GEORGE MASON UNIVERSITY

CS 310 – Fall 2025
Data Structures –L06
Linked List

Archange G. Destiné
adestine@gmu.edu

Outline for Today (W4 – Lec 06)

1. **Linked List**
2. **Today:**
   - **Review: Participation Activity Questions**
   - **Stack and Queue, Introduction**
3. **Notes:**
   - **Project 1 (Deadline is Sept 19)**

Midterm Exam: Week 8 – Wednesday Oct 15^{th} (See Course Schedule)

Questions from Last Week Topics…

Linked List

List

We discussed Dynamic Array and used an array as underlying structure.

- **advantages**
- **limitations with this way of storing the data**

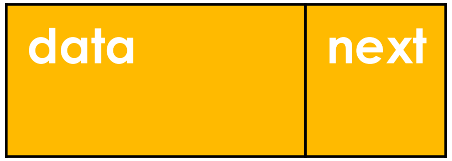
Linked List

Simply a collection of components called nodes

where "Every" node contains the address of the next node.

So, a node will have 2 fields:

- 1 field to store the relevant information
- 1 field to store the address of the next node



How do we know the address of the first node?

Linked List

data: can be a value of a primitive type or reference to an object.

next: can only be a reference to an object.

To have each node storing 2 fields: data and next,
We need to create a class:
for instance LinkedListNode, or just MyNode…

Linked List


```
public class Node {
    public int data;
    public ______ next;
}
```

What should be the "type" of next?

Should the class or the fields be public? Maybe Node should be an inner class.

Linked List

A Node (implementation… example)

```
class ListNode
{
    Object   data;
    ListNode next;
}
```

Important Java Memory Review

`Node a = new Node(…)`

How is this executed?

What is the value of `a`?

Advantages of Linked List

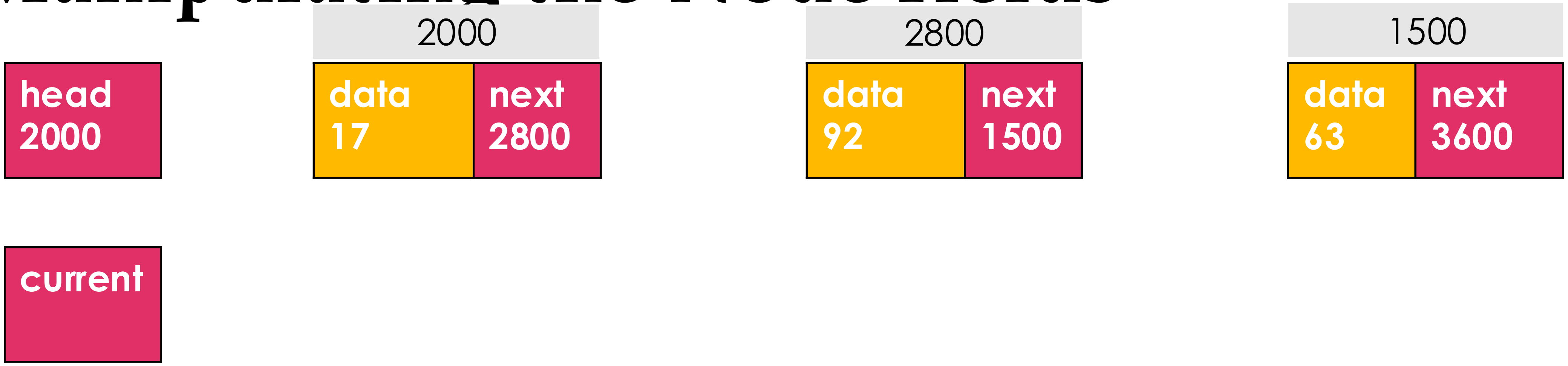
- Uses memory only as needed
- When entry removed, unneeded memory returned to system
- Avoids moving data when adding or removing entries

Linked List

Some properties of linked lists:

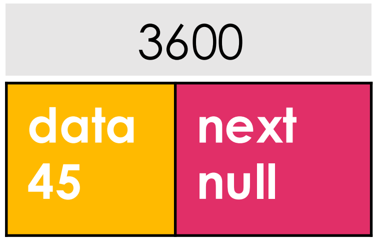
- The address of the first node is stored in <u>head</u>.
- Each node has 2 fields* (data and next)
- If next is null, this means that next is pointing to nothing (indicating the last node)

Manipulating the Node fields

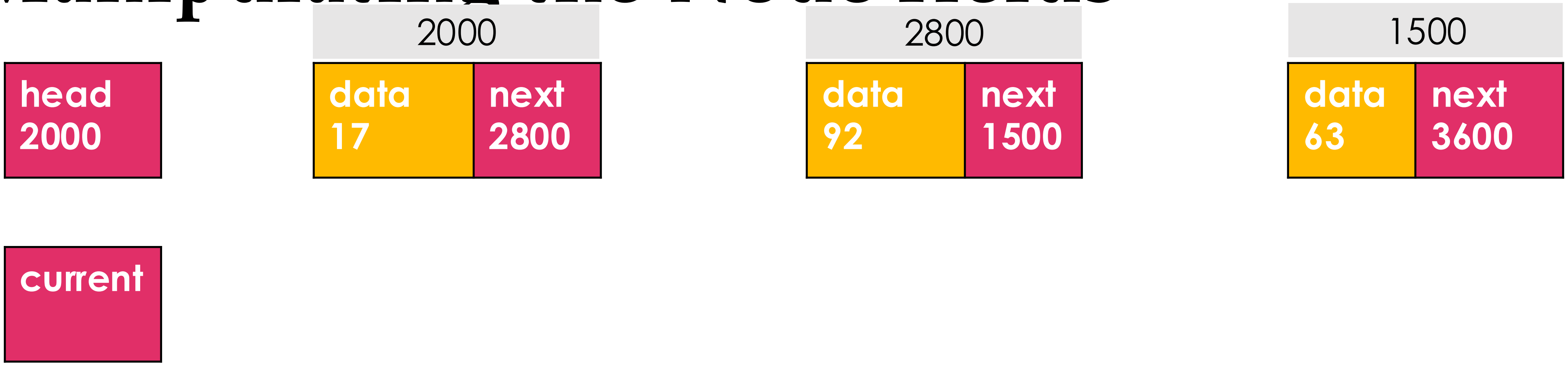


What happens after:

```
current = head
```



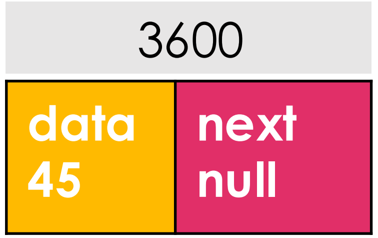
Manipulating the Node fields



After: `current = head`

Let's fill this table:

variable, expression	value
current	
current.data	
current.next	
current.next.data	



