BITS Pilani
Pilani Campus

Course Name : Data Structures & Algorithms Design

A Baskar
Computer Science & Information Systems

Outline

- Abstract Data Types
- Stacks
- Queues

Recap

- Pseudocode
- Primitive operations
- Worst case vs best case
- Asymptotic Analysis, Order notation
- Correctness of Algorithms
- Recursion, Recurrence relations

Abstract Data Types

Mathematical model + Informal algorithm → ADT + Pseudocode → Concrete data Str + Program

Abstract Data Type

- An **abstract data type** (**ADT**) is a mathematical model for a certain class of data structures that have similar behavior.

Abstract Data Types (ADTs)

- A method for achieving abstraction for data structures and algorithms
- ADT = model + operations
- Describes what each operation does, but not how it does it
- An ADT is independent of its implementation

Abstract Data Types

- Typical operations on data
  - Add data to a data collection
  - Remove data from a data collection
  - Ask questions about the data in a data collection

Abstract Data Types

- Data abstraction
  - Asks you to think *what* you can do to a collection of data independently of *how* you do it
  - Allows you to develop each data structure in relative isolation from the rest of the solution
  - A natural extension of procedural abstraction

Examples

- Simple ADTs
  - *Stack* ✓
  - *Queue* ✓
  - Vector
  - Lists ✓
  - Sequences
  - Iterators

  All these are called Linear Data Structures

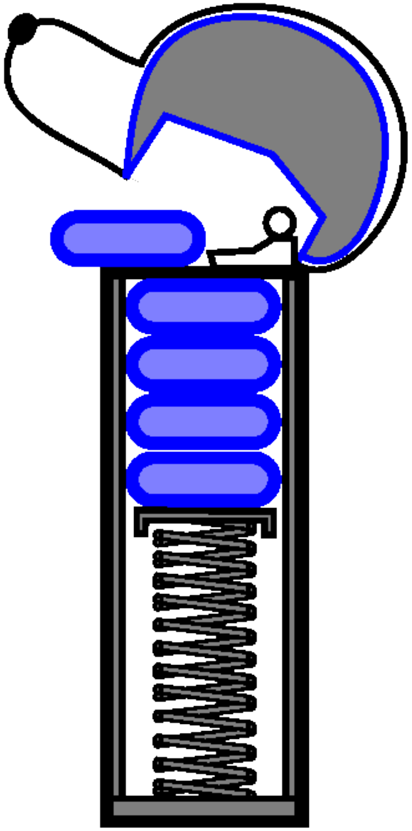
Stacks

Stacks

- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (**LIFO**) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as "**pushing**" onto the stack. "**Popping**" off the stack is synonymous with removing an item.

Stacks

- A coin dispenser as an analogy:



Push (e)

Pop ( ) removes the top element

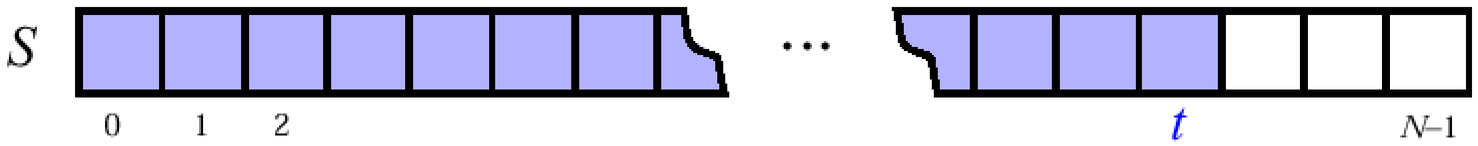
Size ( )

Is empty ( )

top ( ) doesn't remove the top elem

Stacks: An Array Implementation

- Create a stack using an array by specifying a maximum size *N* for our stack.
- The stack consists of an N-element array *S* and an integer variable *t*, the index of the top element in array S.



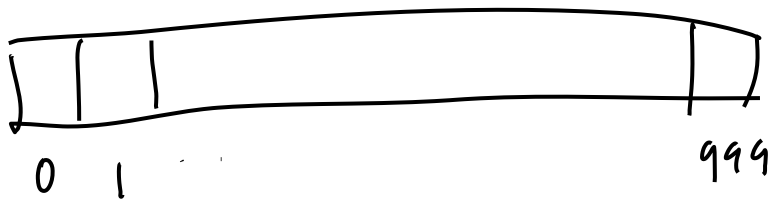
- Array indices start at 0, so we initialize *t* to -1

N = 1000

New( )

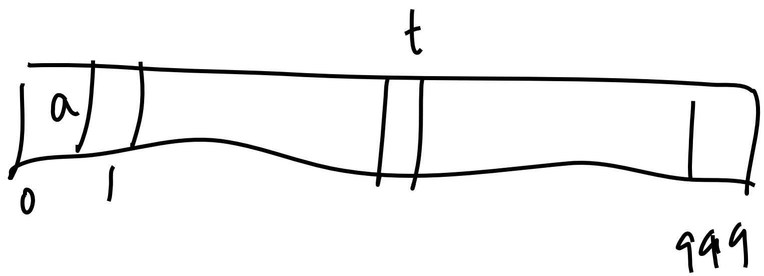
t = -1

creates an array S



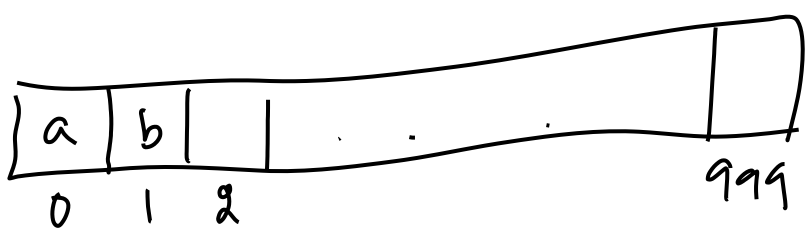
Push(a)

