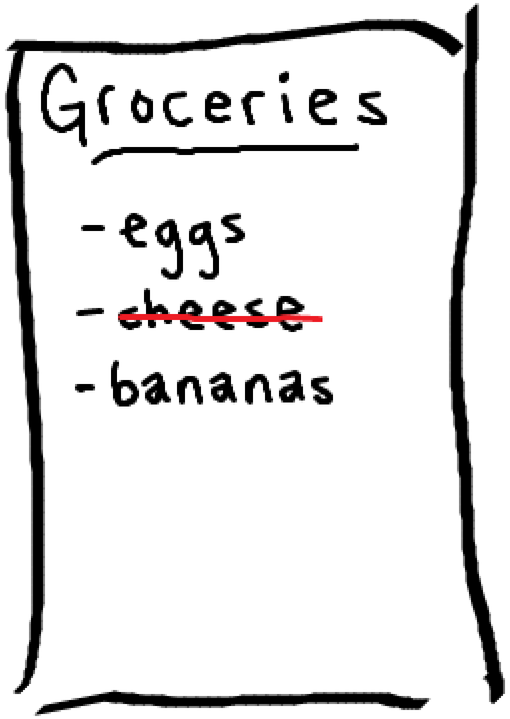
Method 3: $f, b = -1$ when empty

Pros:
- Relatively efficient (only re-assign f or b when the queue is full or empty)
- Array memory fully utilized

Cons:
- Additional conditional operations on every enqueue and dequeue

The Notion of a List

- Collection of items
  - Elements can be inserted and removed in ***any*** order
  - Any element can be accessed at any given time by their position in the list
    - Not just the front (Queue) or top (Stack) element

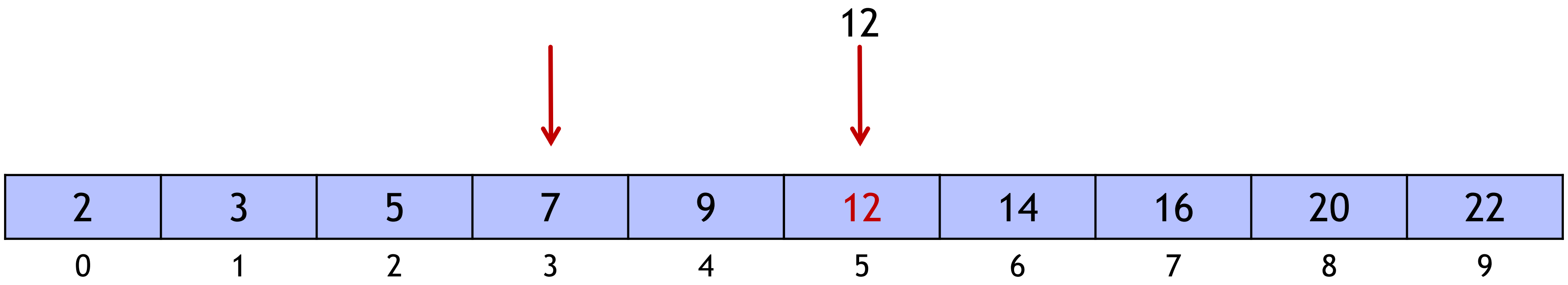


Index-Based Lists

- We can uniquely refer to each element in the list containing n elements using an integer in the range $[0, n-1]$
  - We define the **index** or **rank** of an element e in a list by the number of elements that come before e in the list
  - Hence the first element is at index 0, and the last element is at index $n-1$
- Index-based lists support the following operations:
  - **get(*r*):** Return the element in the list with index r.
  - **set(*r*, *e*):** Replace element at index r with e and return the element replaced.
  - **add(*r*, *e*):** Insert a new element e into the list at index r.
  - **remove(*r*):** Remove the element at index r from the list.

Array Implementation of a List

- The **get**(***r***) and **set**(***r***, ***e***) operations can be done in $O(1)$ time:
  - **get(3)** returns 7
  - **set(5, 12)** returns 13



Lists – Array Implementation

- **add(*r*, *e*)** cannot be done in $O(1)$ time
  - **add(*3*, *6*):**

Algorithm add(r, e):
  if $n = N$ **then**
    return error (List is full)
  if $r < n$ **then**
    for $i \leftarrow n - 1$ to r **do**
      $A[i + 1] \leftarrow A[i]$
  $A[r] \leftarrow e$
  $n \leftarrow n + 1$

