Data Structures: An Abstract View

Pasin Manurangsi

Abstract data type (aka Interface / API)

- Specifications
- Which data can be stored
- Which operations are supported & what are the behaviors of those operations

“Problem statement”

ADT is useful as abstraction for building algorithm on top of it

Data Structure

- Description of how the data is kept in memory
- Algorithms for performing each operation

“Solution”

Different data structures for the same ADT can have different running time

Example ADT: Set

- Set `S` of items
- Operations support:
  - `Search(S, k):`
    - Return whether `k` belongs to the set
  - `Insert(S, x):`
    - Insert item `x` into the set
  - `Delete(S, x):`
    - Remove item `x` from the set if it belongs to the set

```
{}
   Insert(S, 4)
{4}
   Find(S, 3)  ⇒  False
   Insert(S, 5)
{4, 5}
   Find(S, 4)  ⇒  True
   Delete(S, 4)
{5}
```

Example ADT: Set

- Set `S` of items
- Operations support:
  - `Search(S, k)`:
    - Return whether `k` belongs to the set
  - `Insert(S, x)`:
    - Insert item `x` into the set
  - `Delete(S, x)`:
    - Remove item `x` from the set if it belongs to the set

Example Application

Implement a system for registering users

`Register(emailAddress)` should register the user if the email address has not been registered before.

```
Register(emailAddress):
    If Search(S, emailAddress):
        Return False   // Fail
    Insert(S, userEmailAddress)
    Return True    // Succeed
```

Time complexity depends on the data structure for set... Will come back to this later

Queue, Stack & Linked List

(ADT)

Queue

- Queue of items
- Operations support:
  - `Enqueue(Q, x)`:
    - Add `x` to the back of the queue
  - `Dequeue(Q)`:
    - Remove the item at the front of the queue
    - Return that item
      - If empty, return NULL
  - `Front(Q)`:
    - Return item at the front of the queue (without removing it)

First-In First-Out (FIFO)

(ADT) Queue

- Queue of items
- Operations support:
  - `Enqueue(Q, x)`:
    - Add `x` to the back of the queue
  - `Dequeue(Q)`:
    - Remove the item at the front of the queue
    - Return that item
      - If empty, return NULL
  - `Front(Q)`:
    - Return item at the front of the queue (without removing it)

First-In First-Out (FIFO)

```
empty
  ↓ Enqueue(Q, 3)
[3]
  ↓ Enqueue(Q, 6)
[3][6]
  ↓ Enqueue(Q, 2)
[3][6][2]
  ↓ Dequeue(Q) ⇒ 3
[6][2]
```

(ADT)

Stack

- Stack of items
- Operations support:
  - `Push(S, x)`:
    - Add `x` to the top of the stack
  - `Pop(S)`:
    - Remove the item at the top of the stack
    - Return that item
      - If empty, return NULL
  - `Top(S)`:
    - Return item at the top of the stack (without removing it)

Last-In First-Out (LIFO)

(ADT) Stack

- Stack of items
- Operations support:
  - `Push(S, x)`:
    - Add `x` to the top of the stack
  - `Pop(S)`:
    - Remove the item at the top of the stack
    - Return that item
      - If empty, return NULL
  - `Top(S)`:
    - Return item at the top of the stack (without removing it)

Last-In First-Out (LIFO)

bottom ⟷ top

`empty`

↓ `Push(S, 3)`

3

↓ `Push(S, 6)`

3	6

↓ `Push(S, 2)`

3	6	2

↓ `Pop(S)` → 2

3	6

(ADT) Stack

- Stack of items
- Operations support:
  - `Push(S, x)`:
    - Add `x` to the top of the stack
  - `Pop(S)`:
    - Remove the item at the top of the stack
    - Return that item
      - If empty, return NULL
  - `Top(S)`:
    - Return item at the top of the stack (without removing it)

Last-In First-Out (LIFO)

(Data structure) Array-based Stack

- `S` has two attributes
  - `arr`: an array of items
  - `size`: # of items
- Manipulate these correctly for each operation

```
Initialize(S):
    Create S.arr of size 5
    S.size = 0
```

size = 0

Push(S, 3)