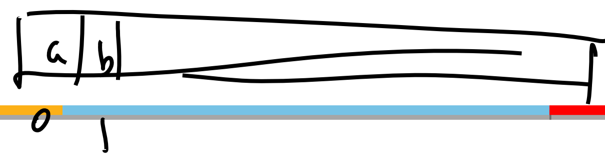
t = 0



Push(b)

t = 1





t = 1

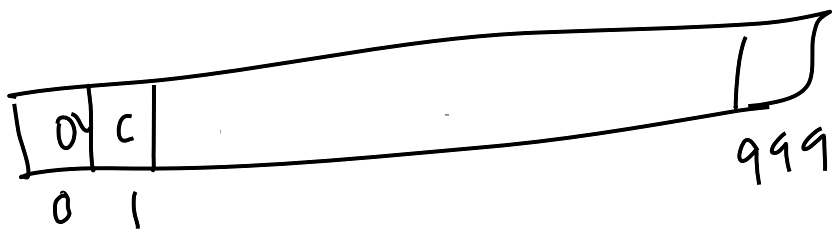
Pop ( )

t = 0

size ( ) return 1

won't access it

Push (c)



t = 1

size ( ) return 2

Push (c)

size = N

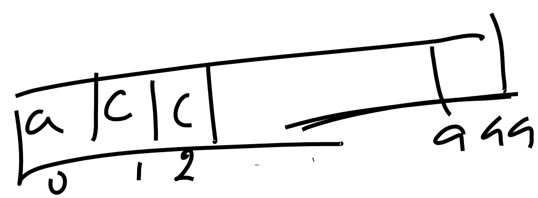
~~t < N~~

t ← t+1

~~t 0~~

t 2

t+1



Stacks: An Array Implementation

- **Pseudo code**

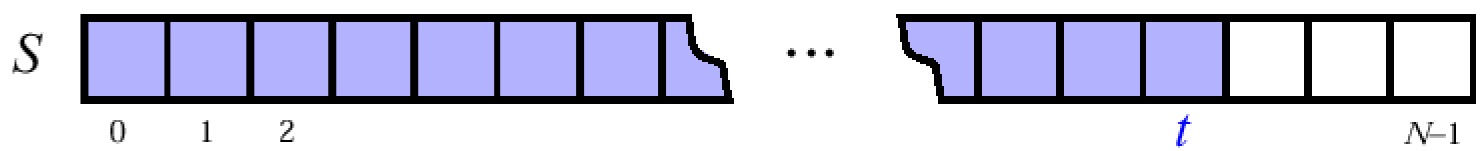
```
Algorithm size()
return t+1

Algorithm isEmpty()
return (t<0)

Algorithm top()
if isEmpty() then
    return Error
return S[t]
```

```
Algorithm push(o)
if size()==N then
    return Error - overflow
t←t+1
S[t]←o

Algorithm pop()
 if isEmpty() then
    return Error
 t←t-1

return S[t+1]
```

return S[t]

t ← t-1

won't be evaluated



Stack - Array implementation

Push()	O(1)
Pop()	O(1)
size()	O(1)
isEmpty()	O(1)
top()	O(1)

O(N) Size

Stacks: An Array Implementation

The array implementation is simple and efficient

(methods performed in O(1)).

Disadvantage

There is an upper bound, *N*, on the size of the stack.

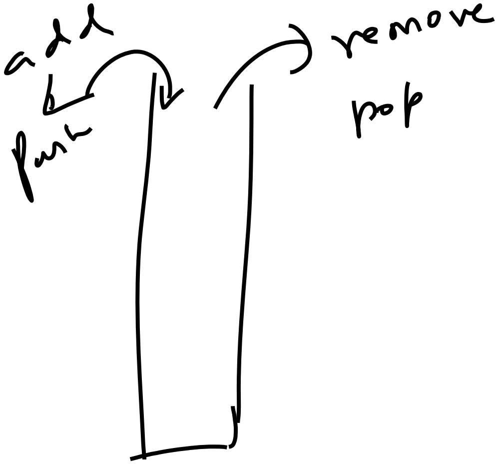
The arbitrary value *N* may be too small for a given application OR

a waste of memory.

N – 1 million

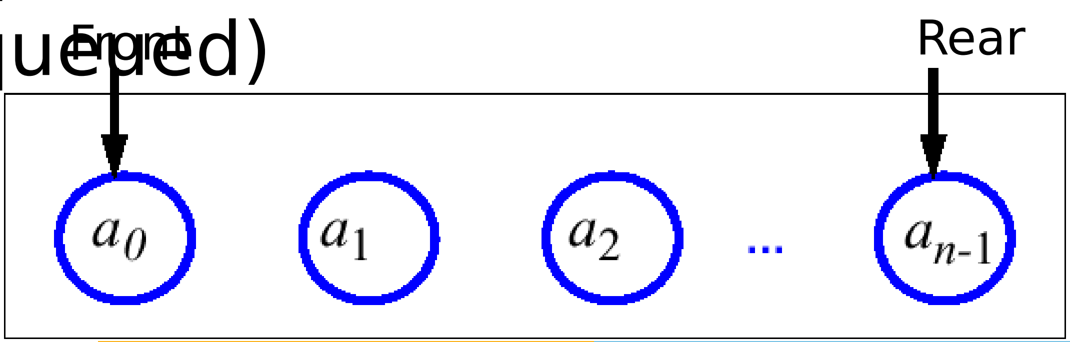
100

QUEUES



Queues

- A queue differs from a stack in that its insertion and removal routines follows the **first-in-first-out** (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)



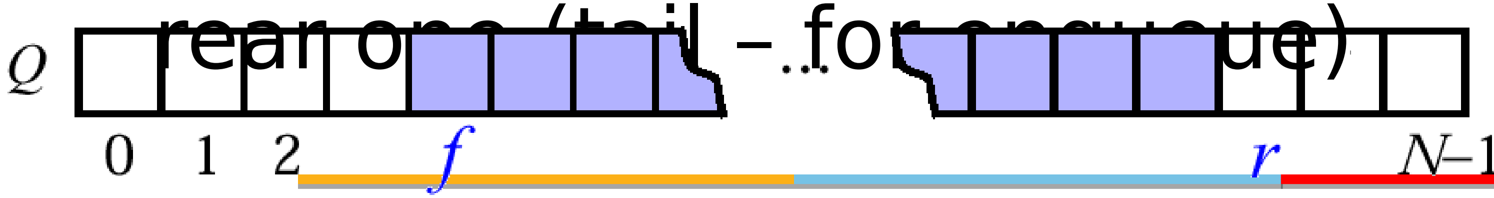
Queues

- The **queue** supports the following methods:
  - **New()** – *Creates an empty queue*
  - **Enqueue(S, o)** - Inserts object *o* at the rear of the queue
  - **Dequeue(S)** - Removes the front element from the queue; an error occurs if **S** is empty
  - **Front(S)** - Returns, but does not remove, the front element; an error occurs if **S** is empty

