Fundamentals

SFWRENG 2CO3: Data Structures and Algorithms

Jelle Hellings

Department of Computing and Software
McMaster University

Winter 2024

Collection types

A *collection type* is an *data type* used to manage a *collection of values*.

Typically an *abstract data type*: the implementation is hidden from the user.

Collection types are implemented via *data structures*.

Collection types: Bag

A *bag* B is a collection to which values can be added, but not removed:

- $\textsc{Add}(B, v)$ add value v to a bag B;
- $\textsc{Empty}(B)$ return `true` if bag B holds no values;
- $\textsc{Size}(B)$ returns the number of values in B.

In addition, one can *iterate* over the values currently in bag B.

Collection types: Bag

A *bag* B is a collection to which values can be added, but not removed:

$\textsc{Add}(B, v)$ add value v to a bag B;

$\textsc{Empty}(B)$ return `true` if bag B holds no values;

$\textsc{Size}(B)$ returns the number of values in B.

In addition, one can *iterate* over the values currently in bag B.

Remark

Note that Empty can be implemented via Size.

Not all data structures provide an efficient Size, however!

Collection types: Stack

Stack: collection to which values can be added to the top, removed from the top.

Collection types: Stack

Stack: collection to which values can be added to the top, removed from the top.

PUSH(S, v)	add value v to the top of stack S;
POP(S)	remove (and return) the value on top of stack S;
EMPTY(S)	return `true` if stack S holds no values;
SIZE(S)	returns the number of values in S.

Collection types: Stack

Stack: collection to which values can be added to the top, removed from the top.

PUSH(S, v)	add value v to the top of stack S;
POP(S)	remove (and return) the value on top of stack S;
EMPTY(S)	return `true` if stack S holds no values;
SIZE(S)	returns the number of values in S.

Stacks are sometimes referred to as *first-in-last-out* (FILO) queues.

Collection types: Stack

Stack: collection to which values can be added to the top, removed from the top.

PUSH(S, v)	add value v to the top of stack S;
POP(S)	remove (and return) the value on top of stack S;
EMPTY(S)	return `true` if stack S holds no values;
SIZE(S)	returns the number of values in S.

Stacks are sometimes referred to as *first-in-last-out* (FILO) queues.

(top)

Collection types: Stack

Stack: collection to which values can be added to the top, removed from the top.

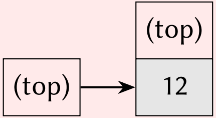
PUSH(S, v) add value v to the top of stack S;

POP(S) remove (and return) the value on top of stack S;

EMPTY(S) return `true` if stack S holds no values;

SIZE(S) returns the number of values in S.

Stacks are sometimes referred to as *first-in-last-out* (FILO) queues.



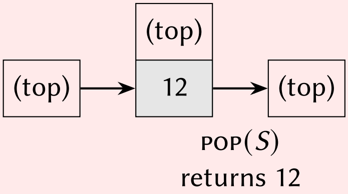
PUSH(S, 12)

Collection types: Stack

Stack: collection to which values can be added to the top, removed from the top.

PUSH(S, v)	add value v to the top of stack S;
POP(S)	remove (and return) the value on top of stack S;
EMPTY(S)	return `true` if stack S holds no values;
SIZE(S)	returns the number of values in S.

Stacks are sometimes referred to as *first-in-last-out* (FILO) queues.

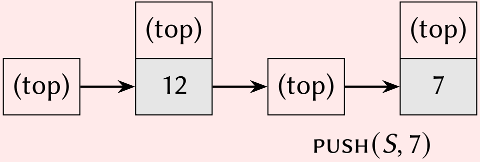


Collection types: Stack

Stack: collection to which values can be added to the top, removed from the top.

Push(S, v)	add value v to the top of stack S;
Pop(S)	remove (and return) the value on top of stack S;
Empty(S)	return `true` if stack S holds no values;
Size(S)	returns the number of values in S.

Stacks are sometimes referred to as *first-in-last-out* (FILO) queues.

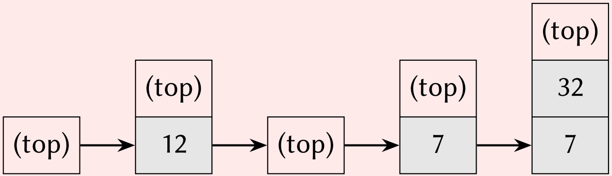


Collection types: Stack

Stack: collection to which values can be added to the top, removed from the top.

PUSH(S, v)	add value v to the top of stack S;
POP(S)	remove (and return) the value on top of stack S;
EMPTY(S)	return `true` if stack S holds no values;
SIZE(S)	returns the number of values in S.

Stacks are sometimes referred to as *first-in-last-out* (FILO) queues.



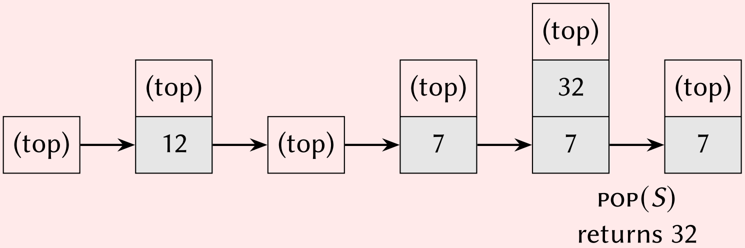
PUSH(S, 32)

Collection types: Stack

Stack: collection to which values can be added to the top, removed from the top.

PUSH(S, v)	add value v to the top of stack S;
POP(S)	remove (and return) the value on top of stack S;
EMPTY(S)	return `true` if stack S holds no values;
SIZE(S)	returns the number of values in S.

Stacks are sometimes referred to as *first-in-last-out* (FILO) queues.

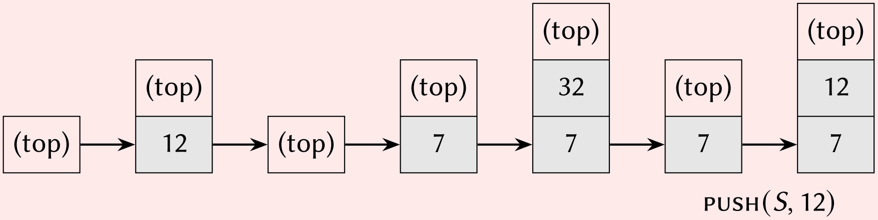


Collection types: Stack

Stack: collection to which values can be added to the top, removed from the top.

PUSH(S, v)	add value v to the top of stack S;
POP(S)	remove (and return) the value on top of stack S;
EMPTY(S)	return `true` if stack S holds no values;
SIZE(S)	returns the number of values in S.

Stacks are sometimes referred to as *first-in-last-out* (FILO) queues.

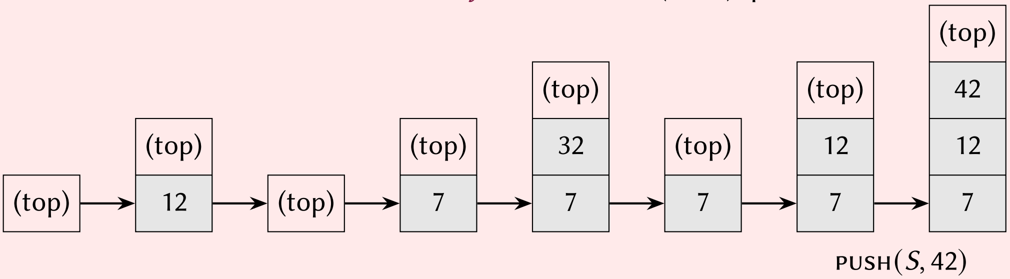


Collection types: Stack

Stack: collection to which values can be added to the top, removed from the top.

PUSH(S, v)	add value v to the top of stack S;
POP(S)	remove (and return) the value on top of stack S;
EMPTY(S)	return `true` if stack S holds no values;
SIZE(S)	returns the number of values in S.

Stacks are sometimes referred to as *first-in-last-out* (FILO) queues.

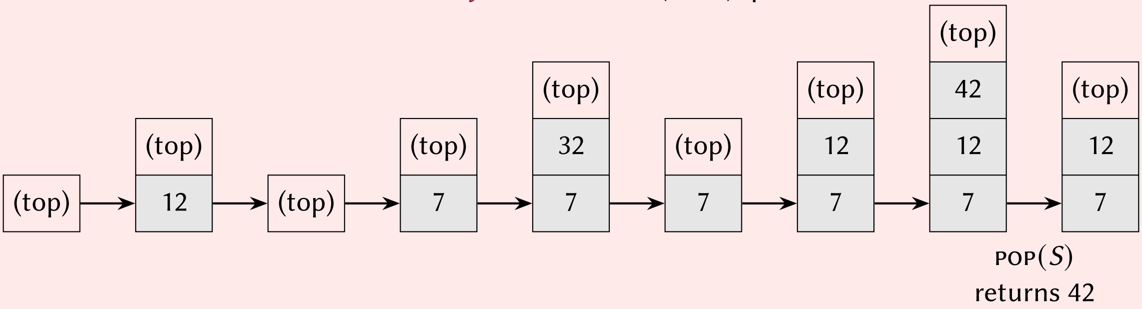


Collection types: Stack

Stack: collection to which values can be added to the top, removed from the top.

- Push(S, v) add value v to the top of stack S;
- Pop(S) remove (and return) the value on top of stack S;
- Empty(S) return `true` if stack S holds no values;
- Size(S) returns the number of values in S.

Stacks are sometimes referred to as *first-in-last-out* (FILO) queues.

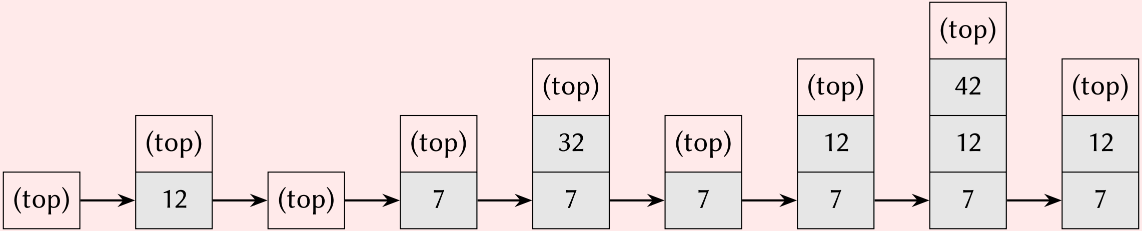


Collection types: Stack

Stack: collection to which values can be added to the top, removed from the top.

- PUSH(S, v) add value v to the top of stack S;
- POP(S) remove (and return) the value on top of stack S;
- EMPTY(S) return `true` if stack S holds no values;
- SIZE(S) returns the number of values in S.

Stacks are sometimes referred to as *first-in-last-out* (FILO) queues.



Stacks are used *everywhere*: e.g., function calls are implemented via stacks.

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

Example

- “1 2 +” is equivalent to $1 + 2$.
- “1 2 3 + -” is equivalent to $1 - (2 + 3)$.
- “1 2 + 3 -” is equivalent to $(1 + 2) - 3$.
- “1 2 3 + 4 5 · - /” is equivalent to $1/((2 + 3) - (4 \cdot 5))$.

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

Example

- "1 2 +" is equivalent to $1 + 2$.
- "1 2 3 + -" is equivalent to $1 - (2 + 3)$.
- "1 2 + 3 -" is equivalent to $(1 + 2) - 3$.
- "1 2 3 + 4 5 · - /" is equivalent to $1/((2 + 3) - (4 \cdot 5))$.

