

# Topics

Sunday, July 20, 2025 10:08 AM

- 1. Introduction to DSA and Python Basics**
- 2. Arrays and Lists**
- 3. Strings**
- 4. Linked Lists**
- 5. Stacks**
- 6. Queues**
- 7. Searching**
- 8. Sorting**
- 9. Recursion**
- 10. Hashing**
- 11. Patterns & Problem Solving**



# 1 - Introduction

Saturday, July 26, 2025 9:29 AM

- **Importance of DSA**
- **Time & Space Complexity**
- **Python Refresher**

## 2. Arrays & Lists

Saturday, July 26, 2025 5:10 PM

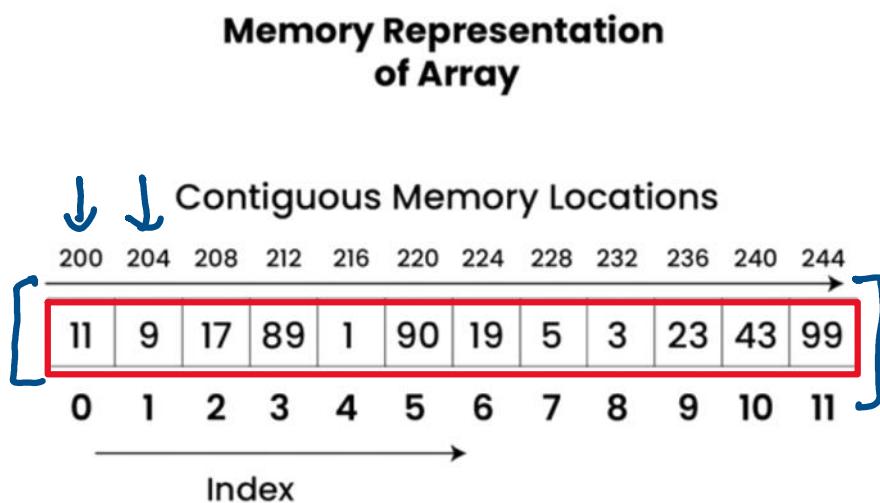
- **Arrays**
- **Lists**
- **Arrays vs Lists**
- **Complexity Analysis**
- **Coding Problems**

## 2.1 - Arrays

Saturday, August 9, 2025 9:04 AM

### What is an Array?

An array is a data structure that stores a collection of elements of the same data type in **contiguous memory locations**



### Properties of an Array:

- ✓ 1. **Homogeneous Elements:** All elements are of the same data type (**model weights of type float32**)
- ✓ 2. **Contiguous Memory Allocation:** Elements are stored one after another in RAM
- ✓ 3. **Fixed Index Access:** Can instantly fetch any element with its index
- ✓ 4. **Efficient Iteration:** Traversing all elements is sequential and cache-friendly

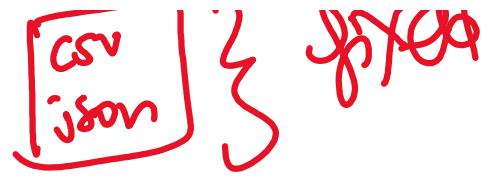
### Fixed-Size Arrays:

- Size is decided at creation ✓
- Stored contiguously in memory
- More memory-efficient, but less flexible ✓
- **Implementation:** *array, numpy* ✓
- **Usage:** Best for predictable, stable data (offline learning)

40

CSV 2 8x80

- **Usage:** Best for predictable, stable data (offline learning)

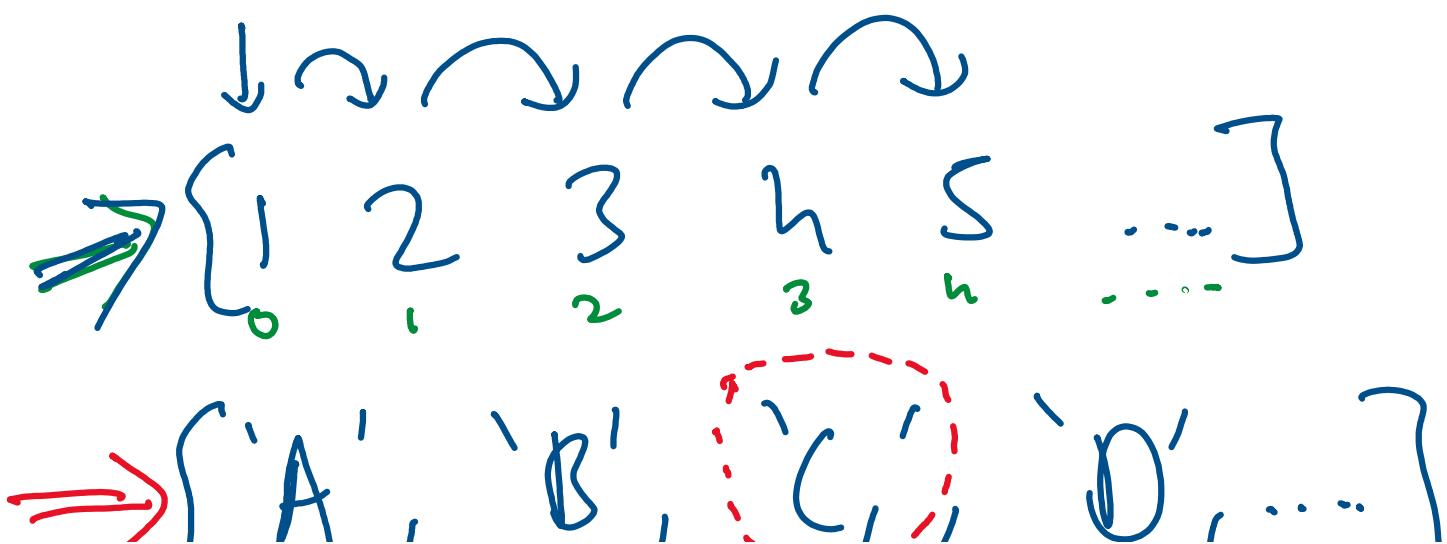


## Dynamic-Size Arrays:

- Can grow or shrink during runtime
- Allocates extra “spare” memory to allow future growth
- **Implementation:** *lists*
- **Usage:** Best for unpredictable, growing datasets (online learning)

## Basic Array Operations:

OPERATION	DESCRIPTION
✓ Traversal	Visiting each element sequentially
✓ Insertion	Adding a new element at a given position
✓ Deletion	Removing an element from a given position
✓ Access (Indexing)	Getting an element at a specific index
✓ Searching	Finding the index of an element
✓ Updating	Changing the value of an element at a specific index
✓ Sorting	Rearranging elements in a certain order
✓ Merging	Combining two arrays into one
✓ Splitting	Dividing one array into multiple arrays



$\Rightarrow [A, B, C, \dots, U, \dots]$

$n \rightarrow n-1$

$[10, 20, 30, 40, 50]$

0 1 2 3 4

The element at index 3 (40) is circled in red.

$[10, 20, 30, 35, 40, 50]$

0 1 2 3 4 5

The element at index 3 (35) is circled in purple. The element at index 3 (35) is also enclosed in a red box.

