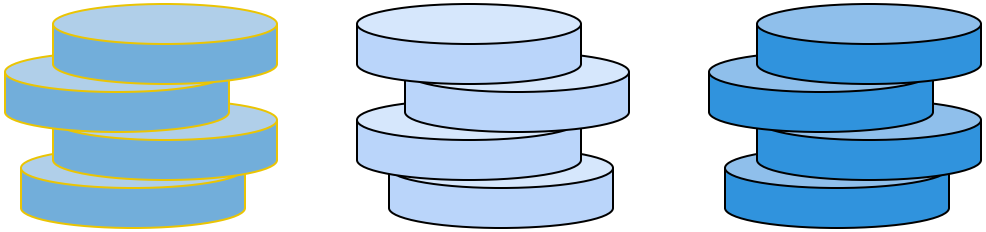
EECS 114:
Engineering Data Structures and Algorithms
Lecture 4

Instructor: Ryan Rusich
E-mail: rusichr@uci.edu
Office: EH 2204

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

Stacks



The Stack

- The concept of stack is derived from the metaphor of a stack of plates in a spring-loaded cafeteria dispenser.
- If you want to remove a plate, you pop the top plate off the stack.
- If you want to replace a plate or insert more plates, you push onto the top of the stack.
- If you wanted to see if a stack of dinner plates were clean, you would need to check the *top* plate, remove that plate, and repeat the process until the entire stack was inspected.

Stack - Examples

- A stack of plates or trays at a cafeteria.
- Call stack for program
- Text editors like emacs and notepad
    - Usually provide an *undo* mechanism that cancels recent changes, reverts document to former states.
    - Accomplish this by keeping text changes in a stack.
    - Are you able to undo changes out of order?
    - Do these stacks have a finite size?

Abstract Data Types (ADTs)

- An abstract data type (ADT) is mathematical model of the set of objects that make-up a data type along with the set of operations allowed on those objects.
- An ADT is a contract between the user of a data structure and its implementer.*
- An ADT specifies:*
  - type of data stored (e.g. any objects, or only ints)
  - available methods, with parameter and return types
  - error conditions associated with methods
  - (optionally) performance guarantees, in terms of space and/or time

* © 2005 Goodrich, Tamassia

Stack ADT

- ***Definition:*** a **stack** is a collection of objects that are inserted and removed according to the *last-in-first-out* (LIFO) principle.
- Objects are inserted (as long as stack not full) onto the **top** of the stack.
- Objects can **ONLY** be removed from the **top** of the stack.
- Objects that have been in the stack the shortest time are first to be removed.
- All stack operations are O(1)

Stack ADT

- Main **stack** operations:
  - ***push***(***Obj o***) : inserts object ***o*** on top of stack
    - An error occurs if the stack is full. (*exception*)
  - ***pop( )*** : removes element from the top of the stack
    - An error occurs if the stack is empty. (*exception*)
  - ***Obj top( )*** : examines the top object on the stack **without** removing it
    - An error occurs if the stack is empty. (*exception*)
    - Use in combination with pop()
    - top() to inspect element, pop() to remove top element

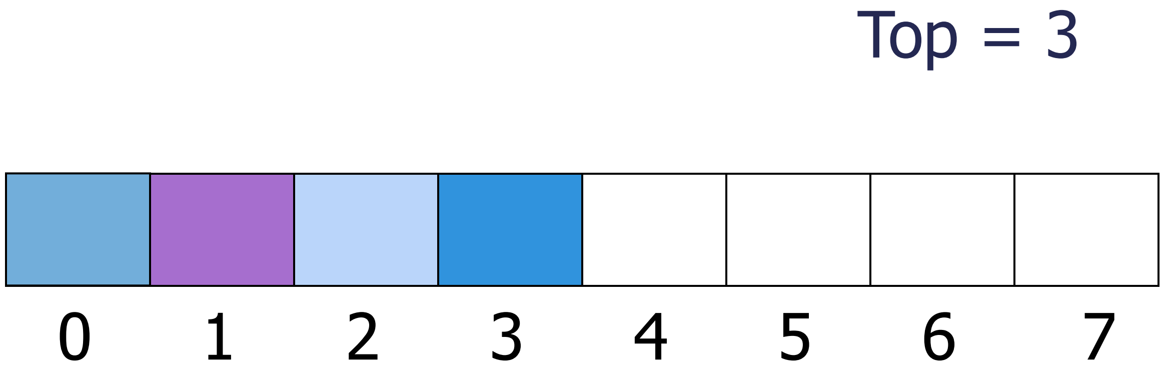
- Auxiliary **stack** operations:
  - ***int size( )*** : returns the number of objects in a stack
  - ***bool isEmpty( )*** : returns **`true`** if the stack is empty, else **`false`**