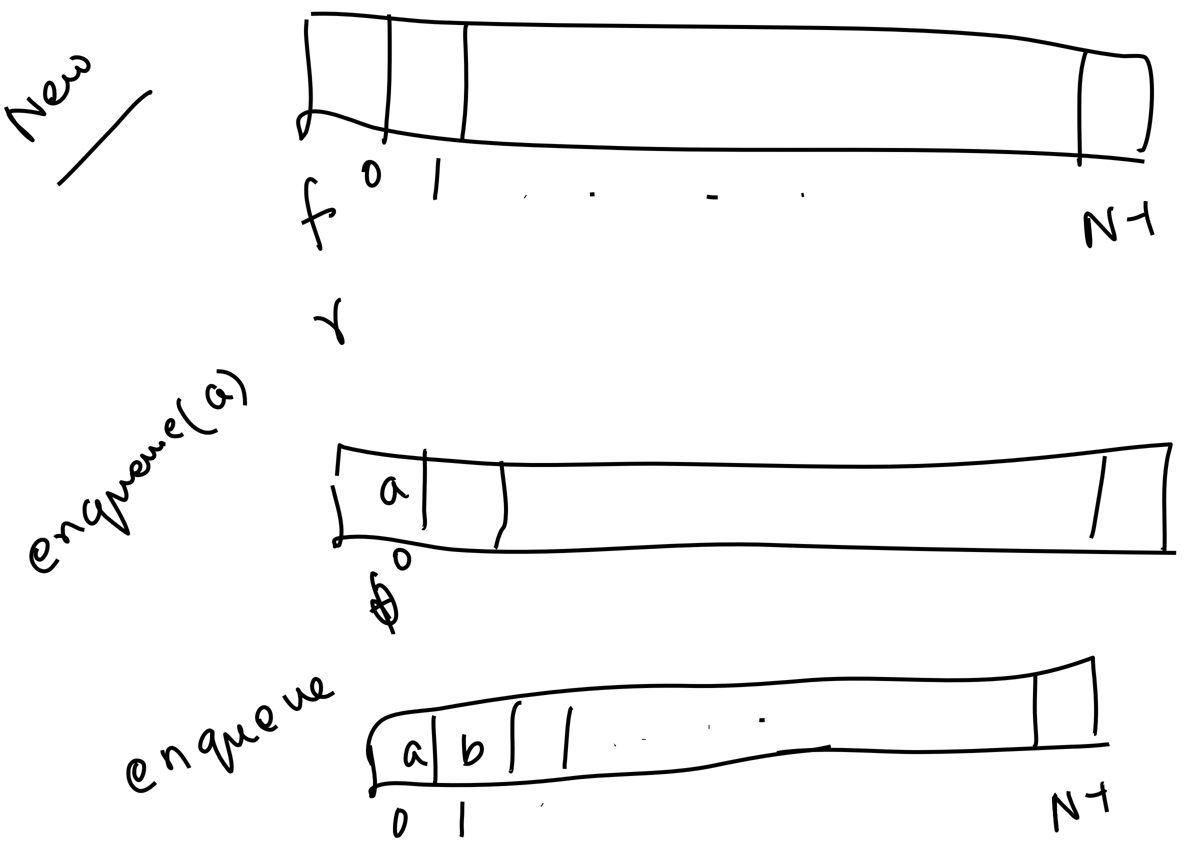
Size(S), isEmpty()

Queues: Array Implementation

- Create a queue using an array
- A maximum size *N* is specified.
- The queue consists of an N-element array *Q* and two integer variables:
  - *f*, index of the front element (head – for dequeue)
  - *r*, index of the element after the rear one (tail – for enqueue)



Naive Implementation



$f = r = 0$

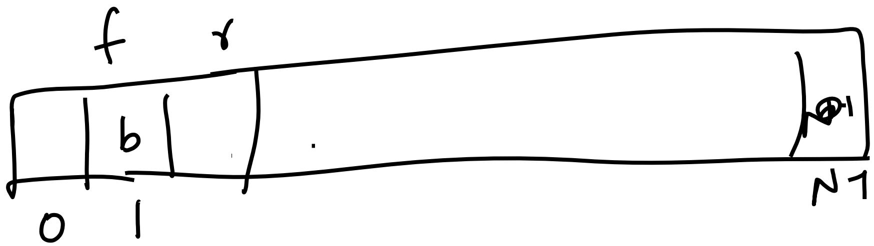
$f = 0$

$r = 1$

$f = 0$

$r = 2$

dequeue

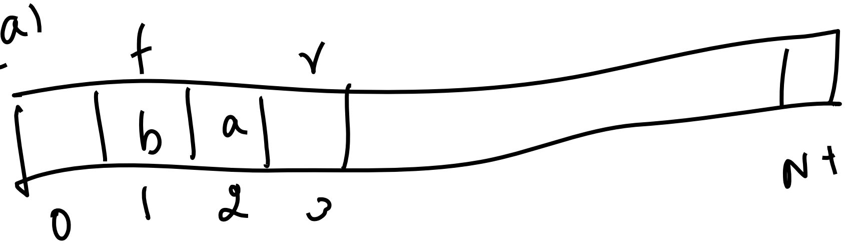


f = 1

r = 2

~~dequ~~

enqueue(a)

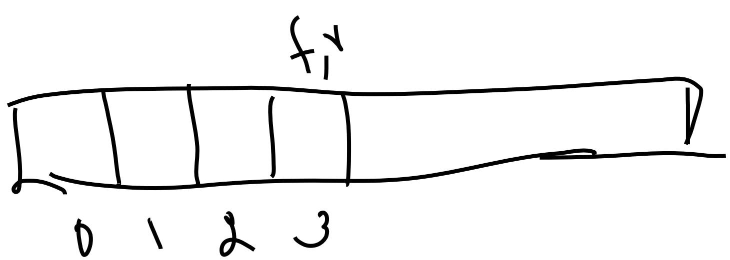


f = 1

r = 3

dequeue

dequeue



f = 3

r = 3

Queue is empty when f = r.