(Data structure)

Array-based Stack

- S has two attributes
  - arr: an array of items
  - size: # of items
- Manipulate these correctly for each operation

```
Initialize(S):
    Create S.arr of size 5
    S.size = 0
```

5 can be changed to arbitrary number of items we expect the stacks to hold

size = 0

Push(S, 3)

size = 1

3				

Push(S, 6)

size = 2

3	6			

Push(S, 2)

size = 3

3	6	2		

Pop(S)

(Data structure)

Array-based Stack

- `S` has two attributes
  - `arr`: an array of items
  - `size`: # of items
- Manipulate these correctly for each operation

```
Push(S, x):
    S.arr[S.size] = x
    S.size ++
```

Running time: O(1)

size = 0

Push(S, 3)

size = 1

3				

Push(S, 6)

size = 2

3	6			

Push(S, 2)

size = 3

3	6	2		

Pop(S) → 2

size = 2

3	6	2		

(Data structure)

Array-based Stack

- `S` has two attributes
  - `arr`: an array of items
  - `size`: # of items
- Manipulate these correctly for each operation

```
Pop(S):
    If S.size = 0:
        Return NULL
    S.size --
    Return S.arr[S.size]
```

Running time: O(1)

(ADT) **Queue**

- Queue of items
- Operations support:
  - `Enqueue(Q, x)`:
    - Add `x` to the back of the queue
  - `Dequeue(Q)`:
    - Remove the item at the front of the queue
    - Return that item
      - If empty, return NULL
  - `Front(Q)`:
    - Return item at the front of the queue (without removing it)

First-In First-Out (FIFO)

(Data structure) **Array-based Queue** Attempt I

- `Q` has two attributes
  - `arr`: an array of items
  - `size`: # of items
- Manipulate these correctly for each operation

size = 0

Enqueue(Q, 3)

(Data structure)

Array-based Queue Attempt I

- Q has two attributes
  - arr: an array of items
  - size: # of items
- Manipulate these correctly for each operation

```
Initialize(Q):
    Create Q.arr of size 5
    Q.size = 0
```

size = 0

Enqueue(Q, 3)

size = 1

3				

Enqueue(Q, 6)

size = 2

3	6			

Enqueue(Q, 2)

size = 3

3	6	2		

Dequeue(Q)

(Data structure)

Array-based Queue Attempt I

- `Q` has two attributes
  - `arr`: an array of items
  - `size`: # of items
- Manipulate these correctly for each operation

```
Enqueue(Q, x):
    Q.arr[Q.size] = x
    Q.size ++
```

Running time: O(1)

(Data structure)

Array-based Queue Attempt I

- Q has two attributes
  - arr: an array of items
  - size: # of items
- Manipulate these correctly for each operation

size = 0

Enqueue(Q, 3)

size = 1

3				

Enqueue(Q, 6)

size = 2

3	6			

Enqueue(Q, 2)

size = 3

3	6	2		

Dequeue(Q) → 3

size = 2

6	2	2		

```
Dequeue(Q):
    If Q.size = 0:
        Return NULL
    currentTop = Q.arr[0]
    Q.size --
    for i = 0 to Q.size:
        Q.arr[i] = Q.arr[i+1]
    Return currentTop
```

(Data structure)

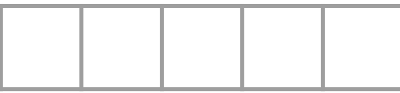
Array-based Queue Attempt I

- `Q` has two attributes
  - `arr`: an array of items
  - `size`: # of items
- Manipulate these correctly for each operation

```
size = 0   [ | | | | ]
    Enqueue(Q, 3)
size = 1   [3| | | | ]
    Enqueue(Q, 6)
size = 2   [3|6| | | ]
    Enqueue(Q, 2)
size = 3   [3|6|2| | ]
    Dequeue(Q) → 3
size = 2   [6|2|2| | ]
```

```
Dequeue(Q):
    If Q.size = 0:
        Return NULL
    currentTop = Q.arr[0]
    Q.size --
    for i = 0 to Q.size:
        Q.arr[i] = Q.arr[i+1]
    Return currentTop
```