Array-based Stack

- Store the elements in an N-element array ***S***
- Have an integer variable ***t*** that gives the index of the top element in the array ***S***
- The top element in the array ***S*** is stored in the cell ***S[t]***
- ***See an example…***

Array-based Stack

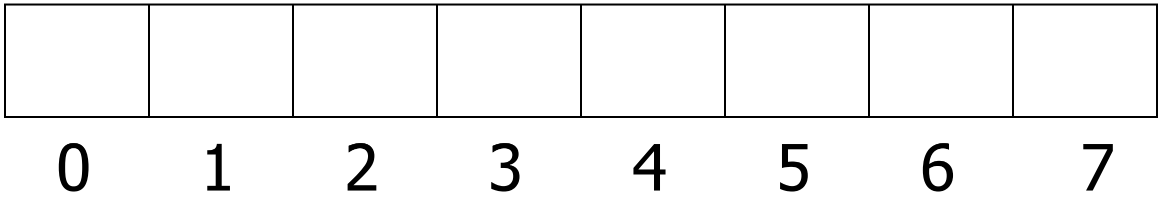
- We push (add) elements from left to right
- A variable keeps track of the index of the last item pushed



Array-based Stack

- We pop (remove) elements from right to left

Top = -1



Stack ADT Pseudocode

```
Algorithm size():
    return t+1
Algorithm isEmpty():
    return (t<0)
Algorithm top():
    if isEmpty() then
        throw a StackEmptyException
    return S[t]
```

Stack ADT Pseudocode

```
Algorithm push(o):
   if size()==N then
        throw a StackFullException
   t = t+1
   S[t] = o

Algorithm pop():
   if isEmpty() then
        throw a StackEmptyException
   t = t-1
```

Stack Class

```
public class Stack
{
    private:
        objectType stack[MAX_STACK_SIZE];
        int top;
    public:
    // constructor sets top to -1
    // functions for stack manipulation
}
```

Stack Implementation - Push

- Array may be full when push called, throw exception

```
public void push ( objectType  obj )
{
          if ( top + 1 == MAX_STACK_SIZE )
              throw FullStackException
          else
              S[++top] = obj;
}
```

Stack Implementation - Pop

- Array may be empty when pop called, throw exception
- getTop() will return top item/objects

```
public void pop ( )
{
    if ( isEmpty ( ) )
        throw EmptyStackException
    else
        --top;
}
```

Stack Implementation- Top

- Array may be empty when pop, throw exception
- Otherwise return top item/object

```
public objectType getTop ( )
{
        if ( isEmpty ( ) )
            throw EmptyStackException
        else
            return S[top];
}
```

Stack Applications

- Postfix Expression Evaluation
- Infix to Postfix Conversion

Reverse Polish Notation (Postfix)

- Operators **`*,/,+,-`** follow their operands:
  - **`3 + 8    (in infix)`**
  - **`3 8 +    (in postfix)`**
- For expressions with multiple operands, operator occurs immediately after its second operand.
  - **`40 4 5 * -,    (in postfix)`**
  - **`40 (4*5) -, -> 40 20 - ,  40 - 20 , 20`**
- Eliminates need for parentheses to force operator precedence.
- Used widely for computation in early desktop calculators.

Stack Application –

Postfix Expression Evaluation

- You may assume I give you a valid postfix expression on exams.
- Algorithm
    - Process postfix expression one item at a time
    - Operand - push
    - Operator – top/pop 2 times
        - evaluate expression push result onto stack

Stack Application –

Postfix Expression Evaluation

```
3 * (5 + ((2 + 3) * 8) + 5) => 3 5 2 3 + 8 * + 5 + *
```

Current Symbol	Stack
3	3
5	3 5
2	3 5 2
3	3 5 2 3
+	3 5 5

Stack Application –

Postfix Expression Evaluation

`3 * (5 + ((2 + 3) * 8) + 5) => 3 5 2 3 + 8 * + 5 + *`

Current Symbol	Stack
8	3 5 5 8
*	3 5 40
+	3 45
5	3 45 5
+	3 50
*	150

Stack Application –
Infix to Postfix Conversion

- Stack can be used to convert infix mathematical expressions to postfix mathematical expressions.