Queues: Array implementation

- Initially, f=r=0
- The queue is empty if f=r

Disadvantage of this implementation

- Repeatedly enqueue and dequeue a single element N times
- Finally, f=r=N
- No more elements can be added to the queue, though there is space in the queue

$N = 5$

0	1	2	3	4

$f = 0$
$r = 0$

1st round

enqueue (a)

a				

$f = 0$
$r = 1$

dequeue (a)

$f = 1$
$r = 1$

2nd round

enque (a)

	a			

$f = 1$
$r = 2$

dequeue (a)

$f = 2$
$r = 2$

5th round

enqueue (a)

				a

$f = 4$
$r = 5$

dequeue (a)

$f = 5$, $r = 5$

After the 5th round

$f = \cancel{6}\ 5$
$r = \cancel{6}\ 5$

enqueue (a)

↓

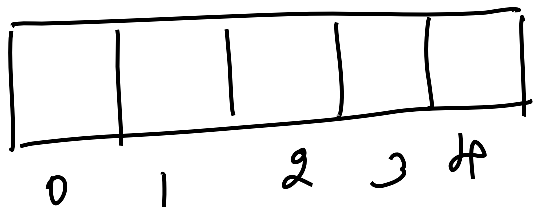
overflow

there is a space

Wrapped Around Implementation

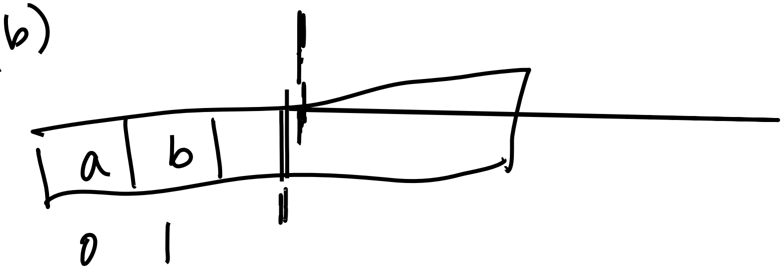
- Let f and r wraparound the end of queue
- Each time r or f is incremented, compute this increment as (r+1)modN or (f+1)modN

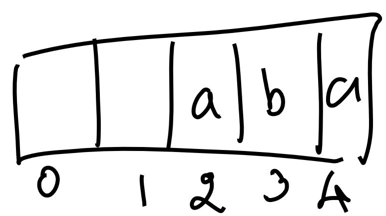
Circular Queue.



$f = 0$
$r = 0$

enqueue (a)
enqueue (b)

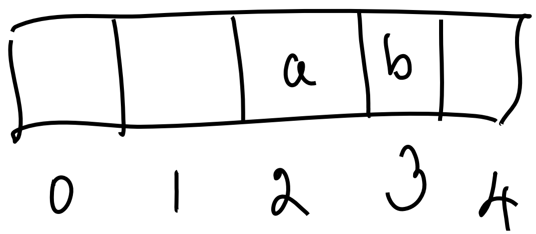




$f = 2$, $r = 5$ } Still there is a space.

Checking $f = N$ and $r = N$ is not enough!

size() returns 3



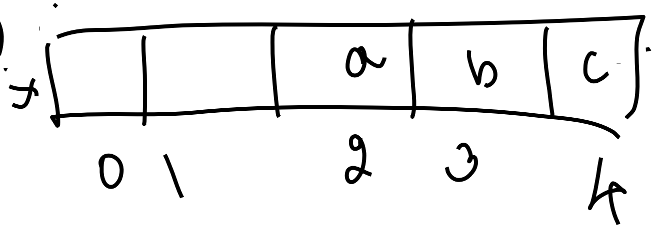
$f = 2$

$r = 4$

$N = 5$

$5 - 2 + 4 = 7 \bmod N$
$= 2$

enqueue(c)

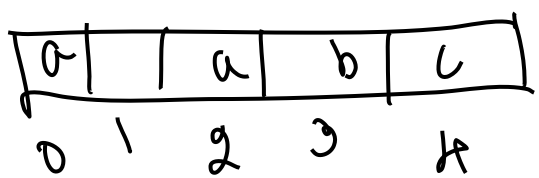


$f = 2 \bmod 5$

$r = 5 \bmod 5$
$= 0$

to wrap around the queue.

enqueue(a)



$f = 2$

$r = 1$

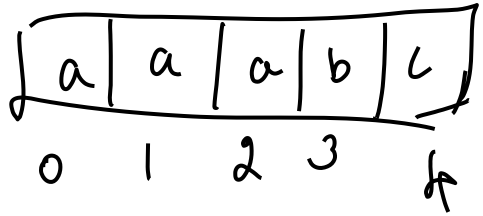
$N = 5$

$N - f + r$

$5 - 2 + 0 \bmod 5$
$= 3$

size returns 4

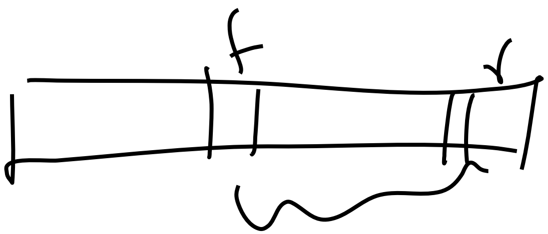
enqueue(a)



$f = 2$

$r = 2$

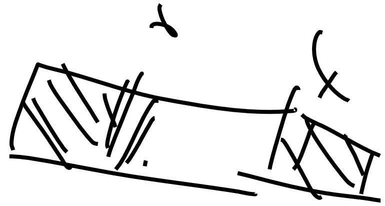
f and r are equal but the queue is not empty.