Running time: O(n)



(Data structure)

Array-based Queue

- `Q` has three attributes
  - `arr`: an array of items
  - `size`: # of items
  - `front`: index of the front

```
Initialize(Q):
    Create Q.arr of size 5
    Q.front = 0
    Q.size = 0
```

(Data structure)

Array-based Queue

- `Q` has three attributes
  - `arr`: an array of items
  - `size`: # of items
  - `front`: index of the front

```
front = 0
size  = 0
```


↓ `Enqueue(Q, 3)`

```
front = 0
size  = 1
```

3				

↓ `Enqueue(Q, 6)`

```
front = 0
size  = 2
```

3	6			

↓ `Enqueue(Q, 2)`

```
front = 0
size  = 3
```

3	6	2		

↓ `Dequeue(Q)`

```
Enqueue(Q, x):
    Q.arr[Q.front+Q.size] = x
    Q.size ++
```

Running time: O(1)

(Data structure)

Array-based Queue

- `Q` has three attributes
  - `arr`: an array of items
  - `size`: # of items
  - `front`: index of the front

```
front = 0
size  = 0
[ | | | | ]

  Enqueue(Q, 3)

front = 0
size  = 1
[3| | | | ]

  Enqueue(Q, 6)

front = 0
size  = 2
[3|6| | | ]

  Enqueue(Q, 2)

front = 0
size  = 3
[3|6|2| | ]

  Dequeue(Q) → 3

front = 1
size  = 2
[3|6|2| | ]
```

```
Dequeue(Q):
    If Q.size = 0:
        Return NULL
    Q.front ++
    Q.size --
    Return Q.arr[Q.front-1]
```

Running time: O(1)

(Data structure)