6

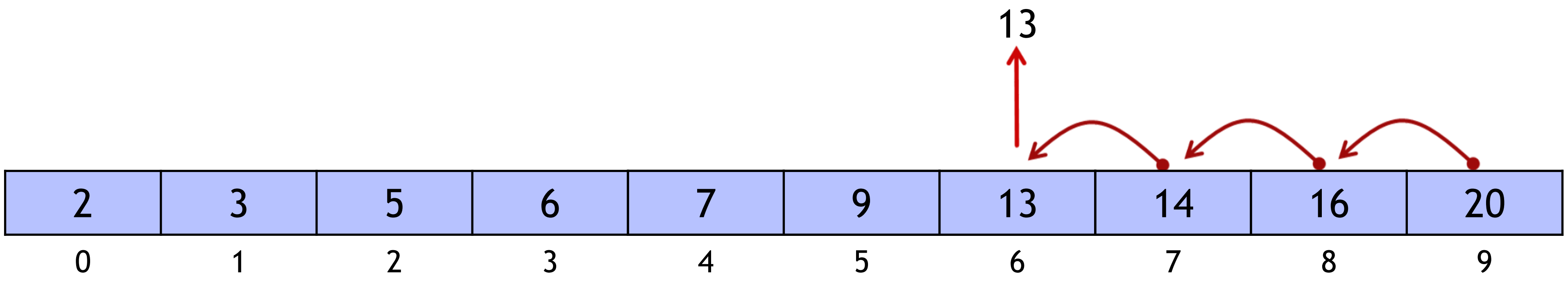
2	3	5	6	7	9	13	14	16	20
0	1	2	3	4	5	6	7	8	9

On average, half of the elements need to be shuffled: $O(n)$

Lists – Array Implementation

- **remove(r)** has the same problem
  - **remove(*6*):**

Algorithm remove(r):
  if $r < n - 1$ **then**
    $e \leftarrow A[r]$
    for $i \leftarrow r$ **to** $n - 2$ **do**
      $A[i] \leftarrow A[i + 1]$
    $n \leftarrow n + 1$
    return e



Runtime of **remove(r)**: $O(n)$

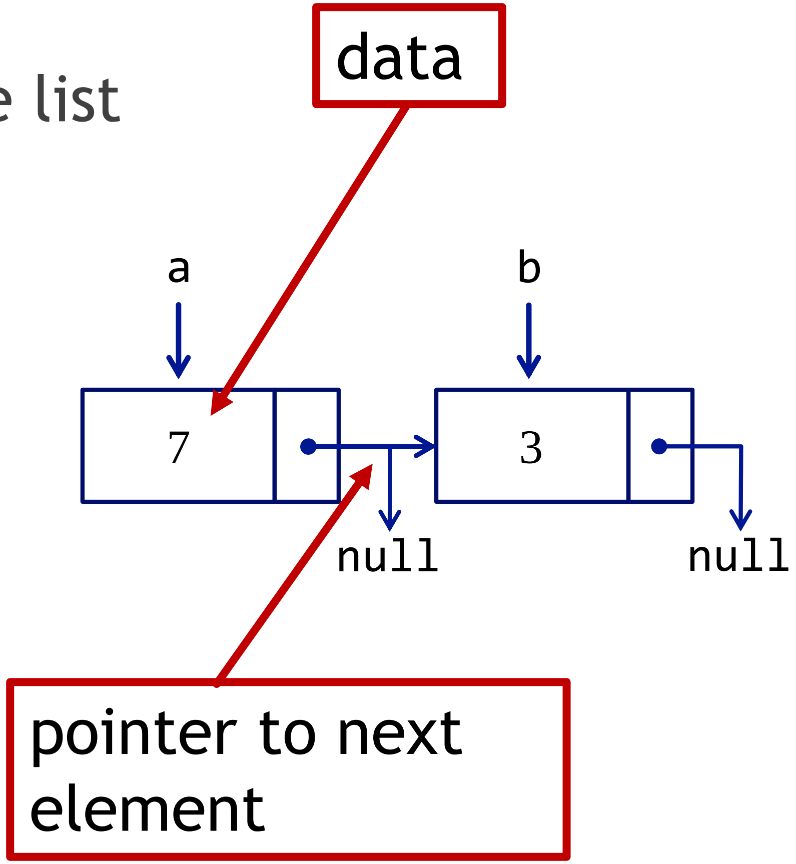
Summary

- Running times of the index-based methods:

Method	Time
get(r)	$O(1)$
set(r, e)	$O(1)$
add(r, e)	$O(n)$
remove(r)	$O(n)$

Node-based (or reference-based) Lists

- A **linked list** is a data structure composed of **nodes** linked together
- A **node** is a data structure that contains:
  - data (whatever we want to store in the list)
  - a pointer to the location of the next element in the list
  - (sometimes a pointer to the previous element too)

