智能软件与工程学院
School of Intelligent Software and Engineering

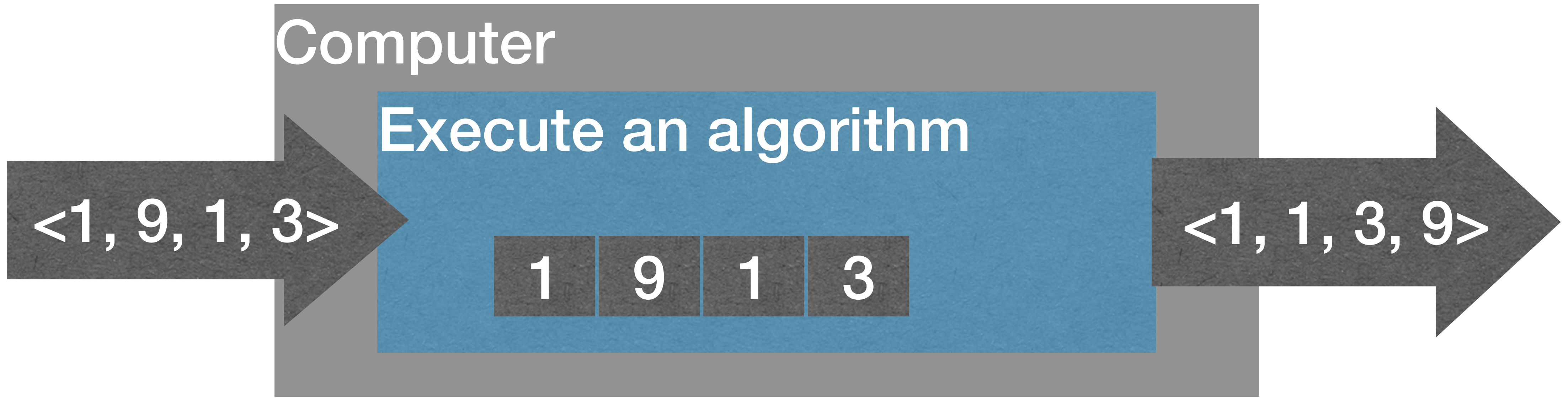
基本数据结构
Basic Data Structures

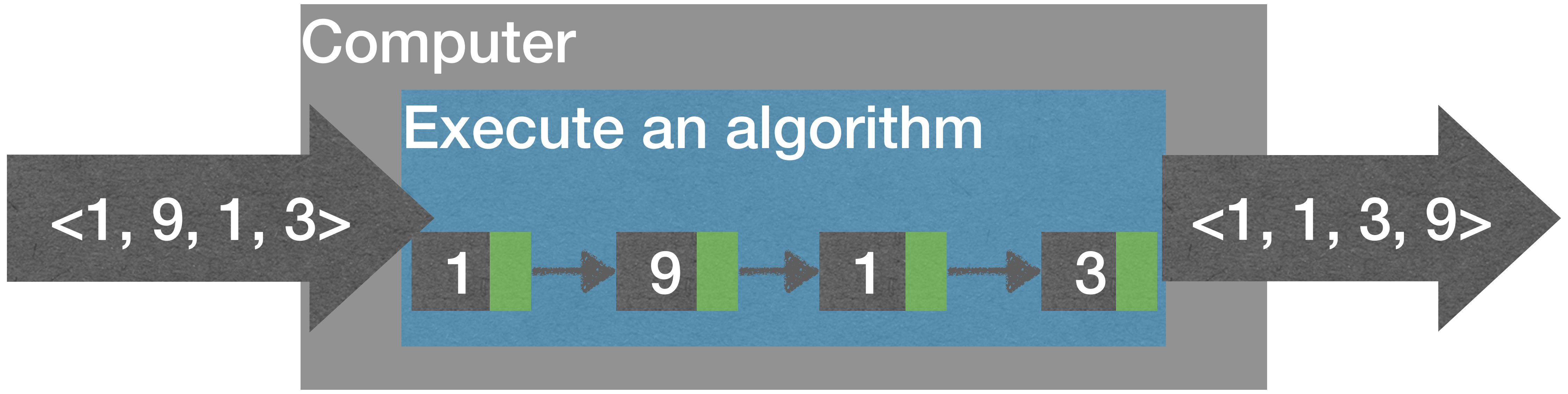
钮鑫涛 王智彬
Nanjing University
2025 Fall

The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne.Thanks for their supports!

What is a “data structure”?

- A data structure is a way to **store and organize data** in order to facilitate **access** and **modifications**.
  - E.g., *array, linked list.*
- Different types of data usually demand different data structures.
- One type of data could be represented by different data structures.





Abstract Data Type (ADT)

- A data structure usually provides an **interface**.
  - Often, the interface is also called an **Abstract Data Type (ADT)**.
  - An ADT specifies what a data structure "can do" and "should do", but *not* "how to do" them.
- ADT: `List`, which supports `get`, `set`, `add`, `remove`, …
- Data structure: `ArrayList, LinkedList, …`
- An ADT is a **logical description**, and a data structure is a **concrete implementation**.
  - Similar to .h file and .cpp file.
  - Different data structures can implement same ADT.

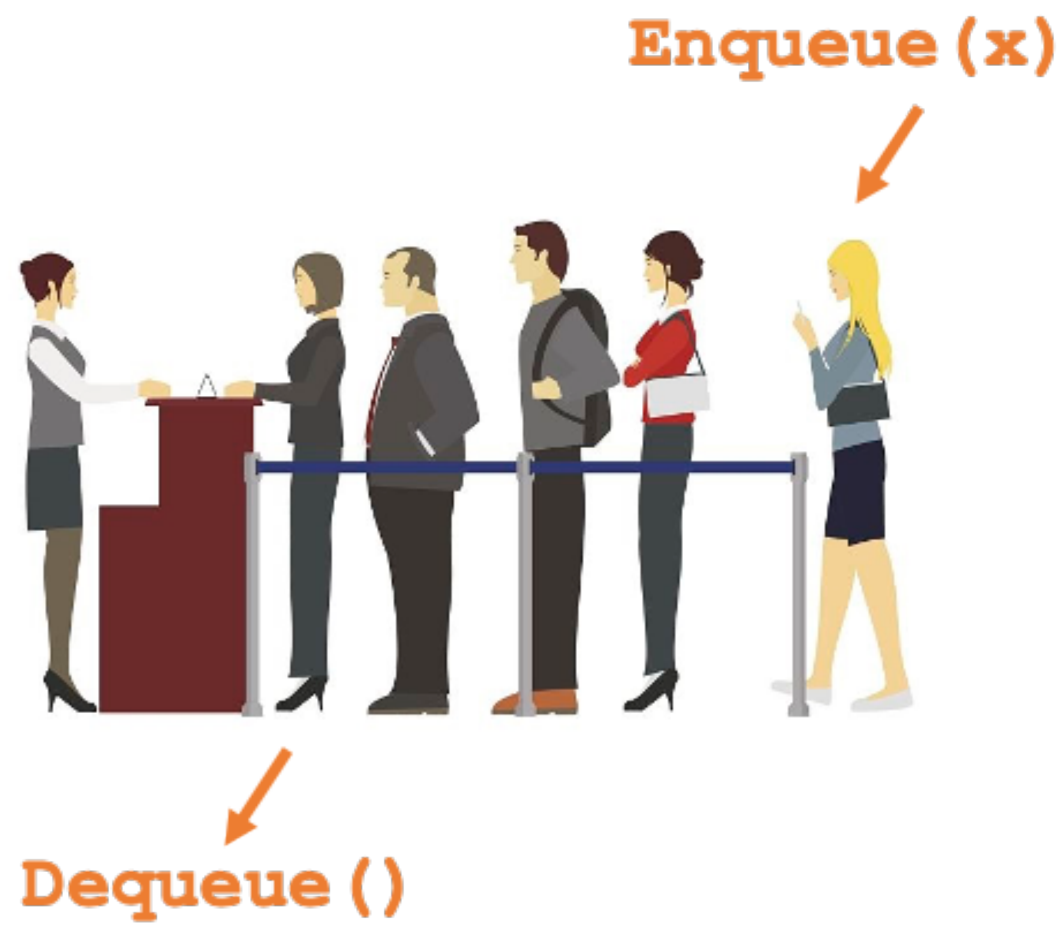
The Queue ADT

- The `Queue` ADT represents a collection of items to which we can **add** items and **remove** the next item.
  - `Add(x)`: add x to the queue.
  - `Remove()`: remove the next item y from queue, return y.
- The *queuing discipline* decides which item to be removed.

