Lecture 2
Linear Data Structures

Subject Lecturer: Kevin K.F. YUEN, PhD.

Acknowledgement: Slides were offered from Prof. Ken Yiu.
Some parts has been revised and indicated.

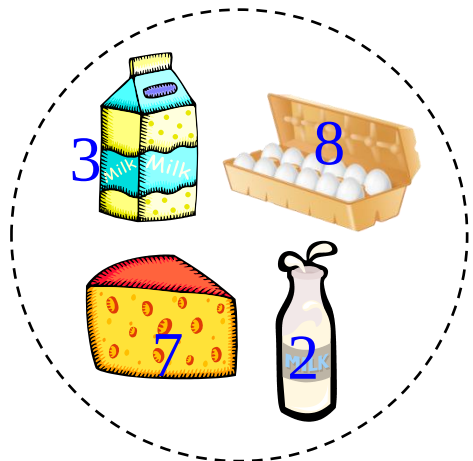
Linear Data Structures

- Data structures support the following operations on a set *S* of items
  - Insert an item into *S*
  - Remove an item from *S*
  - Search for an item in *S*
- What is a linear data structure?
  - Data items are stored sequentially in a linear data structure
  - E.g., stack, queue, linked list

Stack (*Today*)
Queue (*Today*)
Linked List (*Today*)
Tree (*Lecture 3*)
Heap (*Lecture 4*)
……

search key

set



Operation	Stack ***S***	Queue ***Q***	Linked List ***L***
Insert an item	push(*e*)	enqueue(*e*)	insert(*x*)
Remove an item	pop( )	dequeue( )	remove(*x*)
Search for a key	N/A	N/A	search(*k*)

fixed deletion order

- We are going to study the **properties** and the **operations** of stack, queue, and linked list
- Operations with O(1) running time
  - All operations in stack and queue, insert(*x*), remove(*x*) in linked list
- Operations with O(*n*) running time
  - search(*k*) in linked list

n denotes the number of data items stored

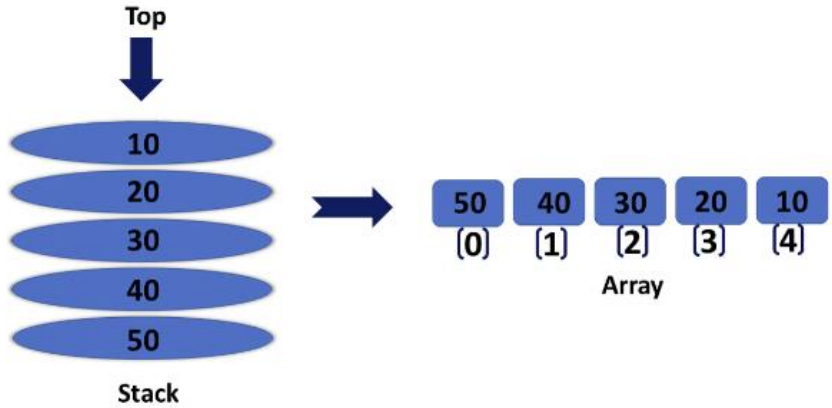
Outline

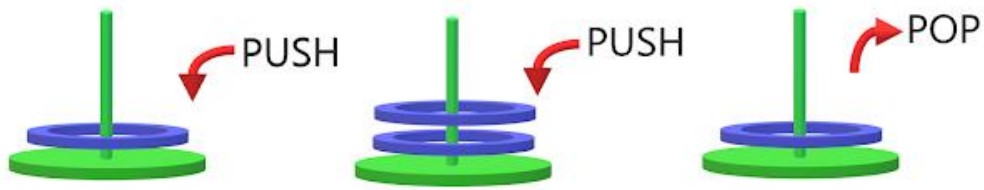
- Stacks
- Queues
- Linked lists
- Variants of linked lists

Stack

Stack: a pile of things arranged one on top of another

https://dictionary.cambridge.org/





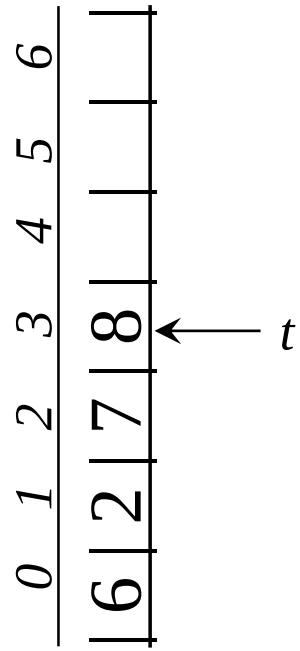
Simple Representation of a Stack with PUSH and POP operations

Built-in Stack in Java

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Stack.html

Stack

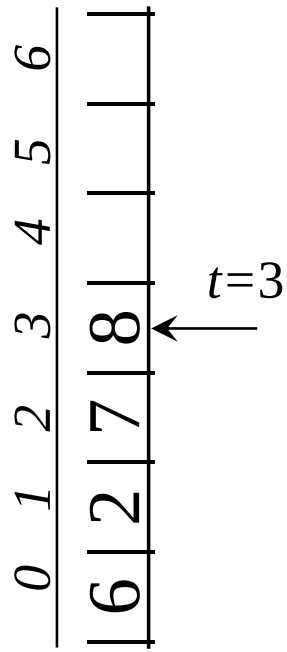
- The last-in first-out (LIFO) property
  - Only the last inserted object can be removed
    - E.g., insert 6, insert 2, insert 7, insert 8, remove → return 8



- Applications
  - Syntax parsing (e.g., for XML, HTML)
  - Searching in puzzle problems
  - Memory management (for procedure call) during program execution

Stack: Array Implementation

- The last-in first-out (LIFO) property
- Instance variables
  - *S*: an array of items
  - *S.length*: the length of the array *S*
  - *t*: the position of the top item
- Operations
  - **push**(*e*): insert item *e* (at the top of the stack)
  - **pop**( ): extract the top item
  - size( ): return the number of items stored
  - isEmpty( ): check whether the stack is empty
  - top( ): return the top item without removing it



Stack: Auxiliary Operations

- isEmpty( ): check whether the stack is empty
- size( ): return the number of items stored
- top( ): return the top item without removing it

size ( )

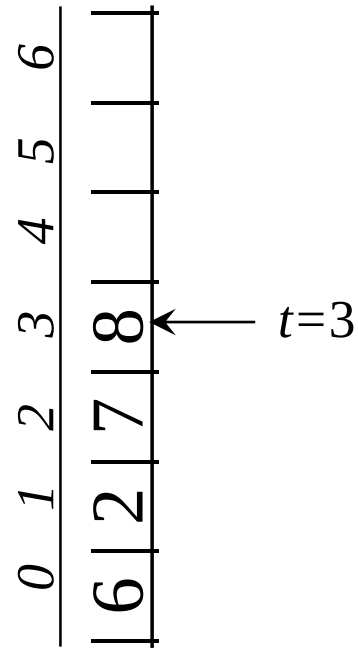
1. return $t+1$

isEmpty ( )

