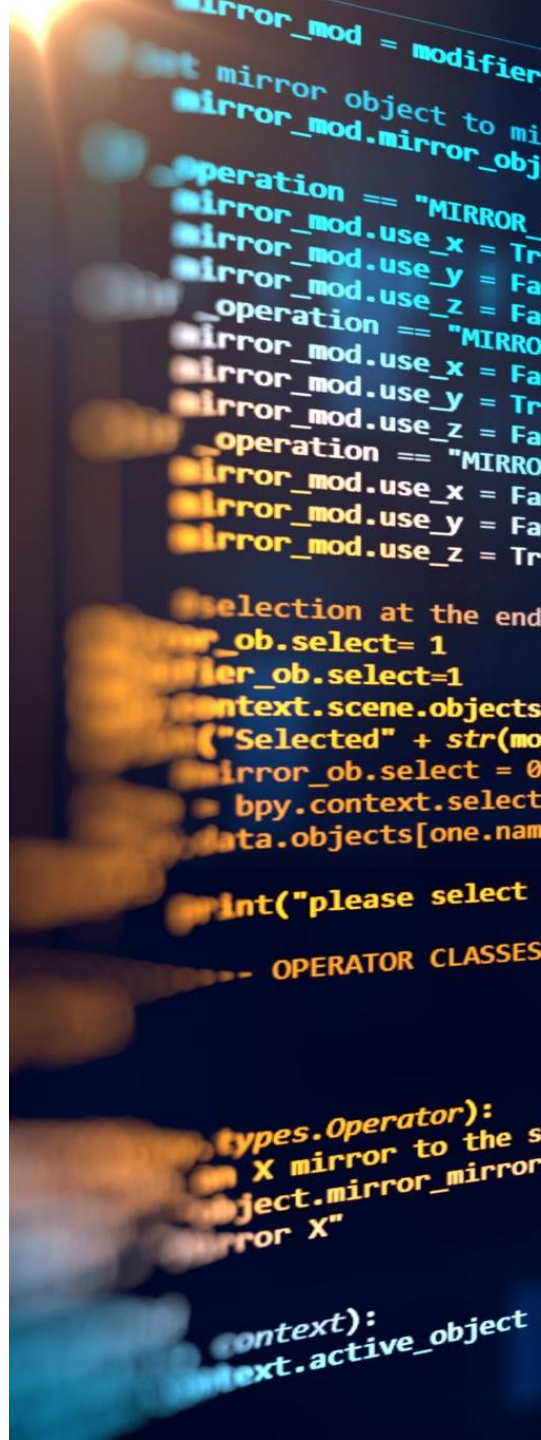
Please use the following QR code to check in and record your attendance.

CS 1037
Fundamentals of Computer Science II

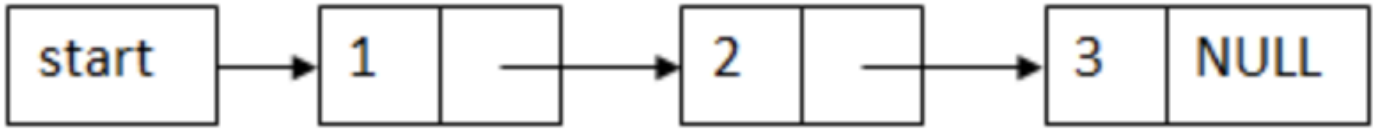
Linked Data Structures (cont.)

Ahmed Ibrahim



Recall: Linear Linked Data Structures

- A Linear or Singly linked list is a data structure in which each element (node) contains:
  - **Data**: The actual value stored in the node.
  - **Next Pointer**: A reference (or pointer) to the next node in the sequence.
- The last node's next pointer is set to null, indicating the end of the list.



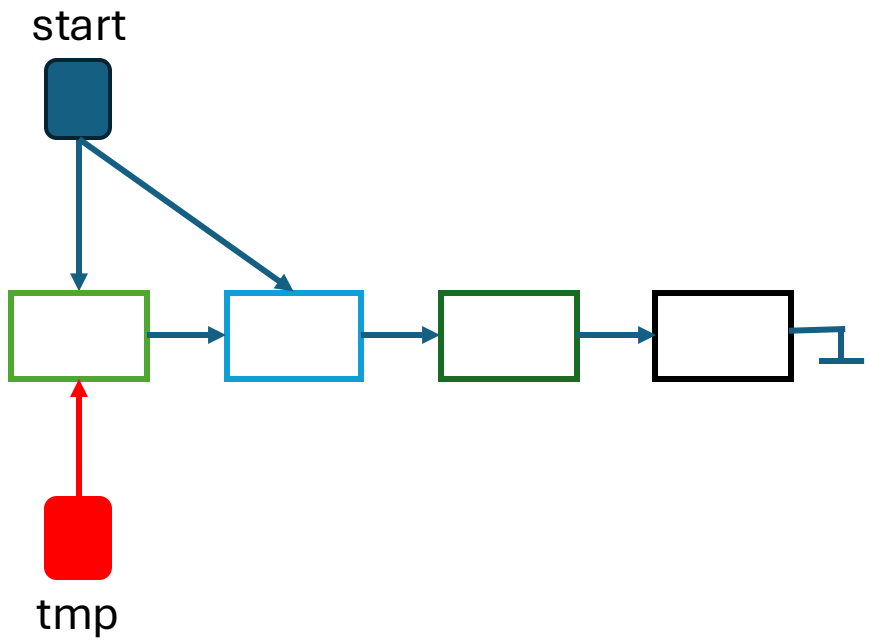
Conceptual Diagram of a Singly-Linked List

Removing Nodes from a Linked List

Delete Node from the Front

- Delete the First Node (Head):