Stack Application –

Infix to Postfix Conversion

- Algorithm
  - Process infix expression one item at a time
  - Operand - write to output
  - Operator - pop and write to output until an entry of lower priority is found (don't pop left parentheses) then push
  - Left parenthesis - push
  - Right parenthesis - pop stack and write to output until left parentheses is found, pop left parenthesis
  - When done processing expression, pop remaining items and write them to output
  - NOTE: Parentheses are not written to the output

Stack Application –

Infix to Postfix Conversion

a + b * c - (d * e + f) * g

Rule	Stack	Output
Operand - write to output		a
	+	a
	+	ab
	+*	ab
	+*	abc
	-	abc*+
	-(	abc*+
	-(	abc*+d
	-(*	abc*+d
	-(*	abc*+de

Stack Application –

Infix to Postfix Conversion

a + b * c - (d * e + f) * g

Rule	Stack	Output
When done processing expression, pop remaining items and write to output	-(+	abc*+de*
	-(+	abc*+de*f
	-	abc*+de*f+
	-*	abc*+de*f+
	-*	abc*+de*f+g
		abc*+de*f+g*-

Linked List-based Stack

```
public bool isEmpty ( ) {
        if ( top == NULL )
            return true;
        else
            return false;
}
```

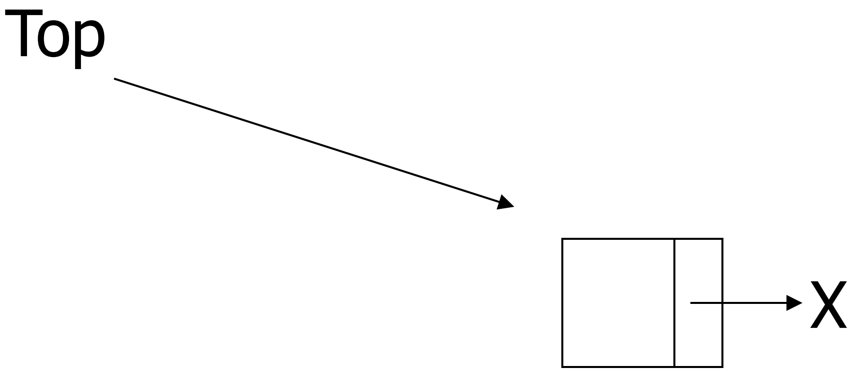
```
public objectType getTop ( ) {
        if ( top )
            return top.obj;
        else
            return null
}
```

```
public void push ( objectType  obj )  {
        Node  newNode = new Node();
        newNode.obj = obj;
        newNode.next = top;
        top = newNode;
}
```