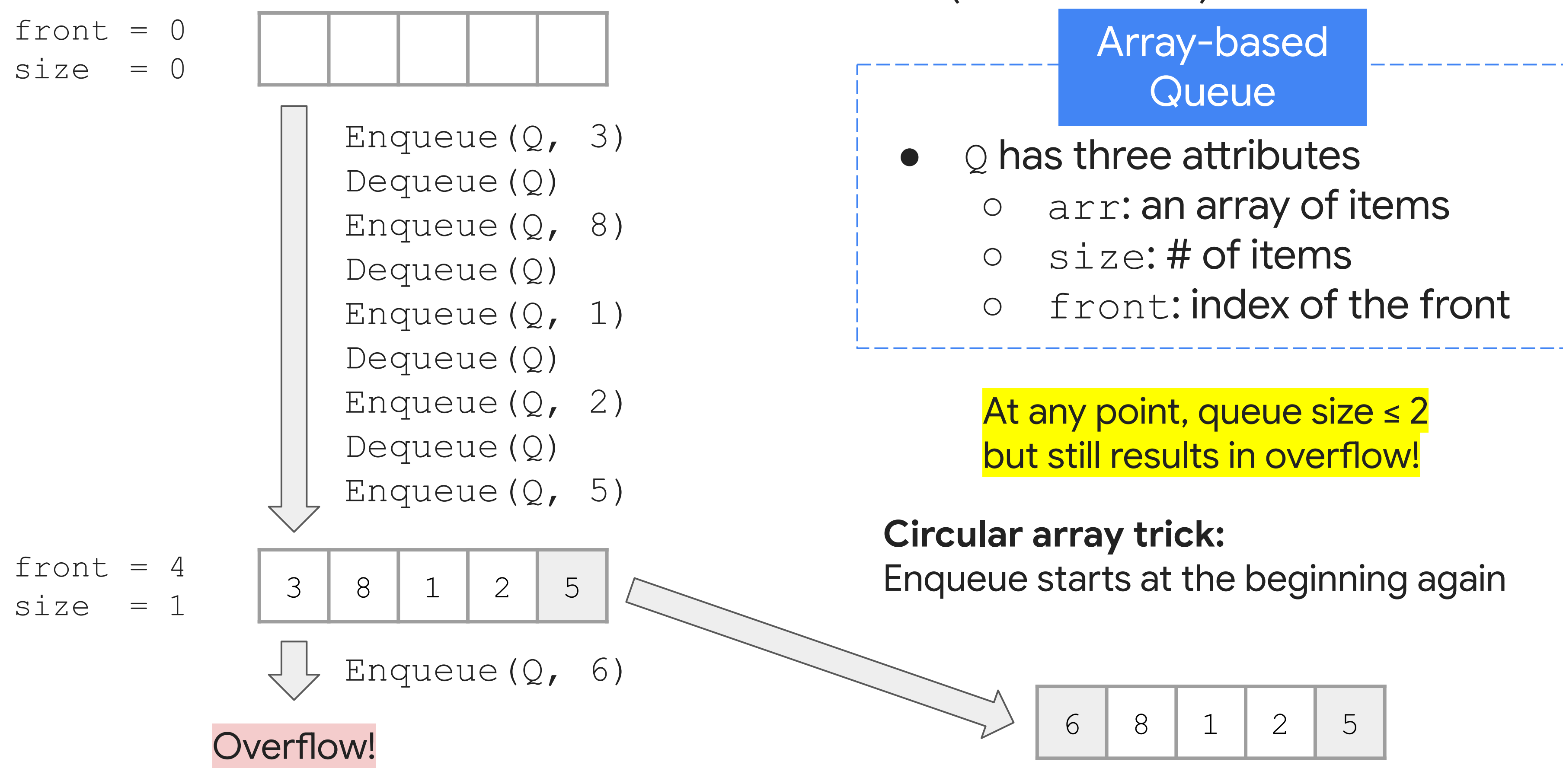
Array-based Stack

- `S` has two attributes
  - `arr`: an array of items
  - `size`: # of items
- Manipulate these correctly for each operation

```
Initialize(S):
    Create S.arr of size 5
    S.size = 0
```

5 can be changed to arbitrary number of items we expect the stacks to hold

How about queue?
If queue size is always ≤ 5, is our data structure ok?



front = 0
size = 0
Enqueue(Q, 3)
Dequeue(Q)
Enqueue(Q, 8)
Dequeue(Q)
Enqueue(Q, 1)
Dequeue(Q)
Enqueue(Q, 2)
Dequeue(Q)
Enqueue(Q, 5)
front = 4
size = 1
3 8 1 2 5
Enqueue(Q, 6)
Overflow!
(Data structure)
Array-based Queue
Q has three attributes
arr: an array of items
size: # of items
front: index of the front
At any point, queue size ≤ 2 but still results in overflow!
Circular array trick:
Enqueue starts at the beginning again
6 8 1 2 5

Circular-Array Trick for Queue

Without circular array

```
Enqueue(Q, x):
    Q.arr[Q.front+Q.size] = x
    Q.size ++
```

```
Dequeue(Q):
    If Q.size = 0:
        Return NULL
    Q.front ++
    Q.size --
    Return Q.arr[Q.front-1]
```

Circular-Array Trick for Queue

Without circular array

```
Enqueue(Q, x):
    Q.arr[Q.front+Q.size] = x
    Q.size ++
```

```
Dequeue(Q):
    If Q.size = 0:
        Return NULL
    Q.front ++
    Q.size --
    Return Q.arr[Q.front-1]
```

With circular array

```
Enqueue(Q, x):
    Q.arr[(Q.front+Q.size)%5] = x
    Q.size ++
```

```
Dequeue(Q):
    If Q.size = 0:
        Return NULL
    Q.front ++
    Q.size --
    Return Q.arr[(Q.front-1)%5]
```

- Wrap around array
- a%b = remainder of a divided by b

Array-based data structures

✔ Easy to implement

❌ Need to know data size beforehand

❌ Need careful overflow handling

(Data structure)

Array-based Stack

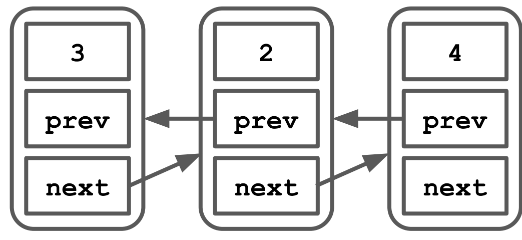
- `S` has two attributes
  - `arr`: an array of items
  - `size`: # of items
- Manipulate these correctly for each operation

```
Initialize(S):
    Create S.arr of size 5
    S.size = 0
```