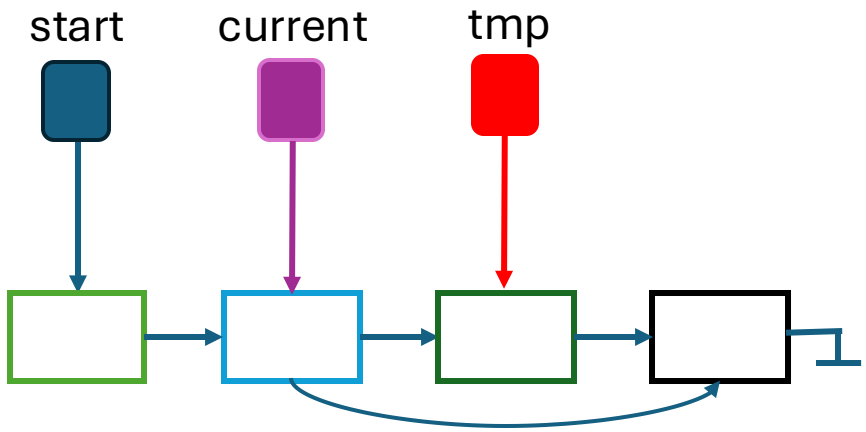
```
…
    // Move head to the next node
    Node *tmp = start;  // Store the current head
    // Update head to the next node
    start = start->next;
    free(tmp);  // Free the old head node
…
```



- The code assumes that the start (the head pointer) points to a valid node, meaning the list contains at least one node.
- If start is NULL, dereferencing start->next would result in undefined behaviour (a segmentation fault in most cases).

Delete Node from the Middle

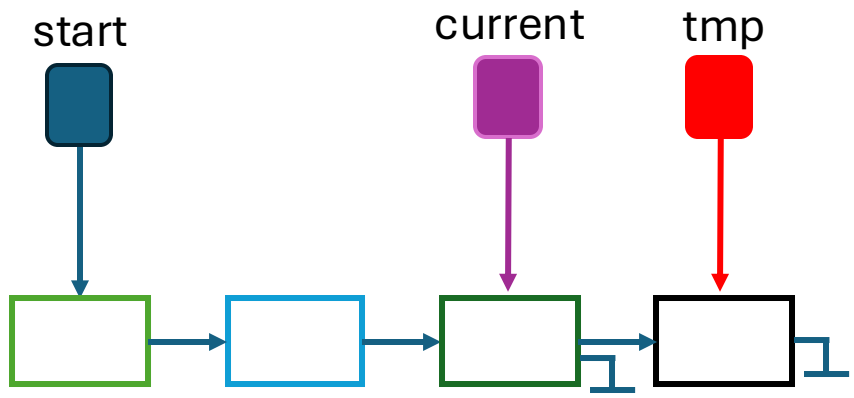
```
…
// Traverse the list to find the node just
//before the desired position
Node *current = start;
for (int i = 0; i < position - 1 && current != NULL; i++) {
    current = current->next;}
// Delete the node at the desired position
Node *tmp = current->next;
current->next = current->next->next;
// Update the link to skip the deleted node
free(tmp);  // Free the node
…
```



- The code assumes that the position provided is valid and that the list has enough nodes to perform this operation safely.

Delete Node from the End

```
…
    // Traverse to the second-to-last node
    Node *current = start;
    while (current->next->next != NULL) {
        current = current->next;
    }
    // Free the last node and set the second-to-last
    //node's next to NULL
    Node *tmp = current->next;
    current->next = NULL;
    free(tmp);
…
```



Question!

You are given the following C code snippet:

```
Node *start = createNode(10);
Start->next = createNode(20);
Start->next->next = createNode(30);
insertAtPosition(&start, 25, 2);
deleteAtPosition(&start, 1);
```

Initial list:
10 -> 20 -> 30 -> NULL
List after inserting 25 at position 2:
10 -> 25 -> 20 -> 30 -> NULL
List after deleting the node at position 1:
25 -> 20 -> 30 -> NULL

What will the final linked list look like after performing both operations, assuming the initial state and the insertion and deletion operations succeed without errors?

A) 25 -> 20 -> 30 -> NULL

B) 10 -> 30 -> NULL

C) 25 -> 30 -> NULL

D) 10 -> 20 -> 25 -> 30 -> NULL

Memory Management in Linked Lists

- Linked lists in C often use dynamic memory allocation (e.g., malloc) to create new nodes.
- To avoid memory leaks, it is essential to deallocate memory using free() when a node is no longer needed.
- Do not call free() on the same pointer more than once; this can cause undefined behavior.

Linked List Traversal

- This operation involves visiting each node in the list and is essential for printing, searching, or processing each node's data.

```
void traverse(Node*start) {
    Node*current = start;
    while (current != NULL) {
        printf("%d\n", current->data);
        current = current->next;
    }
}
```

- Searching allows you to find if a particular data element exists in the list.

```
int search(Node*start, char*data) {
    Node *current = start;
    while (current != NULL) {
  // Compare the current node's
  // data with the target data
        if (strcmp(current->data, data) == 0) {
          return 1; // Data found }
        // Move to the next node
        current = current->next;}
      return 0; // Data not found
}
```

Remember!

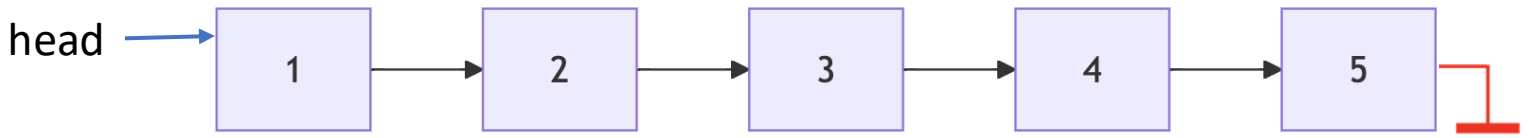
- **Insertion operation**:
  - At the front, The new node becomes the head.
  - In the middle, Insert the node between two existing nodes.
  - At the end, The new node is linked after the last node.
- **Deletion operation**:
  - From the front, The head is removed, and the second node becomes the new head.
  - From the middle, Remove a node and link the previous node to the next one.
  - From the end, Remove the last node and set the second-to-last node's pointer to null.
- **Traversal**: Traverse through all the list nodes, typically from the head to the tail.
- **Search**: Search for a node containing a specific value in the list.