Expressions in postfix notation are very easy to evaluate using a stack.

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

"1 2 3 + 4 5 · - /" is equivalent to $1/((2+3)-(4\cdot 5))$.

Algorithm EVALUATEPN(e):

Input: e is in postfix notation.

```
S := an empty stack.
for each term t ∈ e do
    if t is a value then
        PUSH(S, t).
    else t is an operation ⊕
        v2 := POP(S); v1 := POP(S).
        PUSH(S, v1 ⊕ v2).
return POP(S).
```

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

"1 2 3 + 4 5 · - /" is equivalent to $1/((2+3)-(4\cdot 5))$.

Algorithm EvaluatePN(e):

Input: e is in postfix notation.

1: S := an empty stack.
2: **for each** term $t \in e$ **do**
3:     **if** t is a value **then**
4:         Push(S, t).
5:     **else** t is an operation $\oplus$
6:         v_2 := pop(S); v_1 := pop(S).
7:         Push(S, $v_1 \oplus v_2$).
8: **return** pop(S).

(top)

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

"1 2 3 + 4 5 · - /" is equivalent to $1/((2+3)-(4\cdot 5))$.

Algorithm EVALUATEPN(e):

Input: e is in postfix notation.

$S :=$ an empty stack.
for each term $t \in e$ **do**
  if t is a value **then**
    PUSH(S, t).
  else t is an operation $\oplus$
    $v_2 :=$ POP(S); $v_1 :=$ POP(S).
    PUSH(S, $v_1 \oplus v_2$).
return POP(S).

(top)
1

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

"1 2 3 + 4 5 · - /" is equivalent to $1/((2+3)-(4 \cdot 5))$.

Algorithm EVALUATEPN(e):

Input: e is in postfix notation.

$S :=$ an empty stack.
for each term $t \in e$ **do**
  if t is a value **then**
    PUSH(S, t).
  else t is an operation $\oplus$
    $v_2 :=$ POP(S); $v_1 :=$ POP(S).
    PUSH(S, $v_1 \oplus v_2$).
return POP(S).

(top)
2
1

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

"1 2 3 + 4 5 · - /" is equivalent to $1/((2+3)-(4\cdot 5))$.

Algorithm EVALUATEPN(e):

Input: e is in postfix notation.

```
S := an empty stack.
for each term t ∈ e do
    if t is a value then
        PUSH(S, t).
    else t is an operation ⊕
        v2 := POP(S); v1 := POP(S).
        PUSH(S, v1 ⊕ v2).
return POP(S).
```

(top)
3
2
1

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

$$\text{“}1\ 2\ 3 + 4\ 5 \cdot - /\text{” is equivalent to } 1/((2+3)-(4\cdot 5)).$$

Algorithm EVALUATEPN(e):

Input: e is in postfix notation.

$S :=$ an empty stack.
for each term $t \in e$ **do**
  if t is a value **then**
    PUSH(S, t).
  else t is an operation $\oplus$
    $v_2 := \text{POP}(S)$; $v_1 := \text{POP}(S)$.
    PUSH$(S, v_1 \oplus v_2)$.
return POP(S).

(top)	
5	$2 + 3$
1	

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

"1 2 3 + 4 5 · - /" is equivalent to $1/((2+3)-(4\cdot 5))$.

Algorithm EVALUATEPN(e):

Input: e is in postfix notation.

```
S := an empty stack.
for each term t ∈ e do
    if t is a value then
        PUSH(S, t).
    else t is an operation ⊕
        v2 := POP(S); v1 := POP(S).
        PUSH(S, v1 ⊕ v2).
return POP(S).
```

(top)
4
5
1

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

"1 2 3 + 4 5 · - /" is equivalent to $1/((2+3)-(4 \cdot 5))$.

Algorithm EvaluatePN(e):

Input: e is in postfix notation.

S := an empty stack.
for each term $t \in e$ **do**
    if t is a value **then**
        Push(S, t).
    else t is an operation $\oplus$
        v_2 := pop(S); v_1 := pop(S).
        Push(S, $v_1 \oplus v_2$).
return pop(S).

(top)
5
4
5
1

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

"1 2 3 + 4 5 · - /" is equivalent to $1/((2+3)-(4\cdot 5))$.

Algorithm EvaluatePN(e):

Input: e is in postfix notation.

```
S := an empty stack.
for each term t ∈ e do
    if t is a value then
        Push(S, t).
    else t is an operation ⊕
        v2 := pop(S); v1 := pop(S).
        Push(S, v1 ⊕ v2).
return pop(S).
```

(top)	
20	$4\cdot 5$
5	
1	

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

"1 2 3 + 4 5 · - /" is equivalent to $1/((2+3)-(4\cdot 5))$.

Algorithm EvaluatePN(e):

Input: e is in postfix notation.

```
S := an empty stack.
for each term t ∈ e do
    if t is a value then
        Push(S, t).
    else t is an operation ⊕
        v2 := pop(S); v1 := pop(S).
        Push(S, v1 ⊕ v2).
return pop(S).
```

(top)	
-15	$5-20$
1	

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

"1 2 3 + 4 5 · - /" is equivalent to $1/((2+3)-(4\cdot 5))$.

Algorithm EvaluatePN(e):

Input: e is in postfix notation.

S := an empty stack.
for each term $t \in e$ **do**
    if t is a value **then**
        Push(S, t).
    else t is an operation $\oplus$
        $v_2 :=$ pop(S); $v_1 :=$ pop(S).
        Push$(S, v_1 \oplus v_2)$.
return pop(S).

(top)	
$-.0\overline{6}$	$1/-15$

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

"1 2 3 + 4 5 · - /" is equivalent to $1/((2+3)-(4\cdot 5))$.

Algorithm EVALUATEPN(e):

Input: e is in postfix notation.

S := an empty stack.
for each term $t \in e$ **do**
  if t is a value **then**
    PUSH(S, t).
  else t is an operation $\oplus$
    v_2 := POP(S); v_1 := POP(S).
    PUSH(S, $v_1 \oplus v_2$).
return POP(S).

$\boxed{\text{(top)}}$

returns $-.0\overline{6}$

Using stacks to evaluate expressions in postfix notation

Postfix notation is a notation in which *operators* follow their *operands*.

"1 2 3 + 4 5 · - /" is equivalent to $1/((2+3)-(4\cdot 5))$.

Algorithm EVALUATEPN(e):

Input: e is in postfix notation.

```
S := an empty stack.
for each term t ∈ e do
    if t is a value then
        PUSH(S, t).
    else t is an operation ⊕
        v2 := POP(S); v1 := POP(S).
        PUSH(S, v1 ⊕ v2).
return POP(S).
```

Dijkstra's Two-Stack Algorithm (book) evaluates expressions in *infix* (normal) notation: this by building a postfix notation and simultaneously applying EVALUATEPN.

Collection types: Queue

Queue: collection to which values can be added to the top and removed from the bottom.

Collection types: Queue

Queue: collection to which values can be added to the top and removed from the bottom.

ENQUEUE(Q, v)	add value v to the top of queue Q;
DEQUEUE(Q)	remove (and return) the value at the bottom of queue Q;
EMPTY(Q)	return `true` if queue Q holds no values;
SIZE(Q)	returns the number of values in Q.

Collection types: Queue

Queue: collection to which values can be added to the top and removed from the bottom.

ENQUEUE(Q, v)	add value v to the top of queue Q;
DEQUEUE(Q)	remove (and return) the value at the bottom of queue Q;
EMPTY(Q)	return `true` if queue Q holds no values;
SIZE(Q)	returns the number of values in Q.

Queues are sometimes referred to as *first-in-first-out* (FIFO) queues.

Collection types: Queue

Queue: collection to which values can be added to the top and removed from the bottom.

ENQUEUE(Q, v)	add value v to the top of queue Q;
DEQUEUE(Q)	remove (and return) the value at the bottom of queue Q;
EMPTY(Q)	return `true` if queue Q holds no values;
SIZE(Q)	returns the number of values in Q.

Queues are sometimes referred to as *first-in-first-out* (FIFO) queues.

$\boxed{\text{(top)}}$

Collection types: Queue

Queue: collection to which values can be added to the top and removed from the bottom.

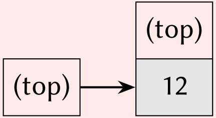
ENQUEUE(Q, v) add value v to the top of queue Q;

DEQUEUE(Q) remove (and return) the value at the bottom of queue Q;

EMPTY(Q) return `true` if queue Q holds no values;

SIZE(Q) returns the number of values in Q.

Queues are sometimes referred to as *first-in-first-out* (FIFO) queues.



ENQUEUE(S, 12)

Collection types: Queue

Queue: collection to which values can be added to the top and removed from the bottom.

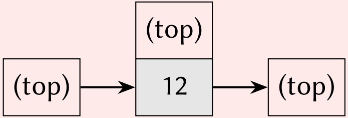
Enqueue(Q, v) add value v to the top of queue Q;

Dequeue(Q) remove (and return) the value at the bottom of queue Q;

Empty(Q) return `true` if queue Q holds no values;

Size(Q) returns the number of values in Q.

Queues are sometimes referred to as *first-in-first-out* (FIFO) queues.



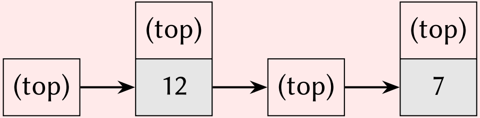
Dequeue(S)
returns 12

Collection types: Queue

Queue: collection to which values can be added to the top and removed from the bottom.

$\textsc{Enqueue}(Q, v)$	add value v to the top of queue Q;
$\textsc{Dequeue}(Q)$	remove (and return) the value at the bottom of queue Q;
$\textsc{Empty}(Q)$	return `true` if queue Q holds no values;
$\textsc{Size}(Q)$	returns the number of values in Q.

Queues are sometimes referred to as *first-in-first-out* (FIFO) queues.



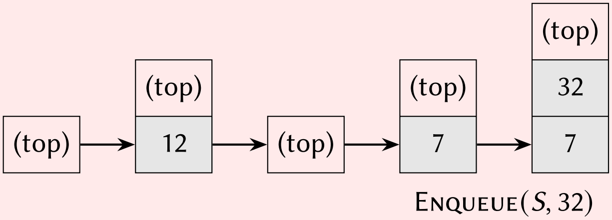
$\textsc{Enqueue}(S, 7)$

Collection types: Queue

Queue: collection to which values can be added to the top and removed from the bottom.

ENQUEUE(Q, v)	add value v to the top of queue Q;
DEQUEUE(Q)	remove (and return) the value at the bottom of queue Q;
EMPTY(Q)	return `true` if queue Q holds no values;
SIZE(Q)	returns the number of values in Q.

Queues are sometimes referred to as *first-in-first-out* (FIFO) queues.



Collection types: Queue

Queue: collection to which values can be added to the top and removed from the bottom.