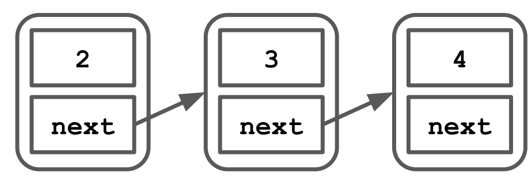
5 can be changed to arbitrary number of items we expect the stacks to hold

(Data structure)

Linked List



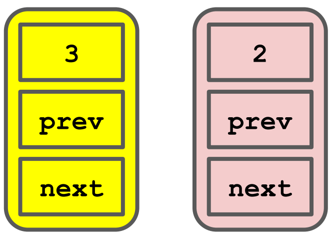
- Each *element* has 3 attributes:
  - `item`: item at this element
  - `prev`: pointer to previous element
  - `next`: pointer to next element
- The linked list `L` then has two attributes:
  - `head`: the first element in `L`
  - `tail`: the last element in `L`

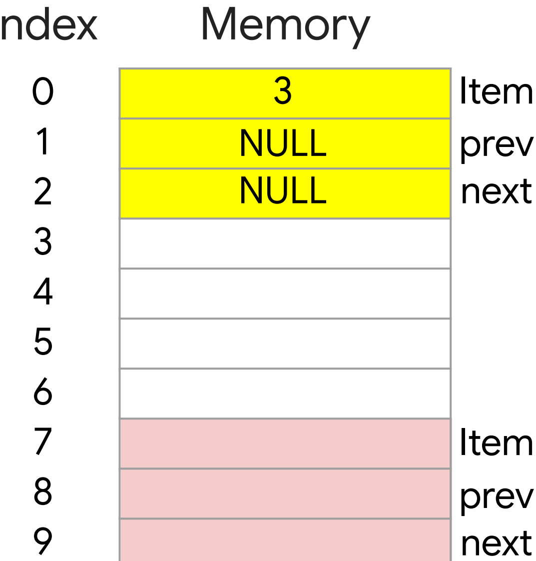


- Sometimes call ***doubly linked list***
- Other variants:
  - ***Singly linked list***: not store `prev`
  - ***Sorted linked list***: `item`s are sorted

(Data structure)

Linked List



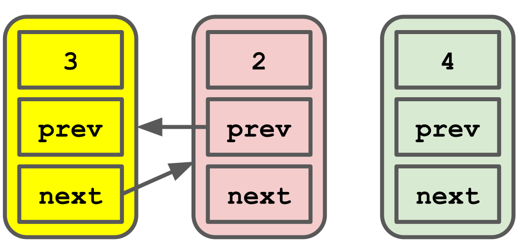


- Each *element* has 3 attributes:
  - `item`: item at this element
  - `prev`: pointer to previous element
  - `next`: pointer to next element
- The linked list `L` then has two attributes:
  - `head`: the first element in `L`
  - `tail`: the last element in `L`

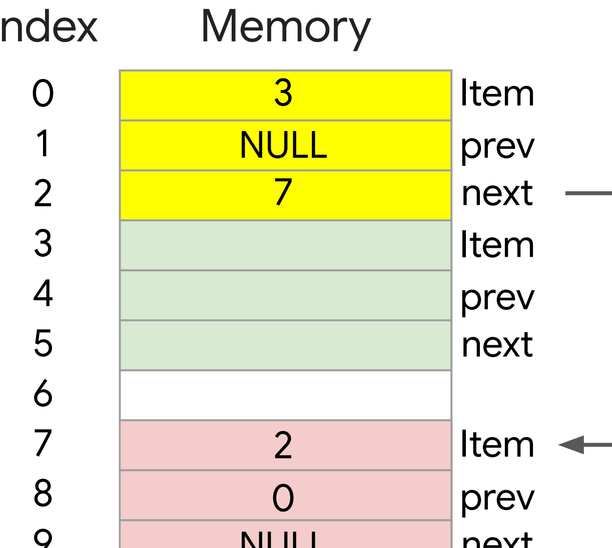
- Sometimes call ***doubly linked list***
- Other variants:
  - ***Singly linked list***: not store `prev`
  - ***Sorted linked list***: `item`s are sorted