Manipulating the Node fields

head
2000

2000
data 17 | next 2800

2800
data 92 | next 1500

1500
data 63 | next 3600

current
2000

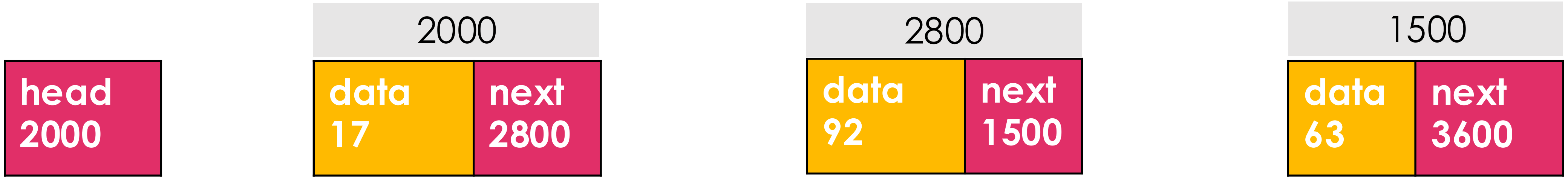
Now, we add: `current = current.next`

Let's fill this table:

variable, expression	value
current	
current.data	
current.next	
current.next.data	

3600
data 45 | next null

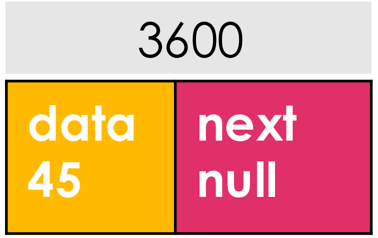
Traversing a list



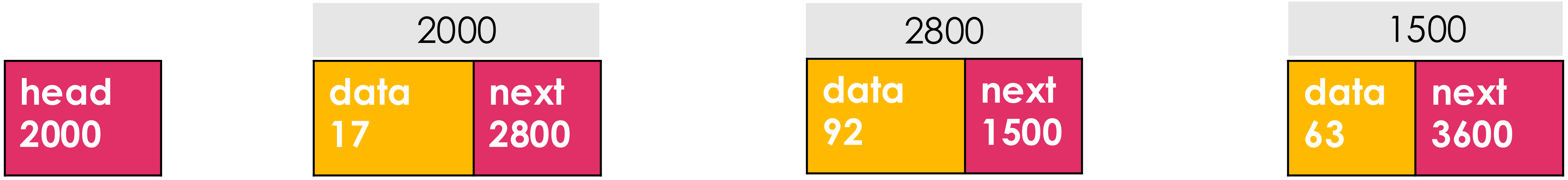
This operation is important.
You will need this in other basic operations such as:

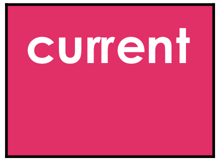
- Search an item
- Insert an item
- Delete an item

We cannot update the variable head to traverse the list. Why?
What would happen if we move head to the second node?

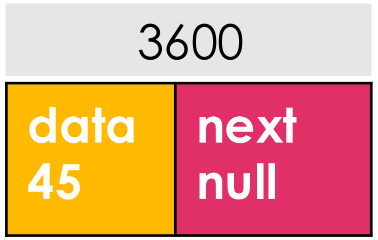


Traversing a list

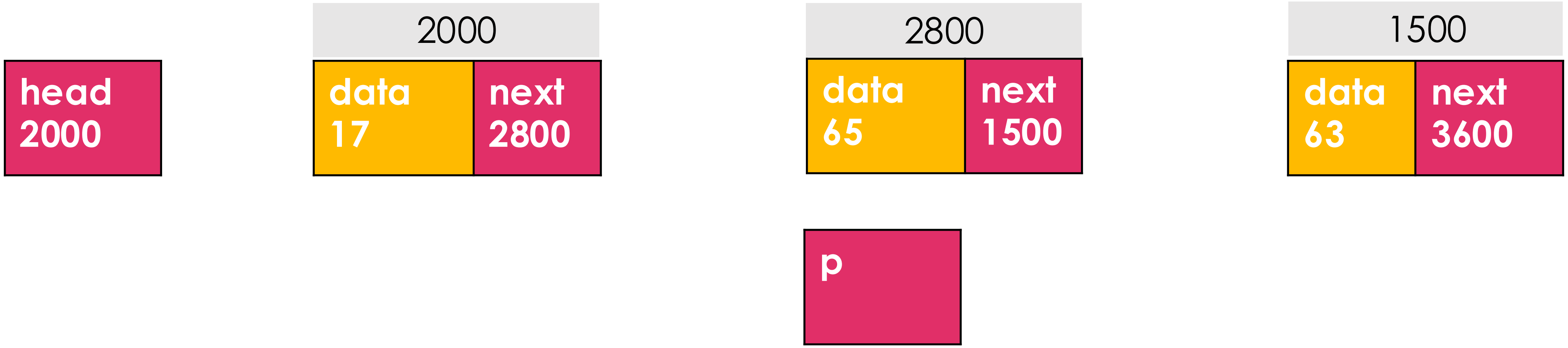





```
current = head;
while(current != null) {
      //Process current
      System.out.println(current.info+” ”);
      current = current.next;
}
```



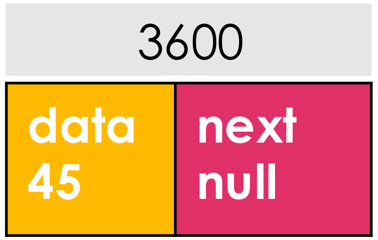
Insertion and Deletion of item



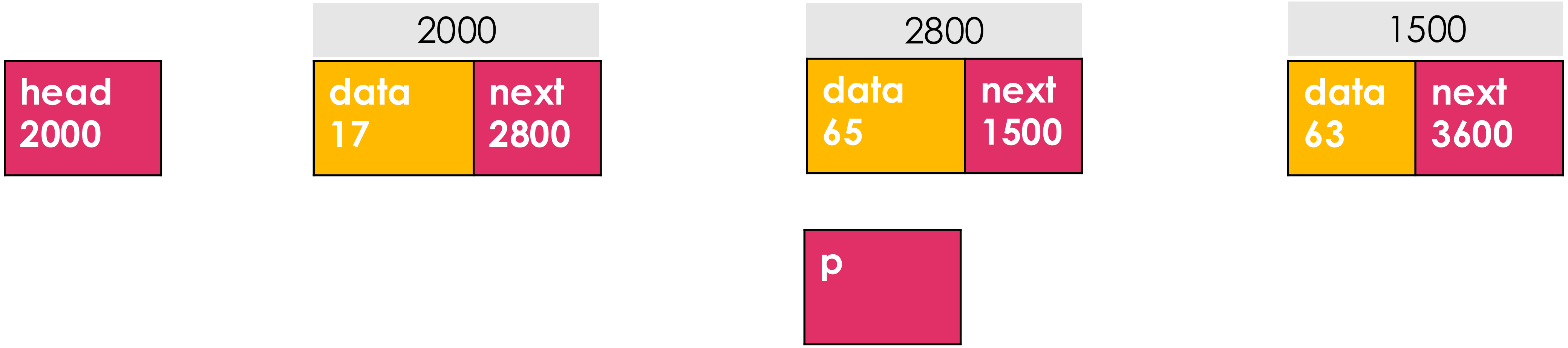
How to add a new node of data 50 after node with data 65 (p).

```
Node nodeInsert = new Node();
nodeInsert.data = 50;
nodeInsert.next = p.next;
p.next = nodeInsert;
```

What would happen when if we switch the last 2 lines?