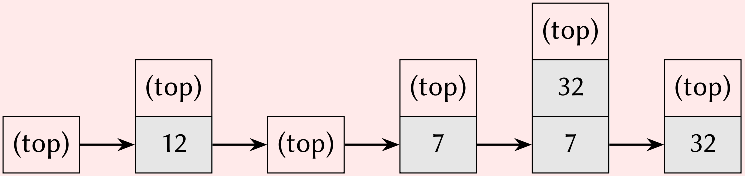
ENQUEUE(Q, v) add value v to the top of queue Q;

DEQUEUE(Q) remove (and return) the value at the bottom of queue Q;

EMPTY(Q) return `true` if queue Q holds no values;

SIZE(Q) returns the number of values in Q.

Queues are sometimes referred to as *first-in-first-out* (FIFO) queues.



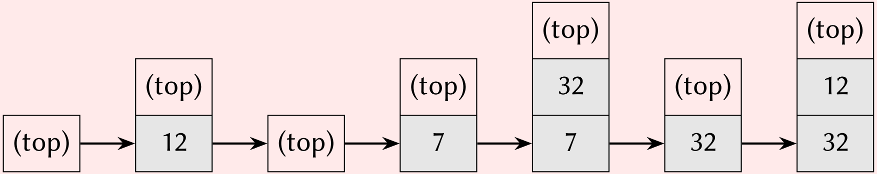
DEQUEUE(S)
returns 7

Collection types: Queue

Queue: collection to which values can be added to the top and removed from the bottom.

ENQUEUE(Q, v)	add value v to the top of queue Q;
DEQUEUE(Q)	remove (and return) the value at the bottom of queue Q;
EMPTY(Q)	return `true` if queue Q holds no values;
SIZE(Q)	returns the number of values in Q.

Queues are sometimes referred to as *first-in-first-out* (FIFO) queues.



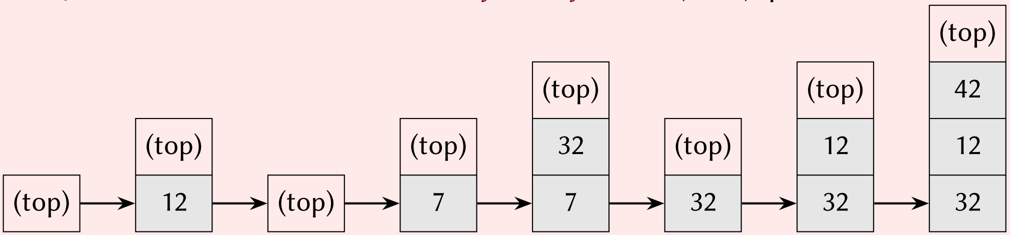
ENQUEUE(S, 12)

Collection types: Queue

Queue: collection to which values can be added to the top and removed from the bottom.

ENQUEUE(Q, v)	add value v to the top of queue Q;
DEQUEUE(Q)	remove (and return) the value at the bottom of queue Q;
EMPTY(Q)	return `true` if queue Q holds no values;
SIZE(Q)	returns the number of values in Q.

Queues are sometimes referred to as *first-in-first-out* (FIFO) queues.



ENQUEUE(S, 42)

Collection types: Queue

Queue: collection to which values can be added to the top and removed from the bottom.

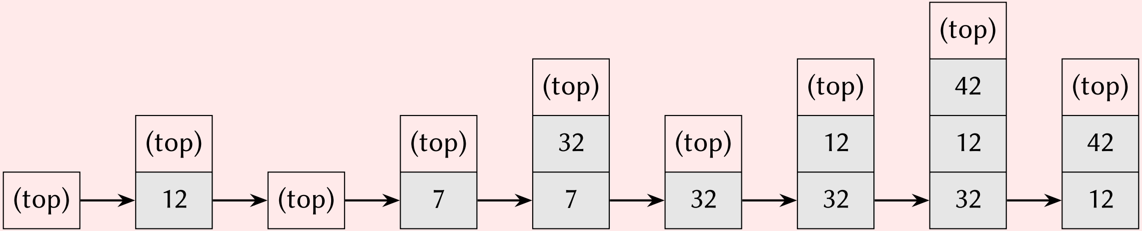
ENQUEUE(Q, v) add value v to the top of queue Q;

DEQUEUE(Q) remove (and return) the value at the bottom of queue Q;

EMPTY(Q) return `true` if queue Q holds no values;

SIZE(Q) returns the number of values in Q.

Queues are sometimes referred to as *first-in-first-out* (FIFO) queues.



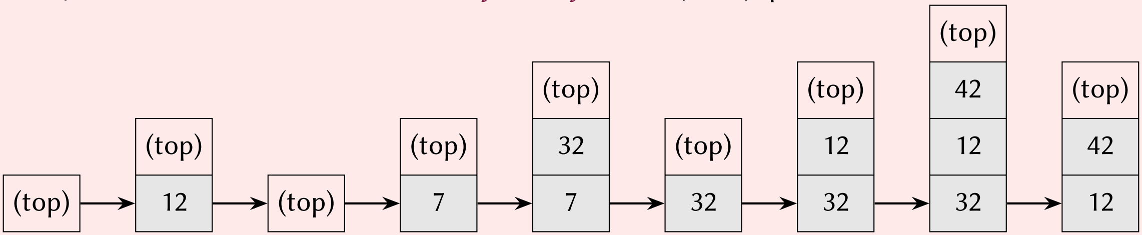
DEQUEUE(S)
returns 32

Collection types: Queue

Queue: collection to which values can be added to the top and removed from the bottom.

ENQUEUE(Q, v)	add value v to the top of queue Q;
DEQUEUE(Q)	remove (and return) the value at the bottom of queue Q;
EMPTY(Q)	return `true` if queue Q holds no values;
SIZE(Q)	returns the number of values in Q.

Queues are sometimes referred to as *first-in-first-out* (FIFO) queues.



Queues are used used *everywhere*: e.g., communication buffers
(for network packages, for tasks exchanged between producer-consumer threads, ...).

Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.

$\mathit{entries}[0 \ldots N)$	An array that can hold N values.
start	The position in *entries* of the *first* value in the buffer.
length	The current number of values in the buffer.

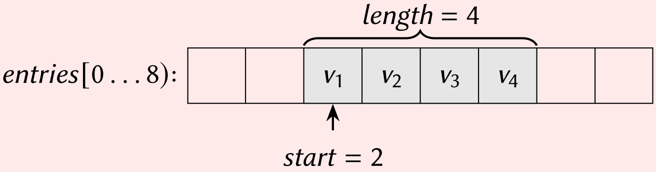
The values start at position *start* and wrap-around at the end of *entries*.

Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.

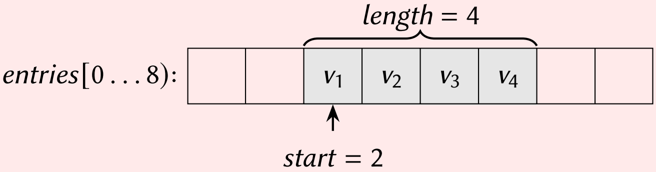
$\mathit{entries}[0 \dots N)$	An array that can hold N values.
start	The position in *entries* of the *first* value in the buffer.
length	The current number of values in the buffer.

The values start at position *start* and wrap-around at the end of *entries*.



Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.



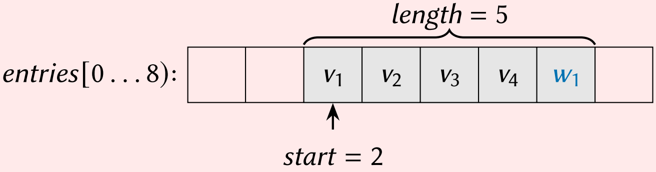
Algorithm PushBack(R, v):

Input: R is a non-full ring buffer ($R.length \neq N$).

if $R.start + R.length < N$ **then**
    $R.entries[R.start + R.length] := v$.
else Wrap-around the end of the list
    $R.entries[R.start + R.length - N] := v$.
$R.length := R.length + 1$.

Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.



Algorithm $\textsc{PushBack}(R, v)$:

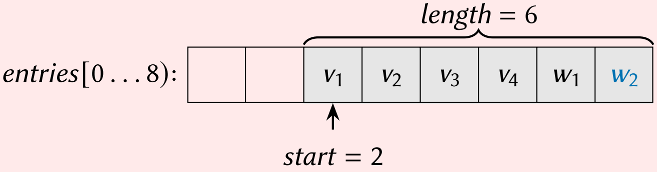
Input: R is a non-full ring buffer ($R.length \neq N$).

```
if R.start + R.length < N then
    R.entries[R.start + R.length] := v.
else Wrap-around the end of the list
    R.entries[R.start + R.length − N] := v.
R.length := R.length + 1.
```

$\textsc{PushBack}(R, w_1)$;
$\textsc{PushBack}(R, w_2)$;
$\textsc{PushBack}(R, w_3)$;
$\textsc{PushBack}(R, w_4)$.

Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.



Algorithm $\textsc{PushBack}(R, v)$:

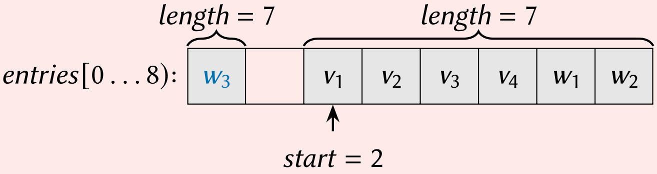
Input: R is a non-full ring buffer ($R.length \neq N$).

```
if R.start + R.length < N then
    R.entries[R.start + R.length] := v.
else  Wrap-around the end of the list
    R.entries[R.start + R.length − N] := v.
R.length := R.length + 1.
```

$\textsc{PushBack}(R, w_1)$;
$\textsc{PushBack}(R, w_2)$;
$\textsc{PushBack}(R, w_3)$;
$\textsc{PushBack}(R, w_4)$.

Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.



Algorithm $\textsc{PushBack}(R, v)$:

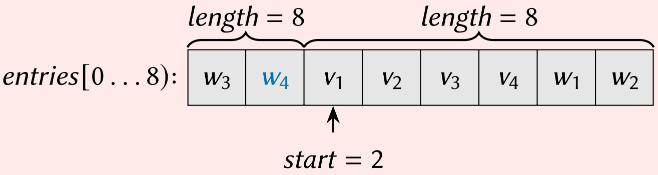
Input: R is a non-full ring buffer ($R.length \neq N$).

if $R.start + R.length < N$ **then**
  $R.entries[R.start + R.length] := v$.
else Wrap-around the end of the list
  $R.entries[R.start + R.length - N] := v$.
$R.length := R.length + 1$.

$\textsc{PushBack}(R, w_1)$;
$\textsc{PushBack}(R, w_2)$;
$\textsc{PushBack}(R, w_3)$;
$\textsc{PushBack}(R, w_4)$.

Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.



Algorithm $\textsc{PushBack}(R, v)$:

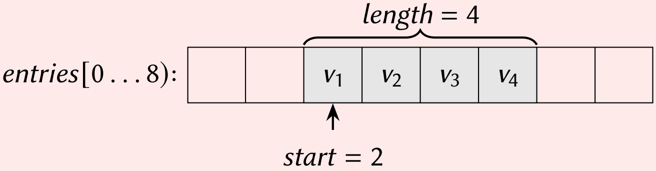
Input: R is a non-full ring buffer ($R.length \neq N$).

if $R.start + R.length < N$ **then**
  $R.entries[R.start + R.length] := v$.
else Wrap-around the end of the list
  $R.entries[R.start + R.length - N] := v$.
$R.length := R.length + 1$.