$r - f$

To avoid this problem we store ~~only~~ at most $(N-1)$ elements in the queue.

$N - f + r$



Queues: Array Implementation

- Pseudo code

```
Algorithm size()
return (N-f+r) mod N

Algorithm isEmpty()
return (f=r)

Algorithm front()
if isEmpty() then
   return Error
return Q[f]
```

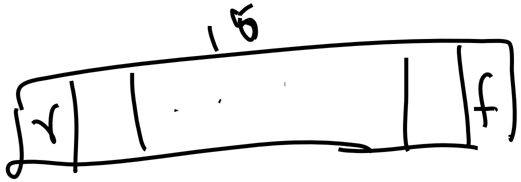
Queues: Array Implementation

- Algorithm **dequeue()**
  ```
  if isEmpty() then
      return Error
  Q[f]=null
  f=(f+1)mod N
  ```

- Algorithm **enqueue(o)**
  ```
  if size = N - 1 then
      return Error
  Q[r]=o
  r=(r +1)mod N
  ```

* O(1) time
Queue operations

* O(N) size.



Size = 1.

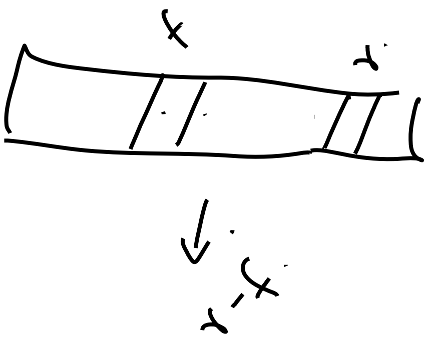
|r - f| = 10.

Q — N

f, r } integers

m Mod n

divide m by n and take the remainder.

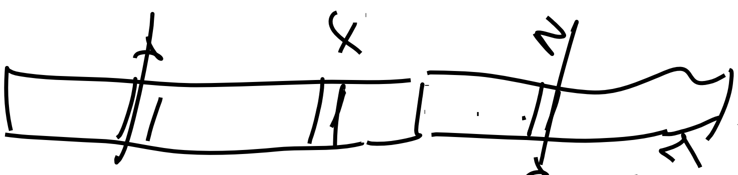


15 mod 7 = 1

15 = 2·7 + 1.

17 mod 7 = 2·7 + 3

3



5 - 2 + 2 = 5 = 0

= 5 mod 5

= 0

Arrays: pluses and minuses

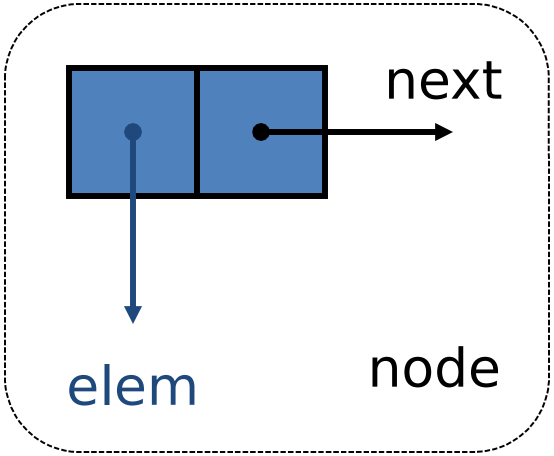
\+ Fast element access.

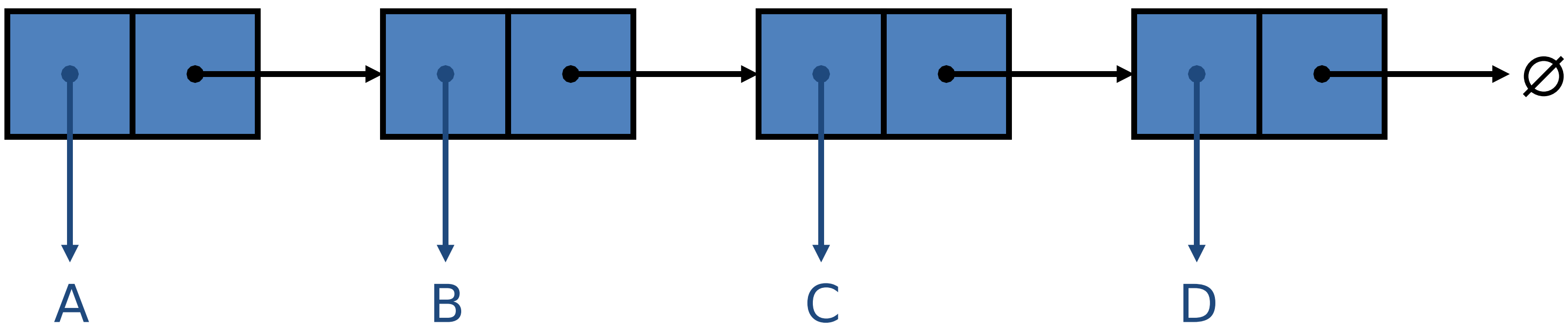
-- Impossible to resize.

- Many applications require resizing!
- Required size not always immediately available.