Insertion and Deletion of item

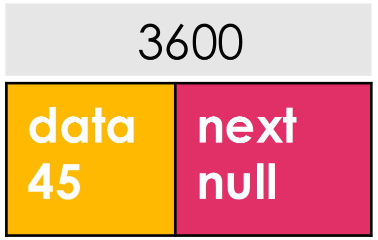


What if we want to delete node with data 63?
Write one line of code for this, considering that p is pointing to node with data 65.

Longer but clearer with an extra variable:

```
q = p.next
p.next = q.next
```

One more statement missing…:



Adding a New Node to the Front

```
class Node<T> {
      private T value;
      private Node<T> next;

      public Node(T value, Node<T> next) {
            this.value = value;
            this.next = next;
      }
```

```
AnyType item1 = new AnyType(100);
AnyType item2 = new AnyType(200);
AnyType item3 = new AnyType(300);

Node<AnyType> head = null;
head = new Node<>(item1, head);    // head -> 100 -> null
head = new Node<>(item2, head);   // head -> 200 -> 100 -> null
head = new Node<>(item3, head);  // head -> 300 -> 200 -> 100 -> null
```

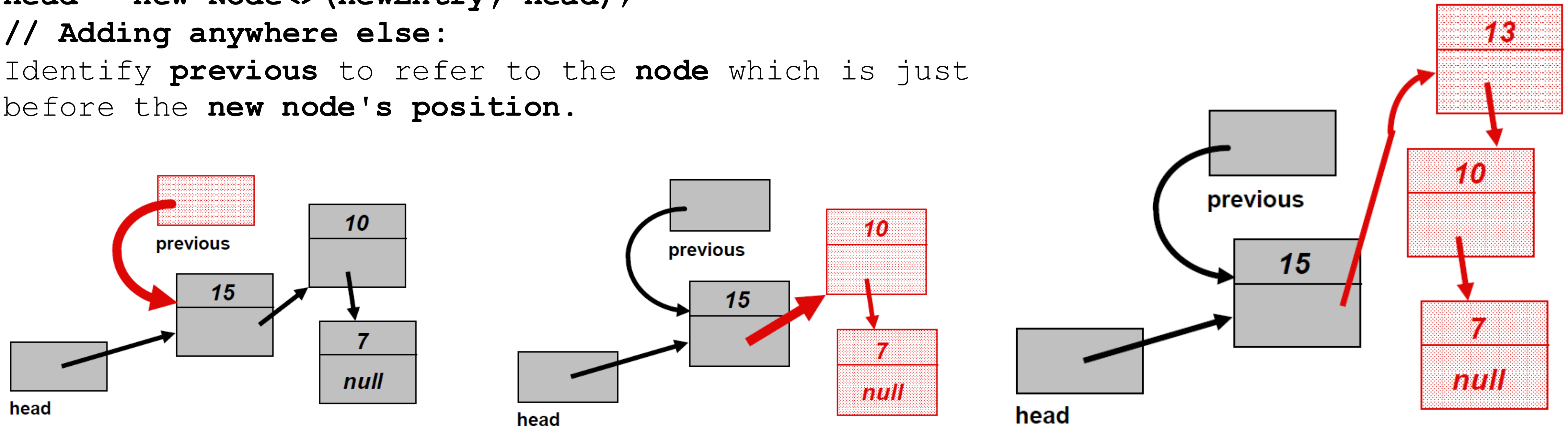
Adding a New Node Anywhere

```
class Node<T> {
      private T value;
      private Node<T> next;

      public Node(T value, Node<T> next) {
            this.value = value;
            this.next = next;
      }
```

```
// Case: Adding to the front:
head = new Node<>(newEntry, head);
// Adding anywhere else:
```

Identify **previous** to refer to the **node** which is just before the **new node's position.**

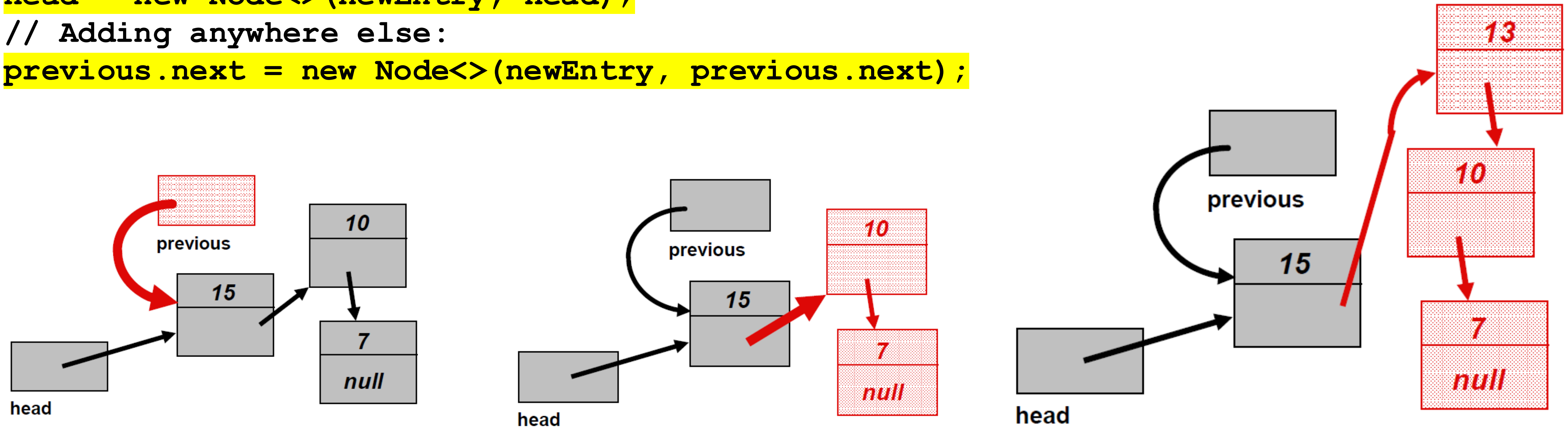


Adding a New Node Anywhere

```
class Node<T> {
       private T value;
       private Node<T> next;

       public Node(T value, Node<T> next) {
              this.value = value;
              this.next = next;
       }
```