(Data structure)

Linked List



- Each *element* has 3 attributes:
  - `item`: item at this element
  - `prev`: pointer to previous element
  - `next`: pointer to next element
- The linked list `L` then has two attributes:
  - `head`: the first element in `L`
  - `tail`: the last element in `L`

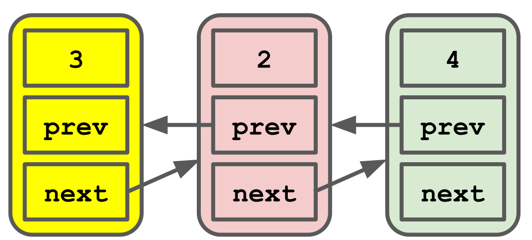


- Sometimes call ***doubly linked list***
- Other variants:
  - ***Singly linked list***: not store `prev`
  - ***Sorted linked list***: `item`s are sorted

(Data structure)

Linked List

- Each *element* has 3 attributes:
  - `item`: item at this element
  - `prev`: pointer to previous element
  - `next`: pointer to next element
- The linked list `L` then has two attributes:
  - `head`: the first element in `L`
  - `tail`: the last element in `L`



Index	Memory	
0	3	Item
1	NULL	prev
2	7	next
3	4	Item
4	7	prev
5	NULL	next
6		
7	2	Item
8	0	prev
9	3	next

- Sometimes call ***doubly linked list***
- Other variants:
  - ***Singly linked list***: not store `prev`
  - ***Sorted linked list***: `item`s are sorted

(Data structure)

Linked List

- Each *element* has 3 attributes:
  - `item`: item at this element
  - `prev`: pointer to previous element
  - `next`: pointer to next element
- The linked list `L` then has two attributes:
  - `head`: the first element in `L`
  - `tail`: the last element in `L`

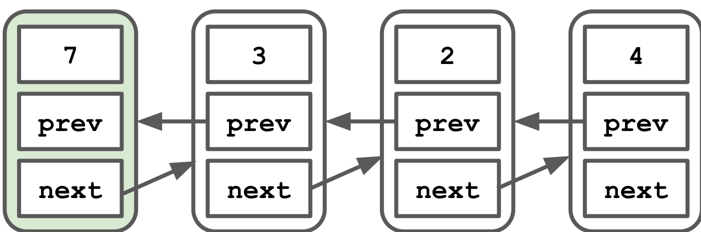
```
Initialize(L):
    L.head = NULL
    L.tail = NULL
```

(Data structure)

Linked List

- Each *element* has 3 attributes:
  - `item`: item at this element
  - `prev`: pointer to previous element
  - `next`: pointer to next element
- The linked list `L` then has two attributes:
  - `head`: the first element in `L`
  - `tail`: the last element in `L`

`Insert(L, 7)`




```
Insert(L, x):
    Create a new element u
    u.item = x
    u.prev = NULL
    u.next = L.head
    L.head.prev = u
    L.head = u
```

Specification

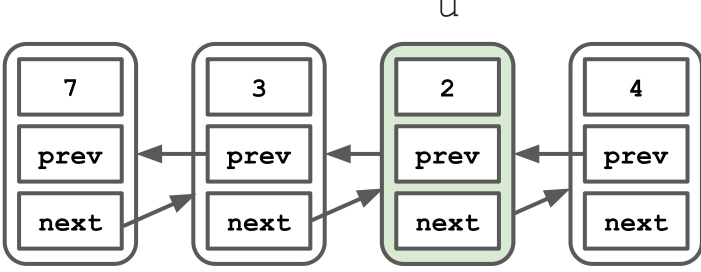
- `Insert(L, x)`: insert item `x` to the beginning of the list

(Data structure)