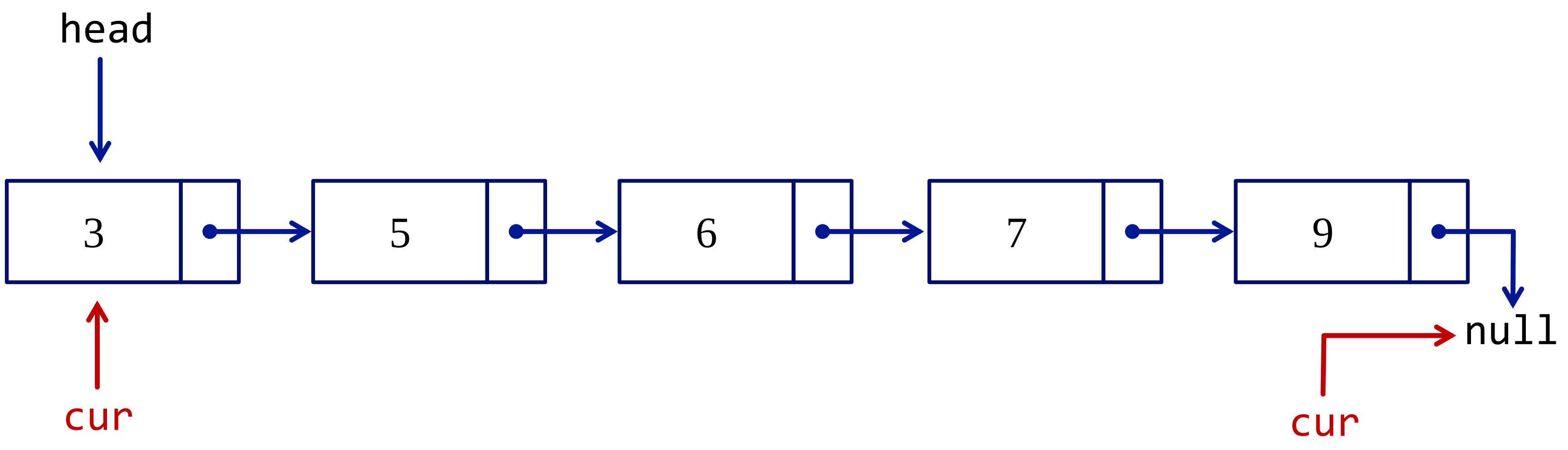



```
public class Node {
        private int data;
        private Node next;
        …
}
```

```
Node a = new Node(7, null);
Node b = new Node(3, null);
a.next = b;
```

Iteration implementation

- With linked lists, we need to keep a reference to the head of the list. From there, we can reach all subsequent elements:




```
for (Node cur = head; cur != null; cur = cur.next) {
        System.out.println(cur.data));
}
```

Iteration

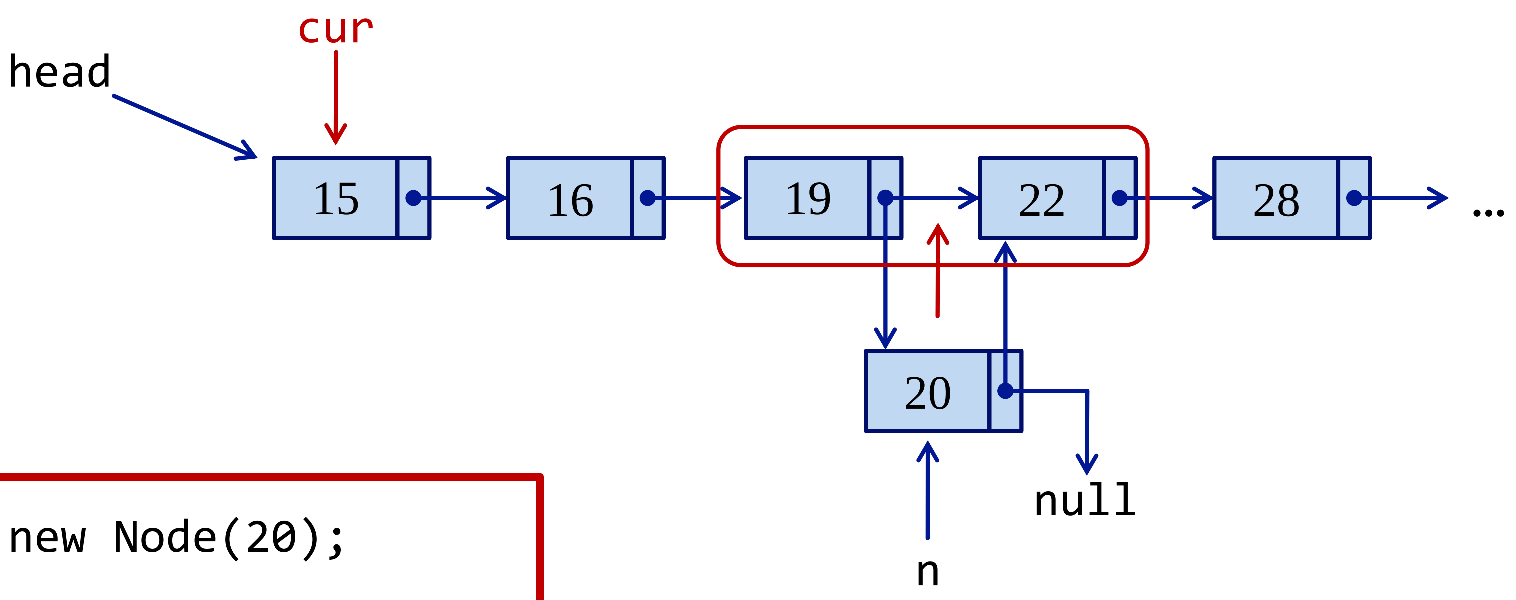
- With linked lists, we need to keep a reference to the head of the list. From there, we can reach all subsequent elements:

```
Node cur = head;
while (cur != null) {
      System.out.println(cur.data));
      cur = cur.next;
}
```

```
for (Node cur = head; cur != null; cur = cur.next) {
      System.out.println(cur.data));
}
```

Insertion

- First, determine where to insert the new node
- Then, update the next pointers appropriately