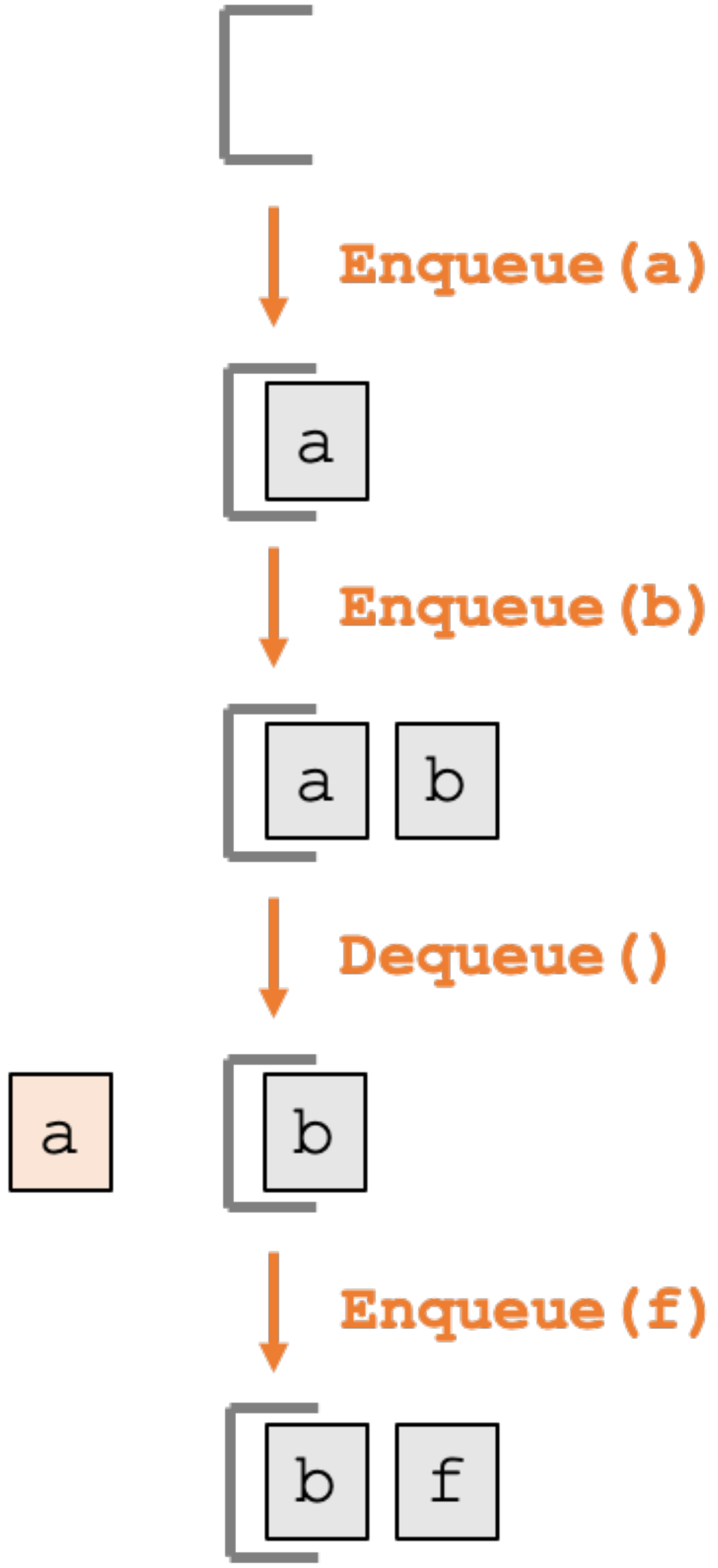
FIFO Queue

> The Queue ADT represents a collection of items to which we can add items and remove the next item.
>
> **Add(x):** add x to the queue.
>
> **Remove():** remove the next item y from queue, return y.

- The **first-in-first-out (FIFO)** queuing discipline: items are removed in the same order they are added.
- **FIFO** Queue:
  - `Add(x)` or **`Enqueue(x)`** : add x to the end of the queue
  - `Remove()` or **`Dequeue()`** : remove the first item from the queue



Example



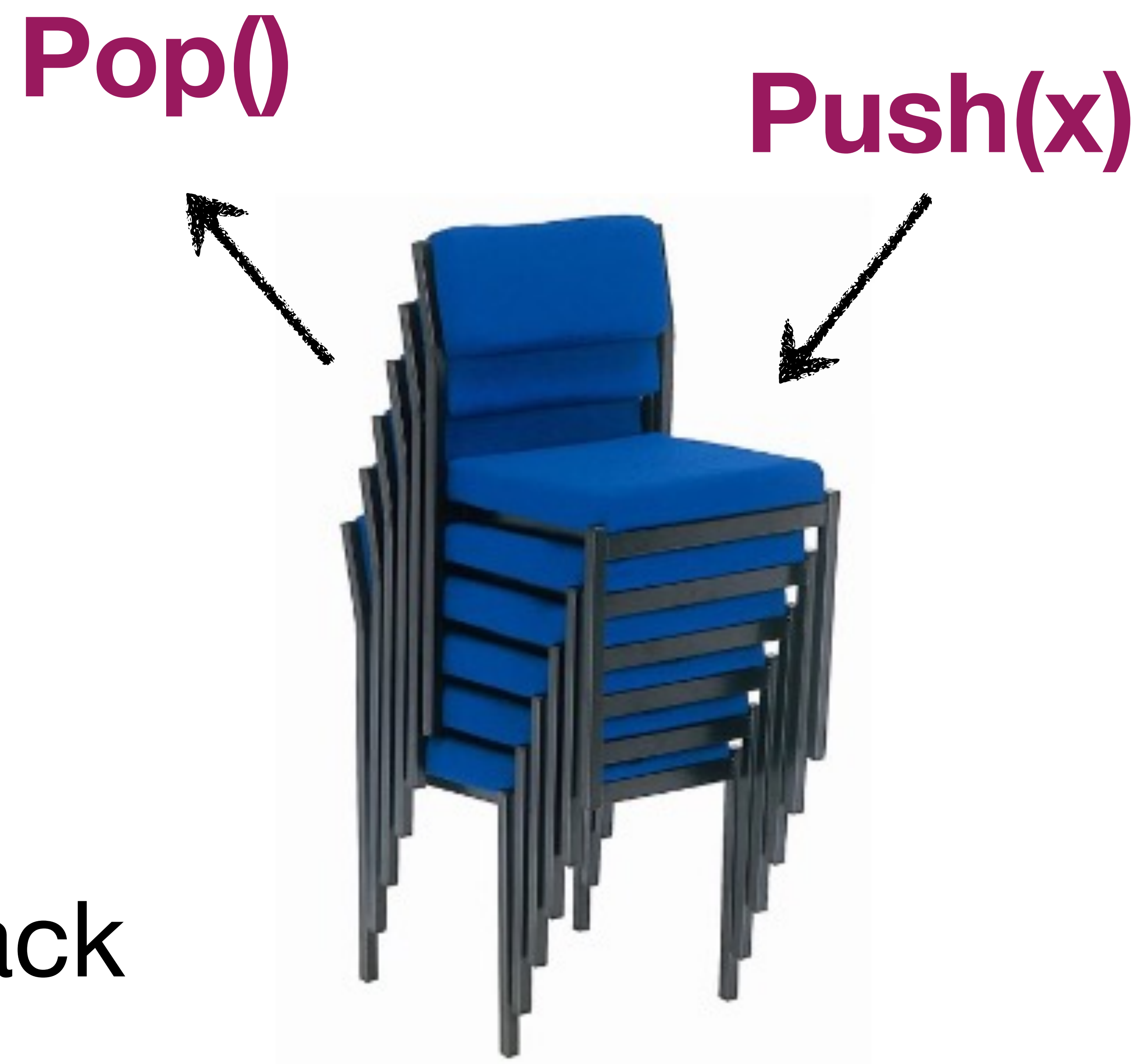
LIFO Queue： Stack

The Queue ADT represents a collection of items to which we can add items and remove the next item.

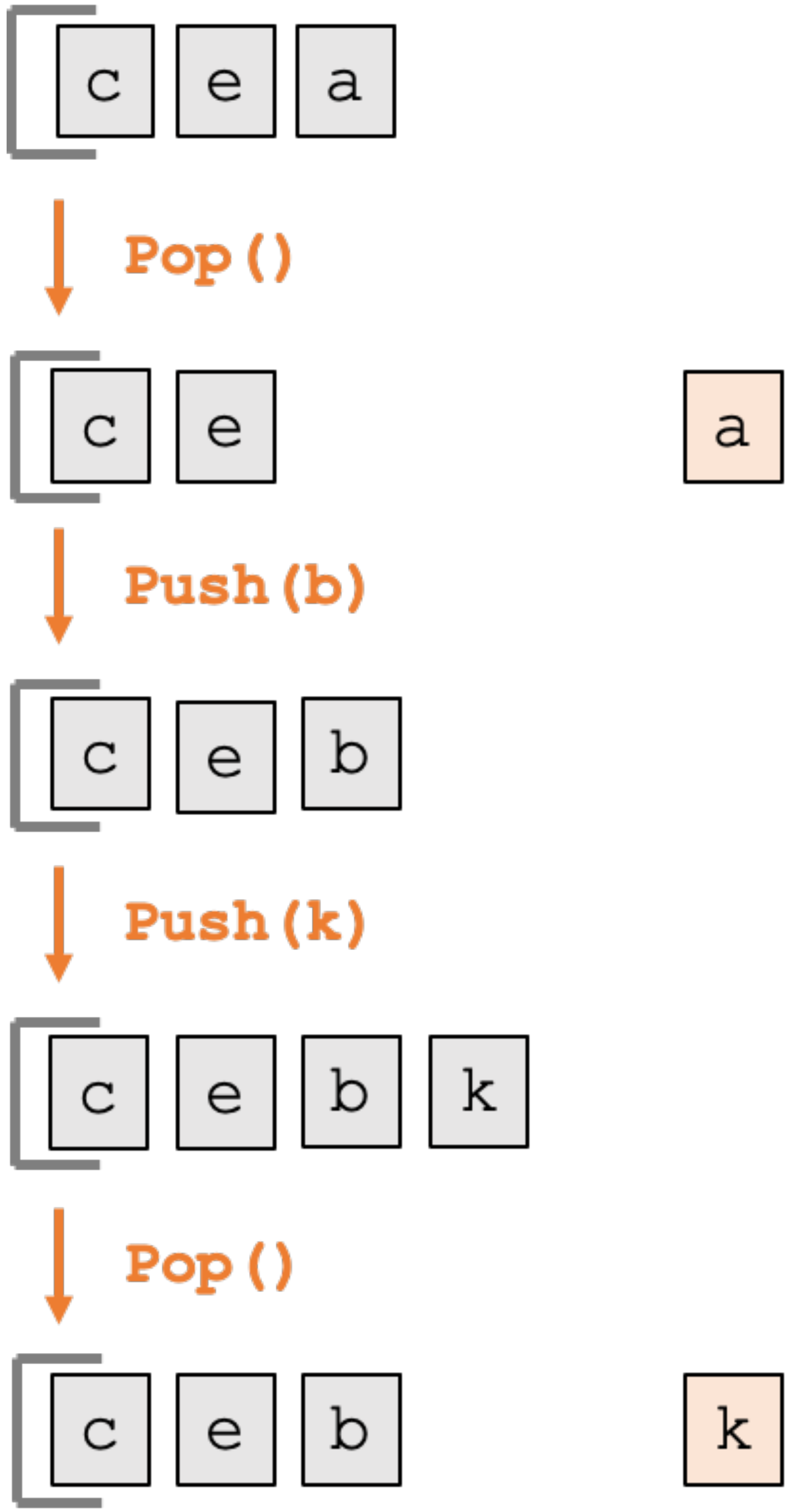
Add(x): add x to the queue.