Basic Operations

Advanced Operations: Reverse a List

Reverse: Reverse the order of nodes in the list.

```
// Function to reverse the linked list
struct Node* reverseLinkedList(struct Node* head) {
struct Node* prevNode = NULL;
struct Node* nextNode = NULL;
struct Node* current = head;

while (current != NULL) {
nextNode = current->next; // Store the next node

current->next = prevNode; // Reverse the link

// Move prevNode and current one step forward
prevNode = current;
current = nextNode;
}

return prevNode; // Return the new head (prevNode)
}
```



More Advanced Operations

Merging Two Lists: Combine two singly linked lists into one.

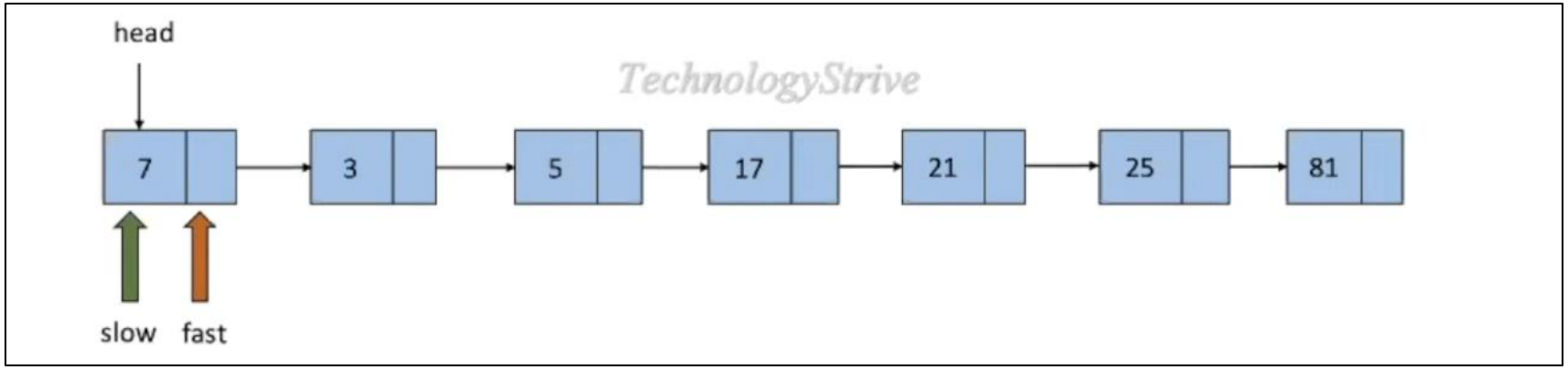
```
// Function to merge two sorted linked lists
struct Node* mergeLists(struct Node* head1, struct Node* head2) {
// If one of the lists is empty, return the other list
if (head1 == NULL) return head2;
if (head2 == NULL) return head1;

// Create a dummy node to help build the new list
struct Node* dummy = (struct Node*)malloc(sizeof(struct Node));
dummy->data = 0; // This value won't be used
struct Node* current = dummy;

// Traverse both lists and merge them in sorted order
while (head1 != NULL && head2 != NULL) {
if (head1->data <= head2->data) {
current->next = head1;
head1 = head1->next;
} else {
current->next = head2;
head2 = head2->next;
}
current = current->next;
}

// If there are remaining nodes in either list, append them
if (head1 != NULL) {
current->next = head1;
} else if (head2 != NULL) {
current->next = head2;
}

// The merged list starts at dummy->next, as dummy was just a helper node
struct Node* mergedHead = dummy->next;
free(dummy); // Free the dummy node
return mergedHead;
}
```

More Advanced Operations

- **Removing Duplicates**: Eliminate duplicate nodes from the list.
- **Splitting the List**: Split the linked list into two sub-lists based on some criteria (e.g., by index or value).
- **Clone the List**: Create an exact replica of the linked list.
- **Find Middle Node**: Identify the middle node in the list using two-pointer techniques.
- **Nth Node from the End**: Find the nth node from the end of the list using two-pointer techniques.

Finding the Middle Node

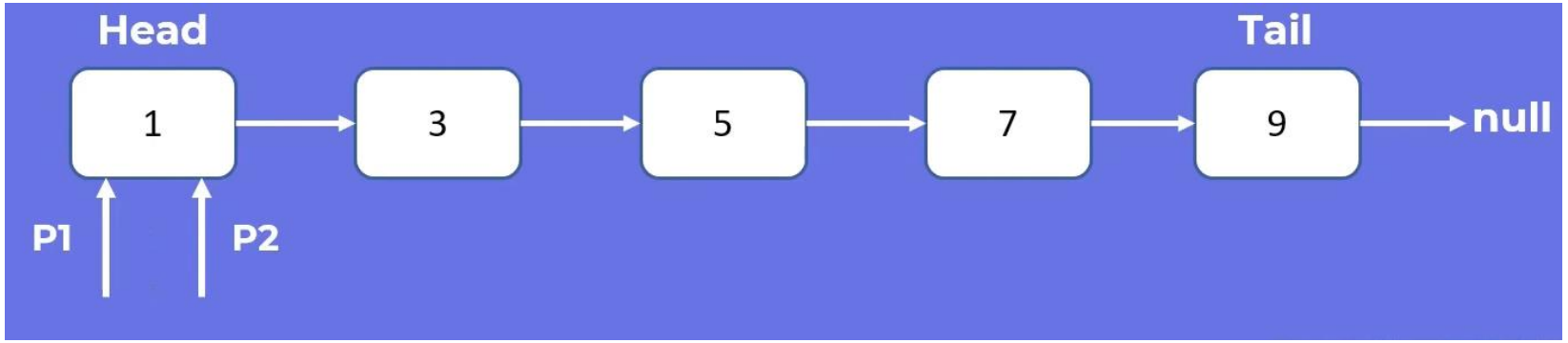
- Given a singly linked list, the task is to find the **middle node** in a single traversal without knowing the list's length in advance.
- Finding the middle of a list is a common problem in algorithms, and optimizing performance requires performing it in a **single traversal using minimal memory**.