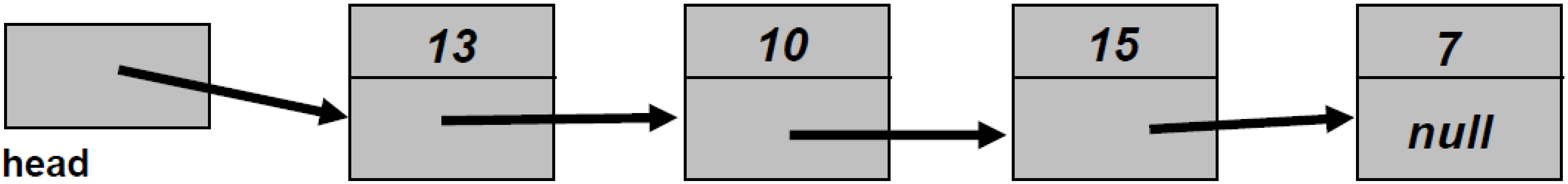
```
// Case: Adding to the front:
head = new Node<>(newEntry, head);
// Adding anywhere else:
previous.next = new Node<>(newEntry, previous.next);
```



Removing the head node…

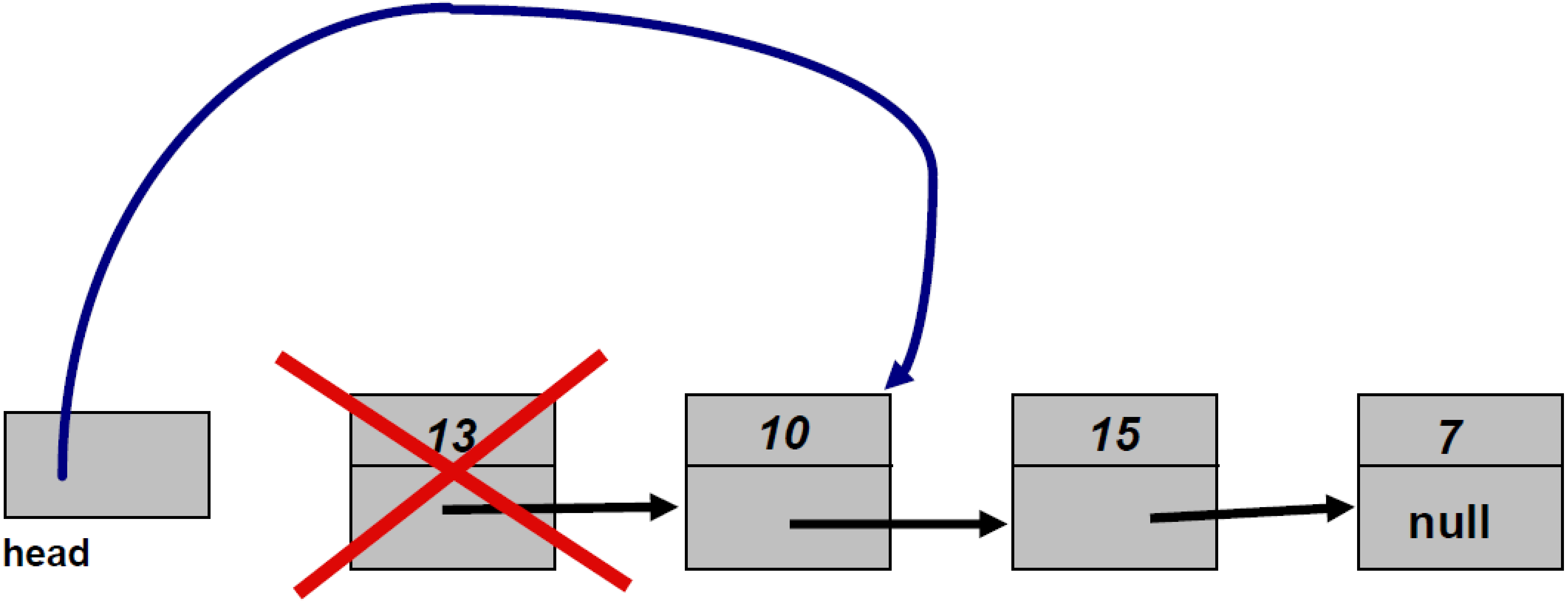
```
class Node<T> {
      private T value;
      private Node<T> next;

      public Node(T value, Node<T> next) {
            this.value = value;
            this.next = next;
      }
```