Remove(): remove the next item y from queue, return y.

- The **last-in-first-out (LIFO)** queuing discipline: the most recently added item is the next one removed
- **Stack** (**LIFO** Queue):
  - `Add(x)` or **`Push(x)`** : add x to the top of the stack
  - `Remove()` or **`Pop()`**: remove the item a the top of the stack

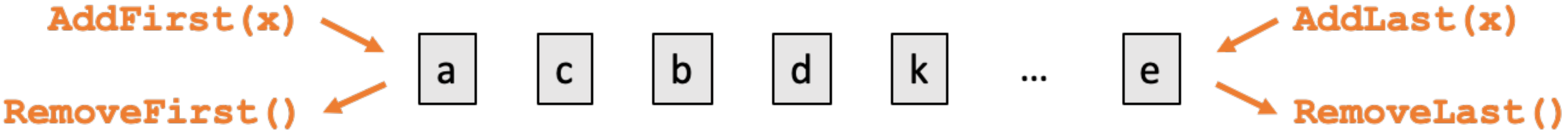


Example



The Deque ADT

- The **Deque** (Double-Ended Queue) ADT represents a sequence of items with a front and a back, which supports the following operations:
  - `AddFirst(x)`: add x to the front of the queue
  - `AddLast(x)`: add x to the back of the queue.
  - `RemoveFirst()`: remove the first item y from queue, return y.
  - `RemoveLast()`: remove the last item y from queue, return y.



The Deque ADT

- A Deque is a generalization of both the FIFO Queue and LIFO Queue (Stack)
  - `Deque` can implement FIFO Queue: `Enqueue(x)` is `AddLast(x)`, `Dequeue()` is `RemoveFirst()`
  - `Deque` can implement `Stack` (LIFO Queue): `Push(x)` is `AddLast(x)`, `Pop()` is `RemoveLast()`

The List ADT

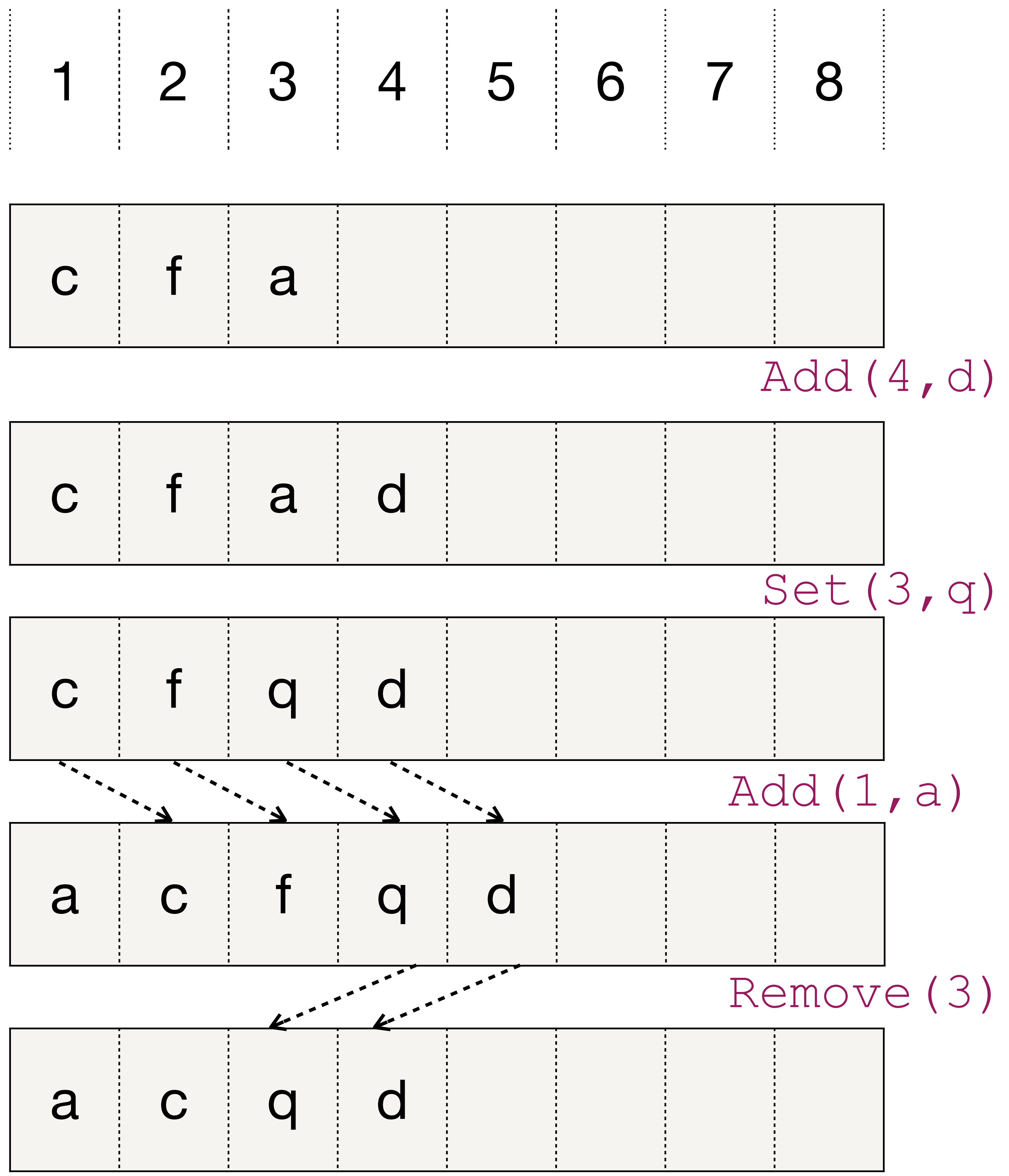
- A **`List`** is a sequence of items $x_1, x_2, \ldots, x_n$, which supports the following operations:
  - `Size()`: return n, the length of the list
  - `Get(i)`: return x_i
  - `Set(i,x)`: set $x_i = x$
  - `Add(i,x)`: set $x_{j+1} = x_j$ for $n \geq j \geq i$, set $x_i = x$, increase list size by 1
  - `Remove(i)`: set $x_j = x_{j+1}$ for $n - 1 \geq j \geq i$, decrease list size by 1

The List ADT

- `List` can implement `Deque`:
  - `AddFirst(x) —> Add(1,x)`
  - `AddLast(x) —> Add(Size()+1,x)`
  - `RemoveFirst() —> Remove(1)`
  - `RemoveLast() —> Remove(Size())`

Using array to implement List — ArrayList

- The list operations implemented by ArrayList
  - `Size()`: always $\Theta(1)$
  - `Get(i)`: always $\Theta(1)$
  - `Set(i,x)`: always $\Theta(1)$
  - `Add(i,x)`: $\Theta(1)$ to $\Theta(n)$
  - `Remove(i)`: $\Theta(1)$ to $\Theta(n)$



Queries and updates are fast

Modifications are fast at “end”, but slow at “front” or “middle”.

Using array to implement List — ArrayList

- The list operations implemented by ArrayList
  - `Size()`: always $\Theta(1)$
  - `Get(i)`: always $\Theta(1)$
  - `Set(i,x)`: always $\Theta(1)$
  - `Add(i,x)`: $\Theta(1)$ to $\Theta(n)$
  - `Remove(i)`: $\Theta(1)$ to $\Theta(n)$

Q: Is `ArrayList` good for `Stack`?

- A: Yes. (`Push` and `Pop` are fast)

Q: Is `ArrayList` good for `FIFO Queue`?

- A: No. Why?

Q: Is `ArrayList` good for `Deque`?

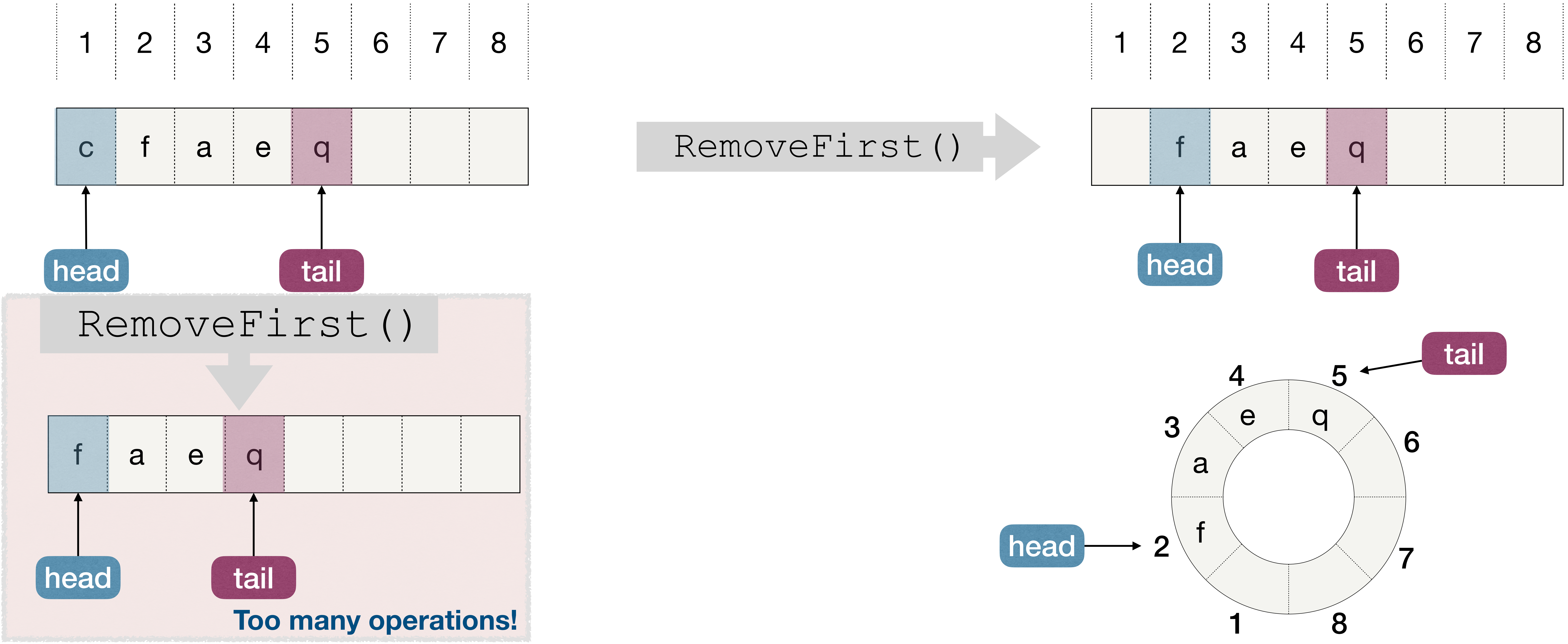
- A: No.