## 2.2 - Lists

Saturday, August 9, 2025 9:29 AM

### What is a List?

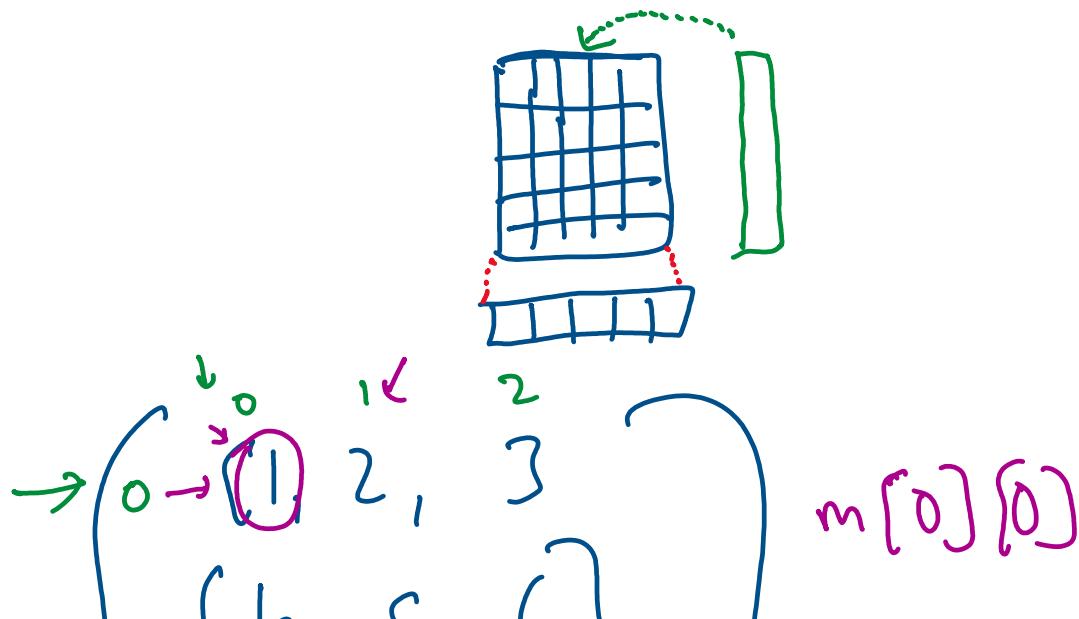
- A list is a built-in and versatile data structure that can hold elements of **different data types**
- A list is a **dynamic, ordered and mutable** sequence in Python

### Properties of Lists:

1. **Ordered:** Items in a list maintain a specific order based on their insertion sequence
2. **Mutable:** Lists are changeable, i.e., elements can be added, removed, or modified after the list has been created
3. **Allows Duplicates:** Multiple identical elements allowed
4. **Indexing:** Supports direct element access using zero-based indexing
5. **Heterogeneous:** Can store elements of mixed data types (e.g., integers, floats, strings, objects)
6. **Dynamic Size:** Can grow or shrink without defining size in advance

### Most Common List Methods:

METHOD	DESCRIPTION	ML ANALOGY
<code>append(x)</code>	Add element x to the end of the list	Add a new training sample to the dataset buffer
<code>insert(i, x)</code>	Insert element x at position i	Insert a missing feature into the correct index of a feature vector
<code>extend(iterable)</code>	Add all elements from another iterable	Add a mini-batch of new samples to an existing dataset
<code>remove(x)</code>	Remove first occurrence of x	Remove a specific label or value from dataset
<code>pop([i])</code>	Remove and return element at index i (default last)	Retrieve and remove the last prediction from stored results
<code>clear()</code>	Remove all elements from the list	Reset the temporary dataset storage
<code>index(x)</code>	Return index of first occurrence of x	Find where a specific label appears in dataset
<code>count(x)</code>	Return number of times x appears	Count samples of a specific class in the dataset
<code>sort()</code>	Sort the list in ascending order (in-place)	Sort prediction scores for ranking tasks
<code>reverse()</code>	Reverse the list in-place	Reverse a time-series sequence for backward RNN processing
<code>copy()</code>	Return a shallow copy of the list	Duplicate dataset before augmentation



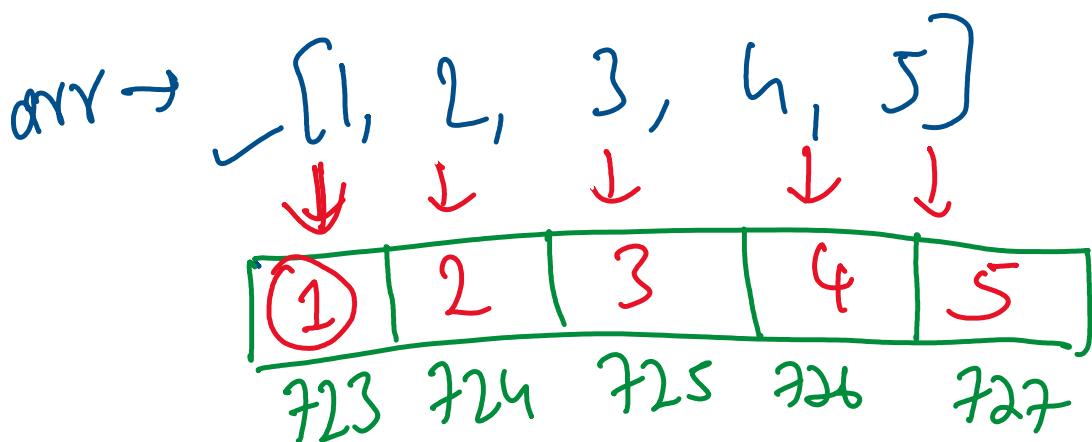
→  $\begin{bmatrix} 1 & [4, 5, 6] \\ 2 & [7, \textcircled{8}, 9] \end{bmatrix}$

$m[2][1]$

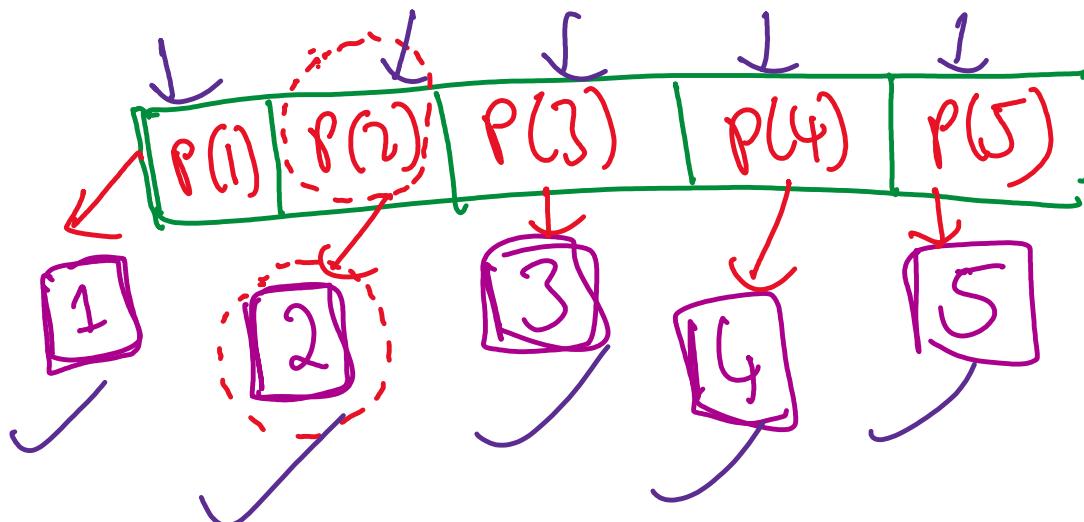
## 2.3 - Arrays vs Lists

Saturday, August 9, 2025 9:29 AM

PROPERTY	ARRAY	LIST
Data Type	- collection of homogeneous (same type) elements	- heterogeneous collection of elements that can store any data types
Memory	- stored in contiguous memory - memory efficient - faster retrieval	- stores references to objects ✓ - less memory-efficient ✓
Performance	- faster for numerical operations - supports <u>vectorized processing</u> (especially with NumPy)	- slower for element-wise math - requires explicit loops
Flexibility	- generally fixed in size	- can grow/shrink dynamically
Functionality	- supports mathematical operations directly (NumPy)	- built-in methods for insertion, deletion, reordering - needs manual loops for math
Use Cases	Scientific computing, ML datasets, image data, numerical simulations	Data collection, mixed-type storage, intermediate data handling



li → [1, 2, 3, 4, 5]