$\textsc{PushBack}(R, w_1)$;
$\textsc{PushBack}(R, w_2)$;
$\textsc{PushBack}(R, w_3)$;
$\textsc{PushBack}(R, w_4)$.

Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.



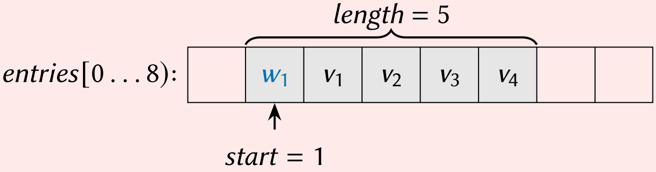
Algorithm $\textsc{PushFront}(R, v)$:

Input: R is a non-full ring buffer ($R.length \neq N$).

if $R.start > 0$ **then**
    $R.start := R.start - 1$.
else Wrap-around the begin of the list
    $R.start := N - 1$.
$R.entries[R.start], R.length := v, R.length + 1$.

Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.



Algorithm PushFront(R, v):

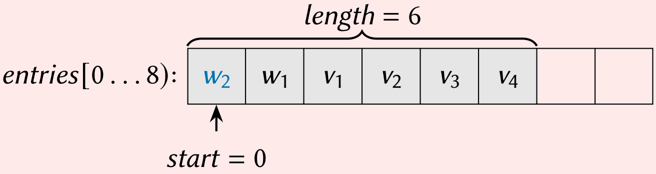
Input: R is a non-full ring buffer ($R.length \neq N$).

if $R.start > 0$ **then**
    $R.start := R.start - 1$.
else Wrap-around the begin of the list
    $R.start := N - 1$.
$R.entries[R.start], R.length := v, R.length + 1$.

PushFront(R, w_1);
PushFront(R, w_2);
PushFront(R, w_3);
PushFront(R, w_4).

Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.



Algorithm $\textsc{PushFront}(R, v)$:

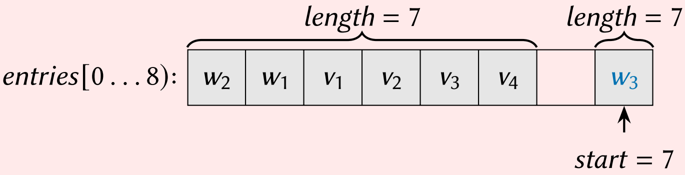
Input: R is a non-full ring buffer ($R.length \neq N$).

if $R.start > 0$ **then**
    $R.start := R.start - 1$.
else Wrap-around the begin of the list
    $R.start := N - 1$.
$R.entries[R.start], R.length := v, R.length + 1$.

$\textsc{PushFront}(R, w_1)$;
$\textsc{PushFront}(R, w_2)$;
$\textsc{PushFront}(R, w_3)$;
$\textsc{PushFront}(R, w_4)$.

Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.



Algorithm PushFront(R, v):

Input: R is a non-full ring buffer ($R.length \neq N$).

if $R.start > 0$ **then**
    $R.start := R.start - 1$.
else Wrap-around the begin of the list
    $R.start := N - 1$.
$R.entries[R.start], R.length := v, R.length + 1$.

PushFront(R, w_1);
PushFront(R, w_2);
PushFront(R, w_3);
PushFront(R, w_4).

Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.

$$\mathit{entries}[0\dots 8):\ \overbrace{\boxed{w_2}\boxed{w_1}\boxed{v_1}\boxed{v_2}\boxed{v_3}\boxed{v_4}}^{\mathit{length}\,=\,8}\overbrace{\underset{\substack{\uparrow\\ \mathit{start}\,=\,6}}{\boxed{w_4}}\boxed{w_3}}^{\mathit{length}\,=\,8}$$

Algorithm PushFront(R, v):

Input: R is a non-full ring buffer ($R.\mathit{length} \neq N$).

if $R.\mathit{start} > 0$ **then**

  $R.\mathit{start} := R.\mathit{start} - 1$.

else Wrap-around the begin of the list

  $R.\mathit{start} := N - 1$.

$R.\mathit{entries}[R.\mathit{start}], R.\mathit{length} := v, R.\mathit{length} + 1$.

PushFront(R, w_1);
PushFront(R, w_2);
PushFront(R, w_3);
PushFront(R, w_4).

Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.

$entries[0 \dots N)$ An array that can hold N values.

start The position in *entries* of the *first* value in the buffer.

length The current number of values in the buffer.

The values start at position *start* and wrap-around at the end of *entries*.

Removing elements from the front or the back is similar:

PopFront(R) undoes PushFront.

PopBack(R) undoes PushBack.

Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.

$\mathit{entries}[0 \ldots N)$	An array that can hold N values.
start	The position in *entries* of the *first* value in the buffer.
length	The current number of values in the buffer.

The values start at position *start* and wrap-around at the end of *entries*.

Removing elements from the front or the back is similar:

$\textsc{PopFront}(R)$	undoes $\textsc{PushFront}$.
$\textsc{PopBack}(R)$	undoes $\textsc{PushBack}$.

- Provides all stack operations in $\Theta(1)$.
- Provides all queue operations in $\Theta(1)$.
- Support *random access* efficiently.
- Can be used to implement a *double-ended queue*.

Data structures: Fixed-size ring buffers

Ring Buffer: a data structure that can hold up-to-N values.

$entries[0 \dots N)$ An array that can hold N values.

start The position in *entries* of the *first* value in the buffer.

length The current number of values in the buffer.

The values start at position *start* and wrap-around at the end of *entries*.

Removing elements from the front or the back is similar:

$\textsc{PopFront}(R)$ undoes $\textsc{PushFront}$.

$\textsc{PopBack}(R)$ undoes $\textsc{PushBack}$.

- Provides all stack operations in $\Theta(1)$.
- Provides all queue operations in $\Theta(1)$.
- Support *random access* efficiently.
- Can be used to implement a *double-ended queue*.
- *Drawback*: can hold at-most N values.

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.

first Pointer to the *first list node*.

last Pointer to the *last list node* (optional, for adding values to the end).

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.

first Pointer to the *first list node*.

last Pointer to the *last list node* (optional, for adding values to the end).

Each value in a linked list is held by a *list node*:

item The value held by the list node.

next A pointer to the next list node in the linked list, if any.

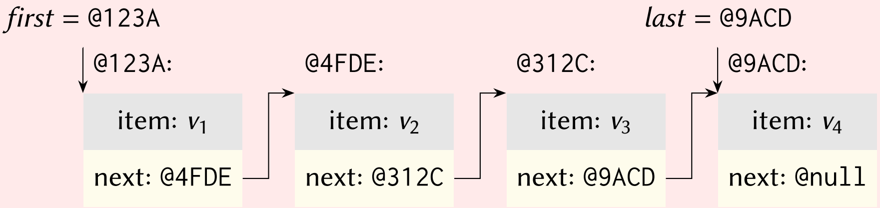
Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.

- *first* Pointer to the *first list node*.
- *last* Pointer to the *last list node* (optional, for adding values to the end).

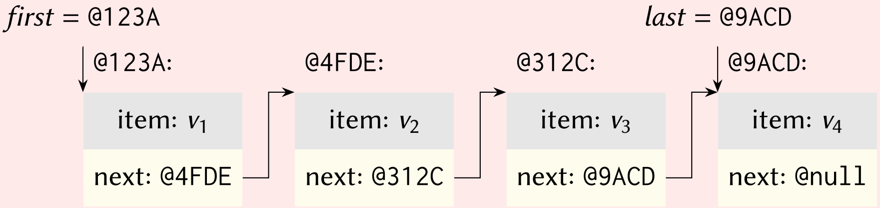
Each value in a linked list is held by a *list node*:

- *item* The value held by the list node.
- *next* A pointer to the next list node in the linked list, if any.



Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



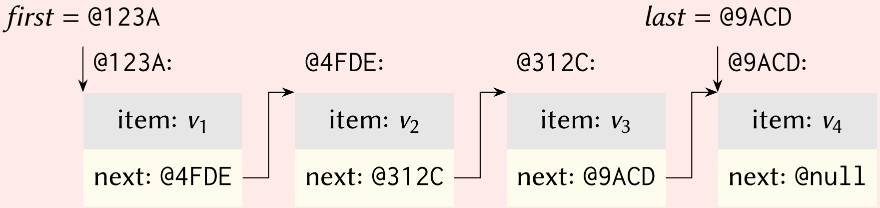
Algorithm PushFront(L, v):

Input: L is a linked list.

Create new list node n for value v.
$n.next := L.first$.
$L.first :=$ pointer to n.
if $L.last =$ `@null` **then** List L was empty
    $L.last :=$ pointer to n.

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



Algorithm PushFront(L, v):

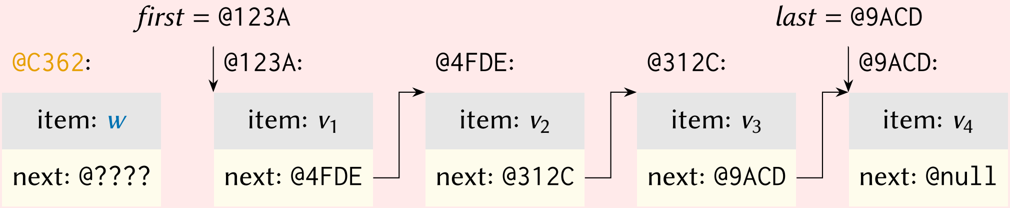
Input: L is a linked list.

1: Create new list node n for value v.
2: $n.next$:= $L.first$.
3: $L.first$:= pointer to n.
4: **if** $L.last$ = @null **then** List L was empty
5: $L.last$:= pointer to n.

PushFront(L, w).

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



Algorithm PushFront(L, v):

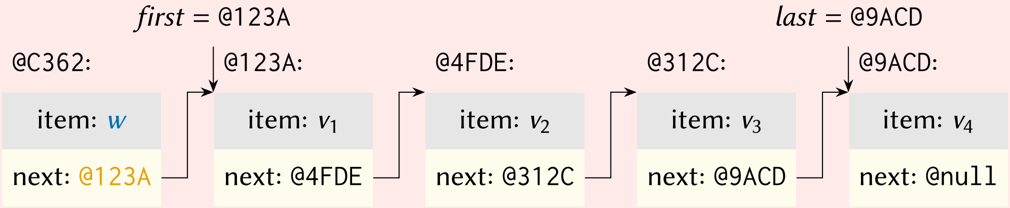
Input: L is a linked list.

Create new list node n for value v.
$n.next$:= $L.first$.
$L.first$:= pointer to n.
if $L.last$ = @null **then** List L was empty
    $L.last$:= pointer to n.

PushFront(L, w).

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



Algorithm PushFront(L, v):

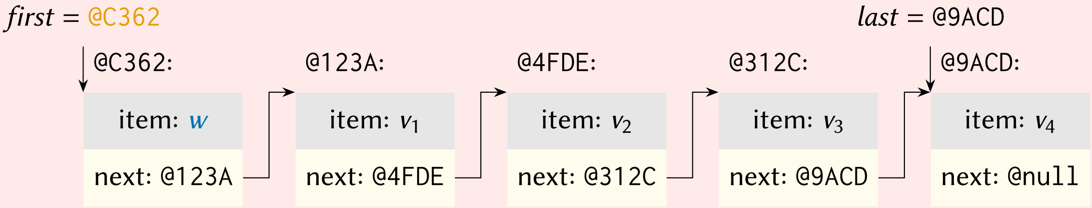
Input: L is a linked list.