1. return ($t<0$)

top ( )

1. if isEmpty( )
2. throw an exception
3. return $S[t]$

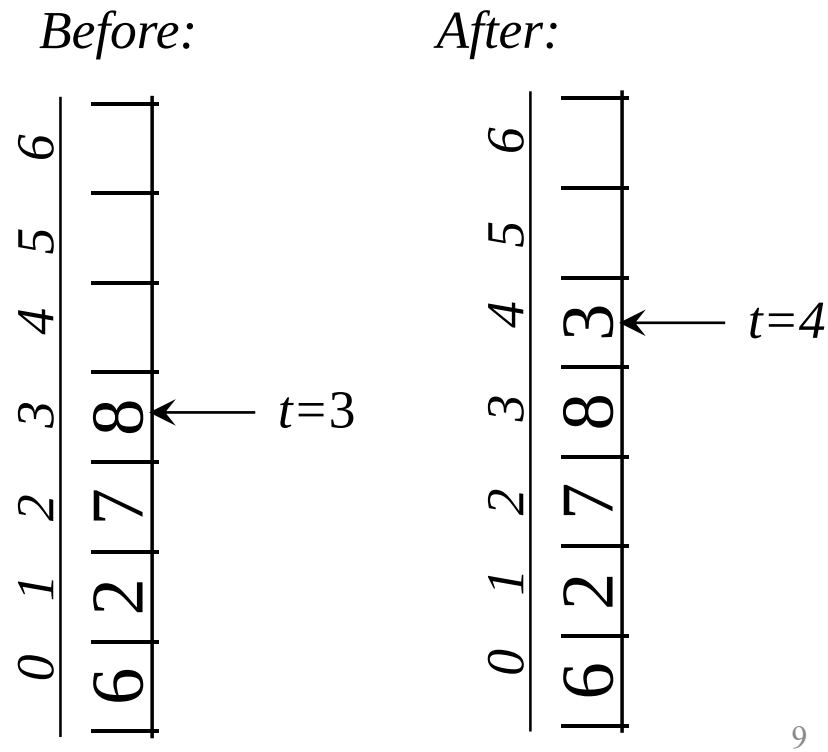


Stack: push(e)

- **push**(*e*): insert *e* to the top of the stack
  - Example: push(3)

push (*e*)

1. if size( ) = *S.length*
2. throw an exception
3. $t \leftarrow t + 1$
4. $S[t] \leftarrow e$

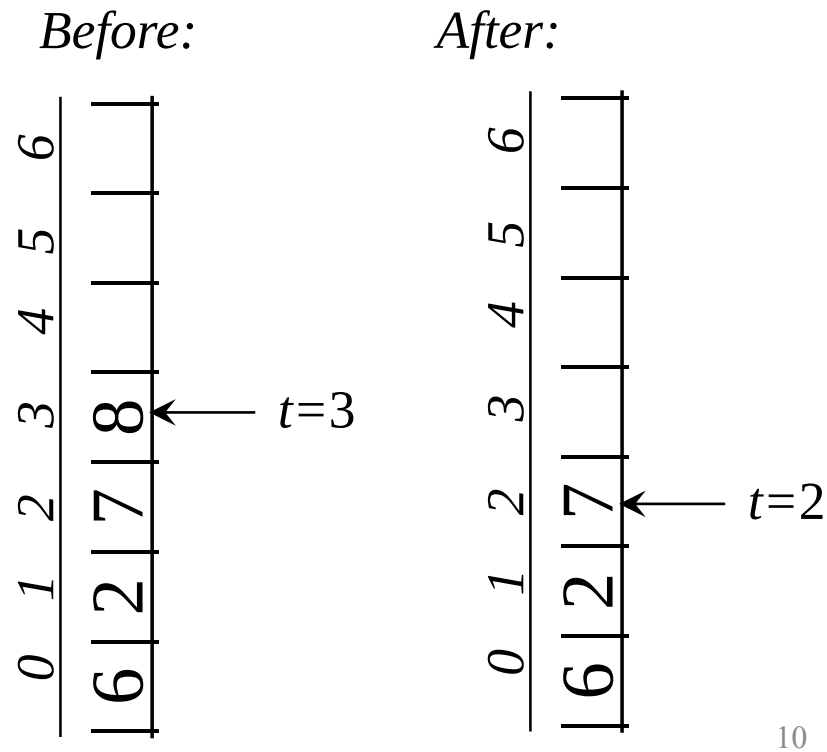


Stack: pop( )

- **pop**( ): extract the top item
  - Example: pop( ) returns 8

pop ( )

1. if isEmpty( )
2. throw an exception
3. *temp* ← *S*[*t*]
4. *S*[*t*] ← null
5. *t* ← *t* – 1
6. return *temp*



Stack: Java Code

```
public class ArrayStack<E> {
  private E[ ] S;
  private int t = -1;
  public ArrayStack(int max_size) {
    ...
  }
  public int size() {
    ...
   }
  public boolean isEmpty() {
    ...
  }
  public E top() throws Exception {
    ...
  }
  public void push(E e) throws Exception {
    ...
  }
  public E pop() throws Exception {
    ...
  }
}
```

- The generic type <E>
  - <E> can be provided at run time
  - E.g., create a stack of integers by `new ArrayStack<int>(10);`
- Class ArrayStack<E>: defines the type of ArrayStack<E> object
  - Instance variables
  - Constructor for creating an object
  - Methods
- *Remark*: This version is simpler than the code in the textbook
  - We use one `class` and one type of `Exception` only

Examples of using a stack

- Reverse an array by using a stack
  - Test case: `[6,2,7,8]` → `[8,7,2,6]`
- Match parentheses by using a stack
  - Test case #1: `[](())` → `true`
  - Test case #2: `(.` → `false`
- Source codes
  - Download at *https://bcs.wiley.com/he-bcs/Books?action=resource&bcsId=8635&itemId=1118771338&resourceId=35121*
  - ReverseWithStack.java, MatchDelimiters.java