OPERATION	DESCRIPTION	TIME COMPLEXITY	ML ANALOGY
Traversal	Visiting each element sequentially	$O(n)$	Iterating over all feature values in a dataset to normalize them
Insertion	Adding a new element at a given position	$O(n)$ (worst case, due to shifting elements)	Adding a new sample into a sorted list of training data
Deletion	Removing an element from a given position	$O(n)$ (due to shifting elements)	Removing an outlier from a dataset stored in an array
Access (Indexing)	Getting an element at a specific index	$O(1)$	Accessing the pixel value at $(x, y)$ in an image tensor
Searching	Finding the index of an element	$O(n)$ (linear search) or $O(\log n)$ if sorted (binary search)	Looking up a feature value in a sorted array of unique IDs
Updating	Changing the value of an element at a specific index	$O(1)$	Updating a model weight in a flattened parameter array
Sorting	Rearranging elements in a certain order	$O(n \log n)$ (typical)	Sorting dataset samples by label before stratified batching
Merging	Combining two arrays into one	$O(n + m)$	Merging two mini-batches into a single batch before training
Splitting	Dividing one array into multiple arrays	$O(n)$	Splitting dataset into training and validation sets

arr → [1, 2, 3, 4, 5]

↑ ↑ ↑ ↑ ↑

case 1: 1, 2, 3, 4, 5, 6  $O(1)$

case 2: 1 2 3 6 4 5

1 2 3 6 4 5  
[ ] [ ] [ ] [ ] [ ]  
 $n/2$  1  $n/2$

$$n/2 + 1 + n/2 \rightarrow n+1 \rightarrow O(n)$$

case 3: 1 2 3 4 5

arr1 → [10 20 30] [40 50] logn

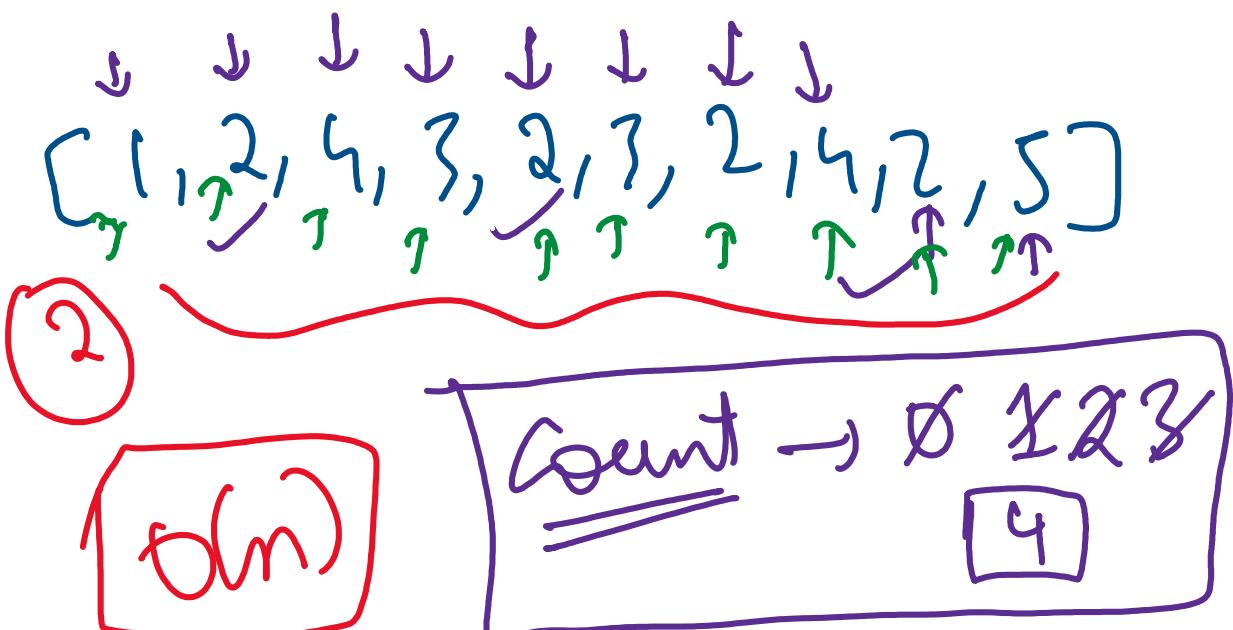
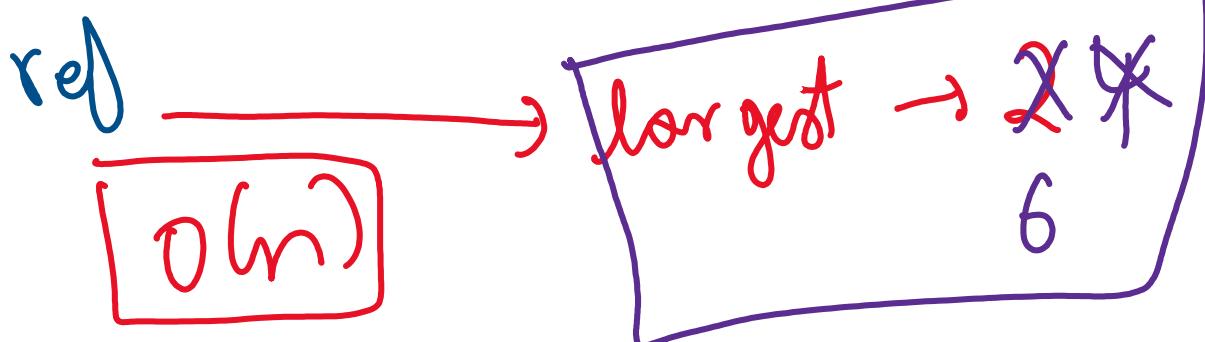
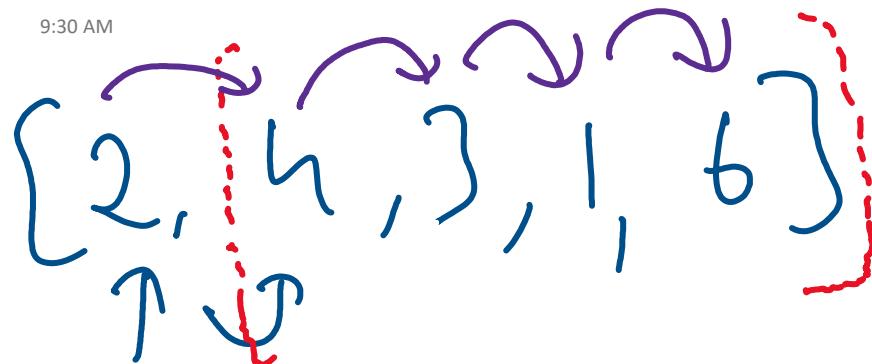
arr2 → 20 30 10 50 40  
 $O(n)$  ↑ ↑ ↑ ↑ ↑  
 $O(1)$   $O(n)$   $O(n)$   $O(n)$   $O(n)$

arr1 → 1 2 3  
 arr2 → 1 2 3  
 $O(n+m)$



## 2.5 - Coding Problems

Saturday, August 9, 2025 9:30 AM



[1, 2, 4, 3, 5]