Create new list node n for value v.
$n.next := L.first$.
$L.first :=$ pointer to n.
if $L.last =$ `@null` **then** List L was empty
    $L.last :=$ pointer to n.

PushFront(L, w).

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



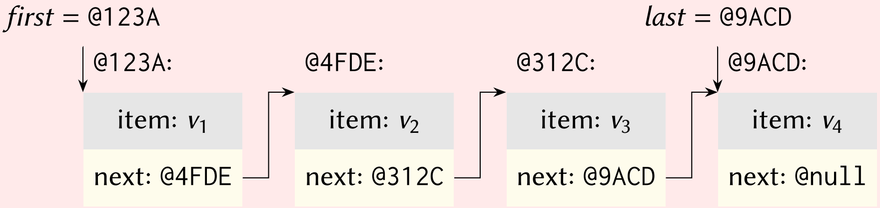
Algorithm PushFront(L, v):

Input: L is a linked list.

Create new list node n for value v.
$n.next$:= $L.first$.
$L.first$:= pointer to n.
if $L.last$ = `@null` **then** List L was empty
    $L.last$:= pointer to n.

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



Algorithm APPENDNODE(L, m, v):

Input: L is a linked list with node m.

Create new list node n for value v.

$n.next := m.next$.

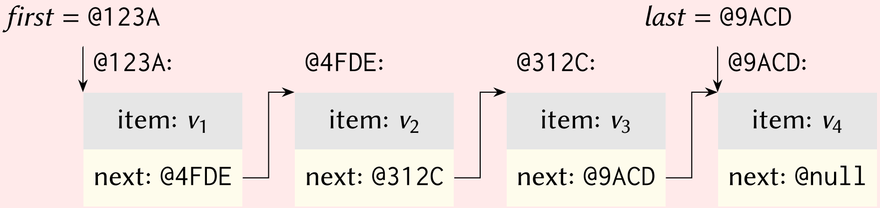
$m.next :=$ pointer to n.

if $L.last = m$ **then** m was the last node in L

    $L.last :=$ pointer to n.

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



Algorithm APPENDNODE(L, m, v):

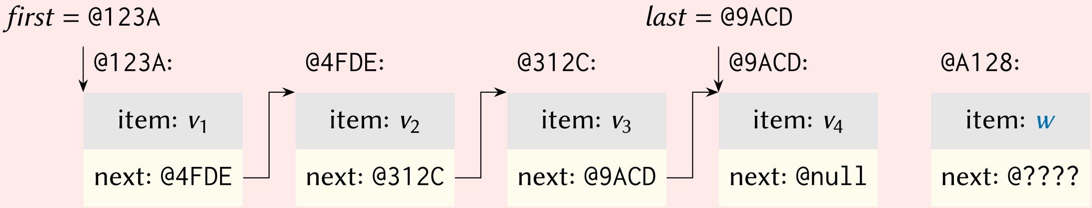
Input: L is a linked list with node m.

Create new list node n for value v.
$n.next$:= $m.next$.
$m.next$:= pointer to n.
if $L.last = m$ **then** *m was the last node in L*
    $L.last$:= pointer to n.

APPENDNODE(L, $L.last$, w).

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



Algorithm APPENDNODE(L, m, v):

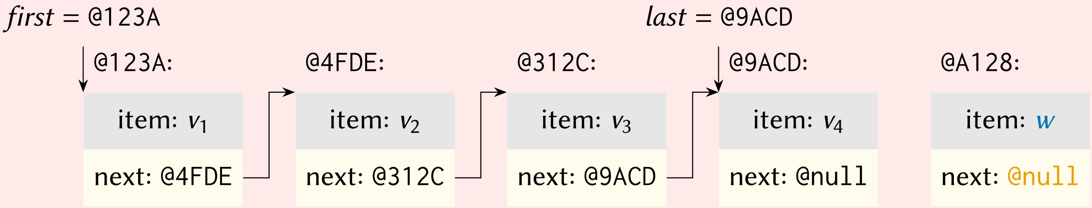
Input: L is a linked list with node m.

Create new list node n for value v.
$n.next := m.next$.
$m.next :=$ pointer to n.
if $L.last = m$ **then** m was the last node in L
    $L.last :=$ pointer to n.

APPENDNODE(L, $L.last$, w).

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



Algorithm AppendNode(L, m, v):

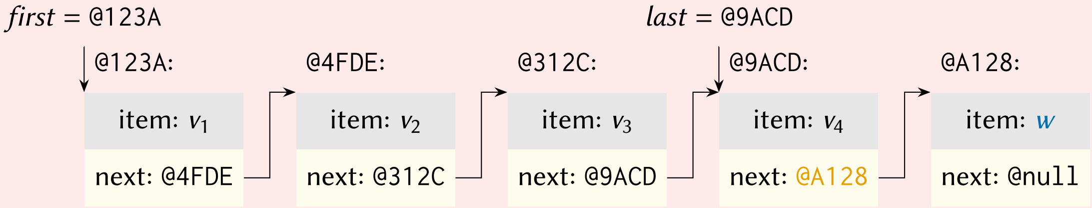
Input: L is a linked list with node m.

Create new list node n for value v.
$n.next := m.next$.
$m.next :=$ pointer to n.
if $L.last = m$ **then** m was the last node in L
    $L.last :=$ pointer to n.

AppendNode(L, $L.last$, w).

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



Algorithm AppendNode(L, m, v):

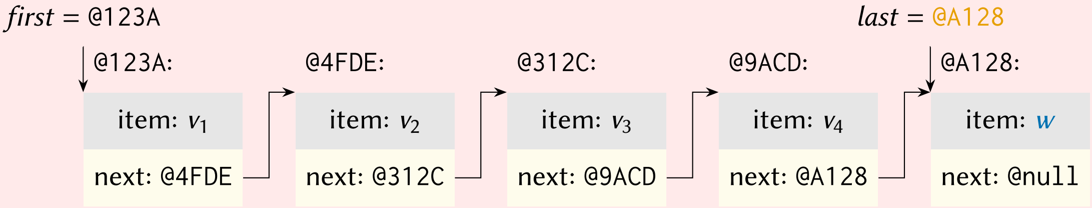
Input: L is a linked list with node m.

Create new list node n for value v.
$n.next := m.next$.
$m.next :=$ pointer to n.
if $L.last = m$ **then** m was the last node in L
    $L.last :=$ pointer to n.

AppendNode(L, $L.last$, w).

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



Algorithm APPENDNODE(L, m, v):

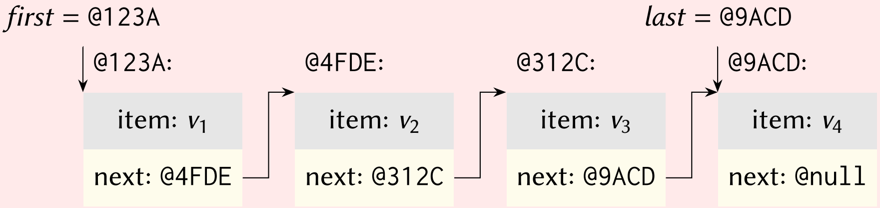
Input: L is a linked list with node m.

Create new list node n for value v.
$n.next := m.next$.
$m.next :=$ pointer to n.
if $L.last = m$ **then** m was the last node in L
    $L.last :=$ pointer to n.

APPENDNODE(L, $L.last$, w).

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



Algorithm AppendNode(L, m, v):

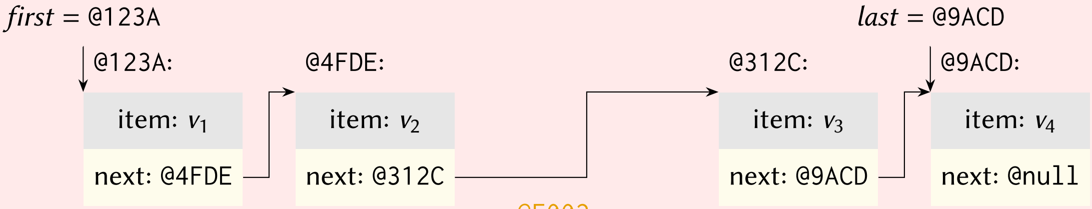
Input: L is a linked list with node m.

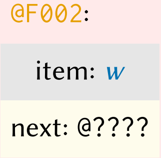
Create new list node n for value v.
$n.next := m.next$.
$m.next :=$ pointer to n.
if $L.last = m$ **then** m was the last node in L
    $L.last :=$ pointer to n.

AppendNode(L, @4FDE, w).

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.





Algorithm APPENDNODE(L, m, v):

Input: L is a linked list with node m.

Create new list node n for value v.
$n.next := m.next$.
$m.next :=$ pointer to n.
if $L.last = m$ **then** m was the last node in L
    $L.last :=$ pointer to n.

APPENDNODE(L, @4FDE, w).

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.

first = `@123A` *last* = `@9ACD`

`@123A:` item: v_1 — next: `@4FDE`

`@4FDE:` item: v_2 — next: `@312C`

`@312C:` item: v_3 — next: `@9ACD`

`@9ACD:` item: v_4 — next: `@null`

`@F002:` item: w — next: `@312C`

Algorithm AppendNode(L, m, v):

Input: L is a linked list with node m.

1: Create new list node n for value v.

2: $n.next := m.next$.

3: $m.next :=$ pointer to n.

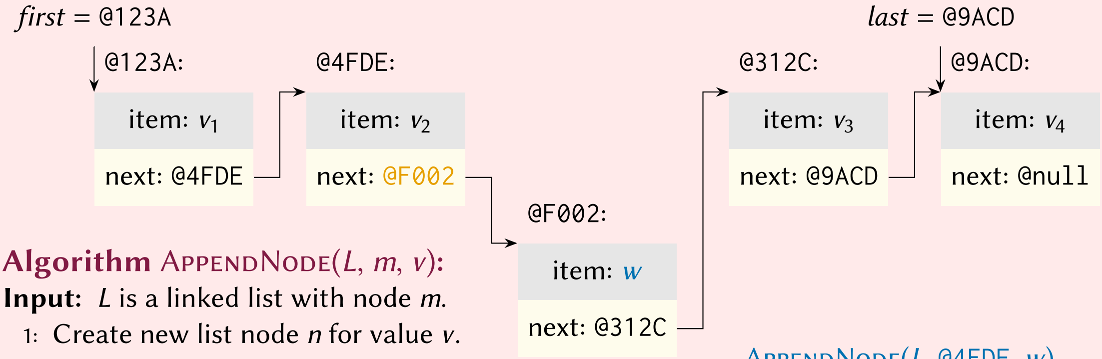
4: **if** $L.last = m$ **then** m was the last node in L

5: $L.last :=$ pointer to n.

AppendNode(L, `@4FDE`, w).

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



Algorithm AppendNode(L, m, v):

Input: L is a linked list with node m.

Create new list node n for value v.
$n.next := m.next$.
$m.next :=$ pointer to n.
if $L.last = m$ **then** m was the last node in L
    $L.last :=$ pointer to n.

AppendNode(L, @4FDE, w).

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.

- *first* Pointer to the *first list node*.
- *last* Pointer to the *last list node* (optional, for adding values to the end).

Each value in a linked list is held by a *list node*:

- *item* The value held by the list node.
- *next* A pointer to the next list node in the linked list, if any.