Singly Linked Lists

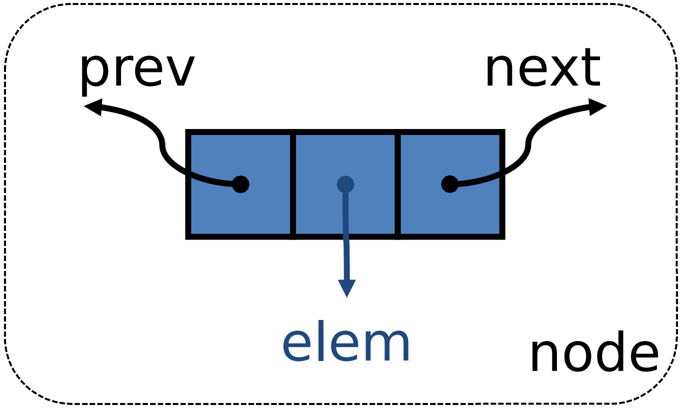
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node

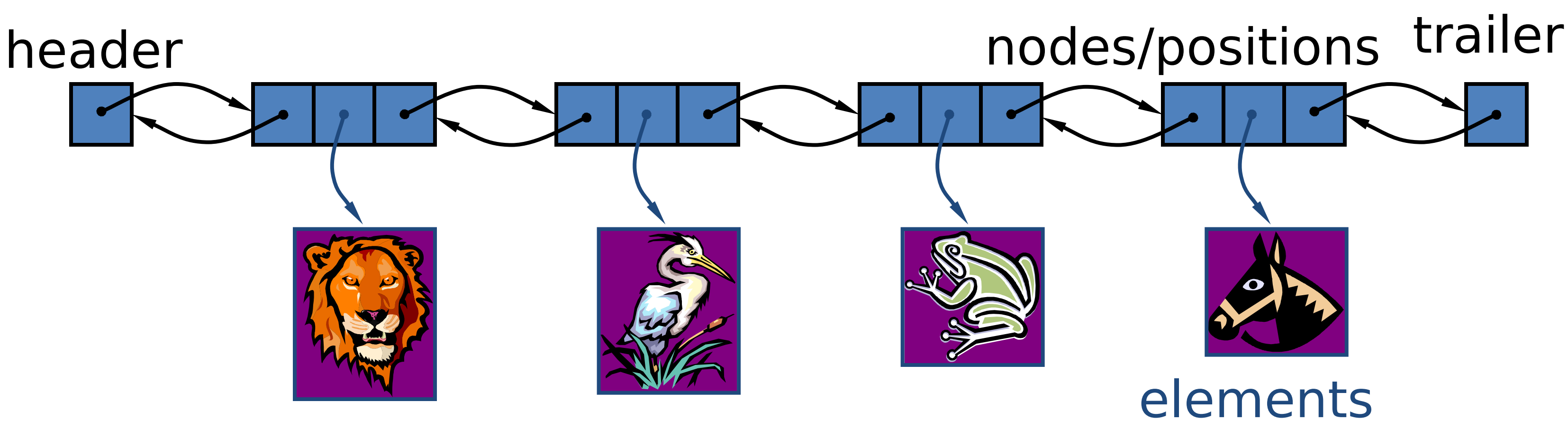




Doubly Linked List

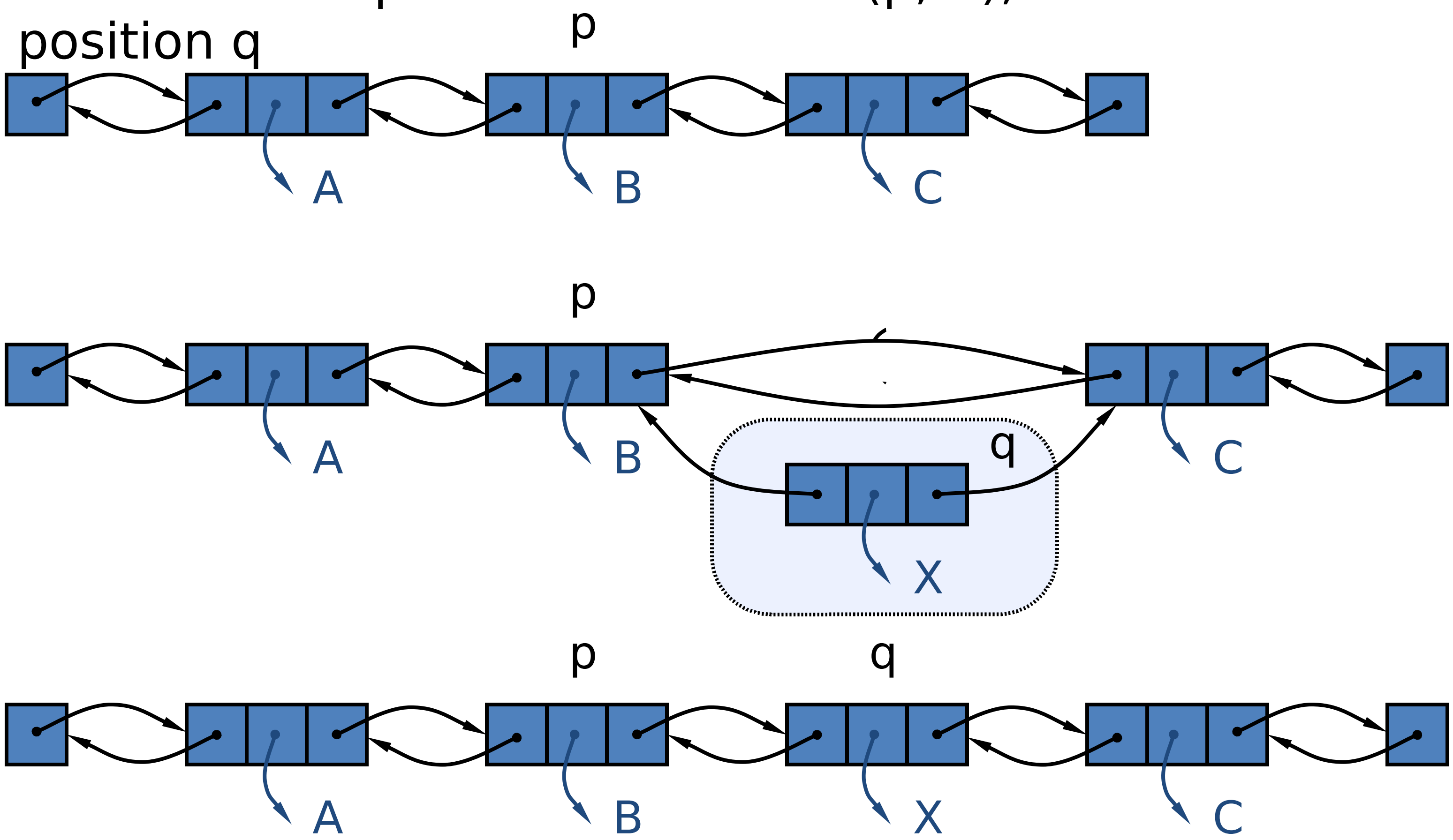
- A doubly linked list is often more convenient!
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