0n)

$[2, 5, 4, 6, 7]$   
 0 1 2↑ 3↑ 4↑

$[7, 6, 4, 5, 2]$

$dn$

a	b	step
↓	↓	↓
n-1	0	-1

$(0, 1, 2, 3)$   
 $\text{range}(0, 4)$

$\text{range}(4, 0)$

$(4, 3, 2, 1)$









## 2.6 - DS Context

Tuesday, December 16, 2025 12:34 AM

### Data Science / ML Correlation:

- Feature vectors, datasets, tensors
- Model inputs & outputs are arrays

### Libraries & Usage:

- **NumPy:** *ndarray*
- **Pandas:** *Series & DataFrame* columns
- **TensorFlow / PyTorch:** *tensors*

### 3. Strings

Saturday, August 16, 2025 4:13 PM

- **Introduction**
- **Properties**
- **Usecases**
- **Common Functions**
- **String Manipulation Techniques**
- **String Comparison**
- **Coding Problems**



## 3.1 - Introduction to Strings

Monday, August 18, 2025 10:23 PM

### What is a String?

- A string is a sequence of characters enclosed within **single ('), double ("), or triple quotes (''' / """")**
- Strings are ordered sequences, meaning characters have a defined index starting from 0

## 3.2 - Properties of Strings

Monday, August 18, 2025 10:36 PM

- **Ordered sequences:** Characters are stored in order & each character has an index
- **Immutable:** Once created, a string cannot be modified; Any operation creates a new string
- **Iterable:** Characters of a string can be looped over
- **Slicing:** Can extract substrings using [start:end:step]
- **Unicode-based:** Each character is stored as a Unicode code point; Python uses UTF-8 internally
- **Concatenation**
- **Repetition**

PROPERTY	DESCRIPTION
Sequence	Ordered collection of characters
Immutable	Cannot be changed
Iterable	Can be looped
Indexed	Positive & negative indexing
Slicing	Extract substrings
Concatenation/Repeat	"+" and "*" are supported
Unicode	Supports all characters & emojis
Homogeneous	Only characters
Built-in Methods	Rich library of functions
Hashable	Usable as dict keys

## 3.3 - Usecases of Strings

Monday, August 18, 2025 10:42 PM

### **1. Handling Textual Data**

- Names, emails, file paths, log entries
- Almost every real-world program deals with strings

### **2. Foundation for Algorithms:**

- **Pattern Matching:** searching substrings (KMP, Rabin-Karp)
- **Hashing:** rolling hash in string problems (finding anagrams)
- **Parsing & Processing:** extracting useful information from text

### **3. Basis for Natural Language Processing (NLP):**

- **Tokenization, sentiment analysis, machine translation** – all work with strings

## 3.4 - Common Functions

Monday, August 18, 2025 10:49 PM

FUNCTION	DESCRIPTION
<code>s.lower()</code>	Converts all characters to lowercase
<code>s.upper()</code>	Converts all characters to uppercase
<code>s.strip()</code>	Removes leading and trailing whitespaces
<code>s.replace(old, new)</code>	Replaces all occurrences of a substring with another
<code>s.split()</code>	Splits string into a list of substrings (default separator = space)
<code>"sep".join(list)</code>	Joins list elements into a single string with a separator

## 3.5 - String Manipulation

Monday, August 18, 2025 10:55 PM

### 1. Case Conversion:

- **lower()**: Converts entire string to lowercase
- **upper()**: Converts entire string to uppercase
- **title()**: Capitalizes the first letter of each word
- **capitalize()**: Capitalizes only the first letter of the string
- **swapcase()**: Swaps uppercase to lowercase and vice versa

### 2. Whitespace Handling:

- **strip()**: Removes spaces (or specific characters) from both ends
- **lstrip() / rstrip()**: Removes spaces from left/right only

### 3. Searching & Finding:

- **find(sub)**: Returns first index of substring, or -1 if not found
- **index(sub)**: Same as find() but raises ValueError if substring not found
- **startswith() / endswith()**: Boolean check if string starts/ends with given text

### 4. Validation Checks:

- **isdigit()**: Returns True if all characters are digits
- **isalpha()**: Returns True if all are alphabets
- **isalnum()**: Returns True if alphanumeric (letters + digits)
- **islower() / isupper()**: Check case
- **isspace()**: Checks if only whitespace

### 5. Splitting & Joining:

- **split(*sep*)**: Splits string into list (default sep = space)
- **join(*list*)**: Joins list elements with a separator into one string

## 6. Replacing Substrings:

- **replace(*old*, *new*)**: Returns a new string where all occurrences of *old* are replaced with *new*

## 7. Counting & Measuring:

- **len(*s*)**: Returns length of string
- **count(*sub*)**: Counts how many times a substring appears

## 8. Formatting Strings:

- **f-strings** (Python 3.6+)

## 3.6 - String Comparison

Monday, August 18, 2025 11:14 PM

In Python, strings are compared **lexicographically** (like words in a dictionary):

- Python checks character by character from left to right
- Comparison is based on **Unicode code points** (the number assigned to each character)
- If all characters match but lengths differ, the shorter string is smaller

### Common Operators:

OPERATOR	DESCRIPTION	EXAMPLE
<code>==</code>	Equality	"hello" == "hello" → True
<code>!=</code>	Inequality	"hi" != "hello" → True
<code>&lt;</code>	Less than (lexicographic)	"apple" < "banana" → True
<code>&gt;</code>	Greater than	"dog" > "cat" → True
<code>&lt;=</code>	Less than or equal	"app" <= "apple" → True
<code>&gt;=</code>	Greater than or equal	"zoo" >= "zebra" → True

## 3.7 - Coding Problems

Monday, August 18, 2025 11:43 PM

{ key: value }

{ 'b': 1,  
  'a': 3,  
  'n': 2 }

A child's drawing of the word "SILENT" in green and red. The letters are stylized and somewhat illegible. Three red arrows point downwards from above the word. Below the word are several purple wavy lines.

$\left[ [0, 0, 0, 1, 1, -1, -1, 1, 0, 0, 0] \right]$

$$n + n + 26$$



$$2n + h$$

四

A simple green line drawing of a face inside a purple-outlined rectangular frame. The face has a wide, curved smile and two short, diagonal lines extending from the bottom right corner of the frame.