Source: https://youtu.be/sqK_gZlecJU

Finding the Nth Node from the End

- Given a singly linked list, the task is to find the **nth node** from the end of the list.
- The challenge is to do this in a **single traversal** without knowing the list's length in advance.
- If the fast pointer reaches the end of the list before completing n steps, this means the list has fewer than n nodes, and the task cannot be completed.

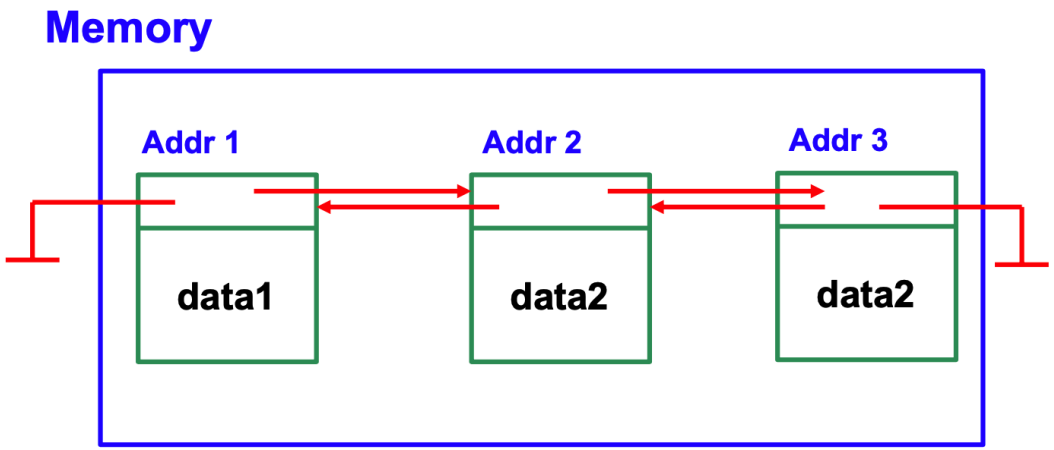


Source: https://www.youtube.com/watch?v=dKFvYm3P6QY

Doubly Linked Data Structures

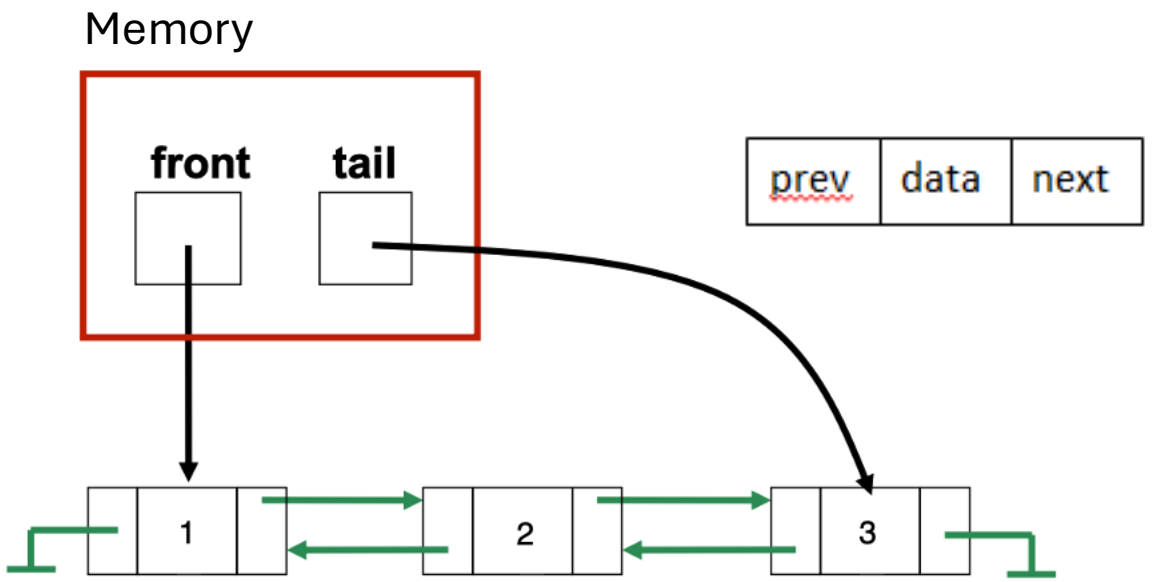
- A doubly linked list is a type of linked list where each node contains three fields:
  - **Data**: The value stored in the node.
  - **Next Pointer**: A reference to the next node in the sequence.
  - **Previous Pointer**: A reference to the previous node in the sequence.
- The first node's previous pointer is NULL, and the last node's next pointer is NULL, signifying the list's boundaries.

```
typdef struct Node {
int data;
struct Node *prev;
struct Node *next;
} NODE;
```