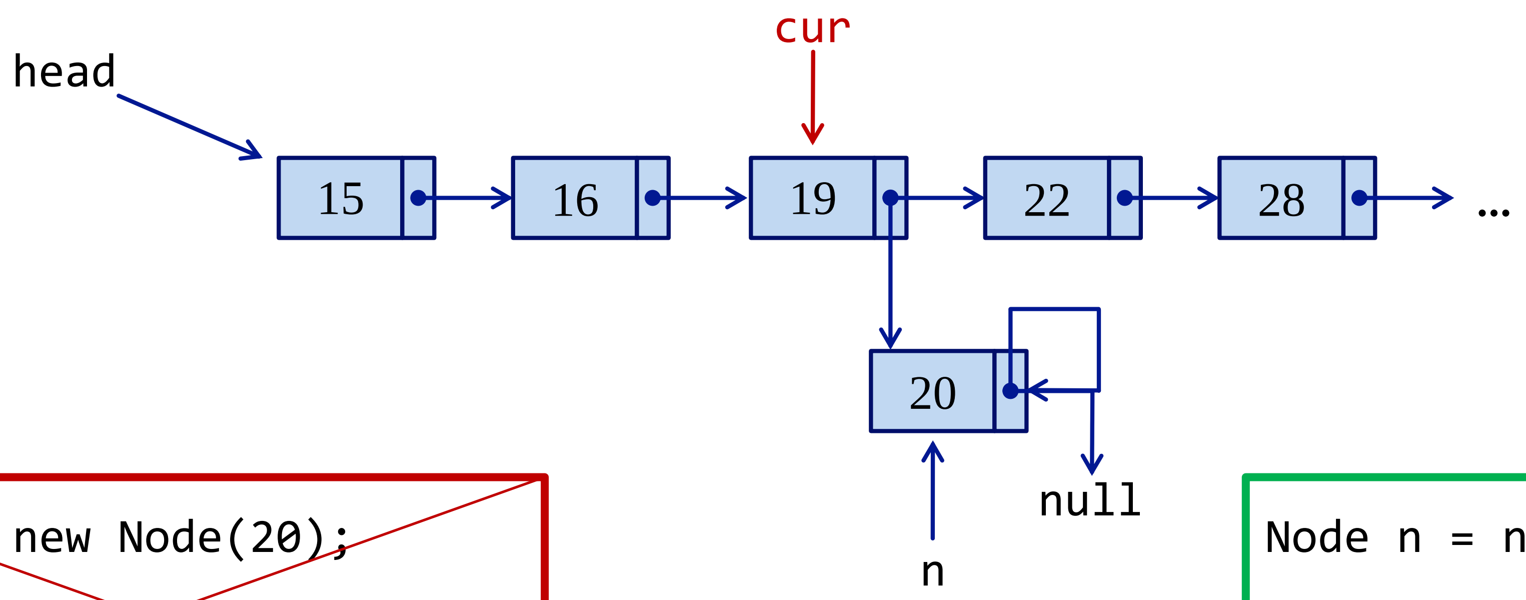

```
Node n = new Node(20);

n.next = cur.next;

cur.next = n;
```

Order of operations is important!

- Let's revisit our insertion example, and assume we want to insert a node with data value 20 between node's 19 and 22.

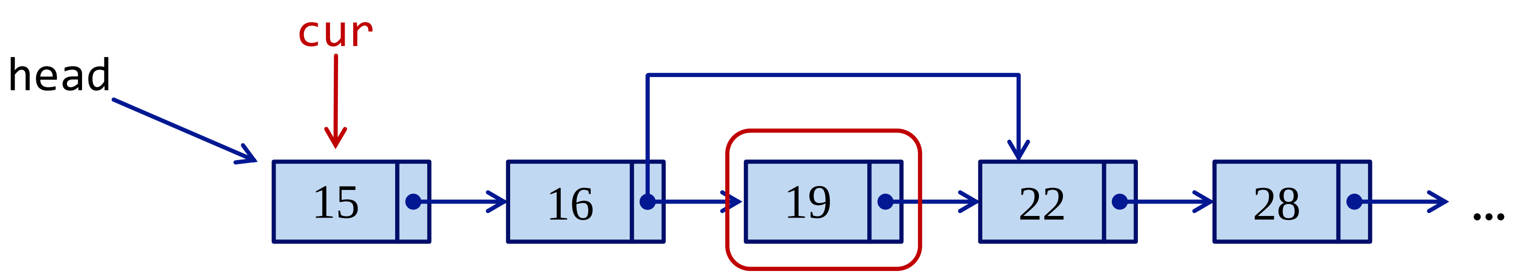



```
Node n = new Node(20);
cur.next = n;
n.next = cur.next;
```

```
Node n = new Node(20);
n.next = cur.next;
cur.next = n;
```