1



## 3.8 - DS Context

Tuesday, December 16, 2025 12:44 AM

### Data Science / ML Correlation:

- Text data processing
- NLP pipelines

### Libraries & Usage:

- **re:** text cleaning
- **NLTK, spaCy:** tokenization
- **transformers:** BERT, GPT tokenizers

## 4. Linked Lists

Wednesday, August 27, 2025 1:55 PM

- **Introduction**
- **Linked Lists in Memory**
- **Linked Lists vs Arrays/Lists**
- **Types of Linked Lists**
- **Singly Linked List**
- **Circular Singly Linked List**
- **Doubly Linked List**
- **Circular Doubly Linked List**
- **Coding Problems**

## 4.1 - Introduction

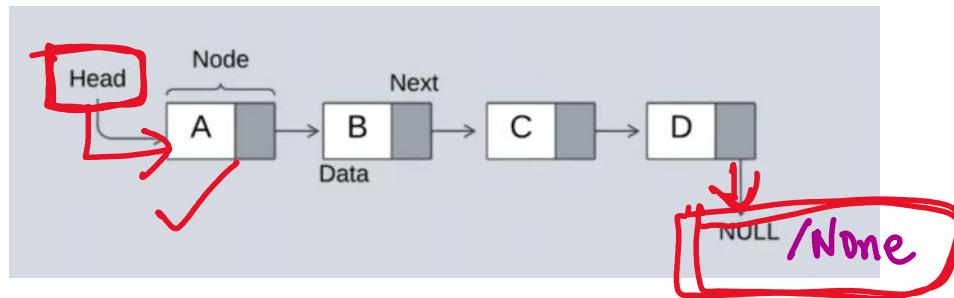
Saturday, August 30, 2025 10:59 AM

### What is a Linked List?

- A linked list is a linear data structure composed of nodes connected by pointers (or references)
- Each node contains a data element and a pointer to the next node in the sequence
- Linked lists do not require contiguous memory and can dynamically change size, making them flexible for operations like insertions and deletions without rearranging the entire structure

### Structure of a Linked List:

- Each element/node of a linked list typically consists of two parts:
  - **Data** - The value stored (e.g., integer, string, object, etc.)
  - **Pointer** - Address/reference to the next node in the sequence



- The first node is called the **head**
- The last node points to **None** (in singly linked list) or back to the **head** (in circular linked list)

### Key Features:

#### 1. Dynamic Size:

- Do not require a fixed size at the time of creation
- Nodes can be added or removed at runtime as needed

#### 2. Non-Contiguous Memory Allocation:

- Nodes can be stored anywhere in memory
- Each node has a pointer/reference that connects it to the next node

#### 3. Efficient Insertions and Deletions:

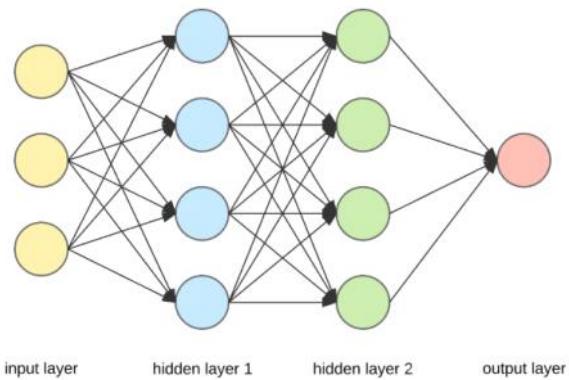
- Insertion/Deletion at the beginning, middle, or end is efficient if the pointer to the node is known
- No shifting of elements is required

#### 4. Sequential Access Only:

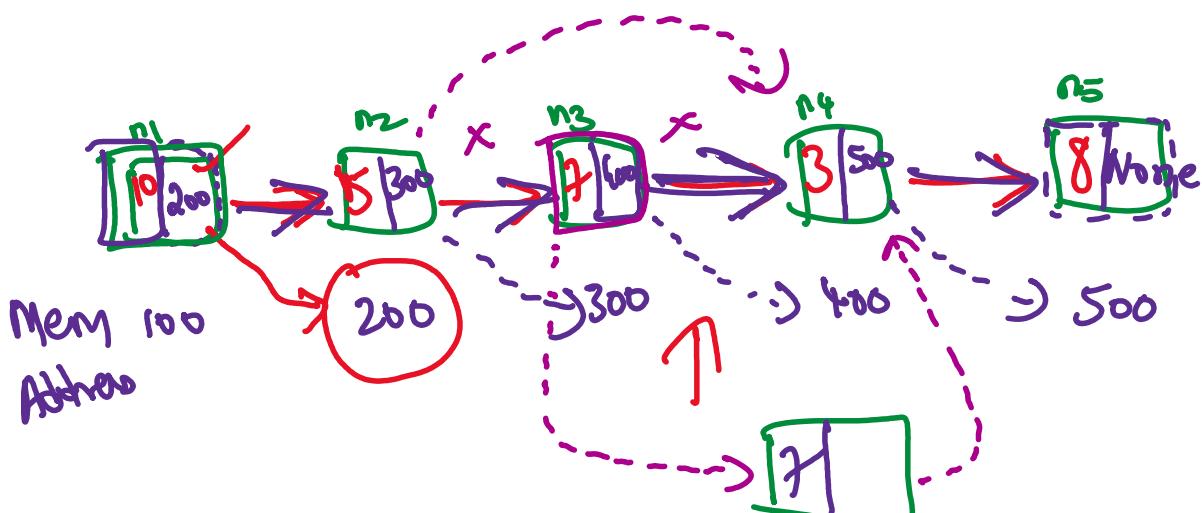
- Elements cannot be accessed directly by index
- To reach a particular node, traversal from the **head** is required

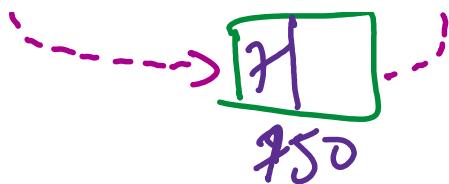
#### ML Analogy:

- **ML Pipelines**
- **Neural Network Architectures:**
  - Layers are stacked and connected sequentially



- **Gradient Descent Steps:**
  - Each update depends on previous state parameters
- **Graph Representation in ML:**
  - ML problems (social networks, knowledge graphs, recommendation systems) are naturally represented using **nodes** and **pointers**





Data Ingestion



Data Validation



Data Trans.



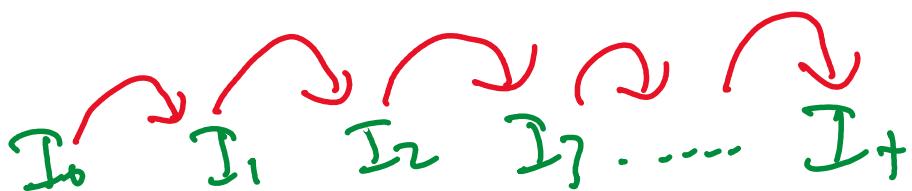
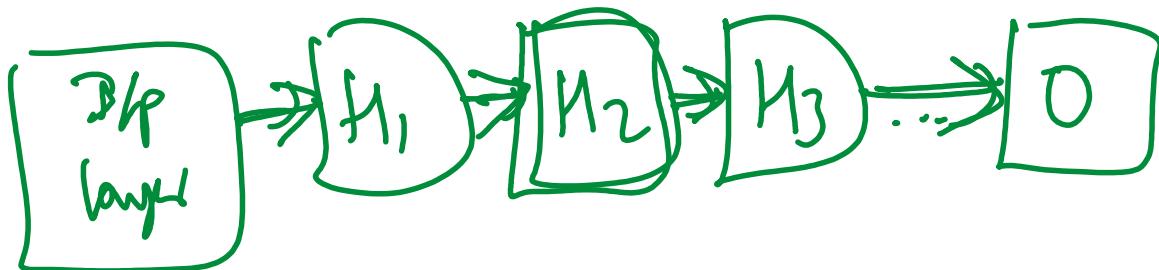
Model Train