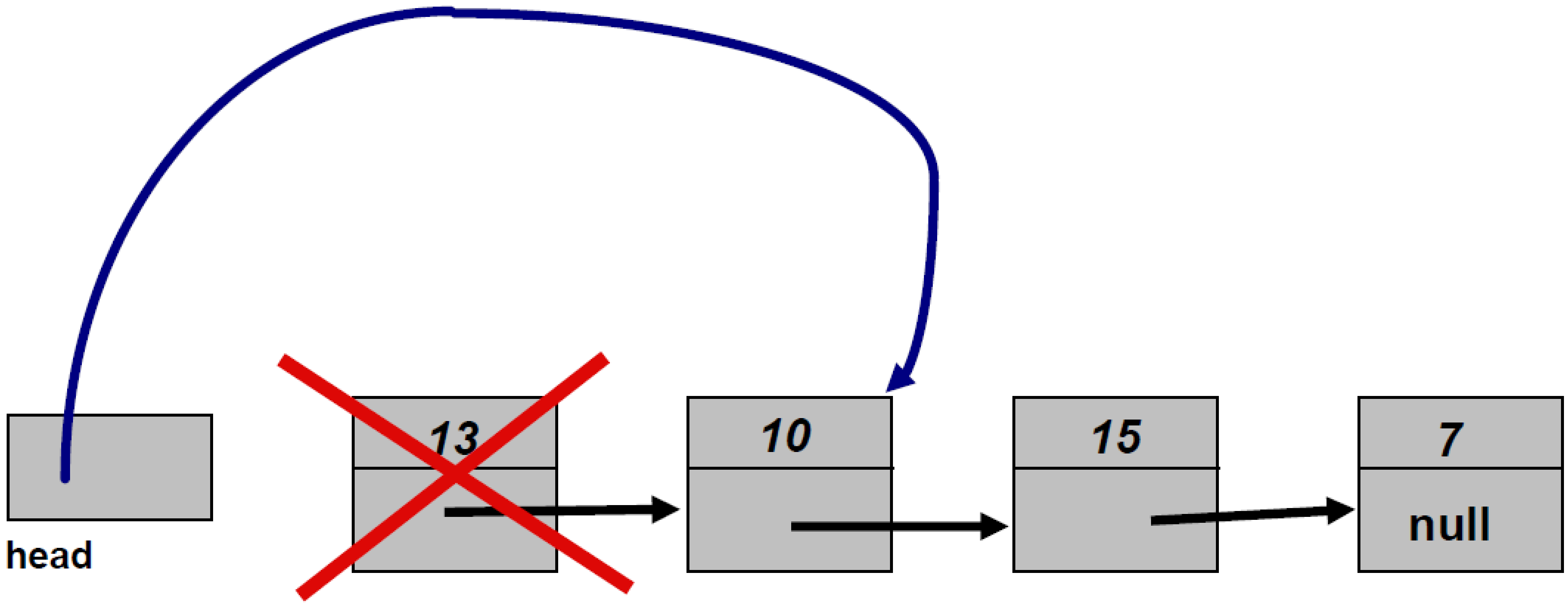
Removing the head node…

`head = ?`



Removing the head node...

```
head = head.next;
```



Linked List, the ADT

set()
- **parameters**: int index, Object to put there
- **return**: ??

–**get()**
- **parameters**: int index
- **return**: Object at index

–**append()**
- **parameters**: Object to add
- **return**: ??

add()
- **parameters**: int index, Object to add
- **return**: ??
- **Other types of add**... parameters: Object?

remove()
- **parameters**: int index
- **return**: ??
- **Other types of remove**... parameters: Object?

search()
- **parameters**: Object to find
- **return**:

Remember to check for edge cases!

Linked List Variants

Node Fields

Reference to **next** node ("singly")

Reference to **previous** and **next** node ("doubly")

List Fields

Keep reference to **head** node

Keep reference to **tail** node

Track **size**? Optional…

Linked List Variants Comparison

Remember: to insert and remove from the middle you first have to search for the correct position which is O(n).

Operation Implementation	get set	add remove (end)	insert remove (start)	insert remove (middle)	search	grow? shrink?
Singly Linked List						
Doubly Linked List						

Linked List Variants Comparison

Remove end not that simple… why?

Operation Implementation	get set	add remove (end)	insert remove (start)	insert remove (middle)	search	grow? shrink?
Singly Linked List	N					
Doubly Linked List	N					

Linked List Variants Comparison

Remember: to insert and remove from the middle you first have to search for the correct position which is O(n).

Operation Implementation	get set	add remove (end)	insert remove (start)	insert remove (middle)	search	grow? shrink?
Singly Linked List	N	1, N				
Doubly Linked List	N	1				

Linked List Variants Comparison

Remember: to insert and remove from the middle you first have to search for the correct position which is O(n).

Operation Implementation	get set	add remove (end)	insert remove (start)	insert remove (middle)	search	grow? shrink?
Singly Linked List	N	1, N	1			
Doubly Linked List	N	1	1			

Linked List Variants Comparison

Remember: to insert and remove from the middle you first have to search for the correct position which is O(n).

Operation Implementation	get set	add remove (end)	insert remove (start)	insert remove (middle)	search	grow? shrink?
Singly Linked List	N	1, N	1	N		
Doubly Linked List	N	1	1	N		

Linked List Variants Comparison

Remember: to insert and remove from the middle you first have to search for the correct position which is O(n).

Operation Implementation	get set	add remove (end)	insert remove (start)	insert remove (middle)	search	grow? shrink?
Singly Linked List	N	1, N	1	N	N	
Doubly Linked List	N	1	1	N	N	

Linked List Variants Comparison

Remember: to insert and remove from the middle you first have to search for the correct position which is O(n).

Operation Implementation	get set	add remove (end)	insert remove (start)	insert remove (middle)	search	grow? shrink?
Singly Linked List	N	1, N	1	N	N	Yes
Doubly Linked List	N	1	1	N	N	Yes

Linked List Variants Comparison

Remember: to insert and remove from the middle you first have to search for the correct position which is O(n).

Operation Implementation	get set	add remove (end)	insert remove (start)	insert remove (middle)	search	grow? shrink?
Singly Linked List	N	1, N	1	N	N	Yes
Doubly Linked List	N	1	1	N	N	Yes

Singly linked list **add** is constant time but **remove** requires searching down to node before last to set to null.
Doubly linked uses more memory, but still O(n)

Which Implementation is Best?

– Array / Static Array-“row” of memory

  – can run out of space

– Dynamic Arrays-arrays that can grow

  – cost to copy repeatedly (not so bad)

  – insert/remove expensive (not good at all -expensive)

– Linked Lists-tiny blocks of memory “linked” together

  – no “quick” memory access

  – extra memory to represent compared to array

  – fewer “expensive” memory moves

  - Can we improve **search?**

General Rule in Data Structures

- Arrays are simple
  - get/set anything
  - add/remove is obvious (need size variable)
  - very clear how data is laid out
- Just about every other data structure is less so
  - get/set **non trivial**
  - must preserve some internal structure -control access
  - element-by-element access takes work (time)

	ArrayList	LinkedList
add/remove at end	$O(1)$	$O(1)$
add/remove at front	$O(N)$	$O(1)$
get/set	$O(1)$	$O(N)$
contains	$O(N)$	$O(N)$

List Implementation Summary

– Though arrays are limited in functionality, constants for arrays are much faster

Operation Implementation	get set	add remove (end)	insert remove (start)	insert remove (middle)	search	grow? shrink?
Array	1	-	-	-	-	No
Static Array	1	1	N	N	N	No
Dynamic Array	1	1*	N	N	N	Yes
Singly Linked List	N	1,N	1	N	N	Yes
Doubly Linked List	N	1	1	N	N	Yes
**	1	-	-	-	1	Yes

* Amortized analysis (We discussed this already!)

** Hash Tables, will cover later

Search is important… But O(n)

How to improve **search** operation?

Operation Implementation	get set	add remove (end)	insert remove (start)	insert remove (middle)	search	grow? shrink?
Array	1	-	-	-	-	No
Static Array	1	1	N	N	N	No
Dynamic Array	1	1*	N	N	N	Yes
Singly Linked List	N	1,N	1	N	N	Yes
Doubly Linked List	N	1	1	N	N	Yes
**	1	-	-	-	1	Yes

* Amortized analysis (We discussed this already!)

** Hash Tables, will cover later

Why does search takes O(n) in List?

Improvement ideas?

Keep items sorted?

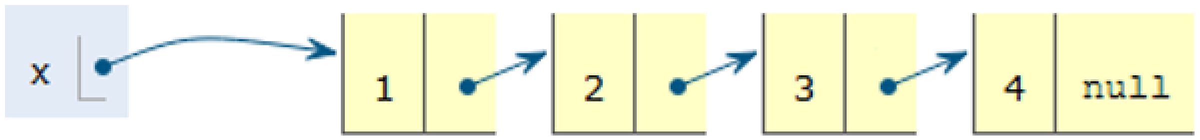
Does this work with Linked List?