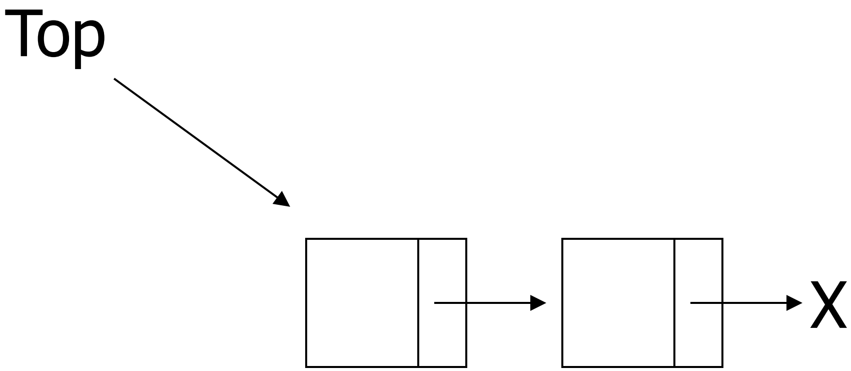
Linked List-based Stack

Top → X

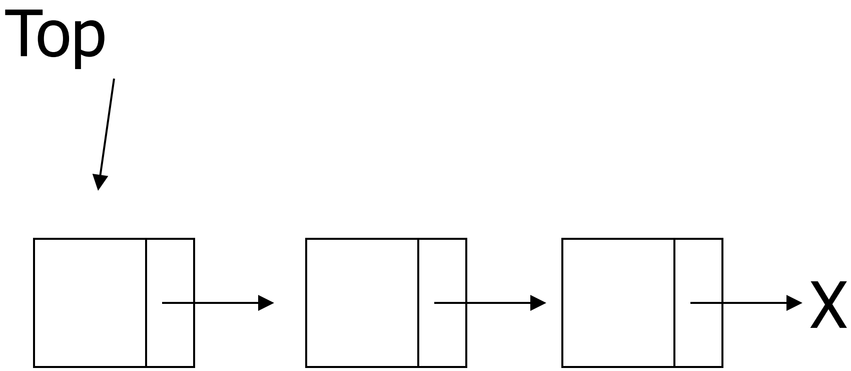
Linked List-based Stack



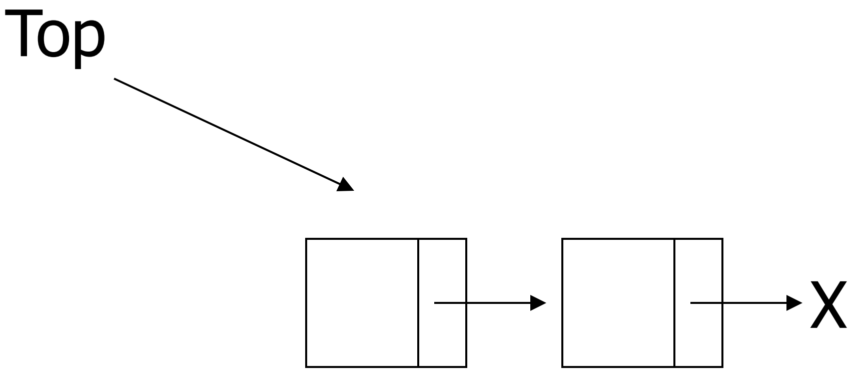
Linked List-based Stack



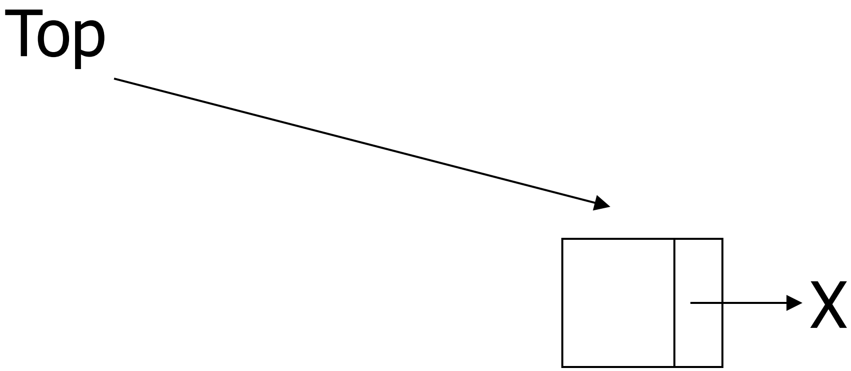
Linked List-based Stack



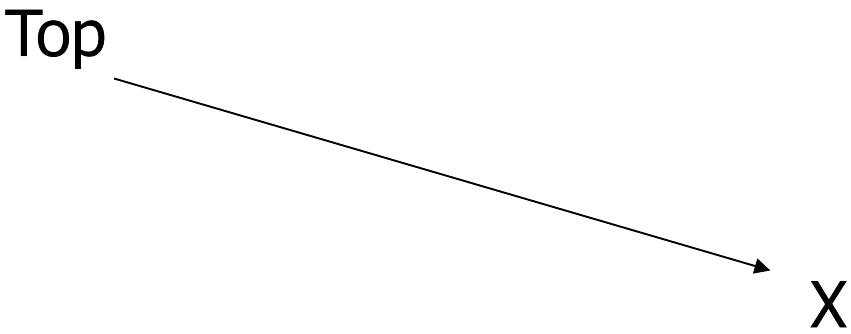
Linked List-based Stack



Linked List-based Stack



Linked List-based Stack



Top

X

Abstract Data Type (ADTs)

- A set of objects together with a set of operations.
- Mathematical abstraction, *i.e.*, hides implementation details.
- Examples: lists, queues, stacks, dictionaries, graphs.
- Typical operations are:
    - Add
    - Remove
    - Size (need a counter or a perform a traversal)
    - Contains (search)
- Allows us to reason about a data structure's behavior.