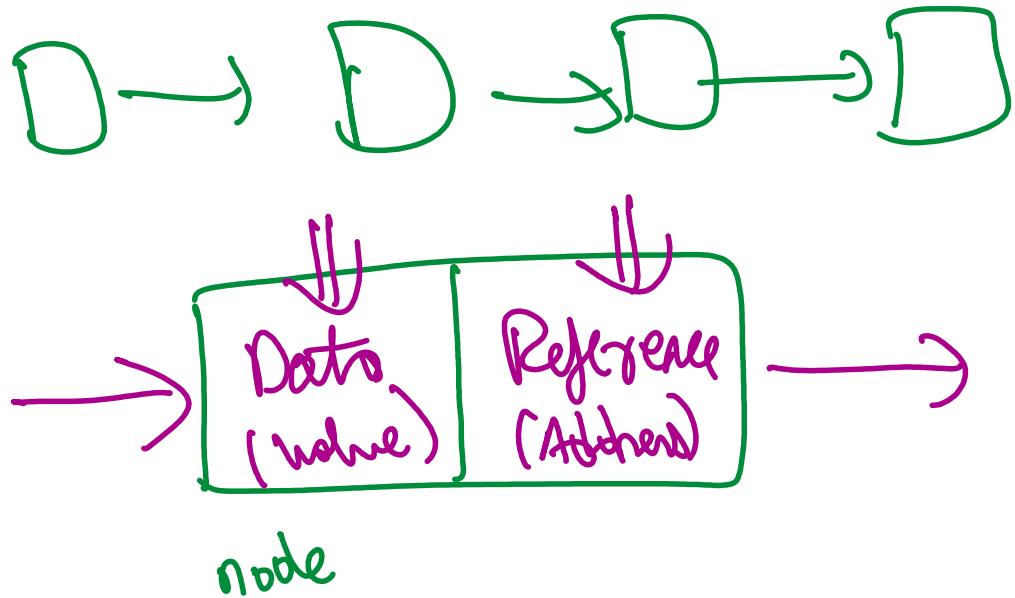
Model Evaluation



Deployment



$I_0 \rightarrow I_1 \rightarrow I_2 \rightarrow I_3 \dots \rightarrow I_t$





## 4.2 - Linked Lists in Memory

Saturday, August 30, 2025 11:12 AM

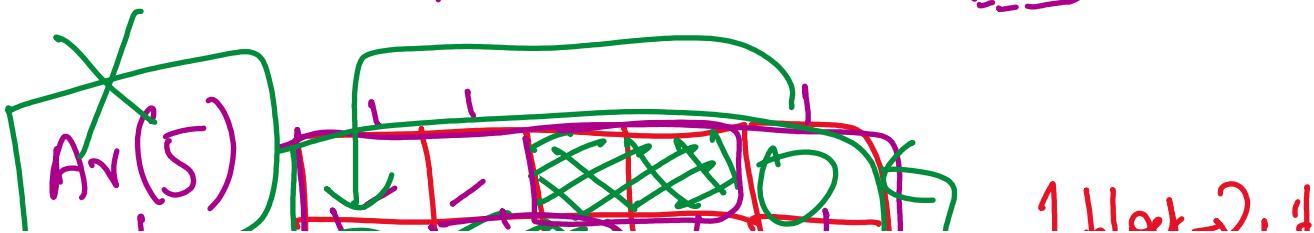
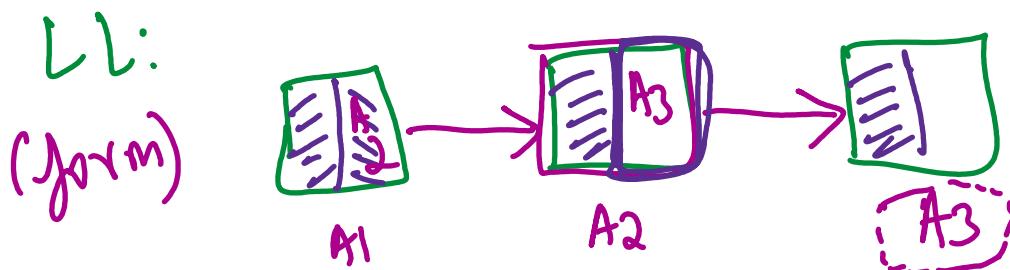
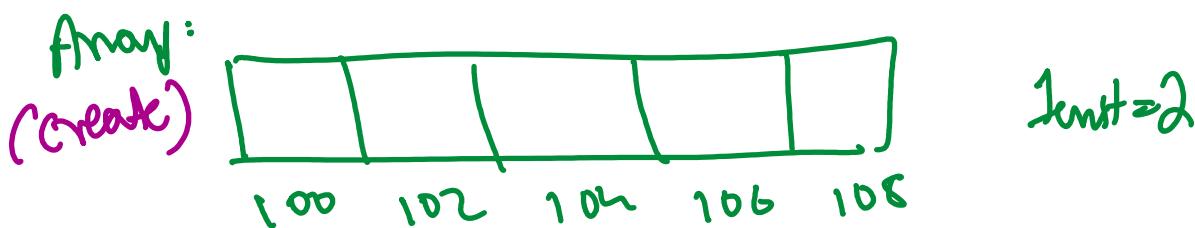
- Linked list nodes are not stored together in a continuous block
- Each node is dynamically allocated whenever a new element is added
- The memory location of the next node is not predictable, so we must store a pointer/reference inside each node

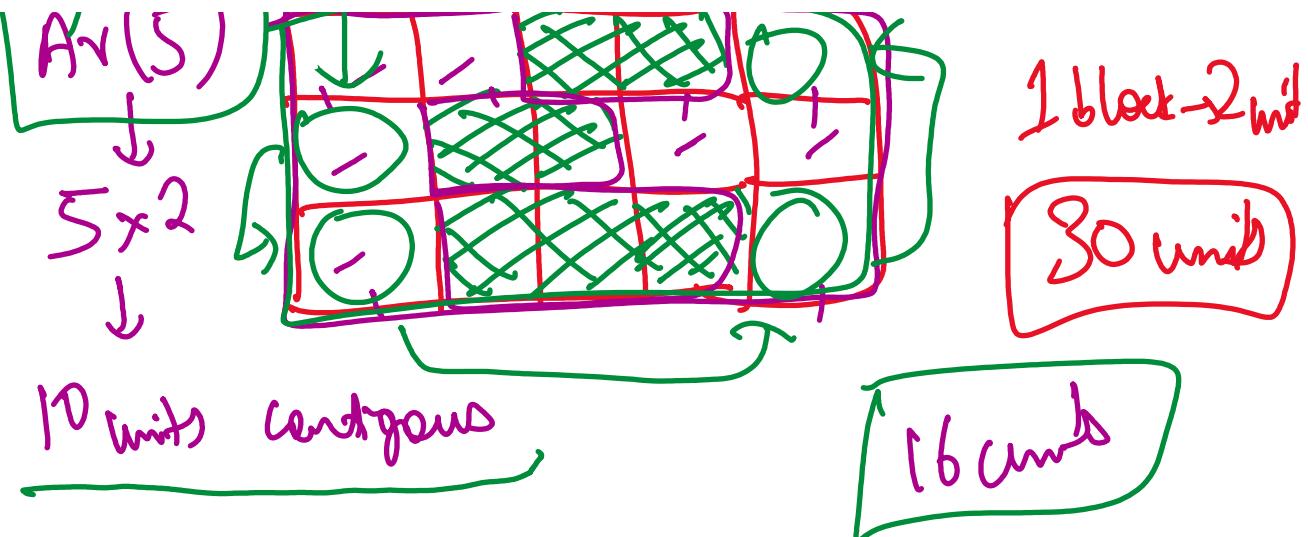
### Pros:

- No need for contiguous free memory
- Prevents memory fragmentation (scattered free memory that cannot always be used for large contiguous allocations)
- Flexible and grows/shrinks at runtime

### Cons:

- Extra memory required for pointers
- Since nodes are scattered, CPU caching is less effective
- Access is sequential since addresses are not predictable

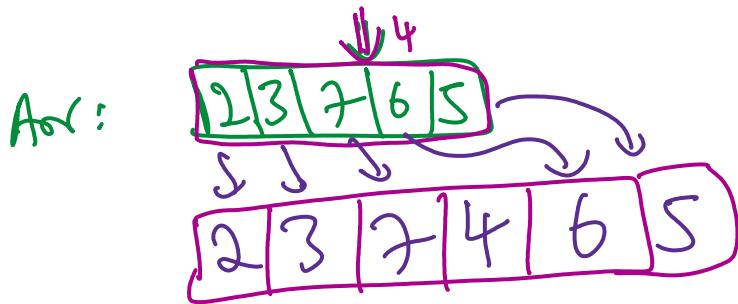




### 4.3 - Linked Lists vs Arrays/Lists

Saturday, August 30, 2025 11:21 AM

FEATURE	ARRAYS / LISTS	LINKED LISTS
Memory storage	Contiguous memory block ✓	Non-contiguous memory (nodes scattered) ✓
Indexing support	Supported (direct access by index) ✓	Not supported (no built-in indexing) ✓
Memory usage	Fixed (or resized in chunks when using Python list) ✓	Flexible (each node allocated separately) ✓
Pointer overhead	None since only data is stored	Extra memory for storing pointers/references in each node
Traversal direction	Only forward using index	Forward in singly, forward/backward in doubly linked lists
Cache friendliness	High (elements stored together, better CPU caching)	Low (nodes scattered across memory, poor cache performance)
Resizing	Needs reallocation when limit is reached	Naturally grows/shrinks with dynamic allocation
Implementation complexity	Simple (built-in in most languages like Python list)	More complex (requires manual implementation of nodes & pointers)
Fragmentation issues	Needs large contiguous block of memory	Works well even with fragmented memory
Example in Python	arr = [10, 20, 30] ✓	Custom <u>Node</u> and <u>LinkedList</u> class
Use cases	Best when frequent random access and fewer insertions/deletions	Best when frequent insertions/deletions and size is dynamic

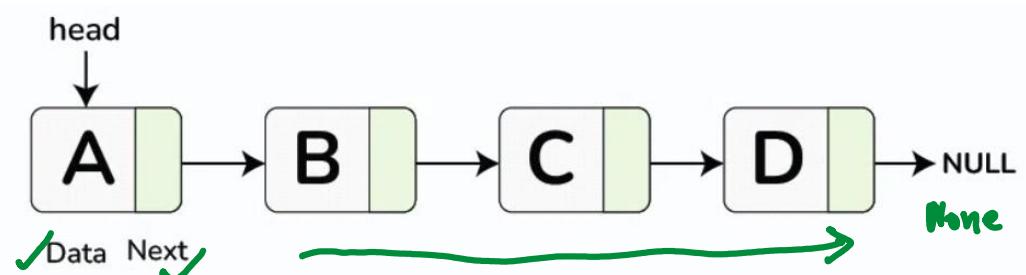


## 4.4 - Types of Linked Lists

Saturday, August 30, 2025 11:45 AM

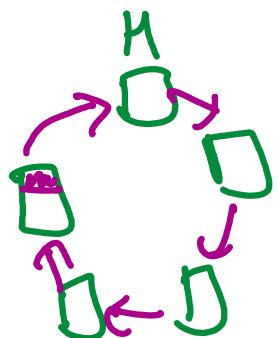
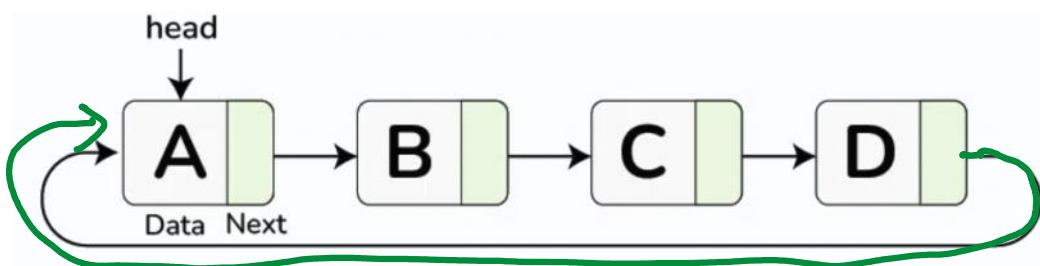
### 1. Singly Linked List:

- Each node has two parts:
  - **Data:** stores the value
  - **Next:** pointer/reference to the next node
- The last node points to **None**
- Traversal is only in one direction (forward)



### 2. Circular Singly Linked List:

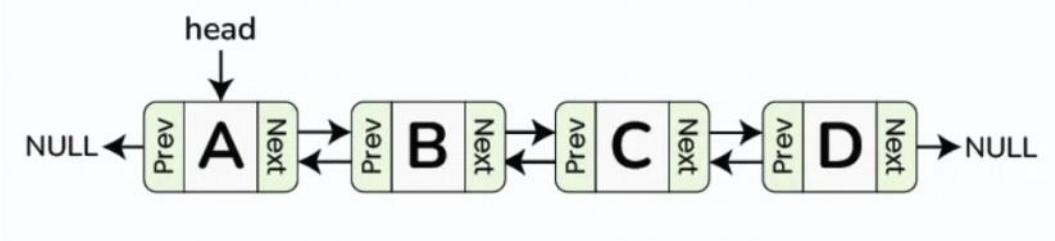
- Similar to singly linked list, but last node points back to **head** instead of **None**
- Forms a circular chain
- Traversal can continue indefinitely only in forward direction