Queries and updates are fast

Modifications are fast at “end”, but slow at “front” or “middle”.

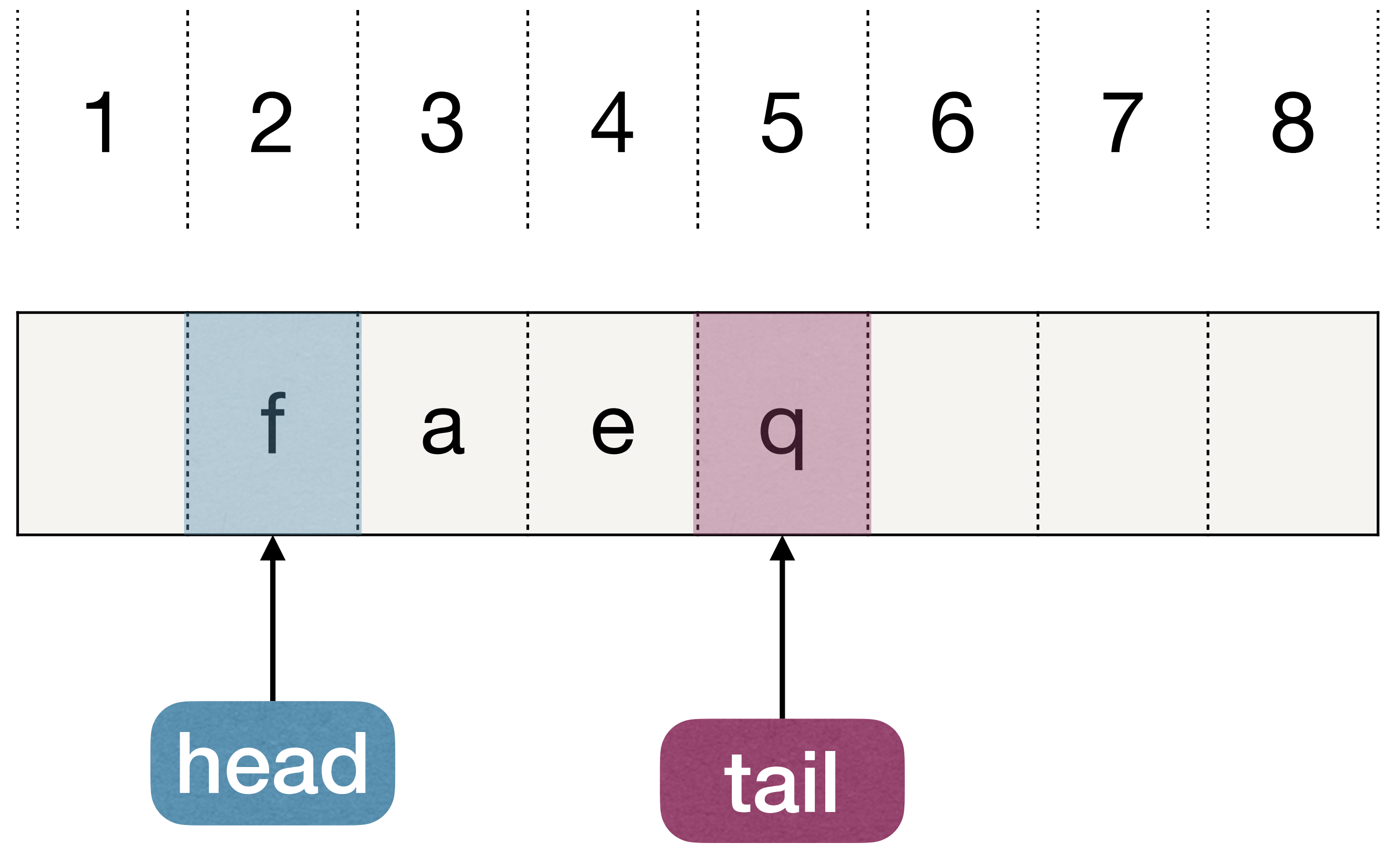
Using circular array to implement Deque — ArrayDeque

- `ArrayList` is good for `Stack`, but not FIFO `Queue` or `Deque`



Using circular array to implement Deque — ArrayDeque

- Maintain `head` and `tail`:
  - `AddFirst` and `RemoveFirst`: move `head`.
  - `AddLast` and `RemoveLast`: move `tail`.
  - Use modular arithmetic to “wrap around” at both ends.



AddLast(x):

$tail := (tail \ \% \ N)+1$

$A[tail] := x$

RemoveLast():

$tail := (tail = 1) \ ? \ N : (tail - 1)$

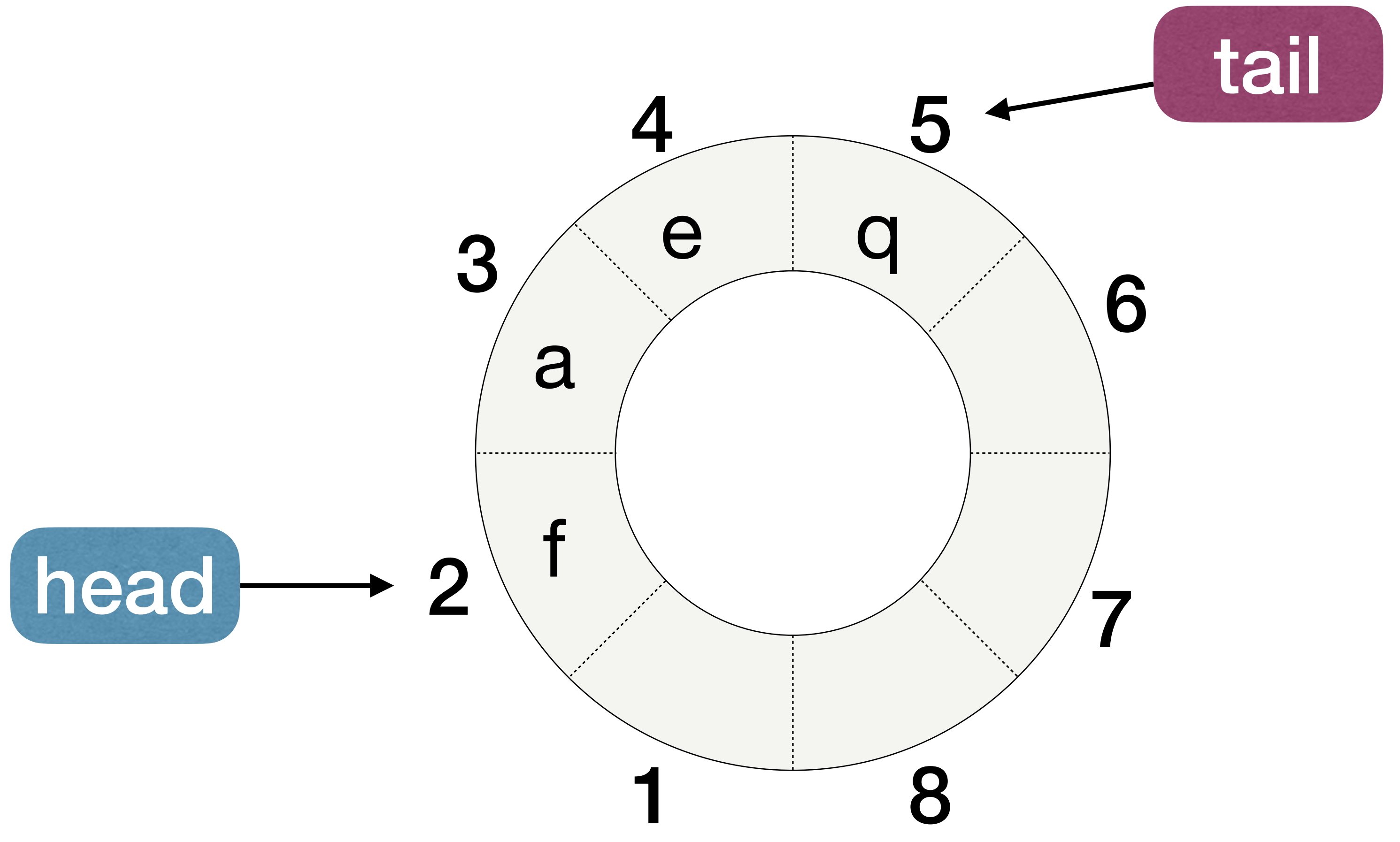
AddFirst(x):

$head := (head = 1) \ ? \ N : (head - 1)$

$A[head] := x$

RemoveFirst():

$head := (head \ \% \ N) + 1$



All of them are $O(1)$

Using circular array to implement Deque — ArrayDeque

- Maintain `head` and `tail`:
  - `AddFirst` and `RemoveFirst`: move `head`.
  - `AddLast` and `RemoveLast`: move `tail`.
  - Use modular arithmetic to "wrap around" at both ends.

AddLast(x):

$tail := (tail \% N)+1$

$A[tail] := x$

RemoveLast():

$tail := (tail = 1)\ ?\ N : (tail - 1)$

AddFirst(x):

$head := (head = 1)\ ?\ N : (head - 1)$

$A[head] := x$

RemoveFirst():

$head := (head \% N) + 1$

All of them are $O(1)$

- Queries and updates are fast
- Modifications are fast at "front" and "end" (i.e., `head` and `tail`), but still slow at "middle".
- `ArrayDeque` is good for `Stack`, FIFO `Queue`, and `Deque`; but can be slow for some `List` operations.
- Capacity of array is also a problem!