Queues

Queue ADT

- ***Definition:*** a **queue** is a collection of objects that are inserted and removed according to the first-in-first-out (FIFO) principle.
- Objects are inserted into the **rear** of the queue.
- Objects can **ONLY** be removed from the **front** of the queue.
- Objects that have been in the queue the longest are first to be removed.
- All queue operations are O(1).
  - *All of the action occurs at the* ***front*** *or* ***rear*** *of queue.*

Queue - Examples

- Movie ticket line
- Amusement park line
- Grocery store checkout
- Access to shared resources (e.g., printer queue)
- Phone calls to large companies
- Freeway off-ramp
- Life ☺

Queue ADT

- Main **queue** operations:
  - ***enqueue(o)*** : insert object ***o*** at the rear of the queue.

    push(o)
  - ***dequeue( )*** : remove from the queue the object in the front.

    pop()

    An error occurs if the queue is empty. (*exception*)
  - ***front( )*** : returns the element at the front **without** removing it.

    front()

    An error occurs if the queue is empty. (*exception*)

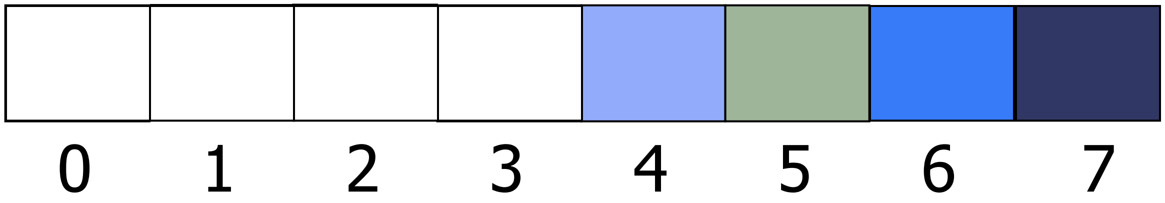
- Auxiliary **queue** operations:
  - ○ ***size( )*** : returns the number of objects in a queue. Either store as a variable counter or calculate it.
  - ○ ***isEmpty( )*** : returns **`true`** if the stack is empty, else **`false`**

Naïve Array-based Queue

- Two variables keep track of the front and rear
  - ***front*** - index of the ***front*** element, initialize to 0
  - ***rear*** - index of the ***rear*** element, initialize to 0
- Variable for number of objects in queue ***Q***
  - ***size***
- Variable for capacity of the queue ***Q***
  - *N*

Naïve Array-based Queue

front = 4
rear = 8

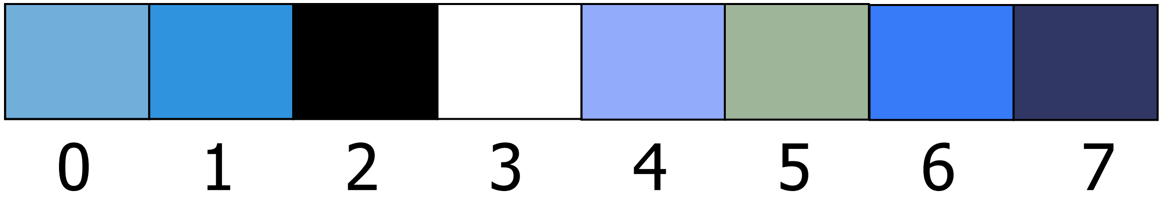


What happens on the next enqueue operation?
What are the possible solutions?