Doubly Linked List

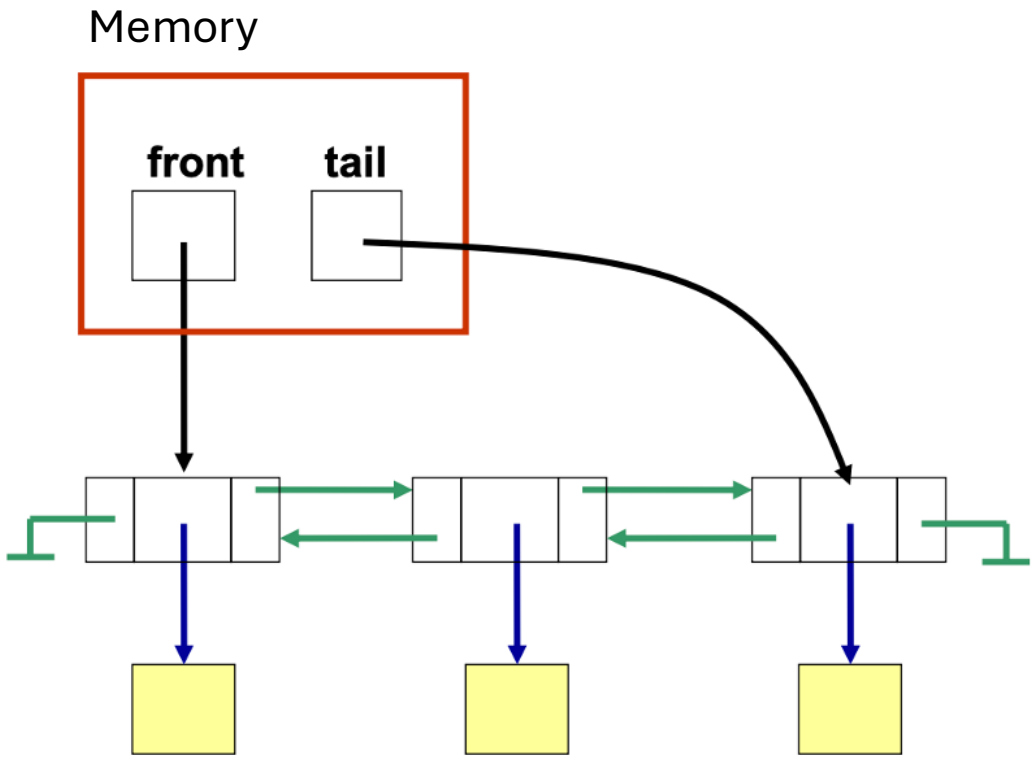
- The following figure shows the doubly linked list structure.
- The front (Head) and Tail pointers track the first and last nodes in the list, respectively.
- Each node is **dynamically** allocated in memory, allowing flexible list sizes that grow and shrink as needed.



Doubly Linked List Structure

Doubly Linked List cont.

- Doubly linked lists support operations like traverse (forward and backward), search (forward and backward), insert (any location), and delete.
- Real-World Use Cases:
  - Undo/Redo functionality
  - Browser navigation
  - Music/playlist management



Doubly vs. **Singly** Linked List

Doubly Linked List:

- **Pros**: It allows bidirectional traversal and efficient insertion/deletion from both ends or the middle of the list.
- **Cons**: Higher memory usage is due to storing two-pointers, which is more complex to implement.

Singly Linked List:

- **Pros**: It is simpler and uses less memory, which makes it good for scenarios where only forward traversal is needed.
- **Cons**: Inefficient for operations that require traversal or insertion/deletion from the middle or end, as there is no easy way to move backwards.

Question!

In a doubly linked list, what happens when you delete a node in the middle of the list?

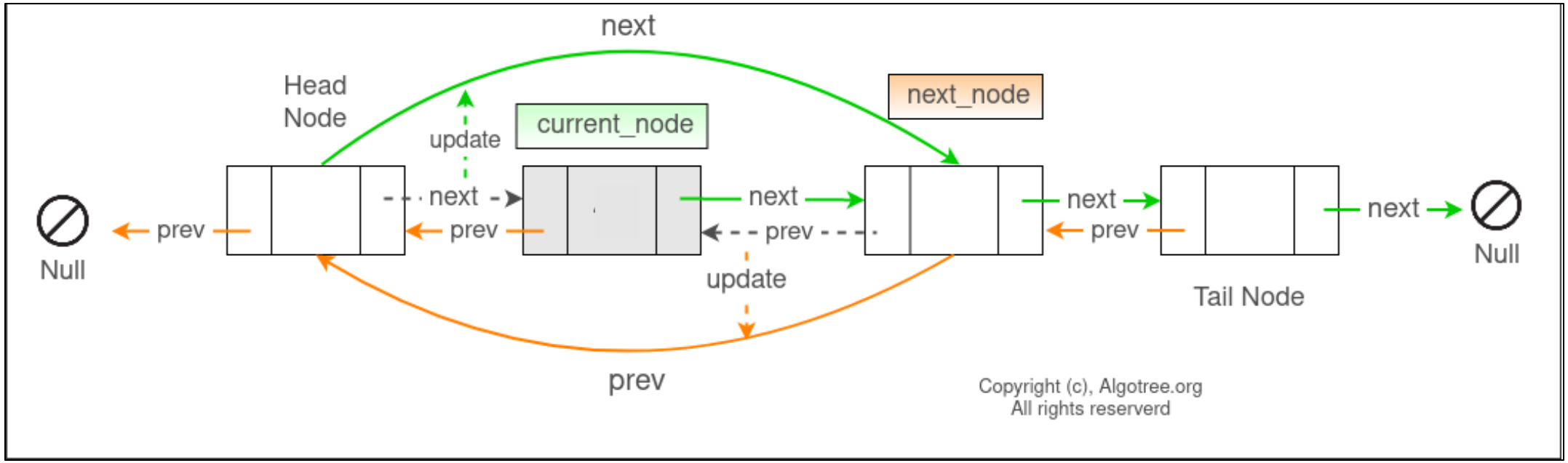
A) The previous pointer of the node before the deleted node is updated to point to the next node, and the next pointer of the next node is updated to point to the previous node of the deleted node.