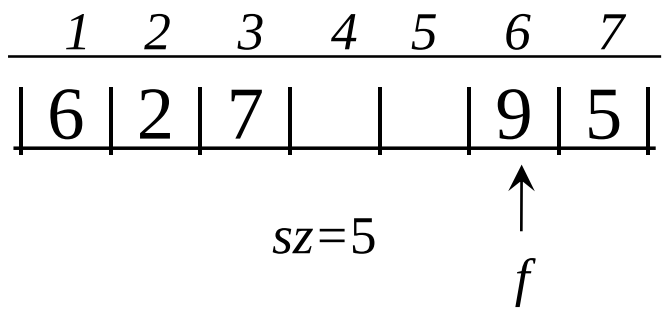
Outline

- Stacks
- Queues
- Linked lists
- Variants of linked lists

Queue

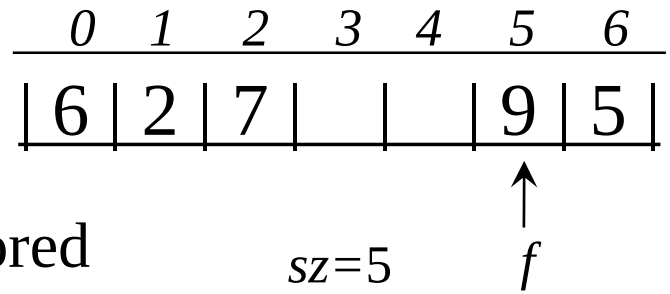
- The first-in first-out (FIFO) property
  - Only the first inserted object can be removed
    - E.g., insert 9, insert 5, insert 6, insert 2, insert 7
      remove → return 9

- Applications
  - CPU scheduling
  - Disk scheduling
  - Playlist in media player



Queue: Array Implementation

- The first-in first-out (FIFO) property
- Instance variables
  - *Q*: an array of items
  - *Q.length*: length of array *Q*
  - *f*: the front position
  - *sz*: the number of items stored



- Operations
  - **enqueue**(*e*): insert item *e* (at the rear of the queue)
  - **dequeue**( ): extract the item from the front
  - size( ): return the number of items stored
  - isEmpty( ): check whether the queue is empty
  - top( ): return the front item without removing it

Queue: Auxiliary Operations

- Use `Q` as a circular array by using the modulo operation
  - Modulo means the reminder of integer division
- The rear position is `r = (f+sz) mod Q.length`
  - E.g., in this example, the items are stored at 5, 6, 0, 1, 2

size ( )

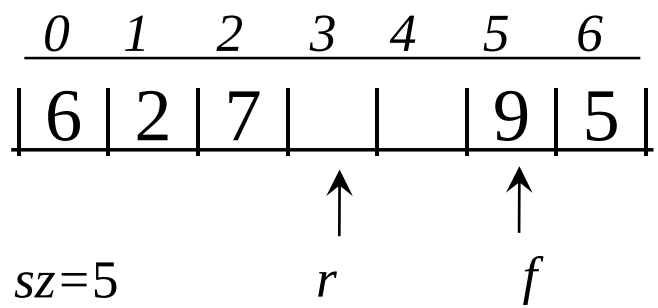
1. return *sz*

isEmpty ( )

1. return (*sz* = 0)

front ( )

1. if isEmpty ( )
2. throw an exception
3. return *Q*[*f*]



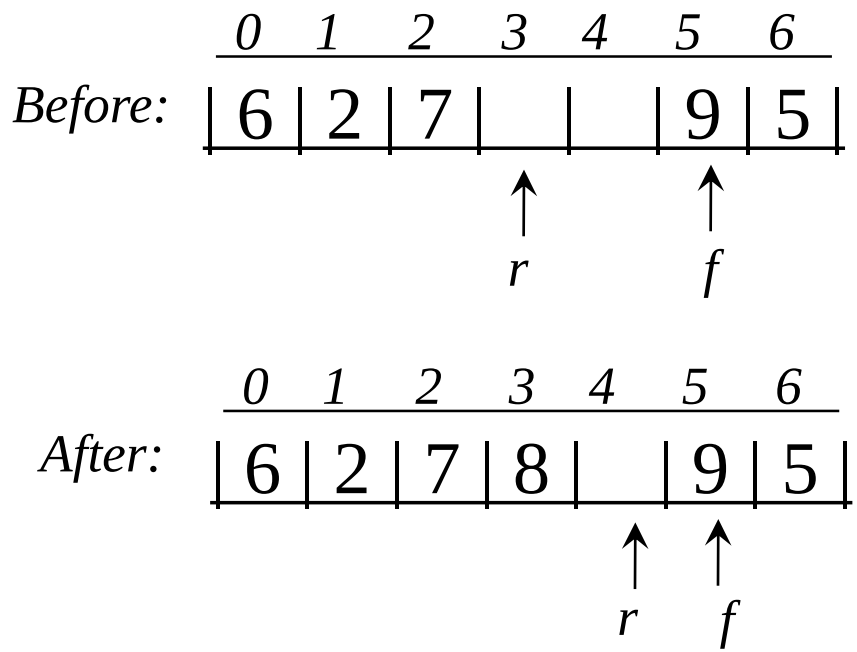
Queue: enqueue(*e*)

- **enqueue**(*e*): insert item *e* (at the rear of the queue)
  - Example: enqueue(8)

What if size( ) = *Q.length* ?

enqueue (*e*)

1. if size( ) = *Q.length* – 1
2. throw an exception
3. else
4. $r \leftarrow (f + sz) \bmod Q.length$
5. $Q[r] \leftarrow e$
6. $sz \leftarrow sz + 1$



Version 1 - slide

isEmpty ( )

1. return (*sz* = 0)

enqueue (*e*)

1. if size( ) = *Q.length* – 1
2. throw an exception
3. else
4. $r \leftarrow (f + sz) \bmod Q.length$
5. $Q[r] \leftarrow e$
6. $sz \leftarrow sz + 1$

Let N = Q.length.
If size() = *N*, then f = r.
If f = r, the queue is full whilst isEmpty is determined by sz=0, but not f =r.
So the queue is full when size()= sz = N.
Both N or (N -1) is correct, but size() = N-1 has one unused unit in the queue.

Version 2 - textbook

Algorithm size():
  return $(N - f + r) \bmod N$

Algorithm isEmpty():
  return $(f = r)$

Algorithm front():
  if isEmpty() **then**
    throw a QueueEmptyException
  return $Q[f]$

Algorithm dequeue():
  if isEmpty() **then**
    throw a QueueEmptyException
  $temp \leftarrow Q[f]$
  $Q[f] \leftarrow$ **null**
  $f \leftarrow (f + 1) \bmod N$
  return *temp*

Algorithm enqueue(e):
  if size() $= N - 1$ **then**
    throw a FullQueueException
  $Q[r] \leftarrow e$
  $r \leftarrow (r + 1) \bmod N$

If size() = N, then f = r.
If f = r, we do not know whether the queue is empty or full.
When size() = N – 1, the queue has one unused unit, and f = r never happens.