Removing elements from the front or after a given node is similar:

- PopFront(L) undoes PushFront.
- RemoveNode(L, w) undoes AppendNode (removes a node after node w).

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.

- *first* Pointer to the *first list node*.
- *last* Pointer to the *last list node* (optional, for adding values to the end).

Each value in a linked list is held by a *list node*:

- *item* The value held by the list node.
- *next* A pointer to the next list node in the linked list, if any.

Removing elements from the front or after a given node is similar:

- PopFront(L) undoes PushFront.
- RemoveNode(L, w) undoes AppendNode (removes a node after node w).

Make sure to free the memory associated with nodes.

In C++: use either `std::unique_ptr` or `std::shared_ptr` to free nodes *for* you.

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.

- *first* Pointer to the *first list node*.
- *last* Pointer to the *last list node* (optional, for adding values to the end).

Each value in a linked list is held by a *list node*:

- *item* The value held by the list node.
- *next* A pointer to the next list node in the linked list, if any.

Removing elements from the front or after a given node is similar:

- PopFront(L) undoes PushFront.
- RemoveNode(L, w) undoes AppendNode (removes a node after node w).

- Provides stack modifications in $\Theta(1)$ (add and remove to front, *last* unnecessary).
- Provides queue modifications in $\Theta(1)$ (add to *last*, remove from front).
- If an $\Theta(1)$ Size(L) is needed, then maintain an explicit counter.

Singly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.

first	Pointer to the *first list node*.
last	Pointer to the *last list node* (optional, for adding values to the end).

Each value in a linked list is held by a *list node*:

item	The value held by the list node.
next	A pointer to the next list node in the linked list, if any.

Removing elements from the front or after a given node is similar:

$\textsc{PopFront}(L)$	undoes $\textsc{PushFront}$.
$\textsc{RemoveNode}(L, w)$	undoes $\textsc{AppendNode}$ (removes a node after node w).

- Provides stack modifications in $\Theta(1)$ (add and remove to front, *last* unnecessary).
- Provides queue modifications in $\Theta(1)$ (add to *last*, remove from front).
- If an $\Theta(1)$ $\textsc{Size}(L)$ is needed, then maintain an explicit counter.
- *Drawback*: Low performance due to *pointer-chasing*, memory overhead of list nodes.

Doubly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.

Doubly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.

Each value in a *doubly* linked list is held by a *list node*:

- *item* The value held by the list node.
- *next* A pointer to the next list node in the linked list, if any.
- *prev* A pointer to the previous list node in the linked list, if any.

Doubly linked lists

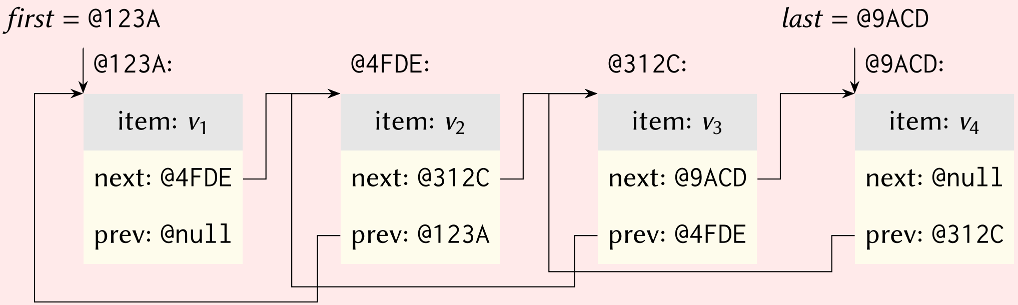
Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.

Each value in a *doubly* linked list is held by a *list node*:

item The value held by the list node.

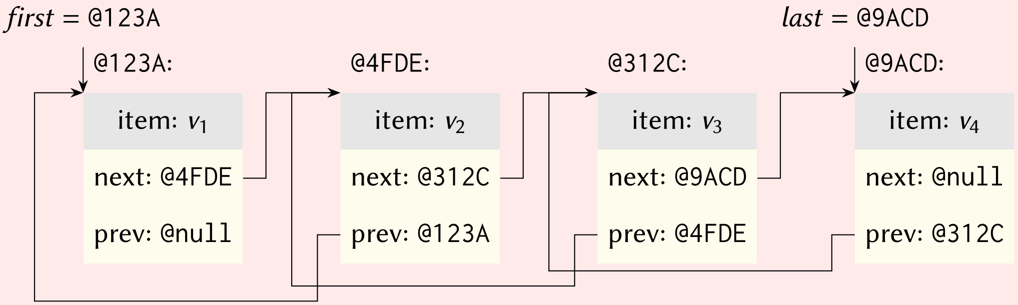
next A pointer to the next list node in the linked list, if any.

prev A pointer to the previous list node in the linked list, if any.



Doubly linked lists

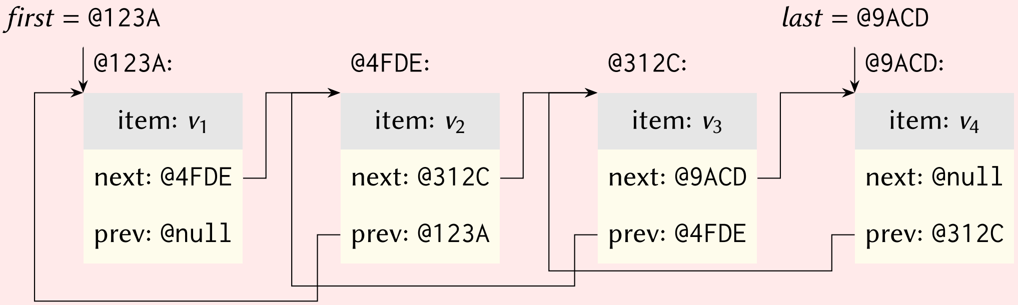
Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



- Doubly-linked lists provide flexible *iteration* and *modifications* of the list: you can easily remove a given doubly linked list node n or visit the node preceding n.

Doubly linked lists

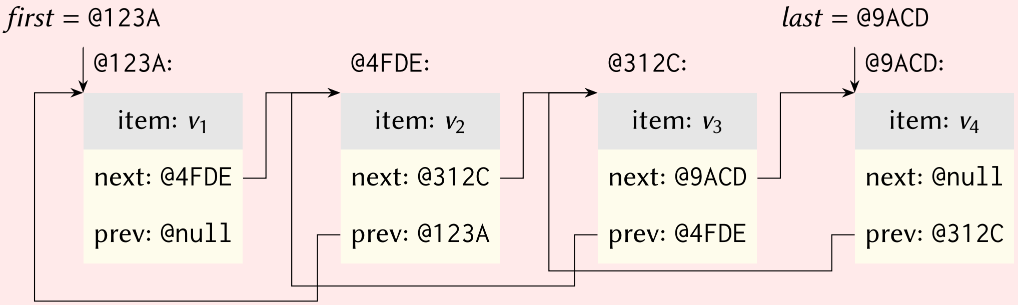
Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



- Doubly-linked lists provide flexible *iteration* and *modifications* of the list: you can easily remove a given doubly linked list node n or visit the node preceding n.
- Doubly-linked lists can be used to implement a *double-ended queue*.

Doubly linked lists

Linked Lists: a data structure that can hold a sequence of values, each stored in a *list node*.



- Doubly-linked lists provide flexible *iteration* and *modifications* of the list: you can easily remove a given doubly linked list node n or visit the node preceding n.
- Doubly-linked lists can be used to implement a *double-ended queue*.
- *Drawback*: Low performance due to *pointer-chasing*, memory overhead of list nodes.

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.

reserved Current internal reserved size for the array of values.

entries An array that can hold up-to-*reserved* values.

length The current number of values in the array, $length \leq reserved$.

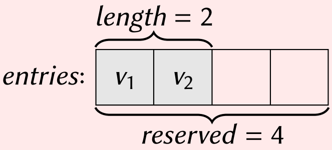
Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.

reserved Current internal reserved size for the array of values.

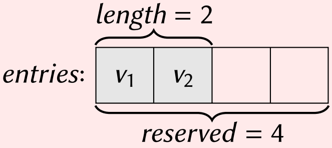
entries An array that can hold up-to-*reserved* values.

length The current number of values in the array, *length* $\leq$ *reserved*.



Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.



Algorithm $\textsc{PushBack}(D, v)$**:**

Input: D is a dynamic array.

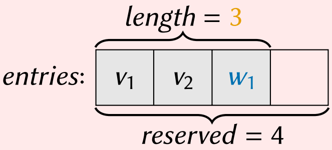
```
if D.reserved = D.length then
    InternalResize(D).
D.entries[D.length] := v.
D.length := D.length + 1.
```

Algorithm $\textsc{InternalResize}(D)$**:**

```
n := max(2 · D.reserved, 1).
Create new array a that can hold n values.
for pos := 0 to D.length do   copy D.entries to a
    a[pos] := D.entries[pos].
Free the memory for array D.entries.
D.reserved, D.entries := n, a.
```

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.



Algorithm PushBack(D, v):

Input: D is a dynamic array.

if $D.reserved = D.length$ **then**
  InternalResize(D).
$D.entries[D.length] := v$.
$D.length := D.length + 1$.

Algorithm InternalResize(D):

$n := \max(2 \cdot D.reserved, 1)$.
Create new array a that can hold n values.
for $pos := 0$ **to** $D.length$ **do** copy $D.entries$ to a
  $a[pos] := D.entries[pos]$.
Free the memory for array $D.entries$.
$D.reserved, D.entries := n, a$.

PushBack(D, w_1); PushBack(D, w_2); PushBack(D, w_3); PushBack(D, w_4).

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.

$$\textit{entries}: \underbrace{\overbrace{\boxed{v_1}\boxed{v_2}\boxed{w_1}\boxed{w_2}}^{\textit{length} = 4}}_{\textit{reserved} = 4}$$

Algorithm $\textsc{PushBack}(D, v)$**:**

Input: D is a dynamic array.

if $D.\textit{reserved} = D.\textit{length}$ **then**
    $\textsc{InternalResize}(D)$.
$D.\textit{entries}[D.\textit{length}] := v$.
$D.\textit{length} := D.\textit{length} + 1$.

Algorithm $\textsc{InternalResize}(D)$**:**

$n := \max(2 \cdot D.\textit{reserved}, 1)$.
Create new array a that can hold n values.
for $\textit{pos} := 0$ **to** $D.\textit{length}$ **do** copy $D.\textit{entries}$ to a
    $a[\textit{pos}] := D.\textit{entries}[\textit{pos}]$.
Free the memory for array $D.\textit{entries}$.
$D.\textit{reserved}, D.\textit{entries} := n, a$.

$\textsc{PushBack}(D, w_1)$; $\textsc{PushBack}(D, w_2)$; $\textsc{PushBack}(D, w_3)$; $\textsc{PushBack}(D, w_4)$.

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.

$$\textit{entries}: \overbrace{\underbrace{\boxed{v_1 \mid v_2 \mid w_1 \mid w_2}}_{\textit{reserved}\,=\,4}}^{\textit{length}\,=\,4}$$

Algorithm PushBack(D, v):

Input: D is a dynamic array.

if $D.reserved = D.length$ **then**
    InternalResize(D).
$D.entries[D.length] := v$.
$D.length := D.length + 1$.