Circular Array-based Queue

- Best solution - use a circular array (wraps around)
    - ***Enqueue*** at the beginning of the array

front = 4
rear = 3



Circular Array-based Queue

- Even though there is plenty of room in the queue, *rear* is at the last cell.
- We want to be able to wrap around.
- We want to index *Q[0] to Q[N-1]* and then immediately go back to *Q[0]*.
- For ***Enqueue***:
  - *rear = (rear + 1) % N , where N=8, Q[0,1,2,…,7]*
  - *rear never points to 8 for N = 8*
  - *rear = (7+1)% 8, wraps around to 0*
- Similarly you can make *front* wrap around.

Queue ADT - Pseudocode

```
Algorithm dequeue():
    if isEmpty() then
        throw a QueueEmptyException
    $f \leftarrow (f + 1) \bmod N$
Algorithm enqueue(o):
    if size()==N-1 then
        throw a QueueFullException
    $Q[r] \leftarrow \mathbf{o}$
    $r \leftarrow (r + 1) \bmod N$
```

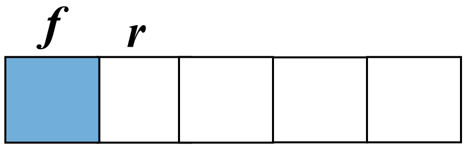
Circular Queue – Pseudocode

```
Algorithm size():
    return $(N - f + r) \bmod N$
Algorithm isEmpty():
    return $(f == r)$
Algorithm front():
    if isEmpty() then
        throw a QueueEmptyException
    return $Q[f]$
```

size = $(N - f + r) \% N$

size = $(5 - 0 + 1) \% 5$

size = 1 = $(6) \% 5$



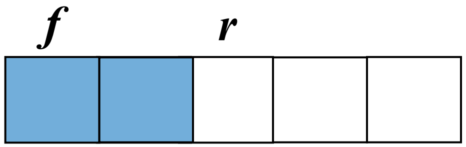
Circular Queue – Pseudocode

```
Algorithm size():
    return $(N - f+r) \bmod N$
Algorithm isEmpty():
    return $(f==r)$
Algorithm front():
    if isEmpty() then
        throw a QueueEmptyException
    return $Q[f]$
```

$size = (N - f + r) \% N$

$size = (5 - 0 + 2) \% 5$

$size = 2 = (7) \% 5$



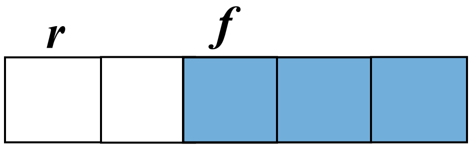
Circular Queue – Pseudocode

```
Algorithm size():
    return $(N - f+r) \bmod N$
Algorithm isEmpty():
    return $(f==r)$
Algorithm front():
    if isEmpty() then
        throw a QueueEmptyException
    return $Q[f]$
```

$size = (N - f + r) \% N$

$size = (5 - ? + ?) \% 5$

$size = 3 = (?) \% 5$



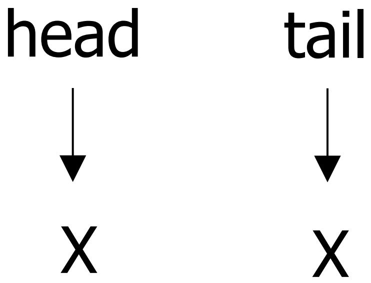
Extendable Array-based Queue

- In an *enqueue* operation, when the array is full, instead of making this an error condition, we can replace the array with a larger one
- Generally every time you increase the size of an array, you will double it in size.
- This disadvantage can also be addressed by using a linked list rather than an array as the underlying data structure.

Linked List Based Queue

- Using a linked list -- can remove the size restrictions of an array
- Queue can grow dynamically
- Linked list with front and rear pointers
  - *front* is the same as *head*
  - *rear* is the same as *tail*
- *head* and *tail* initially point to NULL
  - Similar to array-based queue where *head* and *tail* are set to zero