B) Only the next pointer of the node before the deleted node needs to be updated to point to the next node.

C) Only the previous pointer of the next node needs to be updated to point to the previous node.

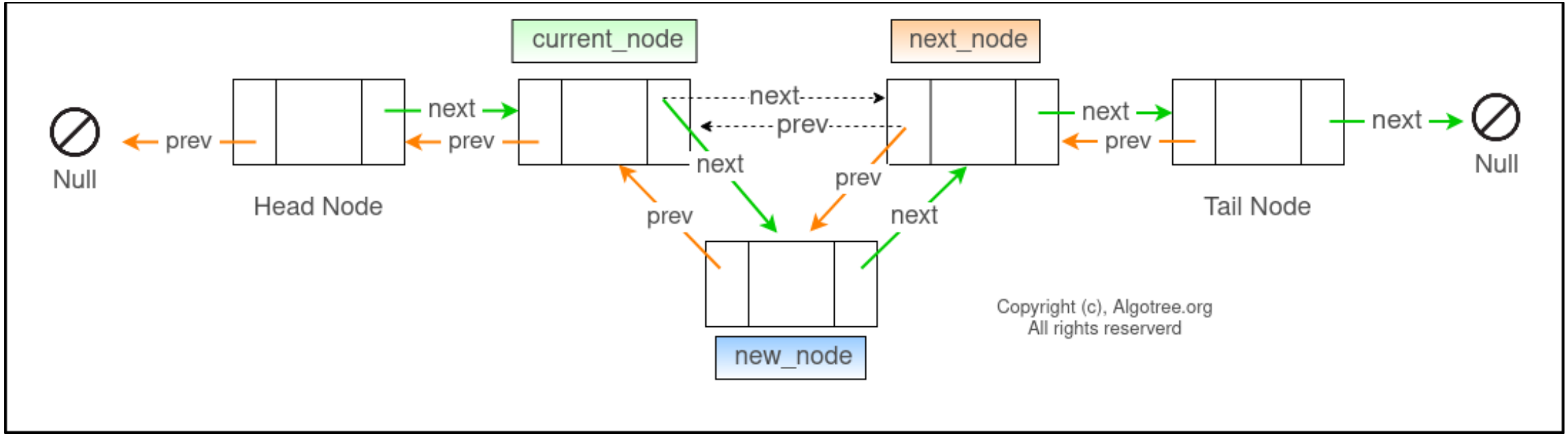
D) Both the next pointer of the previous node and the previous pointer of the next node must be updated to bypass the deleted node.

Deleting a node from a Doubly Linked List



Source: https://www.algotree.org/algorithms/linkedlists/doubly_linked_list/

Insert a node from a Doubly Linked List



Source: https://www.algotree.org/algorithms/linkedlists/doubly_linked_list/

Question!

Consider a doubly linked list implementation. What is the most efficient way to iterate over the list from the head to the tail and print each node's data?

A) Use two pointers, one starting from the head and one from the tail, traverse the list in both directions.

B) Use a single pointer starting from the head and moving forward until the tail is reached.

C) Use recursion to visit each node starting from the head.

D) Use the next pointer of each node, then reverse the list and use the previous pointer.

When might you need two pointers in a doubly linked list scenario?

Scenarios!

- When you need to find the **middle node** of the list in a single traversal.
- You need to reverse a specific section of the list.
  - Mark the start and end of the section with two pointers, then adjust their pointers to reverse the nodes between them.
- When you want to merge two sorted doubly linked lists into one.
- Comparing Two Lists
- Partitioning a List

Abstract Data Structure (ADT)

Abstract Data Structure

An **Abstract Data Type (ADT)** is a model for data types where the type is defined by its **behaviour** (the operations that can be performed) rather than by its **implementation**.

ADTs describe what data is stored and what operations can be done on that data without specifying how these operations are **implemented internally**.

- **Key Characteristics of ADTs:**
  - **Encapsulation**: The internal workings of an ADT (such as data structures or algorithms used) are hidden from the user.
  - **Operations**: ADTs are defined by the operations that can be performed on them, such as insertion, deletion, or access of elements.
  - **Independence from Implementation**: ADTs can be implemented using various data structures, but the user is not informed of these details.

Why?

- **Enforcing Constraints**: ADTs help enforce rules or constraints around the behaviour of the data.
- **Increased flexibility**: Since ADTs are defined by their behaviour, different implementations can be used based on the context.
- **Easier maintenance**: If the ADT implementation needs to change (e.g., to improve performance), only the internal details need to be updated. The external code using the ADT remains unchanged.
- **Designing complex systems**: ADTs help focus on what a data structure should do rather than how it is implemented, which is especially useful when designing complex systems.

Examples of Abstract Data Types

- **Queue ADT:** A queue provides enqueue() and dequeue() operations, but its internal representation can be an array, linked list, or any other structure.
- **Stack ADT:** A stack allows operations like push(), pop(), and peek(). However, how the stack is implemented (using an array or a linked list) is abstracted.
- **List ADT:** A list provides operations like insertion, deletion, and access of elements by index, but the underlying storage mechanism (such as a dynamic array or linked list) is hidden.
- **Set ADT:** A set supports operations like adding, removing, and checking for membership of elements without exposing how the elements are stored.