When the array is full?

- Resizing arrays
  - Create a new array of greater size and copy the elements of the original array into it.
  - abandon the old array and use the new one in its place.
- The question is, how large?

When the array is full?

- Suppose we have array with initial capacity being 1, then insert *N* items
  - Resize it to have one additional cell every time? —> requiring $1 + 2 + 3 + \ldots N - 1 \sim N^2$ copy operations.
  - Resize the array by doubling its size every time?
    - For simplicity, let $N = 2^k$ for some constant k. —> requiring $1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1 \sim N$
  - We could of course do better if we multiplied the size of the array by an even larger value, but then there would likely be a lot more unused cells in the array on average (consider the case that resizing happens infrequently).

Amortized analysis

- Starting from an empty data structure, average running time per operation over a worst-case sequence of operations.
- Thus, if resizing by one more cell each time, the amortized complexity is $\Theta(n)$ for **each operation**.
- If resizing by doubling space each time, the amortized complexity is $\Theta(1)$ for **each operation**.
- We well learn it later…

What about worst?

Introduced by ***Robert Tarjan*** **at 1985**

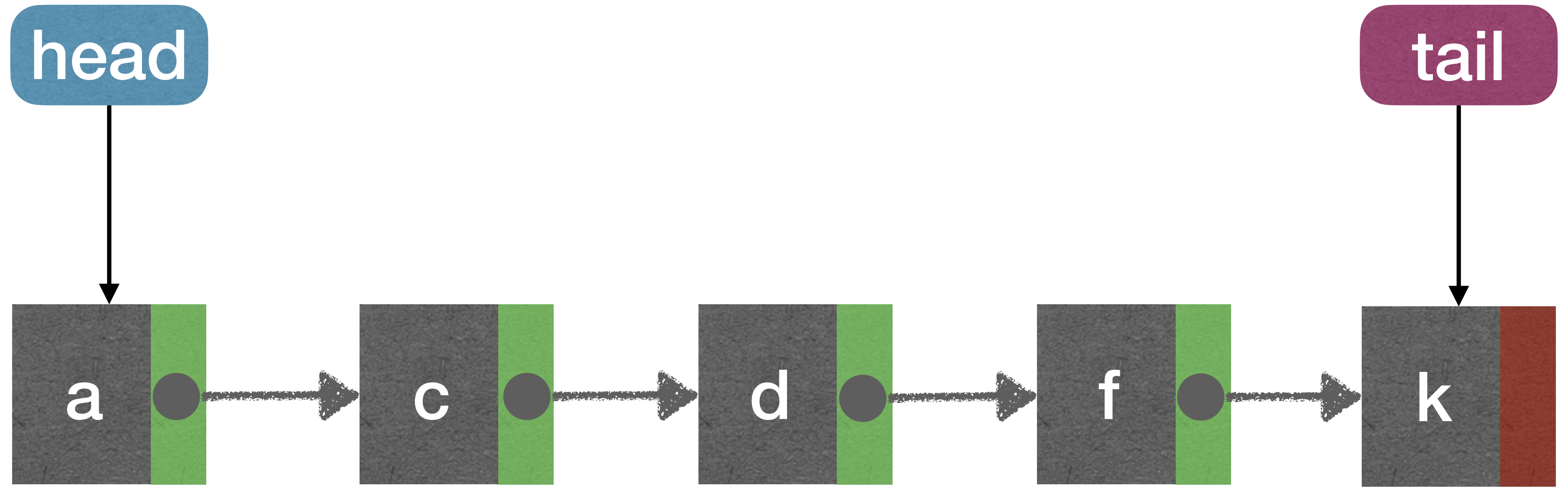
When to shrink array?

- When shrinking an array, we allocate a new array with smaller capacity, and then copy necessary items from the original array into the new array, and abandon the old array.
  - When pop() each time, we shrink the array by 1 less cell?
  - When the array is one-half full, we shrink the array to the halve size?
    - Causing "Thrashing" problem!!! Since, if now we add just one element, we need to resize the array by doubling the size, and then pop one element, we should shrink it back to the halve size —> When pushes and pops come with relatively equal frequency, it will be too expensive!
  - Usually, when the array is 1/4 full, we shrink it to the halve size.
- After all, by doing this we ensure that the array holding the contents of our stack will ALWAYS be between 25% and 100% full!

Using Linked list to implement List — LinkedList

- The list operations implemented by LinkedList
  - `Size()`: always $\Theta(1)$
  - `Get(i)`: $\Theta(1)$ to $\Theta(n)$
  - `Set(i,x)`: $\Theta(1)$ to $\Theta(n)$
  - `Add(i,x)`: $\Theta(1)$ to $\Theta(n)$
  - `Remove(i)`: $\Theta(1)$ to $\Theta(n)$

Traversing backwards from tail is not efficient!



Q: Is `LinkedList` good for `Stack`?

- A: Yes. (`Push` and `Pop` **at head** are fast)

Q: Is `LinkedList` good for `FIFO Queue`?

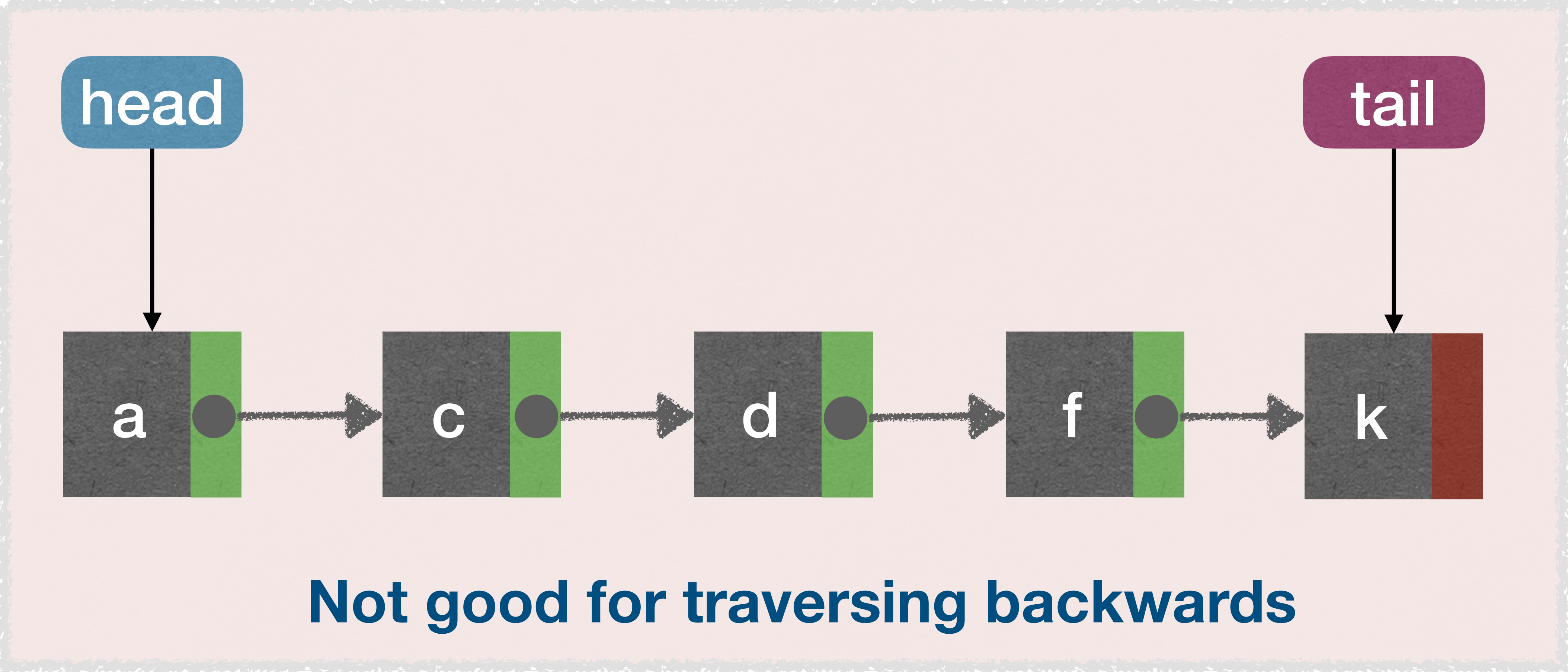
- A: Yes. (`Enqueue` and `Dequeue` are fast)

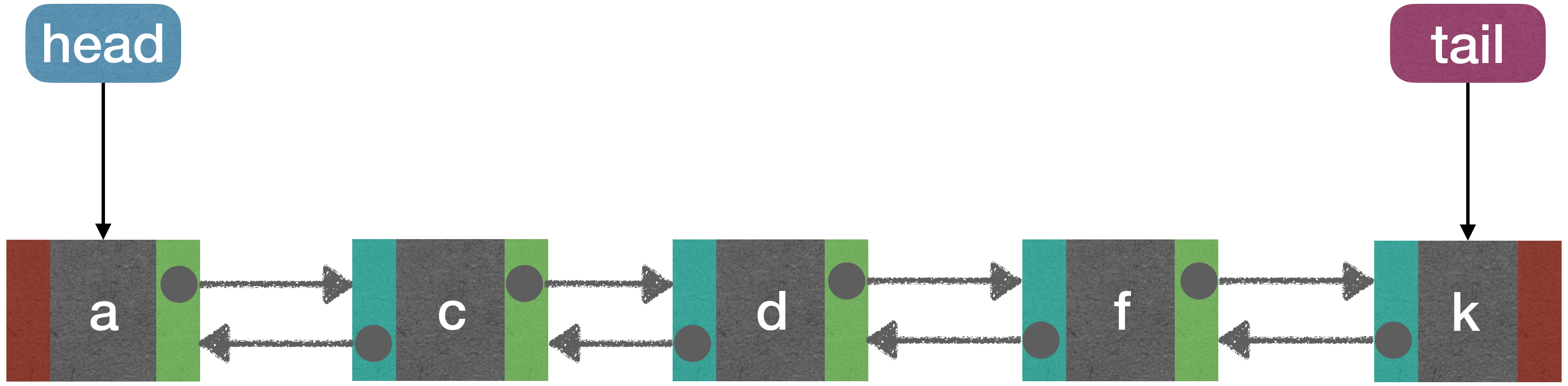
Q: Is `LinkedList` good for `Deque`?