Linked List-based Queue

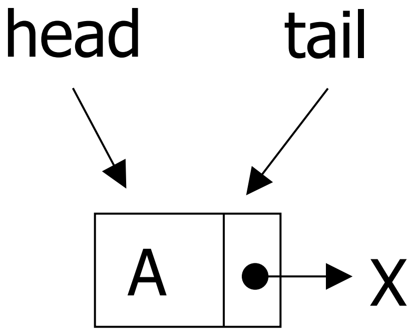


head tail

X X

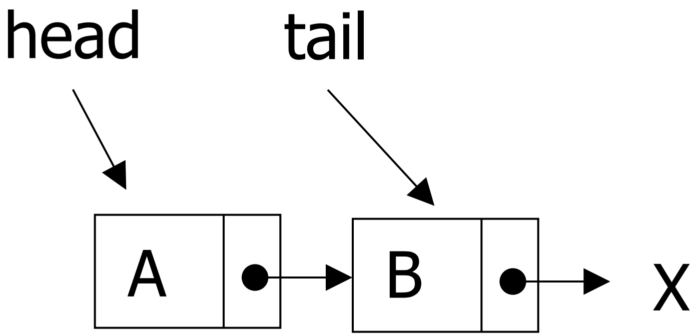
Linked List-based Queue

Enqueue



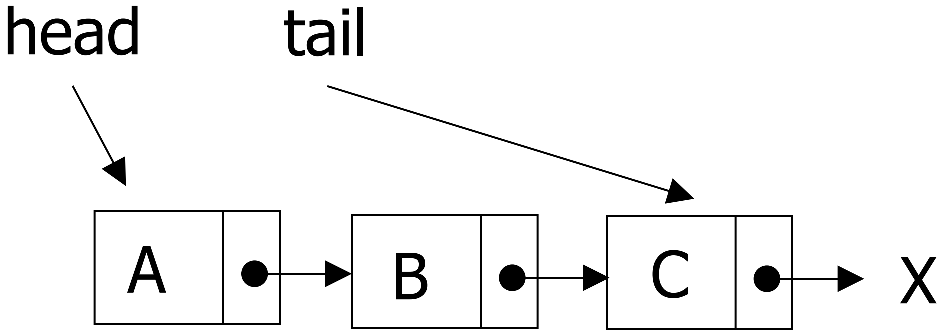
Linked List-based Queue

Enqueue



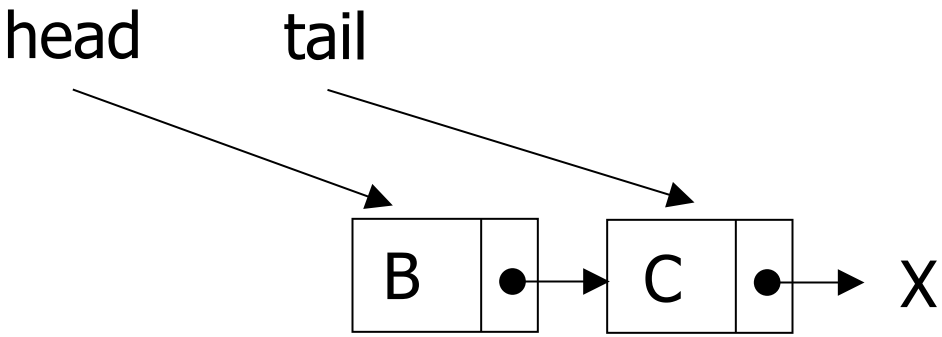
Linked List-based Queue

Enqueue



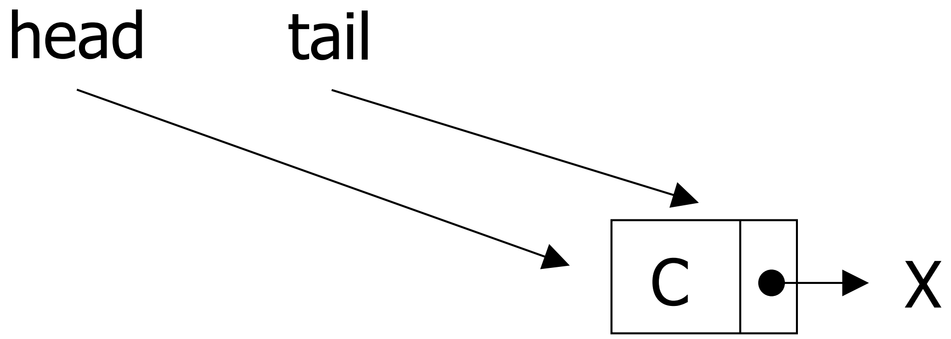
Linked List-based Queue

Dequeue



Linked List-based Queue

Dequeue


head
tail
c
X