Queues (ADT)

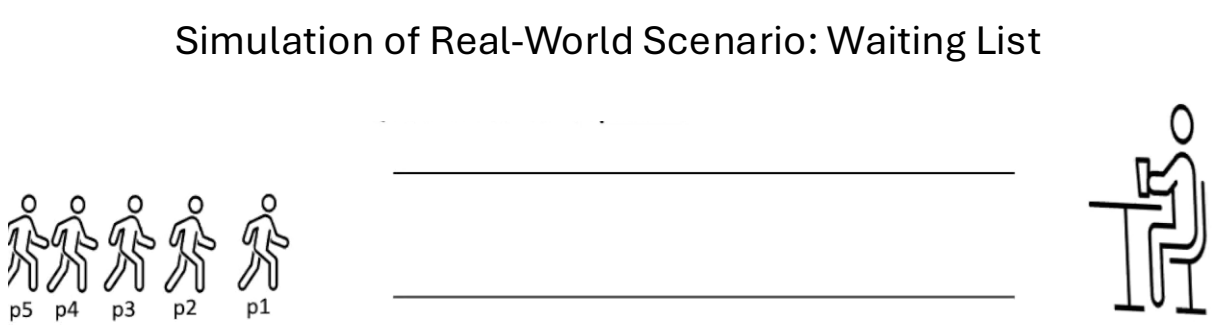
The concepts of **queues** consist of the **abstract queue** and **queue data structures** (the implementations of the abstract queue).

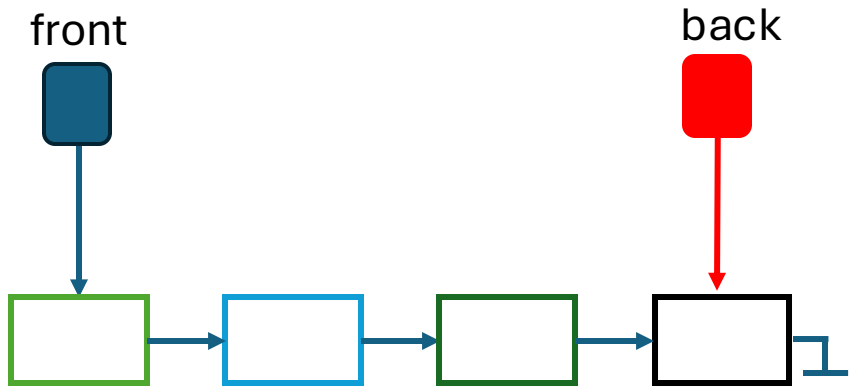
The abstract queue's characteristic is the **First-In-First-Out** (or simply FIFO), which deletes the first element currently in the data structure. The term queue comes from the analogy that people are standing in a queue waiting for services in order of first come, first served.

Examples:

- **Print Queue:** Documents sent to a printer are placed in a queue. The first document in the queue is printed first, followed by the next, and so on.
- **CPU Task Scheduling:** In round-robin scheduling, processes waiting to be executed are placed in a queue, and the CPU handles them in the order they arrive.

Queues (ADT) cont.





- A **queue data structure** is an implementation of the abstract queue.
- A queue can be implemented using an **array** or **linked list** representation, with two accessing variables, front and rear, representing the queue's front and rear (back) positions.

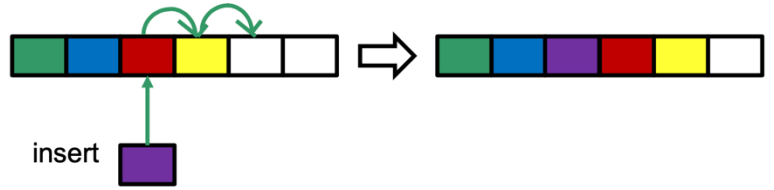
Source:
https://www.youtube.com/watch?app=desktop&v=HcB1P9sJZB4

Underflow vs. Overflow

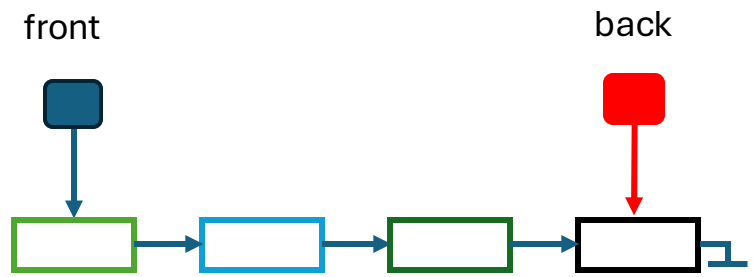
- A queue is said to be empty if it does not contain an element. Deletion cannot be done when a queue is empty; such a situation is called **underflow**.
- The length of a queue is the number of elements in the queue. When the length reaches the maximum length that a queue is allowed, insertion can not be done, and such a situation is called **overflow**.

Linked List Queues

- The array queues have two drawbacks:
  1. The length of its array bounds the length of the queue.
  2. It wastes space if the length of a queue is much shorter than the length of the array.