- A: No.(`RemoveLast` can be slow.)

Using doubly-Linked list to implement List — DLinkedList

- The list operations implemented by DLinkedList
  - `Size()`: always $\Theta(1)$
  - `Get(i)`: $\Theta(1)$ to $\Theta(n)$
  - `Set(i,x)`: $\Theta(1)$ to $\Theta(n)$
  - `Add(i,x)`: $\Theta(1)$ to $\Theta(n)$
  - `Remove(i)`: $\Theta(1)$ to $\Theta(n)$

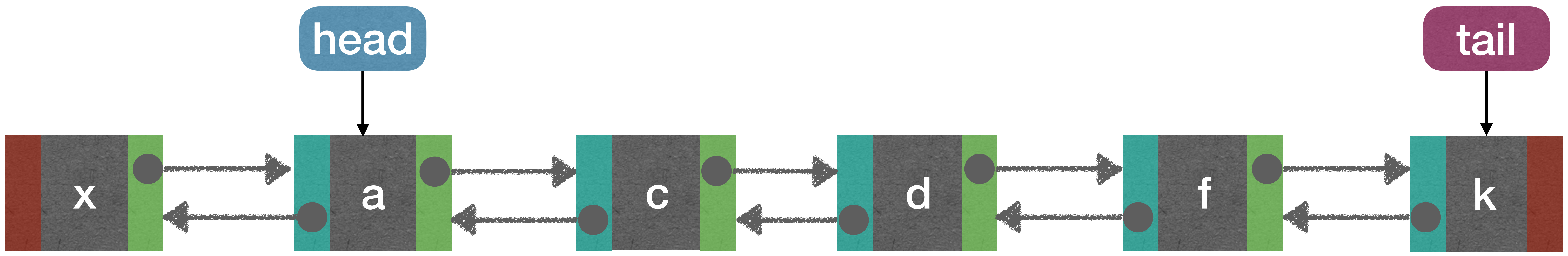




`DLinkedList` **is good for** `Stack`**, FIFO** `Queue`**, and** `Deque`**; but can be slow for some** `List` **operations.**

Using doubly-Linked list to implement List — DLinkedList

- The list operations implemented by DLinkedList
  - `Size()`: always $\Theta(1)$
  - `Get(i)`: $\Theta(1)$ to $\Theta(n)$
  - `Set(i,x)`: $\Theta(1)$ to $\Theta(n)$
  - `Add(i,x)`: $\Theta(1)$ to $\Theta(n)$
  - `Remove(i)`: $\Theta(1)$ to $\Theta(n)$



AddFirst(x):

$x.next := head$
$head.prev := x$
$head := x$
$x.prev := NULL$

What if `head==NULL`? →

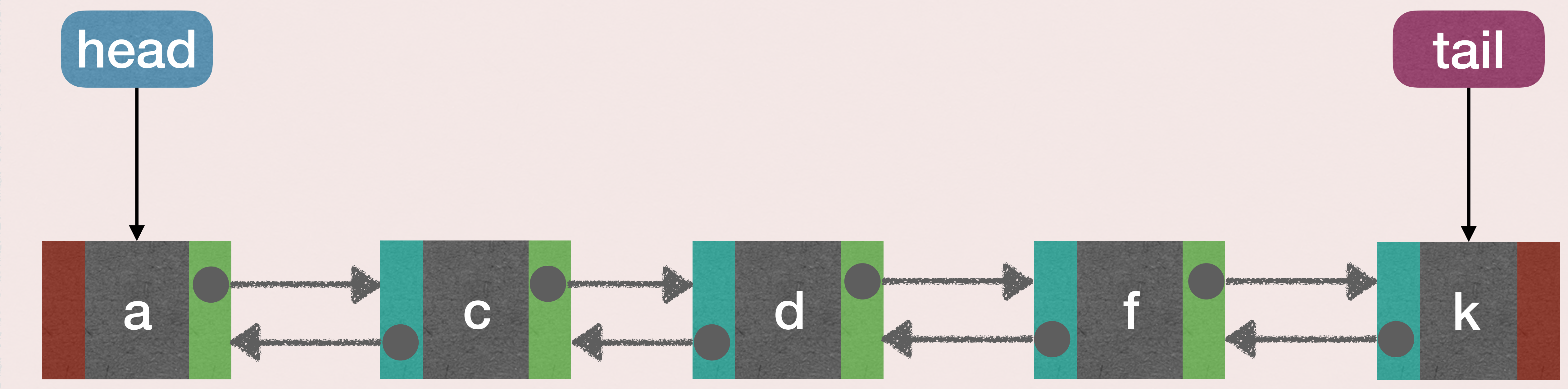
AddFirst(x):

$x.next := head$
if $head \mathrel{!=} NULL$
  $head.prev := x$
$head := x$
$x.prev := NULL$

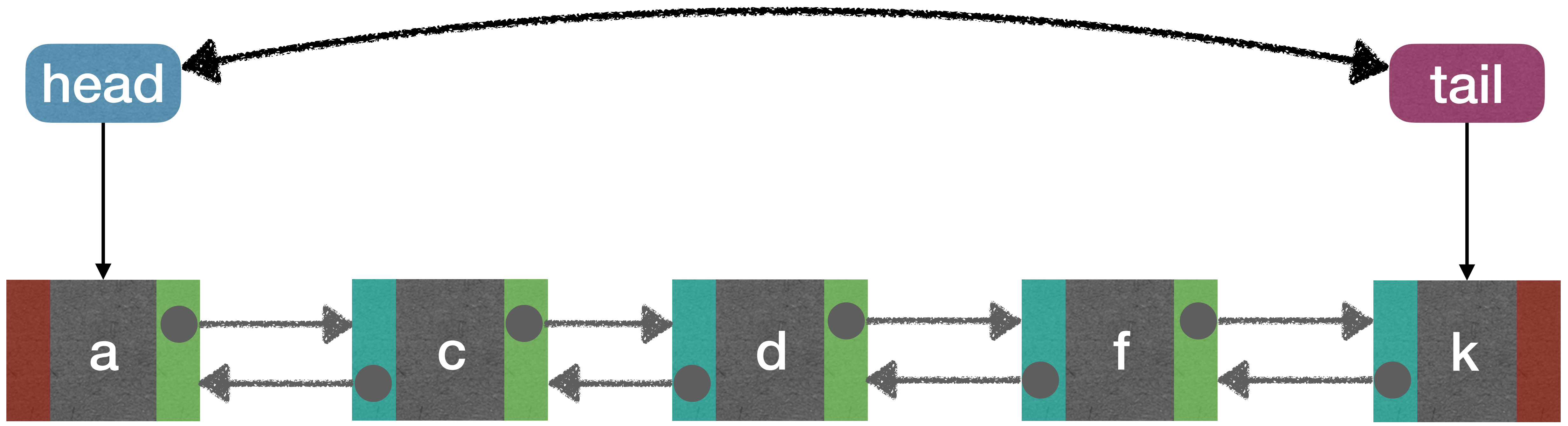
What about `tail`?

Using doubly-Linked list to implement List — DLinkedList

- The list operations implemented by DLinkedList
  - `Size()`: always $\Theta(1)$
  - `Get(i)`: $\Theta(1)$ to $\Theta(n)$
  - `Set(i,x)`: $\Theta(1)$ to $\Theta(n)$
  - `Add(i,x)`: $\Theta(1)$ to $\Theta(n)$
  - `Remove(i)`: $\Theta(1)$ to $\Theta(n)$



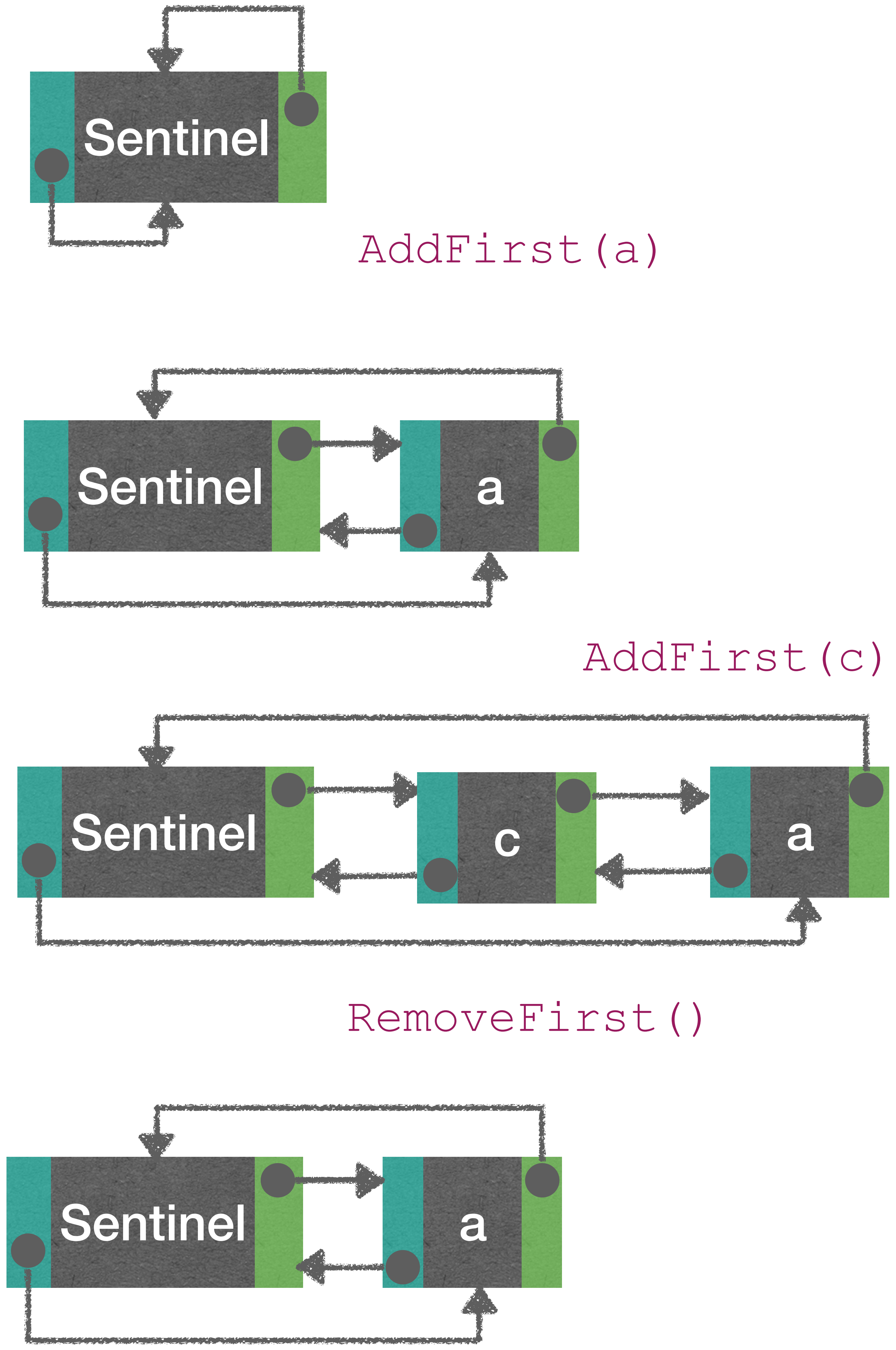
Similar operations for head and tail, and need to consider boundary conditions