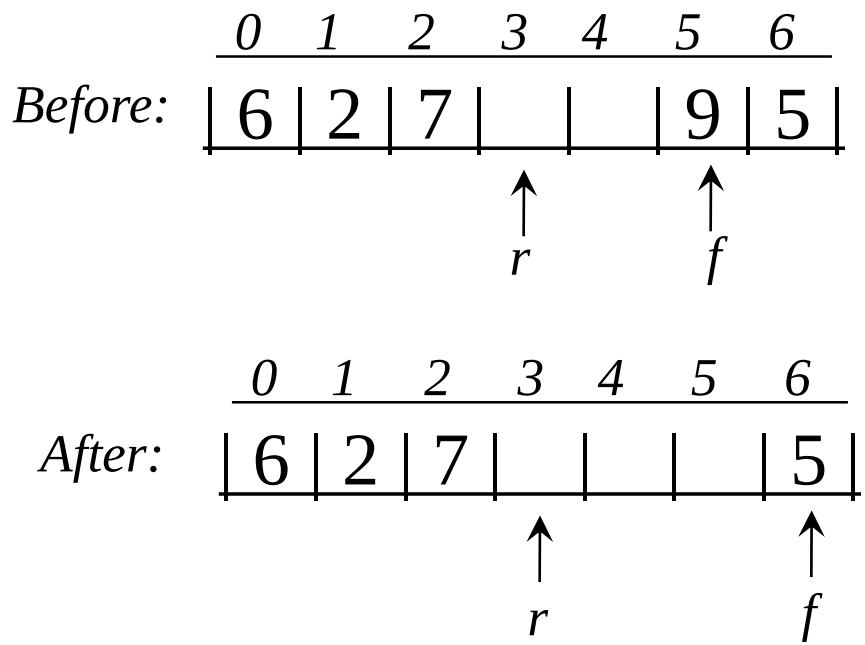
The implementation above contains an important detail, which might be missed at first. Consider the situation that occurs if we enqueue N objects into Q without dequeuing any of them. We would have $f = r$, which is the same condition that occurs when the queue is empty. Hence, we would not be able to tell the difference between a full queue and an empty one in this case. Fortunately, this is not a big problem, and a number of ways for dealing with it exist.

Queue: dequeue( )

- **dequeue**( ): extract the item from the front
  - Example: dequeue( ) returns 9

dequeue ( )

1. if isEmpty( )
2. throw an exception
3. else
4. *temp* ← *Q*[*f*]
5. *Q*[*f*] ← null
6. *f* ← (*f* + 1) mod *Q.length*
7. *sz* ← *sz* – 1
8. return *temp*



Queue: Java Code

```
public class ArrayQueue<E> {
  private E[ ] Q;
  private int f=0;
  private int sz=0;
  public ArrayQueue(int max_size) {
    ...
  }
  public int size() {
    ...
   }
  public boolean isEmpty() {
    ...
  }
  public E front() throws Exception {
    ...
  }
  public void enqueue(E e) throws Exception {
    ...
  }
  public E dequeue() throws Exception {
    ...
  }
}
```

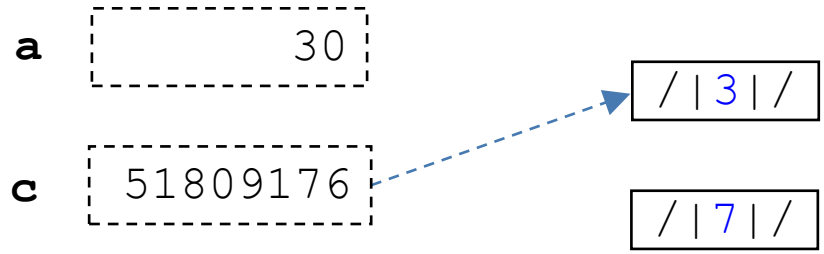
- Class `ArrayQueue<E>`: defines the type of `ArrayQueue<E>` object
  - Instance variables
  - Constructor for creating an object
  - Methods
- *Remark*: This version is simpler than the code in the textbook
  - We use one `class` and one type of `Exception` only

Outline

- Stacks
- Queues
- Linked lists
- Variants of linked lists

Java: primitive type vs. reference type

```
int a = 30;
Node c = x_node;
a = 50;
c = y_node;
```

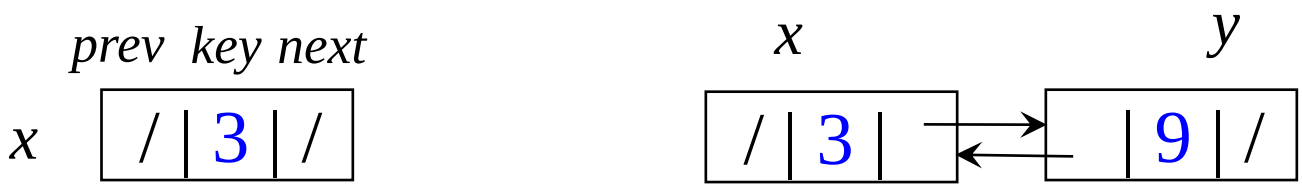


- Primitive-type variable
  - Types: `byte, short, int, long, float, double, char, boolean`
  - A variable stores a *value*
    - E.g., `a` stores the integer value `30`
- Reference-type variable
  - Types that are not primitive (e.g., `Object, String, Node`)
  - A variable stores an *address* of a data object
    - E.g., `c` stores the address of a `Node` object
    - It can also store the **`null`** value, meaning invalid reference
  - Use the dot notation to access an instance variable (e.g., `c.key`)

[Doubly] Linked List: Structure

- In a [doubly] linked list, each node *x* stores:
  - *x.key*: key value of the node
  - *x.prev*: reference to the previous node
  - *x.next*: reference to the next node

- A reference can be set to:
  - The memory address of another node, OR
  - `null` value (shown as '/' in our figures)

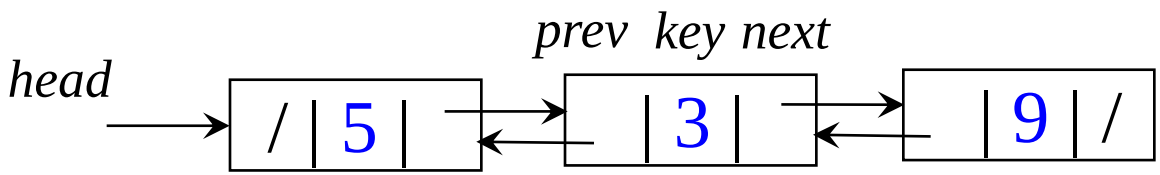


[Doubly] Linked List: Structure

- Linked list *L* is a chain of nodes (see the previous page)
  - ***head*** is the first node in the linked list
  - First node: *prev* = `null`; Last node: *next* = `null`
  - Consecutive nodes *x* & *y* should satisfy:

    $x.next = y$ and $y.prev = x$

- Operations
  - search(*k*): search a node with key value *k* — O(*n*) time
  - insert(*x*): insert a node into *L* — O(1) time
  - remove(*x*): remove an (existing) node from *L* — O(1) time



[Doubly] Linked List Structure: Java Code

Remark: To make the code simple, we just store an integer key in a node and don't use get/set methods

```
class Node {
  int key;
  Node prev;
  Node next;
}
```

A linked list just stores the head

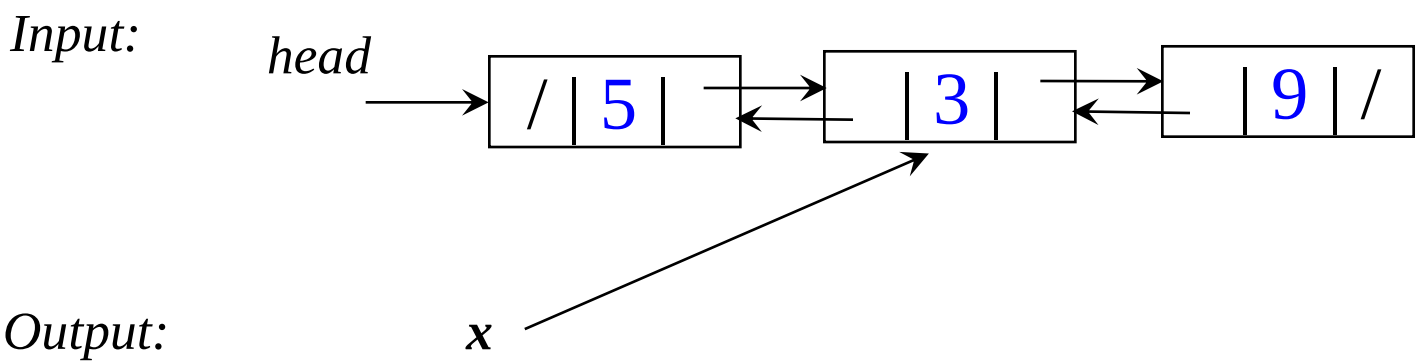
```
class LinkedList {
  Node head;
  LinkedList() {
       head=null;
  }
  Node search(int k) {
    ...
  }
  void insert(Node x) {
    ...
  }
  void remove(Node x) {
    ...
  }
}
```

[Doubly] Linked List: search(k)

- search(*k*): search a node with key value *k*
  - Example: search(3)

search (*k*)

1. *x* ← *head*
2. while *x* ≠ `null` and *x.key* ≠ *k*
3. *x* ← *x.next*
4. return *x*

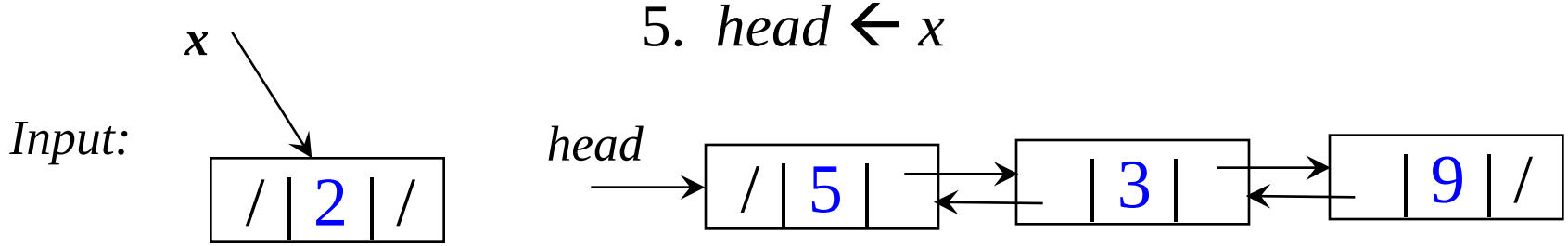


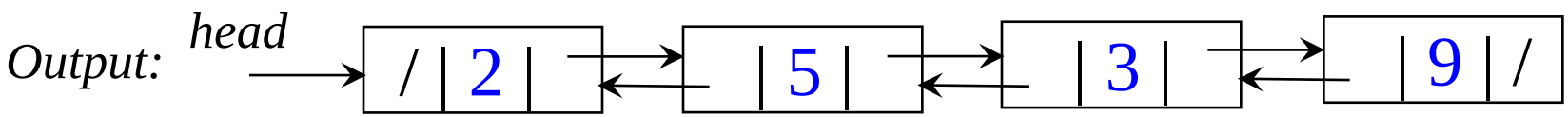
[Doubly] Linked List: insert(x)

- insert(*x*): insert a node into a linked list

insert (*x*)