Removal

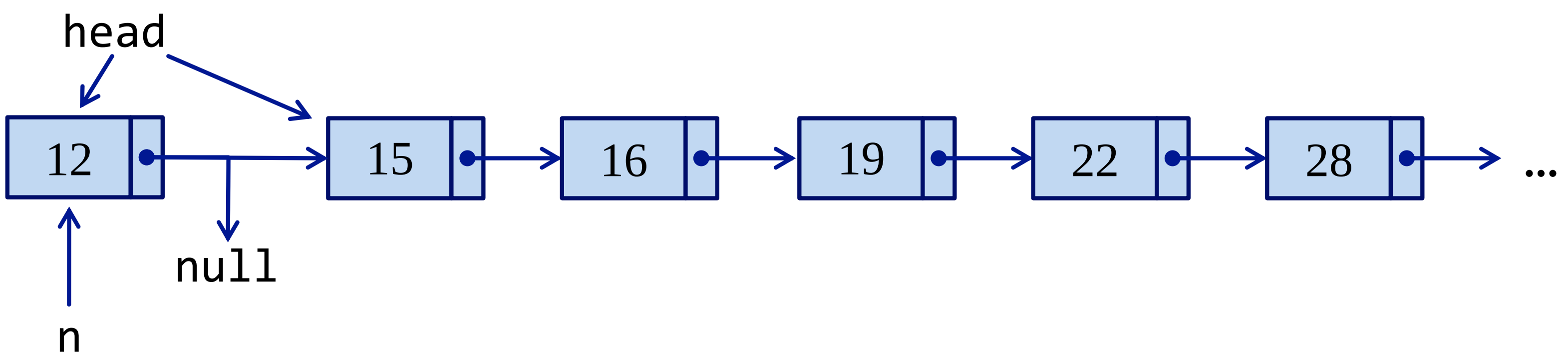
- First, locate the element *preceding* the one to remove
- Then, update the next pointers so that the deleted node is skipped
  - Java’s garbage collection will delete of any object that nothing points to




```
cur.next = cur.next.next;
```

Adding an item to the front of a list

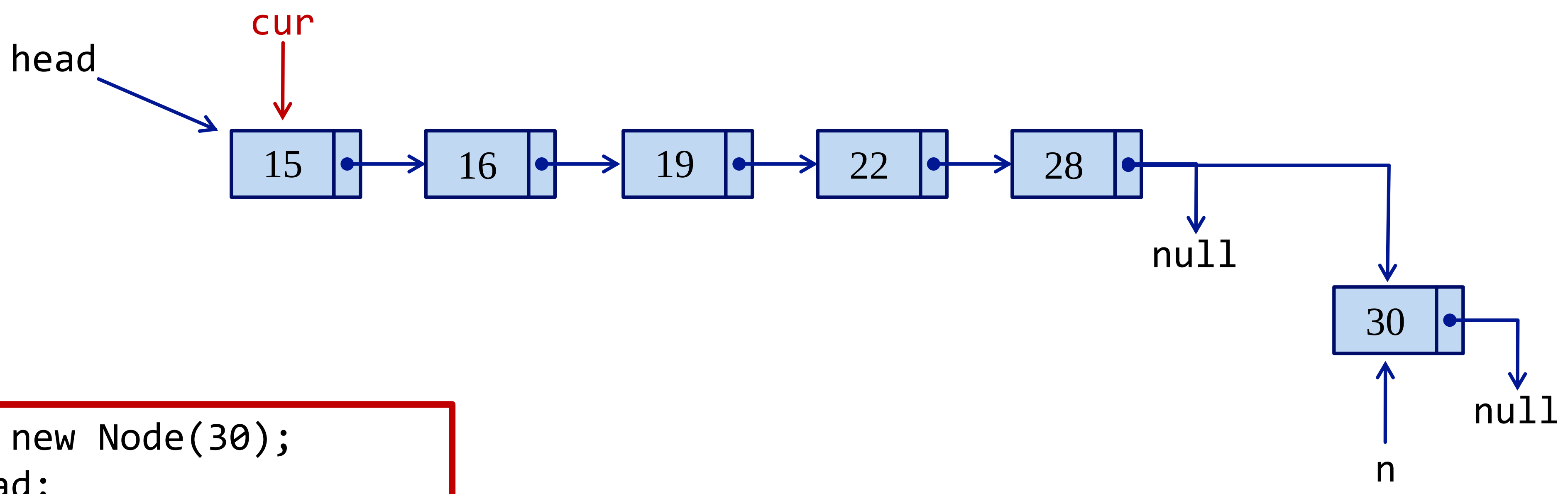
- First, determine where to insert the new node
- Then, update the next pointers appropriately




```
Node n = new Node(12);
n.next = head;
head = n
```