Why does search takes O(n) in List?

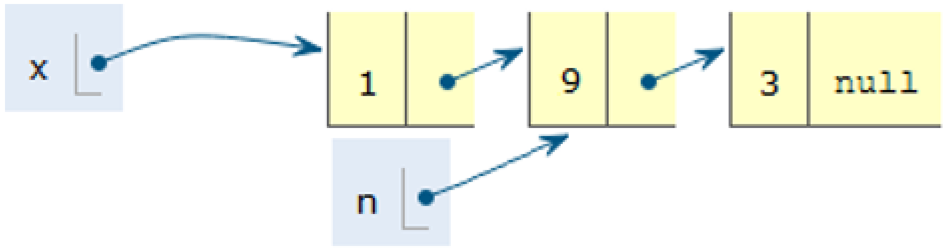
Skip List … Time complexity of search

Extra Activity on Linked List

You have been given the linked list shown below:



Write code to (1) declare a node `n`, (2) and change `x`, such that the memory looks like the picture below after the code has run:



Extra Activity on Linked List

Given the `Node` constructor definition on the right, which statement inserts an item `x` after node `c`?

a. `c = new Node<>(x, c);`
b. `c = new Node<>(x, c.next);`
c. `c.next = new Node<>(x, c);`
d. `c.next = new Node<>(x, c.next);`
e. none of the above

```
public Node(T v, Node<T> n) {
        this.value = v;
        this.next = n;
}
```

Questions?

Stacks

Stacks: a Data Structure that works like a Stack.

Example: Stack of books

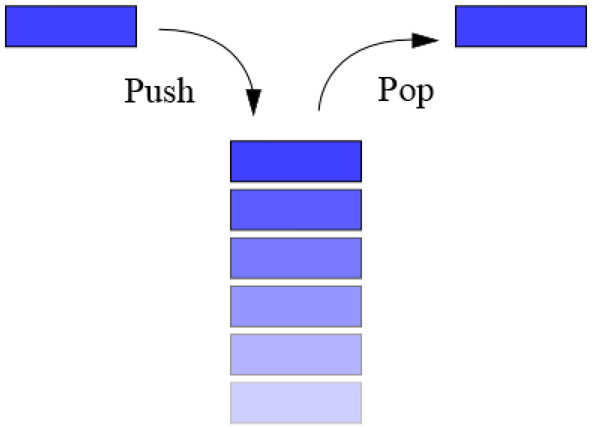
Motivating example: call stack

```
… void main(…) {
    a();
    b();
}

void a() {
    b();
    print(1);
}

void b() {
    c();
    print(2);
}

void c() {
    print(3);
}
```

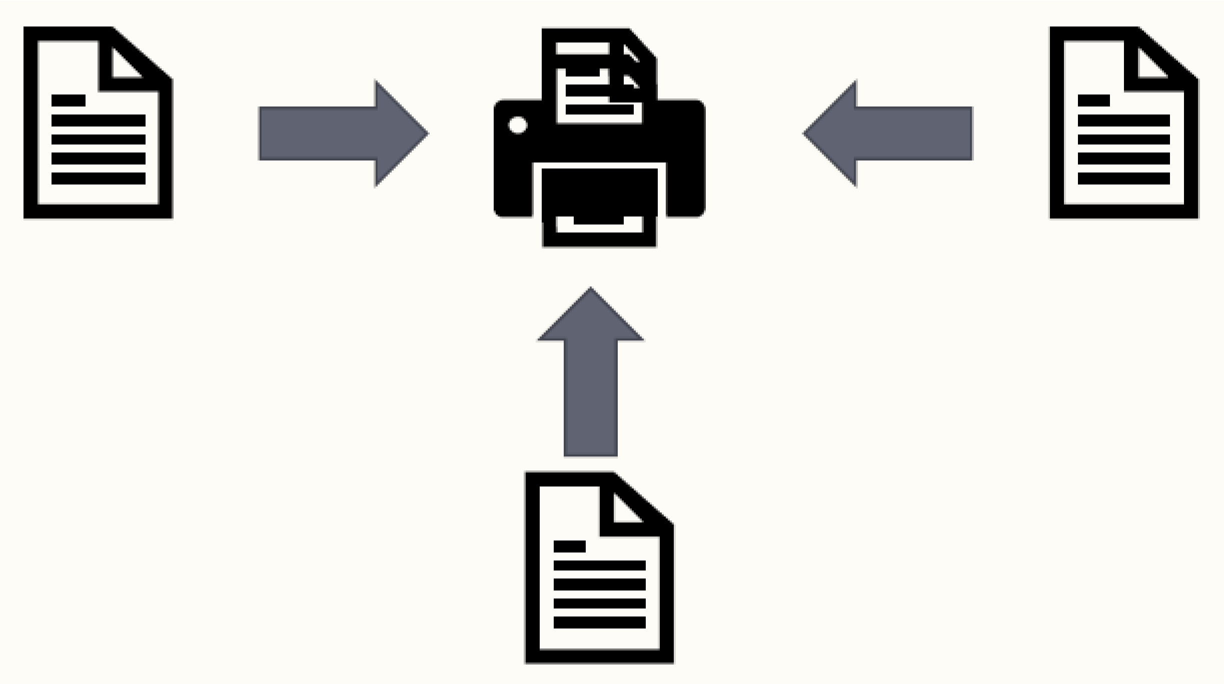


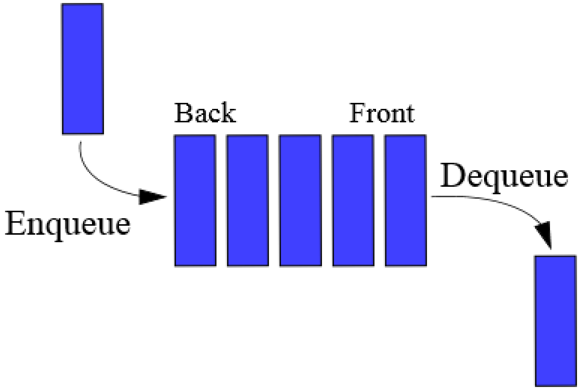
Queues

A Queue is a Data Structure that works like a Queue (line).