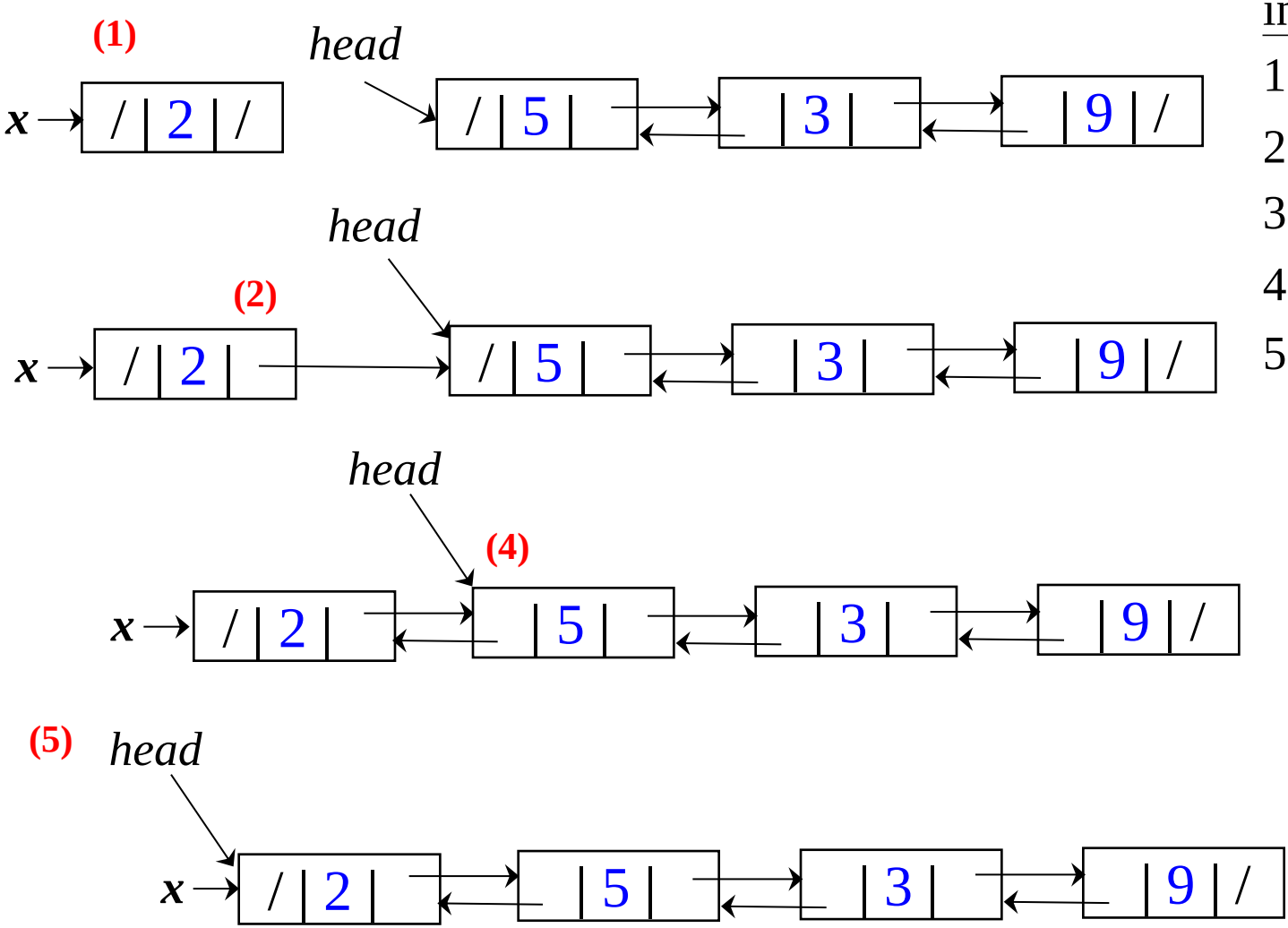
1. *x.prev* ← `null`
2. *x.next* ← *head*
3. if *head* ≠ `null`
4. *head.prev* ← *x*
5. *head* ← *x*





[Doubly] Linked List: insert(x)

- Example



insert (*x*)

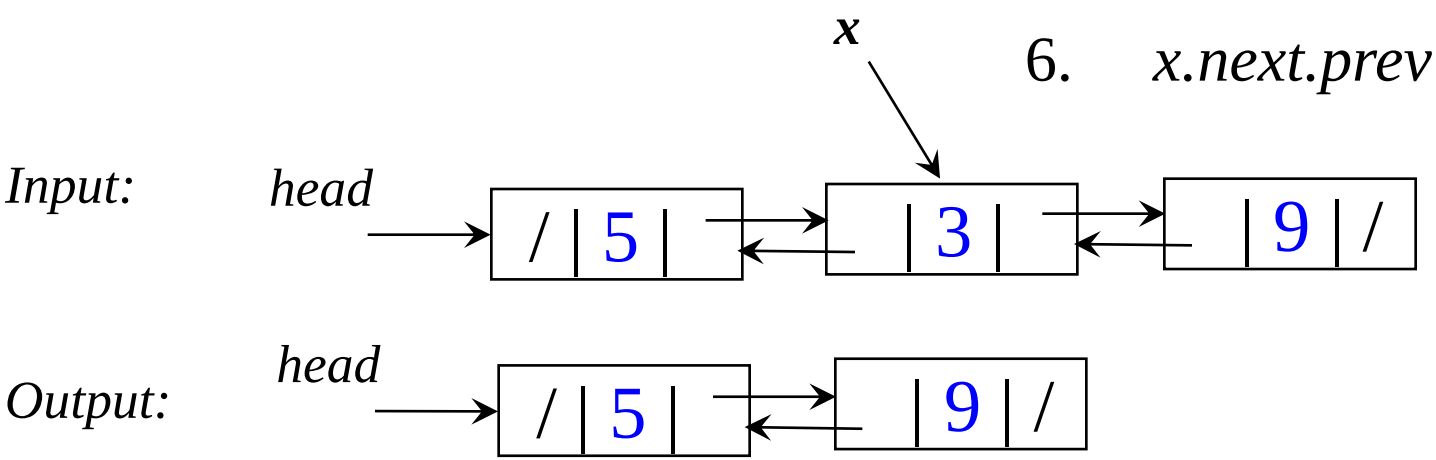
1. *x.prev* ← `null`
2. *x.next* ← *head*
3. if *head* ≠ `null`
4. *head.prev* ← *x*
5. *head* ← *x*

[Doubly] Linked List: remove(x)

- remove(*x*): remove an (existing) item from a linked list
  - *x* must be an existing node in *L*
  - If we wish to remove a node with the key *k*, we must search it first

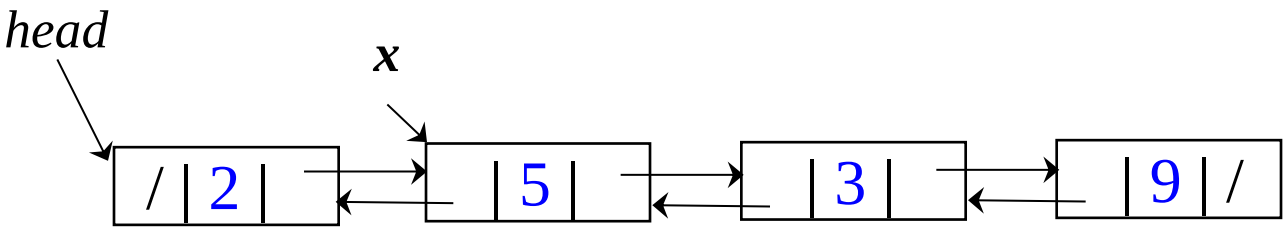
remove (*x*)