Can we connect them?

Using doubly-Linked list to implement List — DLinkedList

- A circular, doubly linked list with a sentinel:
  - A sentinel node is a dummy node used as an alternative over using NULL as the path terminator
  - The sentinel's **next** points to the first node on the list, and its **prev** points to the last node on the list.
  - The first node's **prev** points to the sentinel, as does the last node's **next**.



AddFirst(x):

$x.next := Sentinel.next$

$Sentinel.next.prev := x$

$Sentinel.next := x$

$x.prev := Sentinel$

RemoveFirst():

$Sentinel.next := Sentinel.next.next$

$Sentinel.next.prev := Sentinel$

Using sentinel can marginally increased speed of operations

Summary util now

- `Queue` ADT: FIFO `Queue`, `Stack` (LIFO `Queue`), `Deque`
- `List` ADT: can implement various `Queue`
- Array based implementations (simple/circular):
  - Queries are fast, updates (i.e., `Set`) are also fast
  - Modifications (i.e., `Add` and `Remove`) are fast at “start” and “end”, but slow in “middle”
  - Capacity can be a problem
- Linked list based implementations (singly/doubly linked):
  - Operations (queries, updates, and modifications) are fast at “start” and “end”, but slow in “middle”
  - No capacity issue

Applications of basic data structures

Application of FIFO Queue

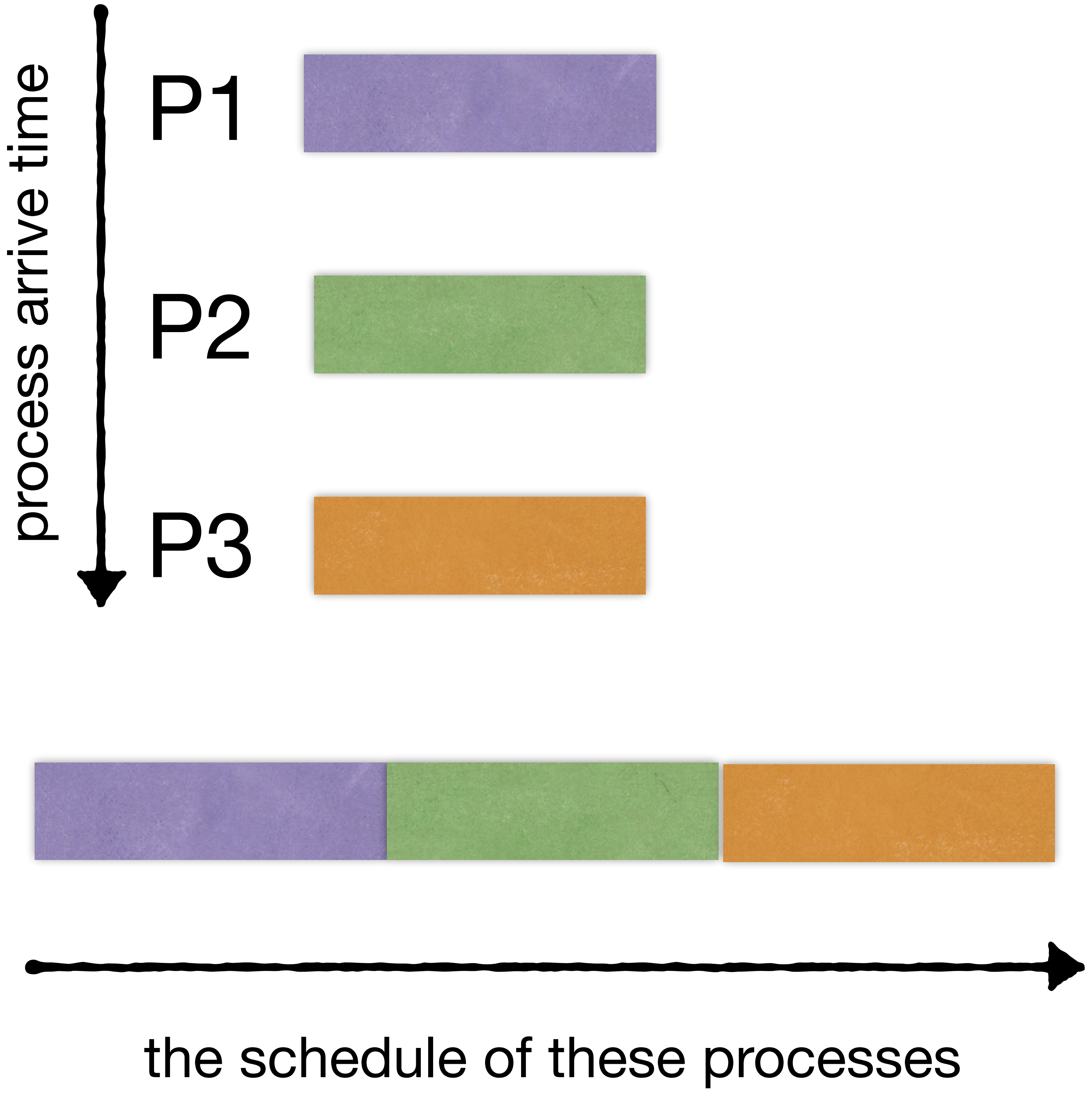
Process Scheduling

ProcessIn(x):

$FIFOqueue.add(x)$

ProcessOut():

$FIFOqueue.remove()$



Application of Stack

Balancing Symbols

- Compiler needs to check whether the parentheses (), brackets [], and braces {} are **matched**.

CheckParen(str):

```
Stack s
int i := 1
while str[i] != NULL
    if str[i] is '(' or '[' or '{'
        s.push(str[i])
    if str[i] is ')' or ']' or '}'
        if s.empty()
            return false
        if s.pop() and str[i] mismatch
            return false
    i++
return s.empty()
```

if(a > b) {b = c[10];} ( ) { [ ] }

if(a > b) {b = c[10]; ( ) { [ ]

if(a > b)) {b = c[10];} ( ))

if(a > b) {b = c[10);} ( ) { [ )

Application of Stack

Function Calls

- How does a function call work?
- Example：
  - Alice: only knows addition.
  - Bob: only knows multiplication.
  - Question: 100+234+35×45+25

```
FuncAlice():
    sum :=100+234
    temp := FuncBob(35,45)
    sum += temp
    sum += 25
    return sum
FuncBob(a,b):
    c=a*b
    return c
```

sum:
temp:

Application of Stack

Function Calls

- How does a function call work?
- Example：
  - Alice: only knows addition.
  - Bob: only knows multiplication.
  - Question: 100+234+35×45+25

FuncAlice():

$sum := 100+234$ ←

$temp := FuncBob(35,45)$

$sum += temp$

$sum += 25$

return sum

FuncBob(a,b):

$c=a*b$

return c

sum: 334
temp:

Application of Stack

Function Calls

- How does a function call work?
- Example：
  - Alice: only knows addition.
  - Bob: only knows multiplication.
  - Question: 100+234+35×45+25

FuncAlice():
  $sum := 100+234$
  $temp := FuncBob(35,45)$ ←
  $sum \mathrel{+}= temp$
  $sum \mathrel{+}= 25$
  return sum

FuncBob(a,b):
  $c=a*b$
  return c

sum: 334
temp:
b: 35
a: 45

Application of Stack

Function Calls

- How does a function call work?
- Example：
  - Alice: only knows addition.
  - Bob: only knows multiplication.
  - Question: 100+234+35×45+25

```
FuncAlice():
    sum :=100+234
    temp := FuncBob(35,45)   ←
    sum += temp
    sum += 25
    return sum
FuncBob(a,b):
    c=a*b
    return c
```

sum: 334
temp:
b: 35
a: 45
return address

Application of Stack

Function Calls

- How does a function call work?
- Example：
  - Alice: only knows addition.
  - Bob: only knows multiplication.
  - Question: 100+234+35×45+25

```
FuncAlice():
    sum :=100+234
    temp := FuncBob(35,45)   ←
    sum += temp
    sum += 25
    return sum
FuncBob(a,b):                ←
    c=a*b
    return c
```

sum: 334
temp:
b: 35
a: 45
return address
c:

Application of Stack

Function Calls

- How does a function call work?
- Example：
  - Alice: only knows addition.
  - Bob: only knows multiplication.
  - Question: 100+234+35×45+25

FuncAlice():

sum :=100+234

temp := *FuncBob*(35,45) ←

sum += *temp*

sum += *25*

return *sum*

FuncBob(a,b):

c=*a***b* ←

return *c*

sum: 334
temp:
b: 35
a: 45
return address
c: 1575

Application of Stack

Function Calls

- How does a function call work?
- Example：
  - Alice: only knows addition.
  - Bob: only knows multiplication.
  - Question: 100+234+35×45+25

```
FuncAlice():
    sum :=100+234
    temp := FuncBob(35,45)   ←
    sum += temp
    sum += 25
    return sum
FuncBob(a,b):
    c=a*b
    return c   ←
```