Example: Line in a Super Market

Motivating example: printer queue





Stacks Whiteboard

– Operations:
  - **Push**, **pop**, **peek**, **isEmpty**, **size**

– **Arrays** and/or **Dynamic Arrays**
  - – Typical solution, **good constants**

– **Linked List**
  - – Can use singly **linked list**
  - – Larger constant, but **consistent performance**

Queue Whiteboard

– Operations:

Enqueue, **dequeue**, **peek**, **isEmpty**, **size**

– **Array** w/size (or **DynamicArray**)

+ "**Circular Queue**"

– **Linked List**

– **Easy** implementation with singly linked list + **head/tail** reference

Stack and Queue / Some Applications

Reversing a word:

Algorithm

- Declare a **stack** of characters
- while (there are more characters of the word to read):

  Read a character

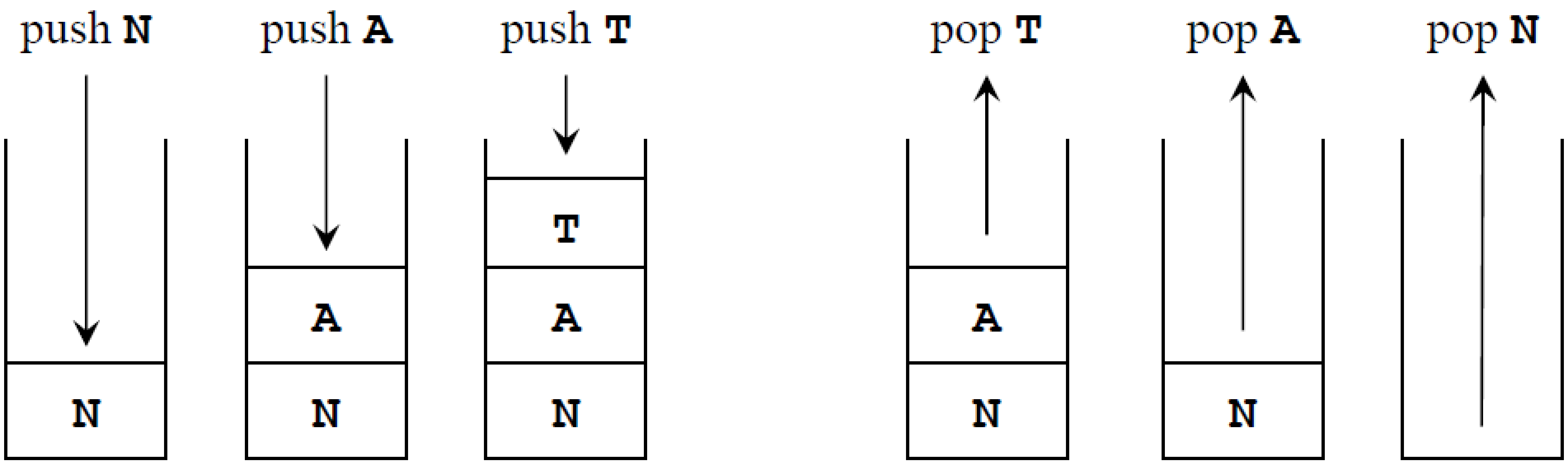
  Push this character onto the stack
- while (the stack is not empty)

  Pop a character off the stack

  Write this character onto the screen

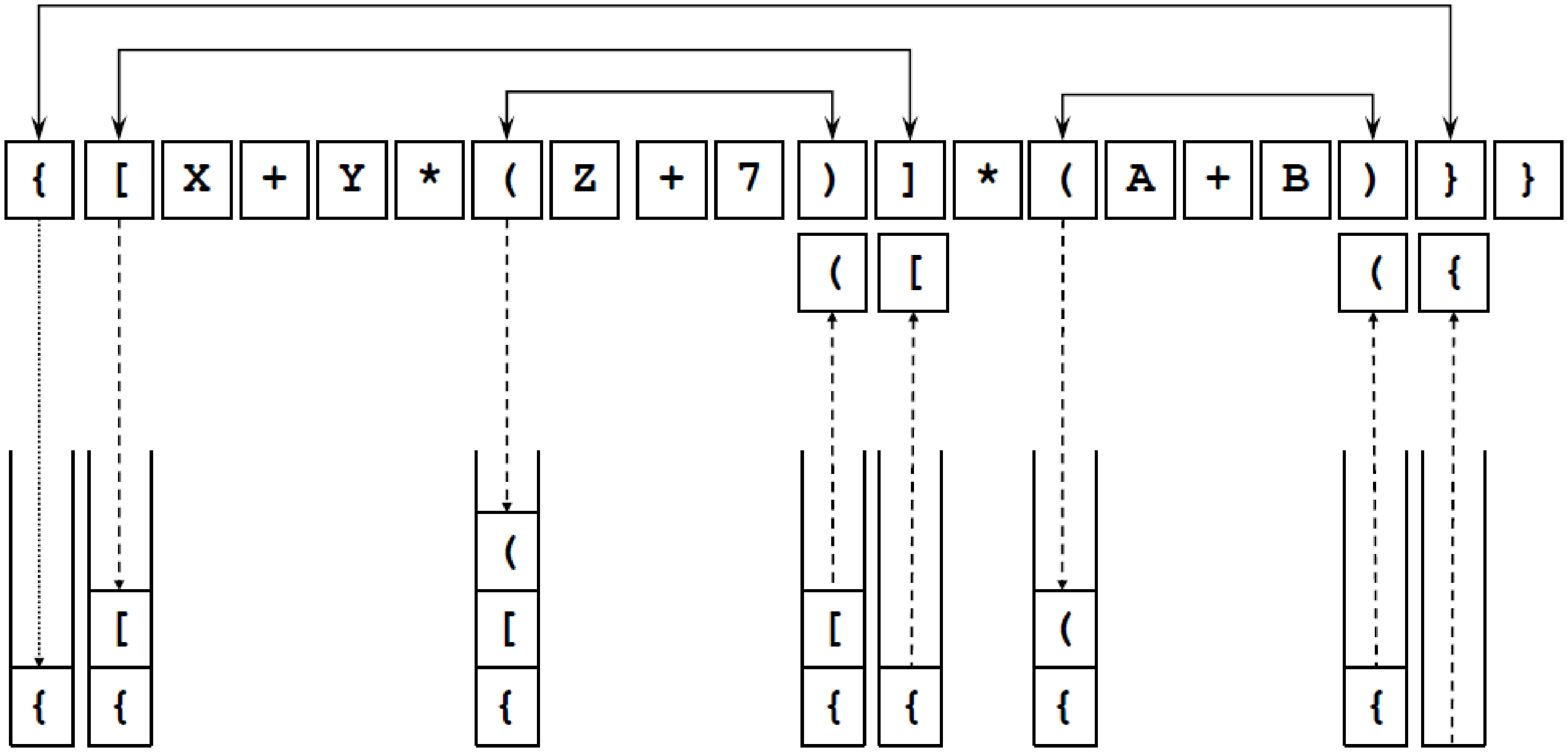
Stack and Queue / Some Applications

- Input: NAT
- Output: TAN



Stack and Queue / Some Applications

Balancing Parentheses



Stack and Queue / Some Applications

Evaluating Arithmetic Expression

Specification

- Program accepts fully parenthesized expressions.
- Returns evaluated expression.
- Assume binary operators only and fully parenthesized expressions (no worry about precedence rules)

Stack-Based Approach—Algorithm

- Declare two stacks: one for operands & one for operators.
- While more tokens
  - **Push** number tokens onto operands stack.
  - **Push** operator tokens onto operators stack.
  - Skip left parentheses and spaces. (assume parentheses are balanced)
  - For right parentheses:
    - **Pop** 2 operands & 1 operator.
    - **Evaluate** expression.
    - **Push** result onto operands stack.