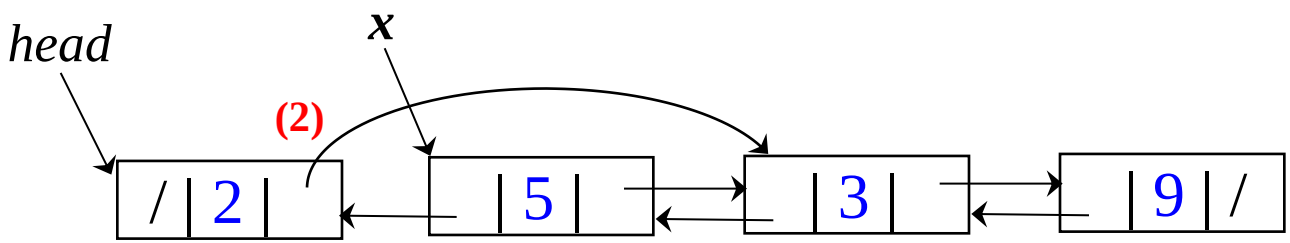
1. if *x.prev* ≠ `null`
2. *x.prev.next* ← *x.next*
3. else
4. *head* ← *x.next*
5. if *x.next* ≠ `null`
6. *x.next.prev* ← *x.prev*

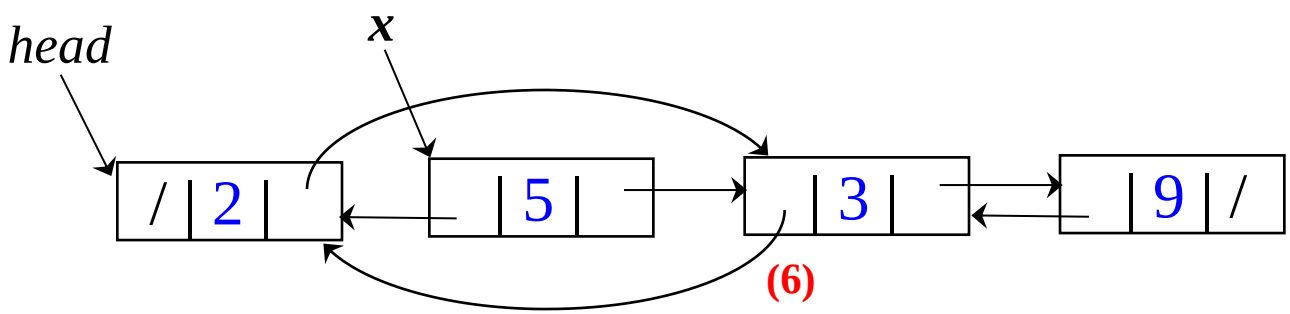


[Doubly] Linked List: remove(x)

- Case I: *x* is not the head of the linked list





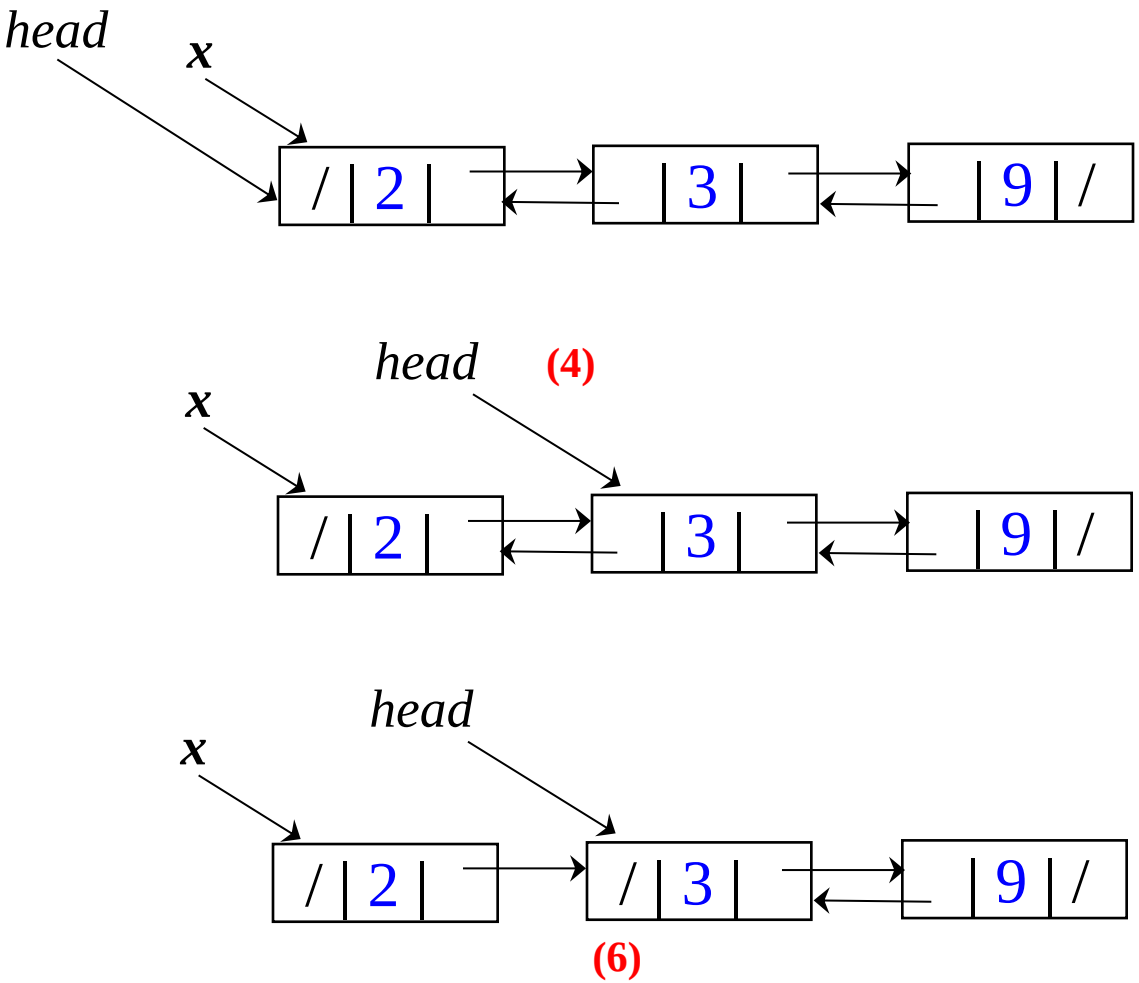


remove (*x*)

1. if *x.prev* ≠ `null`
2. *x.prev.next* ← *x.next*
3. else
4. *head* ← *x.next*
5. if *x.next* ≠ `null`
6. *x.next.prev* ← *x.prev*

[Doubly] Linked List: remove(x)

- Case II: *x* is the head of the linked list



remove (*x*)