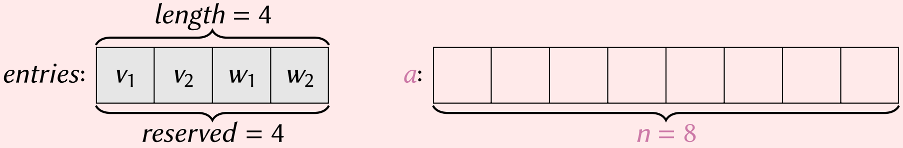
Algorithm InternalResize(D):

$n := \max(2 \cdot D.reserved, 1)$.
Create new array a that can hold n values.
for $pos := 0$ **to** $D.length$ **do** copy $D.entries$ to a
    $a[pos] := D.entries[pos]$.
Free the memory for array $D.entries$.
$D.reserved, D.entries := n, a$.

PushBack(D, w_1); PushBack(D, w_2); PushBack(D, w_3); PushBack(D, w_4).

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.



Algorithm $\textsc{PushBack}(D, v)$:

Input: D is a dynamic array.

if $D.reserved = D.length$ **then**
  $\textsc{InternalResize}(D)$.
$D.entries[D.length] := v$.
$D.length := D.length + 1$.

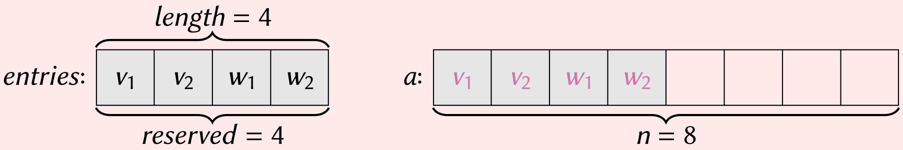
Algorithm $\textsc{InternalResize}(D)$:

$n := \max(2 \cdot D.reserved, 1)$.
Create new array a that can hold n values.
for $pos := 0$ **to** $D.length$ **do** copy $D.entries$ to a
  $a[pos] := D.entries[pos]$.
Free the memory for array $D.entries$.
$D.reserved, D.entries := n, a$.

$\textsc{PushBack}(D, w_1)$; $\textsc{PushBack}(D, w_2)$; $\textsc{PushBack}(D, w_3)$; $\textsc{PushBack}(D, w_4)$.

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.



Algorithm PushBack(D, v):

Input: D is a dynamic array.

```
if D.reserved = D.length then
    InternalResize(D).
D.entries[D.length] := v.
D.length := D.length + 1.
```

Algorithm InternalResize(D):

```
n := max(2 · D.reserved, 1).
Create new array a that can hold n values.
for pos := 0 to D.length do copy D.entries to a
    a[pos] := D.entries[pos].
Free the memory for array D.entries.
D.reserved, D.entries := n, a.
```

PushBack(D, w_1); PushBack(D, w_2); PushBack(D, w_3); PushBack(D, w_4).

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.

?

a: $\underbrace{\begin{array}{|c|c|c|c|c|c|c|c|}\hline v_1 & v_2 & w_1 & w_2 & & & & \\ \hline\end{array}}_{n = 8}$

Algorithm PushBack(D, v):

Input: D is a dynamic array.

```
if D.reserved = D.length then
    InternalResize(D).
D.entries[D.length] := v.
D.length := D.length + 1.
```

Algorithm InternalResize(D):

```
n := max(2 · D.reserved, 1).
Create new array a that can hold n values.
for pos := 0 to D.length do    copy D.entries to a
    a[pos] := D.entries[pos].
Free the memory for array D.entries.
D.reserved, D.entries := n, a.
```

PushBack(D, w_1); PushBack(D, w_2); PushBack(D, w_3); PushBack(D, w_4).

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.

$$\textit{entries}: \underbrace{\overbrace{\boxed{v_1}\,\boxed{v_2}\,\boxed{w_1}\,\boxed{w_2}}^{\textit{length} = 4}\,\boxed{}\,\boxed{}\,\boxed{}\,\boxed{}}_{\textit{reserved} = 8}$$

Algorithm PushBack(D, v):

Input: D is a dynamic array.

```
if D.reserved = D.length then
    InternalResize(D).
D.entries[D.length] := v.
D.length := D.length + 1.
```

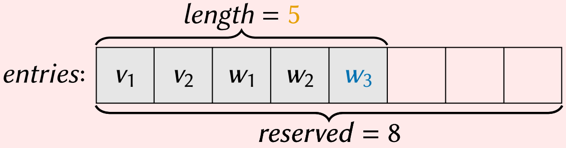
Algorithm InternalResize(D):

```
n := max(2 · D.reserved, 1).
Create new array a that can hold n values.
for pos := 0 to D.length do          copy D.entries to a
    a[pos] := D.entries[pos].
Free the memory for array D.entries.
D.reserved, D.entries := n, a.
```

PushBack(D, w_1); PushBack(D, w_2); PushBack(D, w_3); PushBack(D, w_4).

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.



Algorithm PushBack(D, v):

Input: D is a dynamic array.

if $D.reserved = D.length$ **then**
  InternalResize(D).
$D.entries[D.length] := v$.
$D.length := D.length + 1$.

Algorithm InternalResize(D):

$n := \max(2 \cdot D.reserved, 1)$.
Create new array a that can hold n values.
for $pos := 0$ **to** $D.length$ **do** copy $D.entries$ to a
  $a[pos] := D.entries[pos]$.
Free the memory for array $D.entries$.
$D.reserved, D.entries := n, a$.

PushBack(D, w_1); PushBack(D, w_2); PushBack(D, w_3); PushBack(D, w_4).

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.

$$\textit{entries}: \underbrace{\overbrace{\boxed{v_1}\boxed{v_2}\boxed{w_1}\boxed{w_2}\boxed{w_3}\boxed{w_4}}^{\textit{length} = 6}\boxed{}\boxed{}}_{\textit{reserved} = 8}$$

Algorithm $\textsc{PushBack}(D, v)$**:**

Input: D is a dynamic array.

```
if D.reserved = D.length then
    InternalResize(D).
D.entries[D.length] := v.
D.length := D.length + 1.
```

Algorithm $\textsc{InternalResize}(D)$**:**

```
n := max(2 · D.reserved, 1).
Create new array a that can hold n values.
for pos := 0 to D.length do    copy D.entries to a
    a[pos] := D.entries[pos].
Free the memory for array D.entries.
D.reserved, D.entries := n, a.
```

$\textsc{PushBack}(D, w_1)$; $\textsc{PushBack}(D, w_2)$; $\textsc{PushBack}(D, w_3)$; $\textsc{PushBack}(D, w_4)$.

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.

Removing elements from the the back is similar:
PopBack(D) undoes PushBack (typically without shrinking).

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.

Removing elements from the the back is similar:
PopBack(D) undoes PushBack (typically without shrinking).

Arbitrary inserts at position i, $i \neq D.length$, is costly:
Requires one to copy over by one position all values at-or-after position i.

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.

Removing elements from the the back is similar:
$\textsc{PopBack}(D)$ undoes $\textsc{PushBack}$ (typically without shrinking).

Arbitrary inserts at position i, $i \neq D.\mathit{length}$, is costly:
Requires one to copy over by one position all values at-or-after position i.

What about the complexity?

$\textsc{PushBack}(D, v)$ is either $\Theta(1)$ or $\Theta(D.\mathit{reserved})$ (if $\textsc{InternalResize}$ is called).

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.

Removing elements from the the back is similar:
PopBack(D) undoes PushBack (typically without shrinking).

Arbitrary inserts at position i, $i \neq D.length$, is costly:
Requires one to copy over by one position all values at-or-after position i.

What about the complexity?

PushBack(D, v) is either $\Theta(1)$ or $\Theta(D.reserved)$ (if InternalResize is called).

Can we provide a better analysis?

Intermezzo: Complexity of PushBack

Intermezzo: Complexity of PushBack

Definition (Amortized complexity)

Complexity of a sequence of operations averaged out over all operations performed.

Intermezzo: Complexity of PushBack

Definition (Amortized complexity)

Complexity of a sequence of operations averaged out over all operations performed.

Amortized complexity is not average case complexity!

Average case complexity looks at the average cost of a *single* operation.

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$, and a sequence of M, $M > 0$, PushBack operations.

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.
- PushBack(D, v) only calls InternalResize(D) when $D.length \in \{1, 2, 4, 8, 16, \dots\}$.

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.
- PushBack(D, v) only calls InternalResize(D) when $D.length \in \{1, 2, 4, 8, 16, \dots\}$.
- A call to InternalResize(D) with $D.length = i$ costs $c \cdot i + d$ with c, d constants.

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.
- PushBack(D, v) only calls InternalResize(D) when $D.length \in \{1, 2, 4, 8, 16, \dots\}$.
- A call to InternalResize(D) with $D.length = i$ costs $c \cdot i + d$ with c, d constants.
- Total costs of all InternalResize calls:

$$\sum_{k=0}^{j} c \cdot 2^k + d$$

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.
- PushBack(D, v) only calls InternalResize(D) when $D.length \in \{1, 2, 4, 8, 16, \ldots\}$.
- A call to InternalResize(D) with $D.length = i$ costs $c \cdot i + d$ with c, d constants.
- Total costs of all InternalResize calls:

$$\sum_{k=0}^{j} c \cdot 2^k + d = \left(c \sum_{k=0}^{j} 2^k\right) + \left(\sum_{k=0}^{j} d\right)$$

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.
- PushBack(D, v) only calls InternalResize(D) when $D.length \in \{1, 2, 4, 8, 16, \dots\}$.
- A call to InternalResize(D) with $D.length = i$ costs $c \cdot i + d$ with c, d constants.
- Total costs of all InternalResize calls:

$$\sum_{k=0}^{j} c \cdot 2^k + d = \left(c \sum_{k=0}^{j} 2^k\right) + \left(\sum_{k=0}^{j} d\right)$$
$$= c\left(2^{j+1} - 1\right) + d(j+1)$$

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.
- PushBack(D, v) only calls InternalResize(D) when $D.length \in \{1, 2, 4, 8, 16, \ldots\}$.
- A call to InternalResize(D) with $D.length = i$ costs $c \cdot i + d$ with c, d constants.
- Total costs of all InternalResize calls:

$$\begin{aligned}\sum_{k=0}^{j} c \cdot 2^k + d &= \left(c \sum_{k=0}^{j} 2^k\right) + \left(\sum_{k=0}^{j} d\right) \\ &= c\left(2^{j+1} - 1\right) + d(j+1)\end{aligned}$$

After M PushBacks on D: $D.length = M$ with $2^j < M \leq D.reserved = 2^{j+1}$.

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.
- PushBack(D, v) only calls InternalResize(D) when $D.length \in \{1, 2, 4, 8, 16, \dots\}$.
- A call to InternalResize(D) with $D.length = i$ costs $c \cdot i + d$ with c, d constants.
- Total costs of all InternalResize calls:

$$\sum_{k=0}^{j} c \cdot 2^k + d = \left(c \sum_{k=0}^{j} 2^k\right) + \left(\sum_{k=0}^{j} d\right)$$
$$= c\left(2^{j+1} - 1\right) + d(j + 1)$$

After M PushBacks on D: $D.length = M$ with $2^j < M \leq D.reserved = 2^{j+1}$.
Hence, we must have $j = \lfloor \log_2(M) \rfloor < \log_2(M)$.