Linked List

- Each *element* has 3 attributes:
  - `item`: item at this element
  - `prev`: pointer to previous element
  - `next`: pointer to next element
- The linked list `L` then has two attributes:
  - `head`: the first element in `L`
  - `tail`: the last element in `L`

`Delete(L, u)`




```
Delete(L, u):



    Remove node u from memory
```

Specification

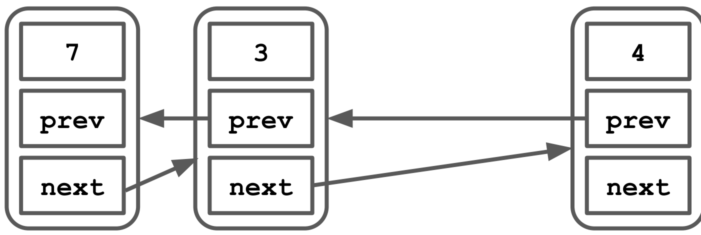
- `Delete(L, u)`: delete element `u` from the list

(Data structure)

Linked List

- Each *element* has 3 attributes:
  - `item`: item at this element
  - `prev`: pointer to previous element
  - `next`: pointer to next element
- The linked list `L` then has two attributes:
  - `head`: the first element in `L`
  - `tail`: the last element in `L`

`Delete(L, u)`




```
Delete(L, u):
    If u.prev != NULL:
        u.prev.next = u.next
    else:
        L.head = u.next
    If u.next != NULL:
        u.next.prev = u.prev
    Remove node u from memory
```

Specification

- `Delete(L, u)`: delete element `u` from the list

(ADT)

Stack

- Stack of items
- Operations support:
  - `Push(S, x)`:
    - Add `x` to the top of the stack
  - `Pop(S)`:
    - Remove the item at the top of the stack
    - Return that item
      - If empty, return NULL
  - `Top(S)`:
    - Return item at the top of the stack (without removing it)

Last-In First-Out (LIFO)

(Data structure)

Linked List-based Stack

`S` has one attribute: a linked-list `L`

```
Initialize(S):
    S.L = new linked list
```

```
Push(S, x):
    Insert(S.L, x)
```

```
Pop(S):
    If S.L.head = NULL:
        Return NULL
    topItem = S.L.head.item
    Delete(S.L, S.L.head)
    Return topItem
```

(ADT)

Queue

- Queue of items
- Operations support:
  - `Enqueue(Q, x)`:
    - Add `x` to the back of the queue
  - `Dequeue(Q)`:
    - Remove the item at the front of the queue
    - Return that item
      - If empty, return NULL
  - `Front(Q)`:
    - Return item at the front of the queue (without removing it)

First-In First-Out (FIFO)

(Data structure)

Linked List-based Queue

`Q` has one attribute: a linked-list `L`

```
Initialize(Q):
    Q.L = new linked list
```

```
Enqueue(Q, x):
    Insert(Q.L, x)
```

```
Dequeue(Q):
    If Q.L.tail = NULL:
        Return NULL
    frontItem = Q.L.tail.item
    Delete(S.L, S.L.tail)
    Return frontItem
```

Linked List-based data structures

✔ Does not need data size beforehand

❌ Harder to implement

❌ Sometimes result in more overhead in running time & memory (to manipulate pointers / allocate new element)

(Data structure)

Linked List

- Each *element* has 3 attributes:
  - `item`: item at this element
  - `prev`: pointer to previous element
  - `next`: pointer to next element
- The linked list `L` then has two attributes:
  - `head`: the first element in `L`
  - `tail`: the last element in `L`

Time Complexity

Data Structure	Push / Enqueue	Pop / Dequeue
Array-based Stack	O(1)	O(1)
Array-based Queue Attempt I	O(1)	O(n)
Array-based Queue	O(1)	O(1)
Linked List-based Stack	O(1)	O(1)
Linked List-based Stack	O(1)	O(1)

n = number of items in the data structure

Brain Teasers

1. Can you implement stack using 2 queues?
    a. Suppose that the queue supports O(1)-time operations.
       What is the running time complexity of your stack?
2. How about the opposite? Can you implement queue using 2 stacks?