EAX: 1575

sum: 334
temp:
b: 35
a: 45
return address
c: 1575

Application of Stack

Function Calls

- How does a function call work?
- Example：
  - Alice: only knows addition.
  - Bob: only knows multiplication.
  - Question: 100+234+35×45+25

FuncAlice():

sum :=100+234

temp := *FuncBob*(35,45) ←

sum += *temp*

sum += 25

return *sum*

FuncBob(a,b):

c=*a***b*

return *c* ←

EAX: 1575

sum: 334
temp:
b: 35
a: 45
return address

Application of Stack

Function Calls

- How does a function call work?
- Example：
  - Alice: only knows addition.
  - Bob: only knows multiplication.
  - Question: 100+234+35×45+25

EAX: 1575

sum: 334
temp:
b: 35
a: 45

```
FuncAlice():
    sum :=100+234
    temp := FuncBob(35,45)   ←
    sum += temp
    sum += 25
    return sum
FuncBob(a,b):
    c=a*b
    return c                 ←
```

Application of Stack

Function Calls

- How does a function call work?
- Example：
  - Alice: only knows addition.
  - Bob: only knows multiplication.
  - Question: 100+234+35×45+25

```
FuncAlice():
    sum :=100+234
    temp := FuncBob(35,45)   ←
    sum += temp
    sum += 25
    return sum
FuncBob(a,b):
    c=a*b
    return c                 ←
```

EAX: 1575

sum: 334
temp: 1575
b: 35
a: 45

Application of Stack

Function Calls

- How does a function call work?
- Example：
  - Alice: only knows addition.
  - Bob: only knows multiplication.
  - Question: 100+234+35×45+25

```
FuncAlice():
    sum :=100+234
    temp := FuncBob(35,45)
    sum += temp          ←
    sum += 25
    return sum
FuncBob(a,b):
    c=a*b
    return c             ←
```

EAX: 1575

sum: 1909
temp: 1575
b: 35
a: 45

Application of Stack

Function Calls

- How does a function call work?
- Example：
  - Alice: only knows addition.
  - Bob: only knows multiplication.
  - Question: 100+234+35×45+25

FuncAlice():

sum :=100+234
temp := *FuncBob*(35,45)
sum += *temp*
sum += *25*
return *sum*

FuncBob(a,b):

c=*a***b*
return *c*

EAX: 1575

sum: 1934
temp: 1575
b: 35
a: 45

Application of Stack

Function Calls

- How does a function call work?
- Example：
  - Alice: only knows addition.
  - Bob: only knows multiplication.
  - Question: 100+234+35×45+25

FuncAlice():

sum :=100+234
temp := *FuncBob*(35,45)
sum += *temp*
sum += 25
return *sum*

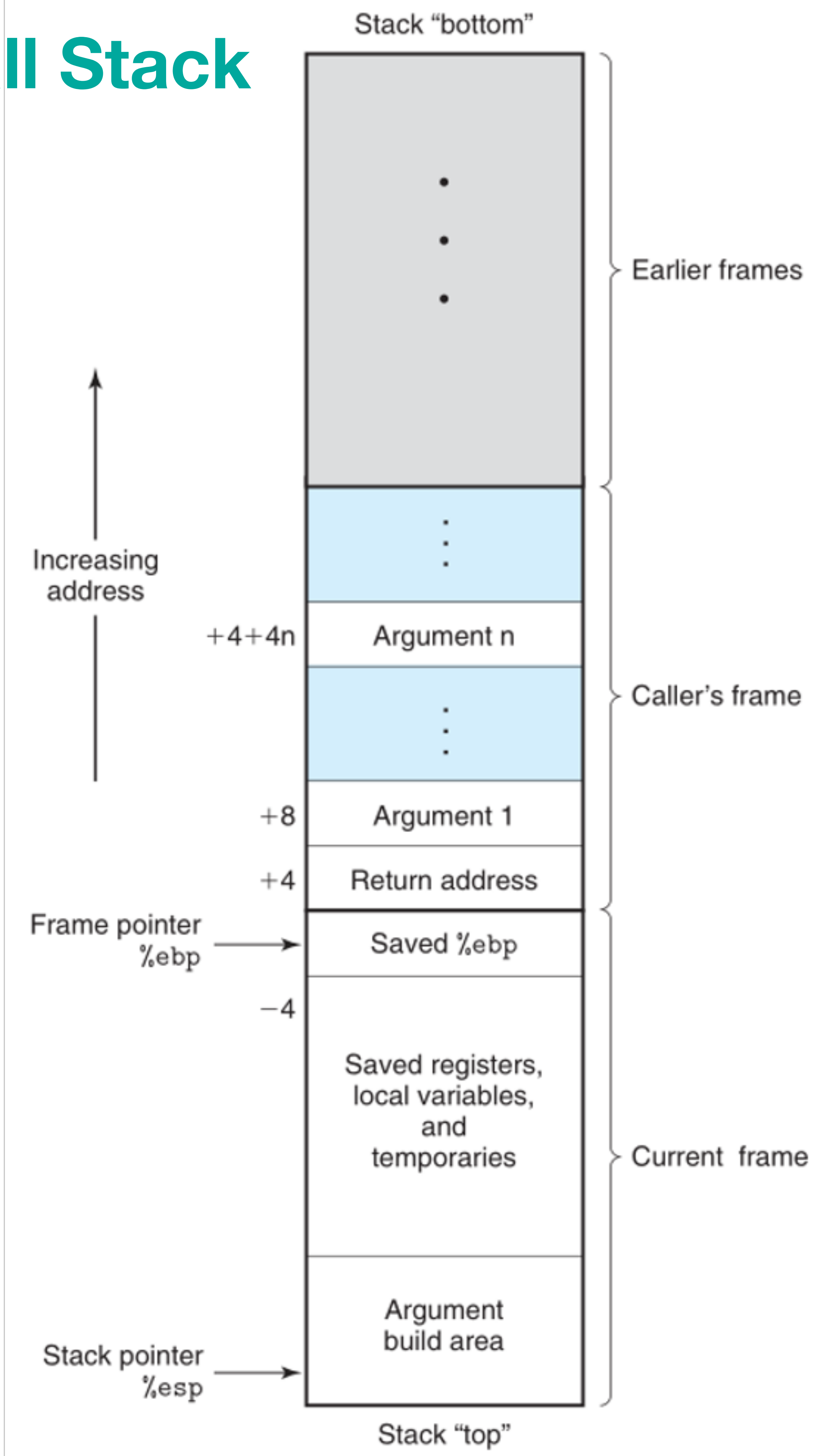
FuncBob(a,b):

c=*a***b*
return *c*

EAX: 1575

sum: 1934
temp: 1575
b: 35
a: 45

Call Stack



Eliminating Recursion

- Function calls are implemented via a "call stack"
- Recursion is a specific type of function call

With the help of a stack, recursion can be replaced by iteration

```
FactRec(val):
if  val = 1
    acc := 1
else
    acc := FactRec(val - 1)
res := val*acc
return res
```

```
class Frame {
  int val
  int acc
  Frame prevFrame
}
```

```
FactIter(n):
Stack s
s.push(Frame(n, -1, NULL))
while !s.empty()
    frame := s.peek()
    if  frame.val <= 1
        frame.acc := 1
    if  frame.acc != -1
        res := (frame.val)*(frame.acc)
        if  frame.prevFrame!=NULL
                (frame.prevFrame).acc := res
        s.pop()
    else
        s.push(Frame(frame.val - 1, -1, frame))
return res
```

Get the top element of the stack

Eliminating Recursion

- Q: Why recursion can be undesirable?
  - A: Recursion can be slow and memory consuming due to the creation and maintenance of stack frames.
- Q: Why recursion can be desirable?
  - A: Recursion can make the code clearer, concise, and intuitive.

Tail recursion

- A function is called **tail-recursive** if each activation of the function will make **at most** one single recursive call, and will **return immediately after that call**.

```
FactRec(n):
if n = 1
  return 1
else
  return n*FactRec ( n - 1 )
```

Not immediately!

```
EuclidGCDRec(m, n):
if n = 0
   return m
else
   rem := m % n
   return EuclidGCDRec(n, rem)
```

Tail Recursion

- A function is called **tail-recursive** if each activation of the function will make **at most** one single recursive call, and will **return immediately after that call**.

Euclid (m, n):

if $n = 0$
  return m
else
  $rem := m \ \% \ n$
  return $Euclid(n, rem)$

Euclid(6, 4):
$4 = 0$?
$rem := 6 \ \% \ 4$
… … …
return 2

Euclid(4, 2):
$2 = 0$?
$rem := 4 \ \% \ 2$
… … …
return 2

Euclid(2, 0):
$0 = 0$?
return 2

Once reaching the base case,
can safely return result immediately!

Tail Recursion to Iteration

- Each function parameter is a variable.
- Convert the main body of the function into a loop:
  - Base cases: do computation and return results.
  - Recursive cases: do computation and update variables.

```
EuclidGCDRec(m, n):
if n = 0
   return m
else
   rem := m % n
   return EuclidGCDRec(n, rem)
```

```
EuclidGCDIter (m, n):
while true
   if n = 0
     return m
   else
     rem := m % n
     m := n
     n := rem
```

Iteration versus Recursion

- Recursion can be converted into iteration
  - Generic method: simulate a call stack
  - Special case: tail recursion
- Iteration can be converted into tail recursion
  - No one is always perfect
  - Iteration can be faster and more memory efficient
  - Recursion can be clearer, more concise and intuitive

Further reading

- [CLRS] Ch10 (10.1-10.3)
- [Morin] Ch1 (1.1, 1.2), Ch2 (2.1-2.4), Ch3 (3.1, 3.2)
- [Deng] Ch1 (1.4*), Ch4 (4.1-4.4)
- [Weiss] Ch3 (3.6)
- [CSAPP] Ch3 (3.7*)