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.
- PushBack(D, v) only calls InternalResize(D) when $D.length \in \{1, 2, 4, 8, 16, \dots\}$.
- A call to InternalResize(D) with $D.length = i$ costs $c \cdot i + d$ with c, d constants.
- Total costs of all InternalResize calls:

$$\begin{aligned}\sum_{k=0}^{j} c \cdot 2^k + d &= \left(c \sum_{k=0}^{j} 2^k\right) + \left(\sum_{k=0}^{j} d\right)\\ &= c\left(2^{j+1} - 1\right) + d(j+1)\\ &\leq c\left(2^{\log_2(M)+1} - 1\right) + d(\log_2(M) + 1)\end{aligned}$$

After M PushBacks on D: $D.length = M$ with $2^j < M \leq D.reserved = 2^{j+1}$.
Hence, we must have $j = \lfloor \log_2(M) \rfloor < \log_2(M)$.

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.
- PushBack(D, v) only calls InternalResize(D) when $D.length \in \{1, 2, 4, 8, 16, \dots\}$.
- A call to InternalResize(D) with $D.length = i$ costs $c \cdot i + d$ with c, d constants.
- Total costs of all InternalResize calls: $c(2M - 1) + d(\log_2(M) + 1)$.

$$\begin{aligned}\sum_{k=0}^{j} c \cdot 2^k + d &= \left(c \sum_{k=0}^{j} 2^k\right) + \left(\sum_{k=0}^{j} d\right) \\ &= c\left(2^{j+1} - 1\right) + d(j+1) \\ &\leq c\left(2^{\log_2(M)+1} - 1\right) + d(\log_2(M) + 1) = c(2M - 1) + d(\log_2(M) + 1).\end{aligned}$$

After M PushBacks on D: $D.length = M$ with $2^j < M \leq D.reserved = 2^{j+1}$.
Hence, we must have $j = \lfloor \log_2(M) \rfloor < \log_2(M)$.

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.
- PushBack(D, v) only calls InternalResize(D) when $D.length \in \{1, 2, 4, 8, 16, \ldots\}$.
- A call to InternalResize(D) with $D.length = i$ costs $c \cdot i + d$ with c, d constants.
- Total costs of all InternalResize calls: $c(2M - 1) + d(\log_2(M) + 1)$.

- *Amortized complexity* of M PushBacks:

$$\frac{bM + c(2M - 1) + d(\log_2(M) + 1)}{M}$$

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.
- PushBack(D, v) only calls InternalResize(D) when $D.length \in \{1, 2, 4, 8, 16, \dots\}$.
- A call to InternalResize(D) with $D.length = i$ costs $c \cdot i + d$ with c, d constants.
- Total costs of all InternalResize calls: $c(2M - 1) + d(\log_2(M) + 1)$.

- *Amortized complexity* of M PushBacks:

$$\frac{bM + c(2M - 1) + d(\log_2(M) + 1)}{M} < b + 2c + d$$

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.
- PushBack(D, v) only calls InternalResize(D) when $D.length \in \{1, 2, 4, 8, 16, \dots\}$.
- A call to InternalResize(D) with $D.length = i$ costs $c \cdot i + d$ with c, d constants.
- Total costs of all InternalResize calls: $c(2M - 1) + d(\log_2(M) + 1)$.

- *Amortized complexity* of M PushBacks:

$$\frac{bM + c(2M - 1) + d(\log_2(M) + 1)}{M} < b + 2c + d = \Theta(1).$$

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.
- PushBack(D, v) only calls InternalResize(D) when $D.length \in \{1, 2, 4, 8, 16, \ldots\}$.
- A call to InternalResize(D) with $D.length = i$ costs $c \cdot i + d$ with c, d constants.
- Total costs of all InternalResize calls: $c(2M - 1) + d(\log_2(M) + 1)$.

- *Amortized complexity* of M PushBacks:

$$\frac{bM + c(2M - 1) + d(\log_2(M) + 1)}{M} < b + 2c + d = \Theta(1).$$

The above analysis can be generalized to include other operations, e.g., PopBack.

Intermezzo: Complexity of PushBack

Consider an empty dynamic array D with $D.reserved = 1$,
and a sequence of M, $M > 0$, PushBack operations.

- Cost of PushBack: some base cost b plus the cost of InternalResize.
- PushBack(D, v) only calls InternalResize(D) when $D.length \in \{1, 2, 4, 8, 16, \dots\}$.
- A call to InternalResize(D) with $D.length = i$ costs $c \cdot i + d$ with c, d constants.
- Total costs of all InternalResize calls: $c(2M - 1) + d(\log_2(M) + 1)$.

- *Amortized complexity* of M PushBacks:

$$\frac{bM + c(2M - 1) + d(\log_2(M) + 1)}{M} < b + 2c + d = \Theta(1).$$

The above analysis can be generalized to include other operations, e.g., PopBack.

To support PopBack efficiently: do not *shrink* too soon:
either *never shrink* or shrink *if requested* or when $4 \cdot D.length < D.reserved$.

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.

Removing elements from the the back is similar:
PopBack(D) undoes PushBack (typically without shrinking).

Arbitrary inserts at position i, $i \neq D.length$, is costly:
Requires one to copy over by one position all values at-or-after position i.

What about the complexity?

PushBack(D, v) is either $\Theta(1)$ or $\Theta(D.reserved)$ (if InternalResize is called).

Can we provide a better analysis?

Amortized complexity of PushBack: $\Theta(1)$.

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.

Removing elements from the the back is similar:
$\textsc{PopBack}(D)$ undoes $\textsc{PushBack}$ (typically without shrinking).

Arbitrary inserts at position i, $i \neq D.length$, is costly:
Requires one to copy over by one position all values at-or-after position i.

- Provides stack modifications in amortized $\Theta(1)$.
- Queue modifications in $\Theta(D.length)$.
- Support *random access* efficiently.

Dynamic arrays

Dynamic Array: a data structure that can hold an *array* of values that is resizes upon need.

Removing elements from the the back is similar:
$\textsc{PopBack}(D)$ undoes $\textsc{PushBack}$ (typically without shrinking).

Arbitrary inserts at position i, $i \neq D.length$, is costly:
Requires one to copy over by one position all values at-or-after position i.

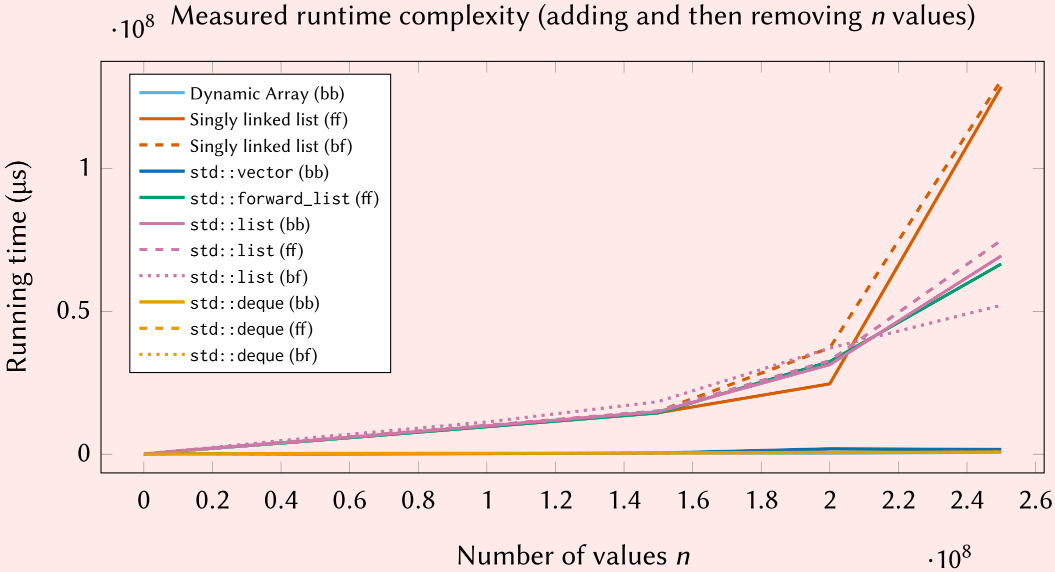
- Provides stack modifications in amortized $\Theta(1)$.
- Queue modifications in $\Theta(D.length)$.
- Support *random access* efficiently.
- *Drawback*: sometimes expensive *resizes*, no efficient queue operations.

A summary of elementary containers

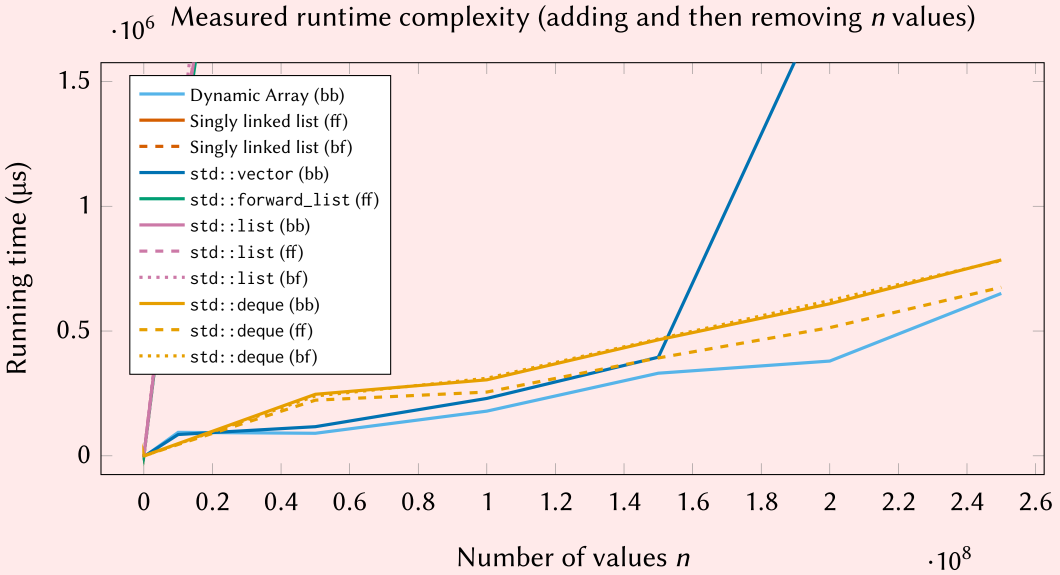
Data structure	Supports ADT: Queue	Stack	Dequeue	Random Access	Memory Usage
Ring Buffer	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	Always NT
Singly linked list	$\Theta(1)$	$\Theta(1)$			$T + P + M$ per value
Doubly linked list	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$		$T + 2P + M$ per value
Dynamic array		$\Theta(1)$ (amortized)		$\Theta(1)$	$\leq M + 2T$ $\leq 2M + 3T$ (during resize)

T is the size of a value, P is the size of a pointer, M is the overhead per memory allocation.

Comparing common containers



Comparing common containers



Elementary containers in practice

Data Collection or Structure	C++	Java
Ring Buffer		`java.util.ArrayDeque`
Singly Linked List	`std::forward_list`	
Doubly Linked List	`std::list`	`java.util.LinkedList`
Dynamic Array	`std::vector`	`java.util.ArrayList`
Other	`std::deque`	
Stack	`std::stack`	Use `ArrayDeque` or `ArrayList`
Queue	`std::queue`	Use `ArrayDeque`

Java provides `java.util.Vector` and `Stack` and `Queue` on top of `Vector`. These are ancient and their usage is *not recommended*. Use `ArrayList` or `ArrayDeque` instead!