Adding an item to the back of a list

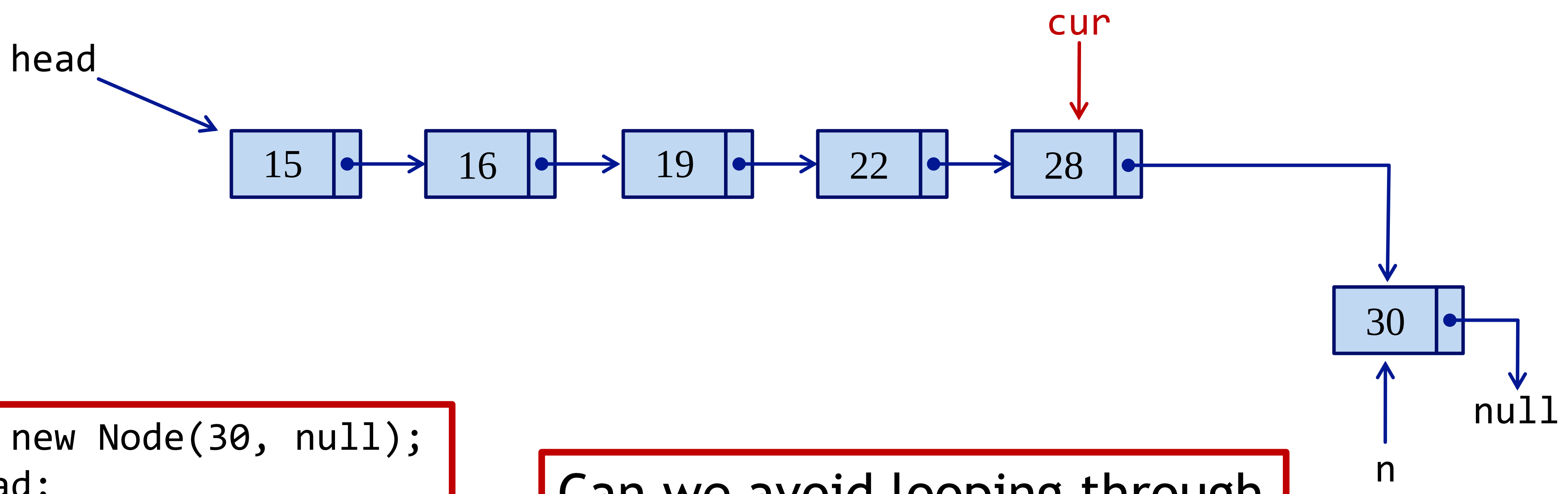
- First, determine where to insert the new node
- Then, update the next pointers appropriately




```
Node n = new Node(30);
cur = head;
while (cur.next != null) {
       cur = cur.next;
}
cur.next = n;
```

Adding an item to the back of a list

- First, determine where to insert the new node
- Then, update the next pointers appropriately

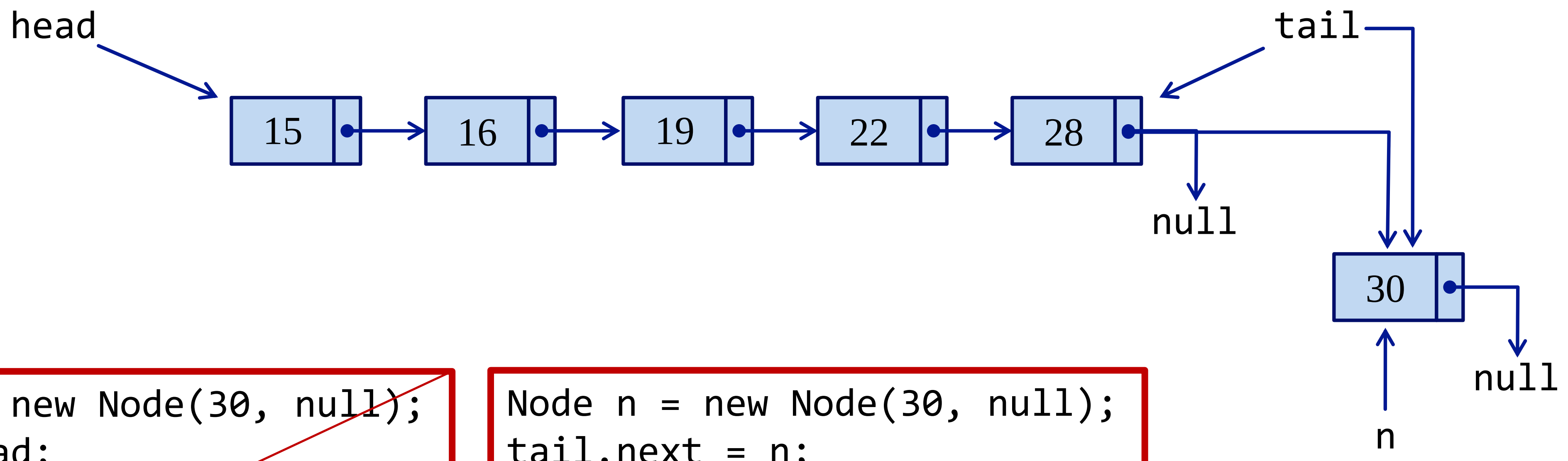



```
Node n = new Node(30, null);
cur = head;
while (cur.next != null) {
        cur = cur.next;
}
cur.next = n;
```

Can we avoid looping through the whole list in order to insert an item at the back?

Tail Reference

- Idea: We have a reference to the front (head) of our list
  - Why don't we do the same with the back (tail)




```
Node n = new Node(30, null);
cur = head;
while (cur.next != null) {
        cur = cur.next;
}
cur.next = n;
```

```
Node n = new Node(30, null);
tail.next = n;
tail = n;
```