Insertion

- We visualize operation insertAfter(p, X), which returns position q



Insertion Algorithm

Algorithm insertAfter(*p*,*e*):

Create a new node *v*

v.setElement(*e*)

Order is important

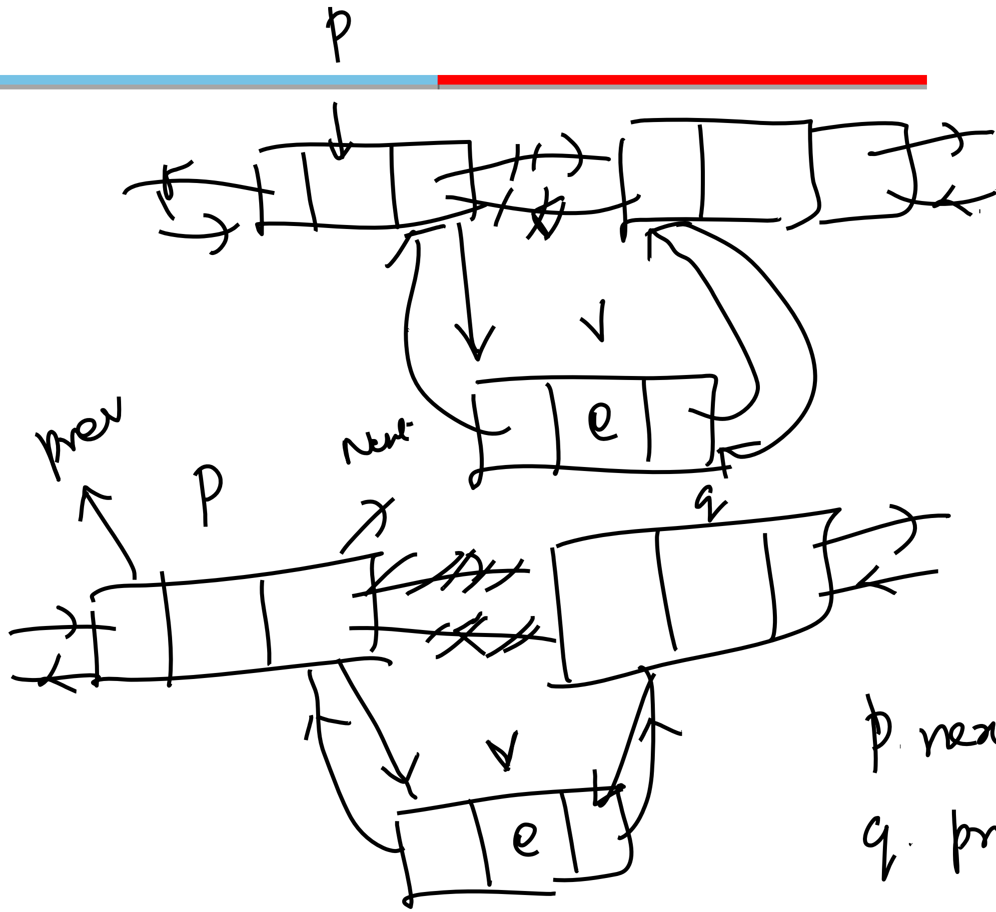
v.setPrev(*p*) {link *v* to its predecessor}