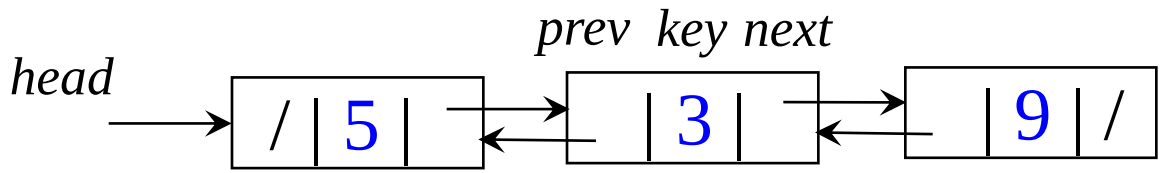
1. if *x.prev* ≠ `null`
2. *x.prev.next* ← *x.next*
3. else
4. *head* ← *x.next*
5. if *x.next* ≠ `null`
6. *x.next.prev* ← *x.prev*

Outline

- Stacks
- Queues
- Linked lists
- Variants of linked lists

Variants: Singly Linked List

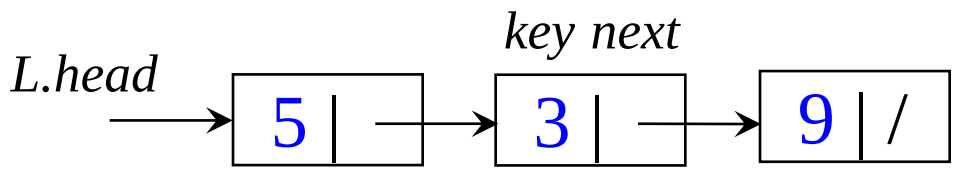
Doubly linked list:



- In a doubly linked list, each node stores *key*, *prev*, *next*
  - This is the type of linked list used in previous slides
  - Require more space ☹
  - Simple algorithm for deleting a node ☺
- In a singly linked list, each node stores *key*, *next* only
  - Require less space ☺
  - More complicated algorithm for deletion ☹

  [Question] How do we remove the node "3" in this linked list?

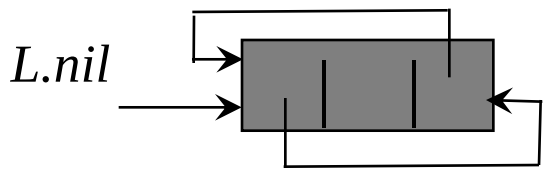
Singly linked list:



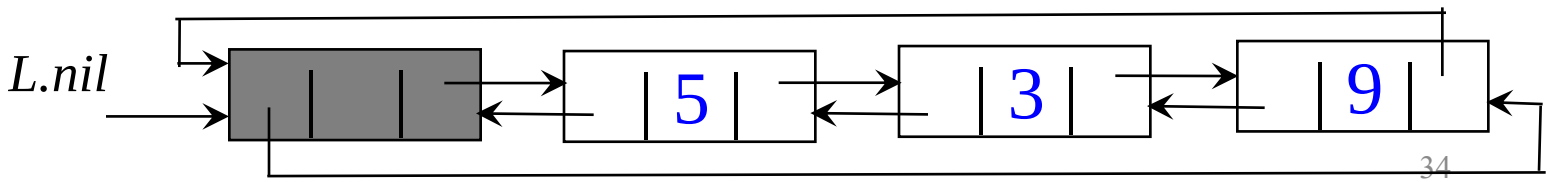
Variants: Linked List with Dummy Node

- How to simplify the operations on linked lists?
- Use a circular linked list, with a dummy node
  - Replace the `null` value by a **dummy** item *L.nil*
  - The "next" of *L.nil* is the first node (i.e., *L.head* is *L.nil.next*)
  - The "prev" of *L.nil* is the last node
- Simpler algorithms can be used on such linked list
  - E.g., *L.nil.next* is always not `null` (no need to check `null`)

Empty linked list:



Linked list with 3 items:



Variants: Array as Linked List

- We can also implement a linked list by using an array
  - Use an array $A[0..n-1]$ of items; each $A[i]$ is an item
  - $A[i].prev$, $A[i].next$ store array positions instead of reference values (use –1 to represent `null`)
- *head_pos*: the head position of the linked list (of used items)
- *free_pos*: the head position of the free list (of unused items)
  - When we *create* an item, just extract it from the free list
- Consider the following example:
  - *head_pos* = 0 → positions of used items: 0, 3, 5
  - *free_pos* = 1 → positions of unused items: 1, 2, 4, 6

head_pos = 0

free_pos = 1

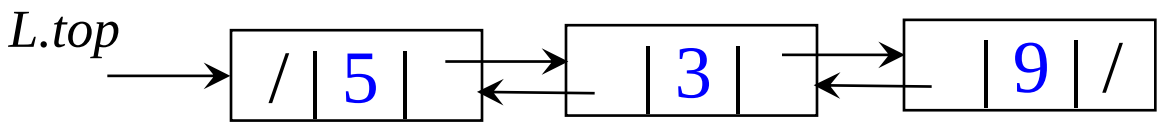
	0	1	2	3	4	5	6
key:	5			3		9	
prev:	-1	-1	1	0	2	3	4
next:	3	2	4	5	6	-1	-1

Questions to you

- How to implement a **stack** by a linked list?

  How about the stack operations (push, pop)

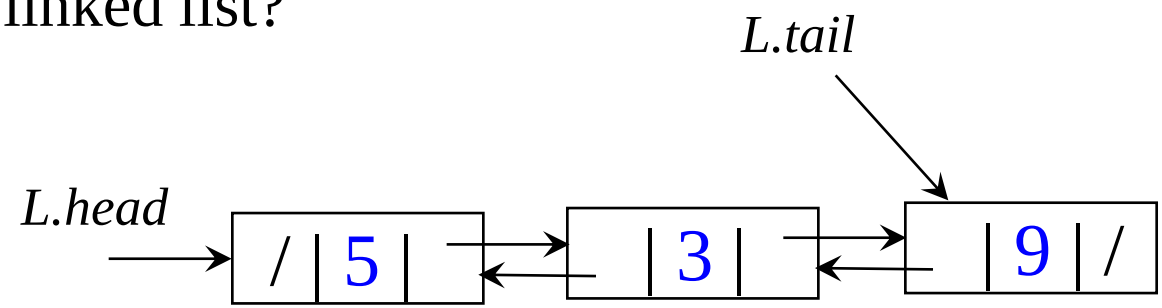
  on this linked list?



- How to implement a **queue** by a linked list?

  How about the queue operations (enqueue, dequeue)

  on this linked list?



Summary

- Stack: LIFO property, push, pop
- Queue: FIFO property, enqueue, dequeue
- Linked list and its operations

- Please read Chapters 3 and 6 in the book "*Data Structures and Algorithms in Java*"