Summary

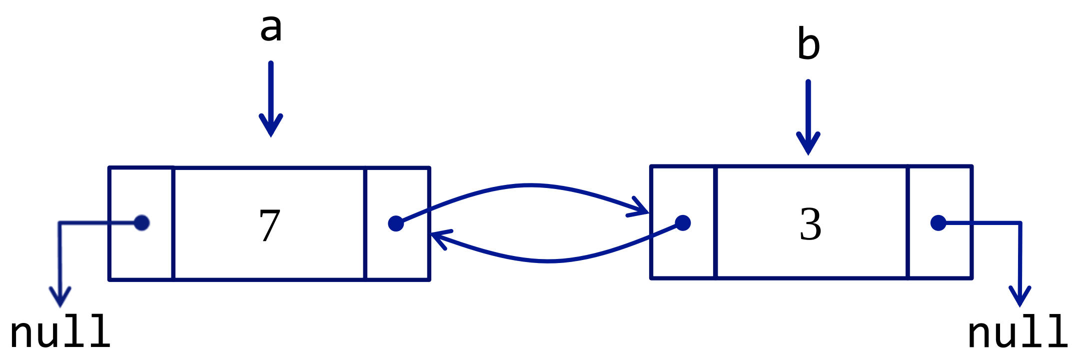
- A linked list allows quick insertion and removal, without the need to reshuffle all other items in the list
  - This allows insertion/removal from the front and back in $O(1)$ time
  - But accessing the middle elements still requires a traversal to get to the location where the insertion or removal should take place. Thus, $O(n)$.

- Next, we will discuss a few variations of a linked list implementation

Doubly-linked list

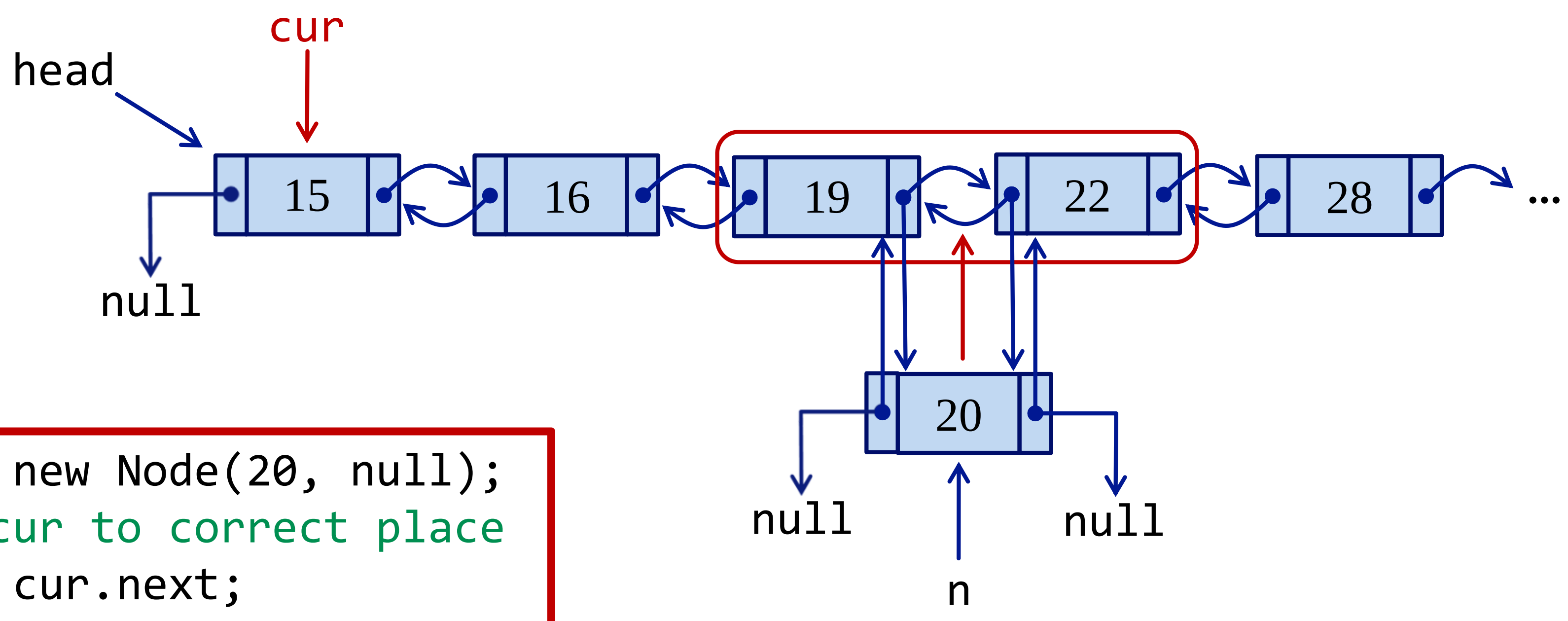
- A **doubly-linked list** is a linked list where each node keeps a reference to both the preceding *and* following nodes in the chain.
- A **node** is a data structure that contains:
  - data (whatever we want to store in the list)
  - a pointer to the location of the next element in the list
  - **(sometimes a pointer to the previous element too)**

```
public class Node {
        private int data;
        private Node prev;
        private Node next;
        …
}
```