Stack and Queue / Some Applications

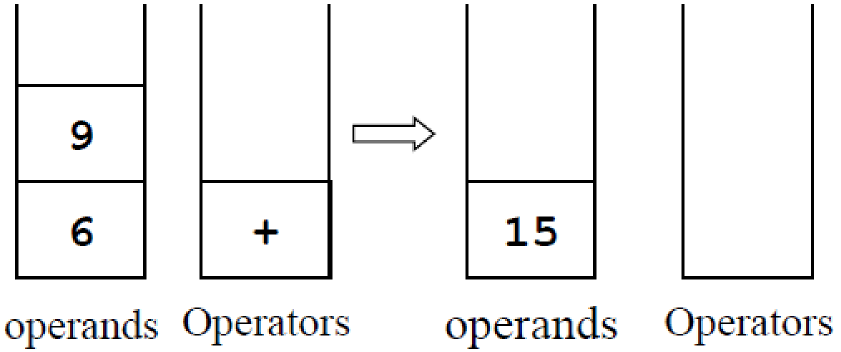
<u>Evaluating Arithmetic Expression</u>

Stack-Based Approach—Animation

`(((6 + 9) / 3) * (6 - 4))`

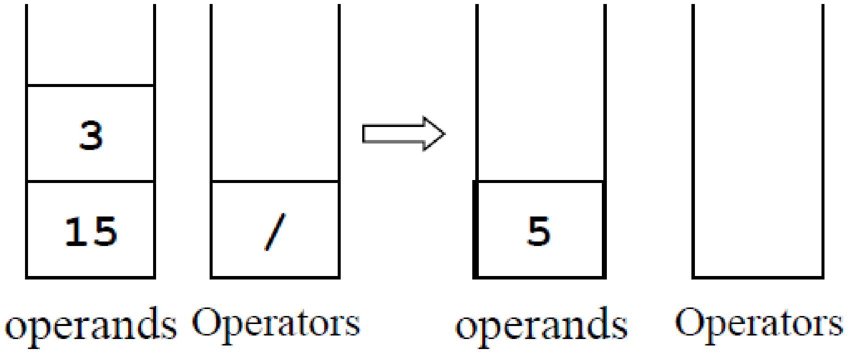
Evaluation 1 (first right paren.)

<u>`(((6 + 9)`</u> `/ 3) * (6 - 4))`



Evaluation 2 (second right paren.)

<u>`(((6 + 9) / 3)`</u> `* (6 - 4))`



Stack and Queue / Some Applications

<u>Evaluating Arithmetic Expression</u>

Stack-Based Approach—Animation

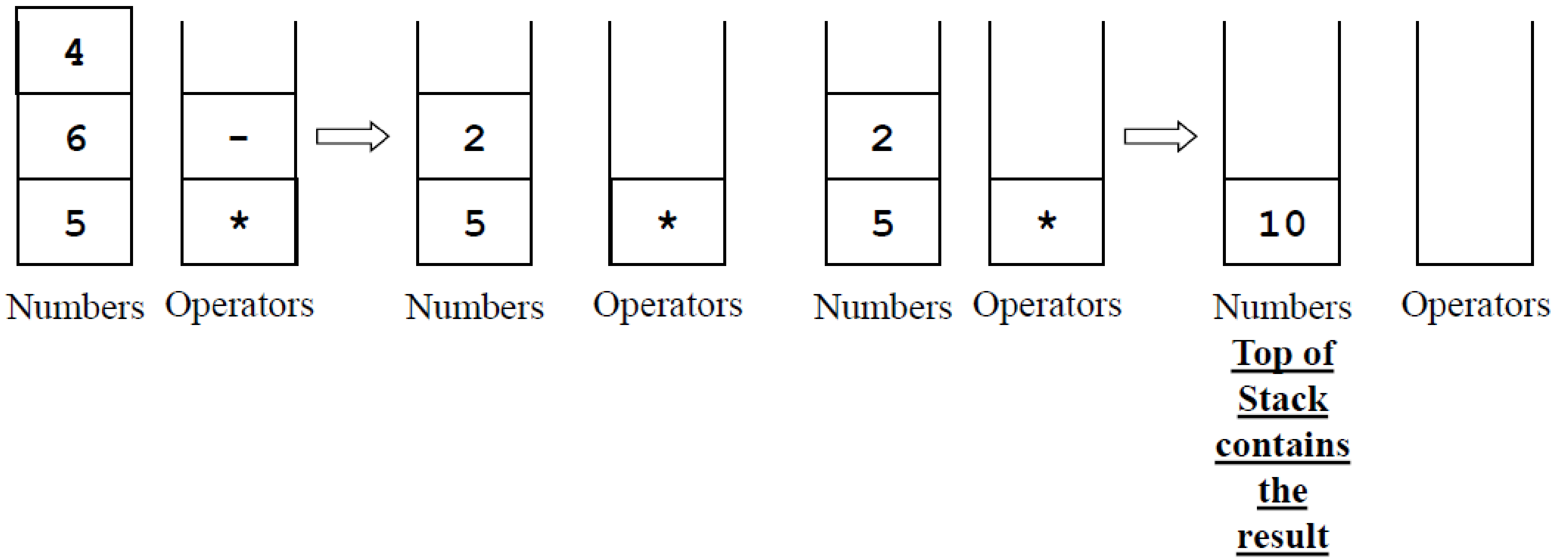
`(((6 + 9) / 3) * (6 - 4))`

Evaluation 3 (third right paren.)

<u>`(((6 + 9) / 3) * (6 - 4))`</u>)

Evaluation 4 (fourth right paren.)

<u>`(((6 + 9) / 3) * (6 - 4))`</u>



Activity (Your Turn)

Evaluating Arithmetic Expression

Use the same approach to evaluate:

((4+5)/3)

Show the states of the two stack for each step

Next Lecture (Lecture 07)

1. **Stacks and Queues**

Reminders:

Keep working on **Project_1**

Do the readings (Ch 11 and 16)

Midterm Exam:

Oct 15: Midterm Exam 10:30am - 11:45pm