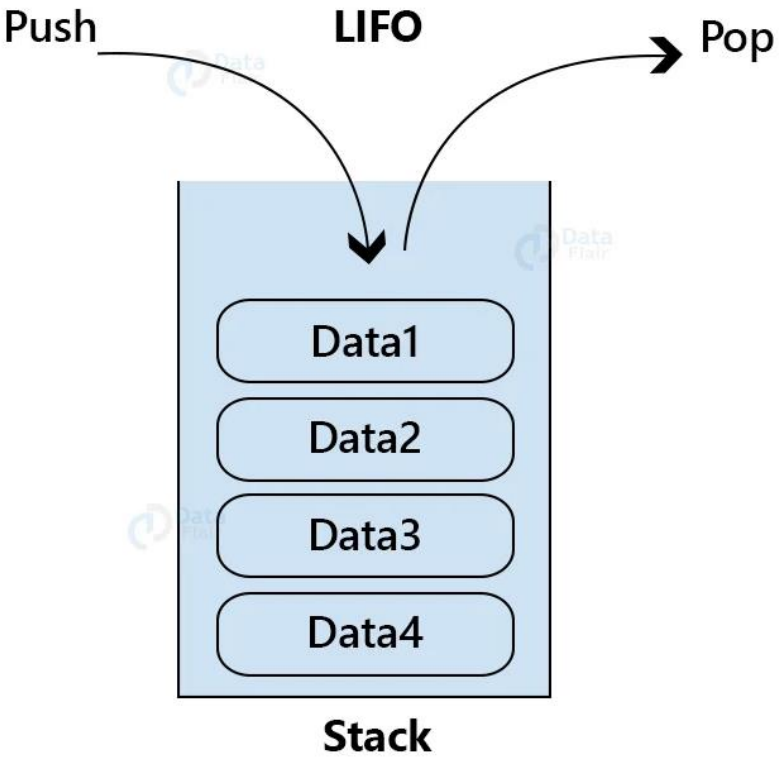
- A **linked list queue** stores queue data values in a singly linked list and uses two pointers, front and rear, to represent the front and rear positions.



- A linked list queue is empty if both front and rear are NULL.
- The queue operations are defined as follows.
  - The **enqueue** operation first creates a node containing the data value, inserts the node after the rear (back) node, and updates both front and rear.
  - The **dequeue** operation deletes the front node (i.e., the node pointed by the front pointer) and updates the front and rear.
- The peek operation returns the data value in the front node.

Stack (ADT)

- The stack concept consists of the abstract stack and stack data structures (the implementations of the abstract queue).
- The following properties specify the abstract stack (or simply **stack**):
  - A linearly ordered collection of data elements with **push**, **pop**, and **peek** operations.
  - The **push** operation inserts an element into the collection. The linear ordering of elements is determined by the time they are pushed; a later-pushed element precedes an early-pushed element.
  - The **pop** operation deletes the latest pushed element (the top element).
  - The **peek** operation gets the latest pushed element.



Source: https://data-flair.training/blogs/stack-in-data-structure/

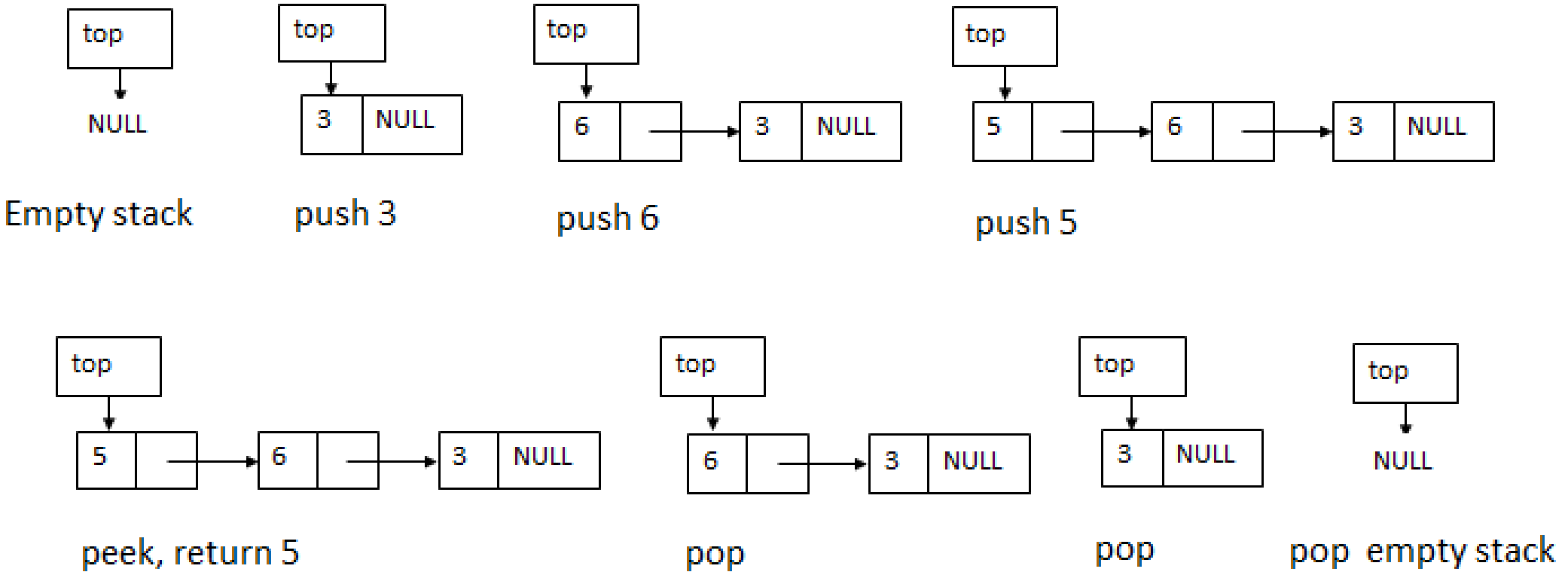
Stack (ADT) cont.

- A stack's characteristic is the **Last-In-First-Out** (or simply **LIFO**), meaning that it pops the latest pushed element.
- The term stack comes from the analogy of a pile of plates. We put (push) a plate on top of the pile and take (pop) the top plate from the pile.
- A stack is **empty** if there is no element in the stack.
- The **length** of a stack is the number of elements in the stack.

Linked List Stack

- Using a singly linked list to implement a stack, a pointer points to the first node. The stack is empty if top = NULL. The stack operations are done as follows.
  - **Push**: create a node containing the element and insert the node at the beginning.
  - **Pop**: delete the first node of the singly linked list.
  - **Peek**: return the first node.

Linked List Stack cont.




thank you