Insertion

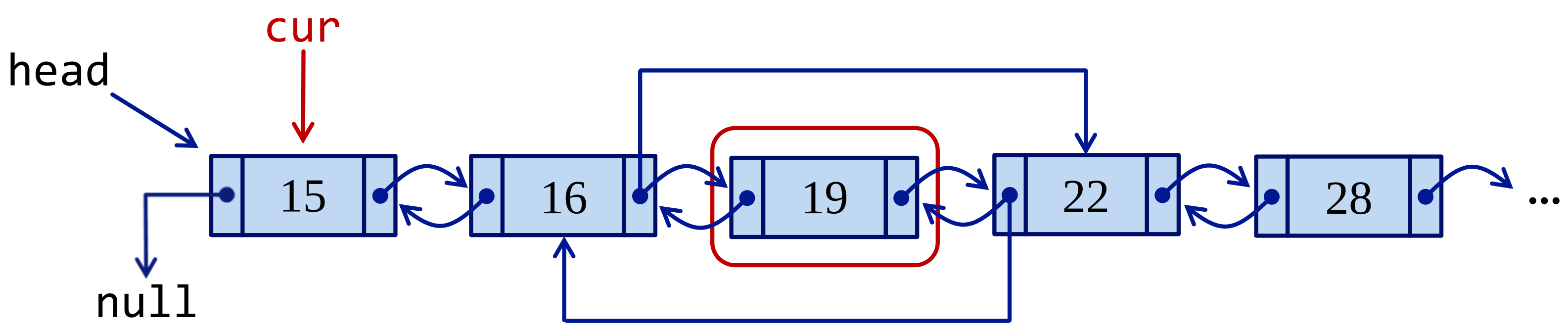
- First, determine where to insert the node
- Then, update pointers so that the order is correct




```
Node n = new Node(20, null);
// move cur to correct place
n.next = cur.next;
n.prev = cur;
cur.next.prev = n;
cur.next = n;
```

Removal

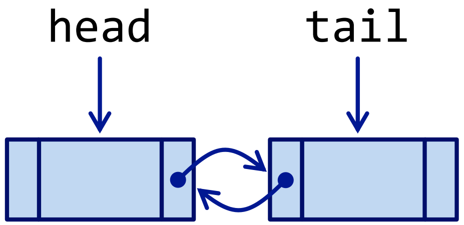
- First, locate the element to remove
- Then, update the next pointers so that the deleted node is skipped



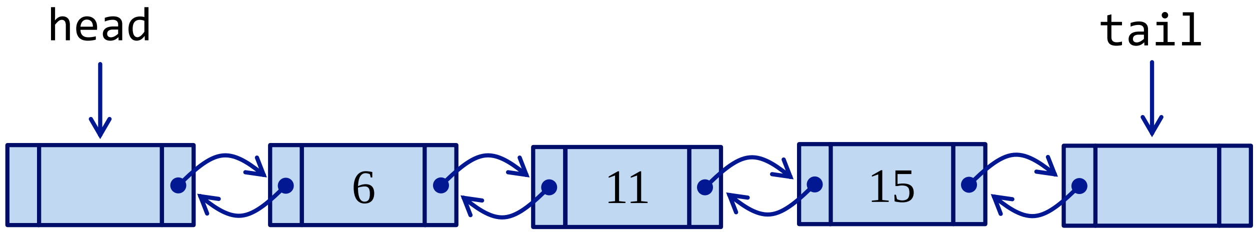

```
cur.next.prev = cur.prev;
cur.prev.next = cur.next;
cur = null;
```

Sentinel Nodes

- A variation of a Linked List implementation is to use **sentinel** nodes
- Sentinel nodes are nodes that go at the front and end of the list
  - They are essentially **head** and **tail** nodes, but they never store any list data!
  - They are just position markers
- An empty list consists of just the sentinel nodes:



- And list elements are added between them:



Position-based List

- In our linked lists, we can think of each node as having a **position**
- We can view a linked list as a container of elements where each element is stored at a position
  - And the positions are arranged in a linear order relative to one another
  - Each position also has a data element (the data being stored at that position)
- A position is defined relative to its neighbours:
  - Position p will always be "after" some position q and "before" some position s

Position-based List

- Using the concept of a position to encapsulate the idea of node in a list, we can define a linked list that supports the following operations:
  - **first():** return the position of the first element in the list
  - **last():** return the position of the last element in the list
  - **before($\boldsymbol{p}$):** return the position of the element in the list preceding p
  - **after($\boldsymbol{p}$):** return the position of the element in the list following p
  - **insertBefore($\boldsymbol{p}$):** insert a new element e into the list before position p
  - **insertAfter($\boldsymbol{p}$):** insert a new element e into the list after position p
  - **remove($\boldsymbol{p}$):** remove the element at position p from the list

Position-based List Implementation

- We will work through a Java example together

Summary

- Running times of the position-based methods:

Method	Time
first()	$O(1)$
last()	$O(1)$
before(p)	$O(1)$
after(p)	$O(1)$
insertBefore(p, e)	$O(1)$
insertAfter(p, e)	$O(1)$
remove(p)	$O(1)$

At first glace, it appears this implementation is clearly better than an array implementation

It is important to remember that accessing an element n spots from the front of the list still requires a traversal ($O(n)$). The array implementation is more efficient for getting or replacing at an index ($O(1)$)

Summary

- Running times of the position-based methods:

Method	Time
first()	$O(1)$
last()	$O(1)$
before(p)	$O(1)$
after(p)	$O(1)$
insertBefore(p, e)	$O(1)$
insertAfter(p, e)	$O(1)$
remove(p)	$O(1)$

It is important to consider the way the program will typically be used.

Will there be frequent insertions before or after another item in the list? A **linked list** doesn't require shuffling of all subsequent items

Will there be frequent accesses from a rank/index? An **array** accesses items at any rank immediately