v.setNext(*p*.getNext()) {link *v* to its successor}

(*p*.getNext()).setPrev(*v*) {link *p*'s old successor to *v*}

p.setNext(*v*) {link *p* to its new successor, *v*}

return *v* {the position for the element *e*}



p. next ← v

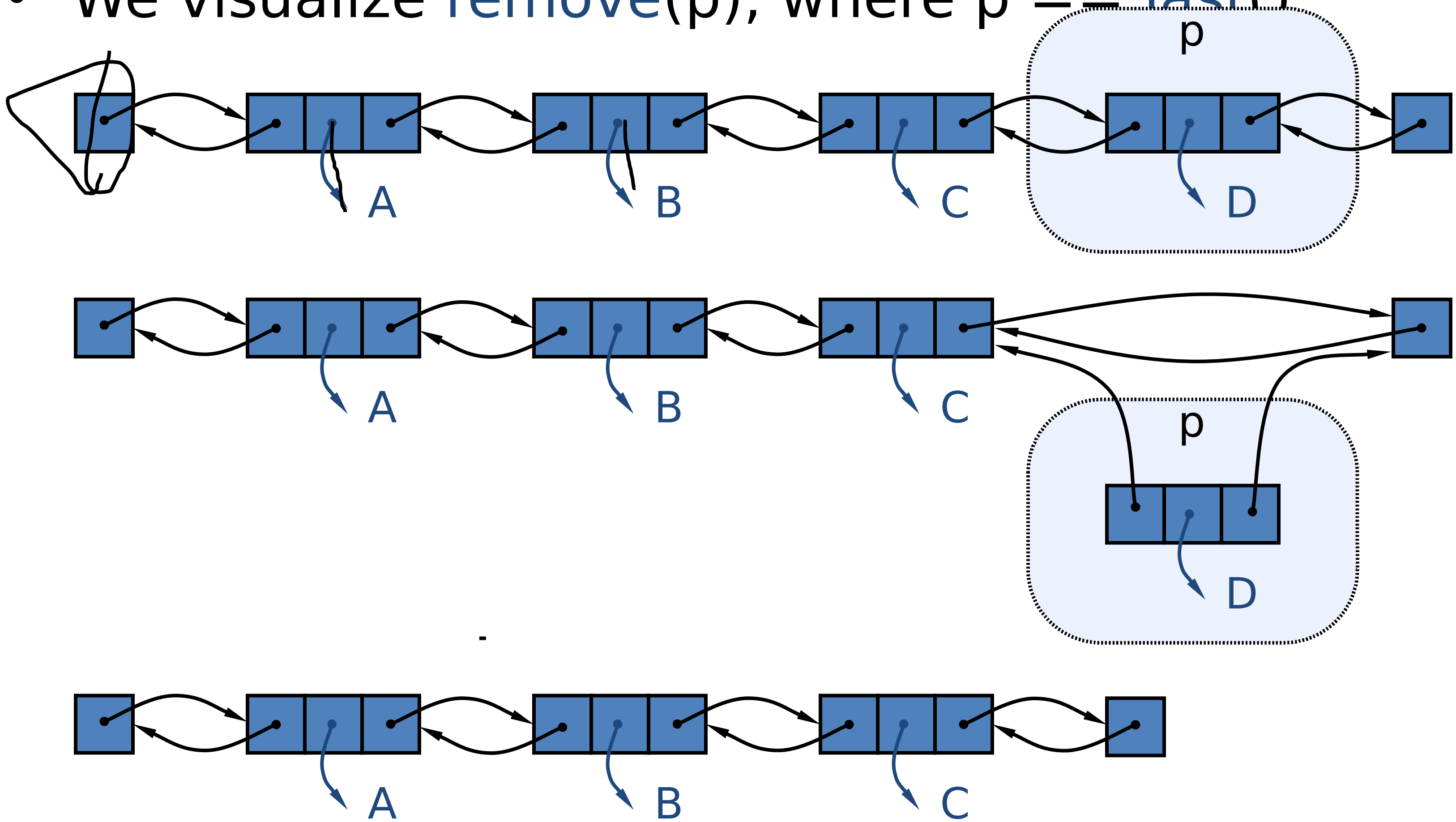
q. prev ← v

v. next

v. prev

Deletion

- We visualize remove(p), where p == last()



Deletion Algorithm

Algorithm remove(*p*):
t = *p*.element {a temporary variable to hold the return value}
(*p*.getPrev()).setNext(*p*.getNext()) {linking out *p*}
(*p*.getNext()).setPrev(*p*.getPrev())
p.setPrev(**null**) {invalidating the position *p*}
p.setNext(**null**)
return *t*

Worst-cast running time

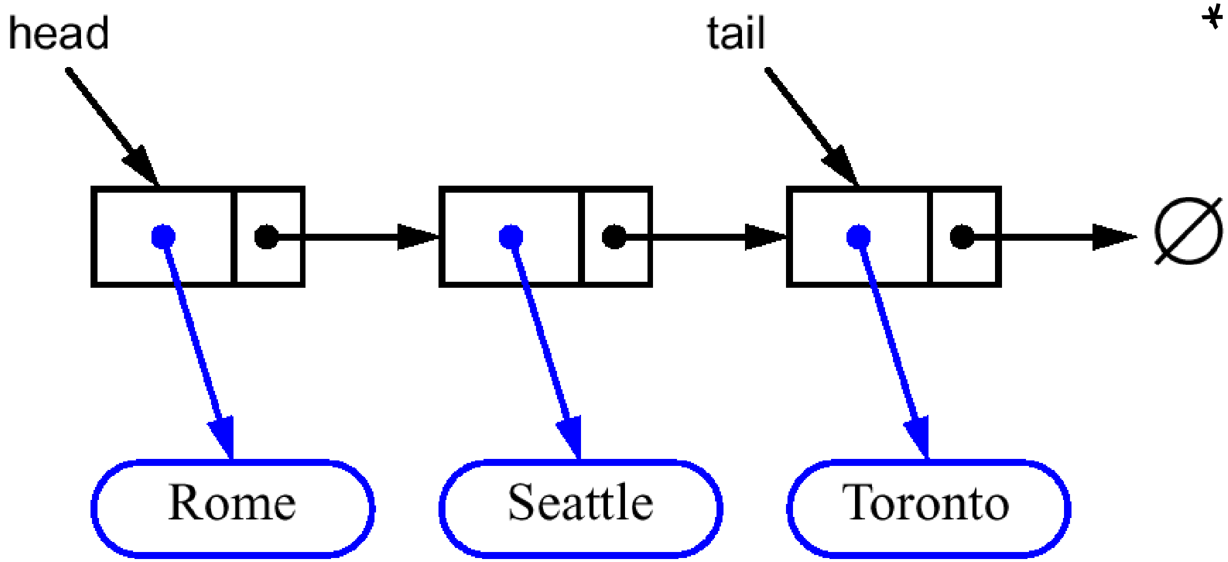
- In a doubly linked list
  - + insertion at head or tail is in O(1)
  - + deletion at either end is on O(1)
  - -- element access is still in O(n)

Know the position inserting or deleting anywhere.

will be O(1)

Stacks: Singly Linked List implementation

- Nodes *(data, pointer)* connected in a chain by links

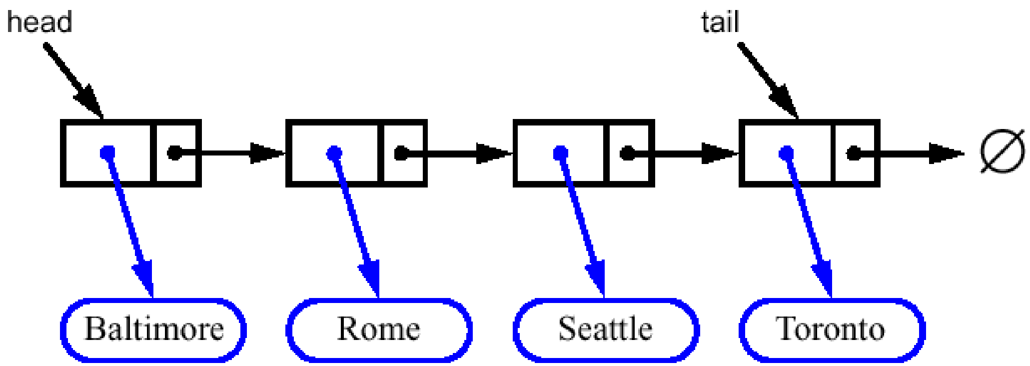


* O(1) time operation

* O (n) where n is the no of elements

- the head or the tail of the list could serve as the top of the stack

Queues: Linked List Implementation



O(1) time operatio

O(n) Size

- Dequeue - advance head reference