### 3. Doubly Linked List:

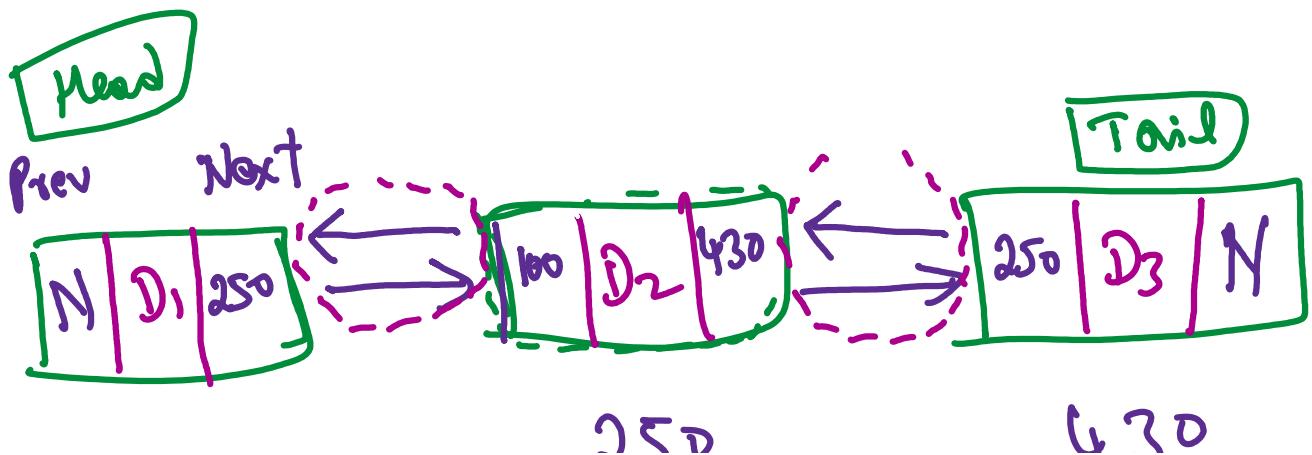
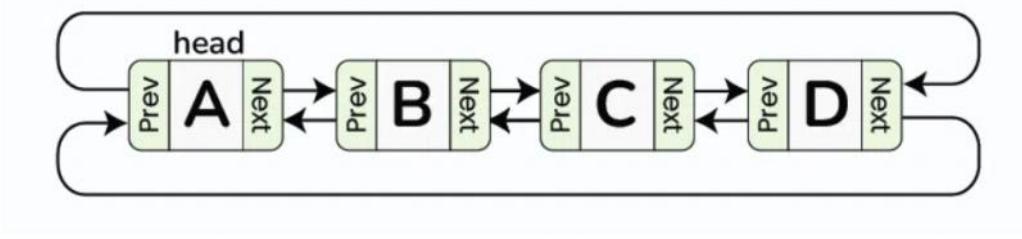
- Each node has three parts:
  - **Prev:** pointer to previous node

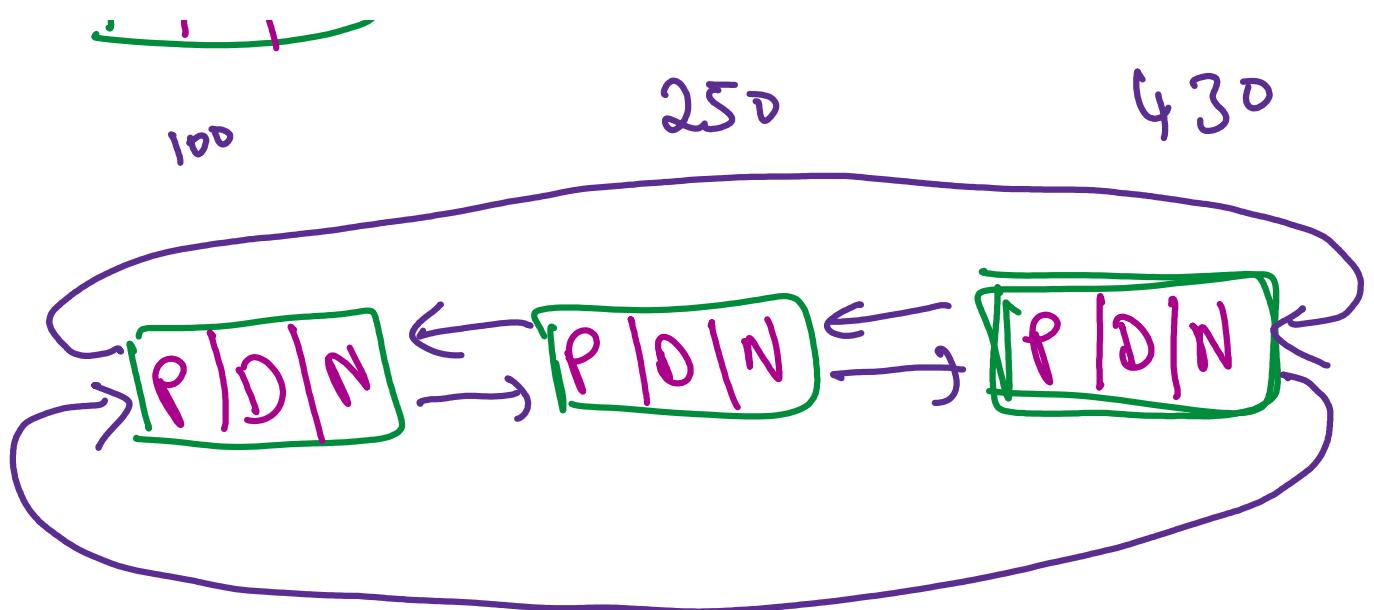
- **Data:** stores the value
- **Next:** pointer to next node
- Allows bidirectional traversal (forward & backward)



#### 4. Circular Doubly Linked List:

- Combines features of circular + doubly linked lists
- Last node points back to **head**, and **head's Prev** points to last node
- Flexible traversal in both directions
- This is the most complex implementation



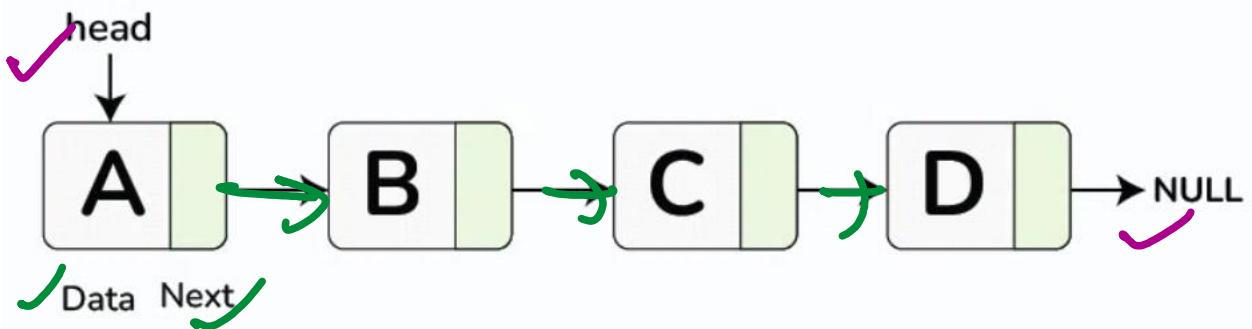




## 4.5 - Singly Linked List

Saturday, August 30, 2025 12:06 PM

- A **Singly Linked List** is the simplest form of a linked list
- Each node has two fields:
  - **Data:** stores the element
  - **Next:** pointer/reference to the next node
- The first node is called the **head**
- The last node's next points to **None**, marking the end of the list
- Traversal is only in one direction (forward)



### Pros:

- Dynamic size (can grow/shrink easily)
- Efficient insertion/deletion at the beginning or middle (if pointer is known)

### Cons:

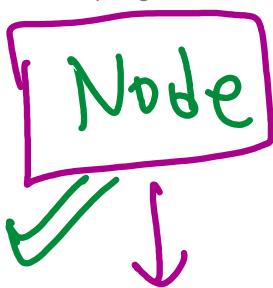
- Cannot traverse backward
- Sequential access only
- Extra memory for pointer



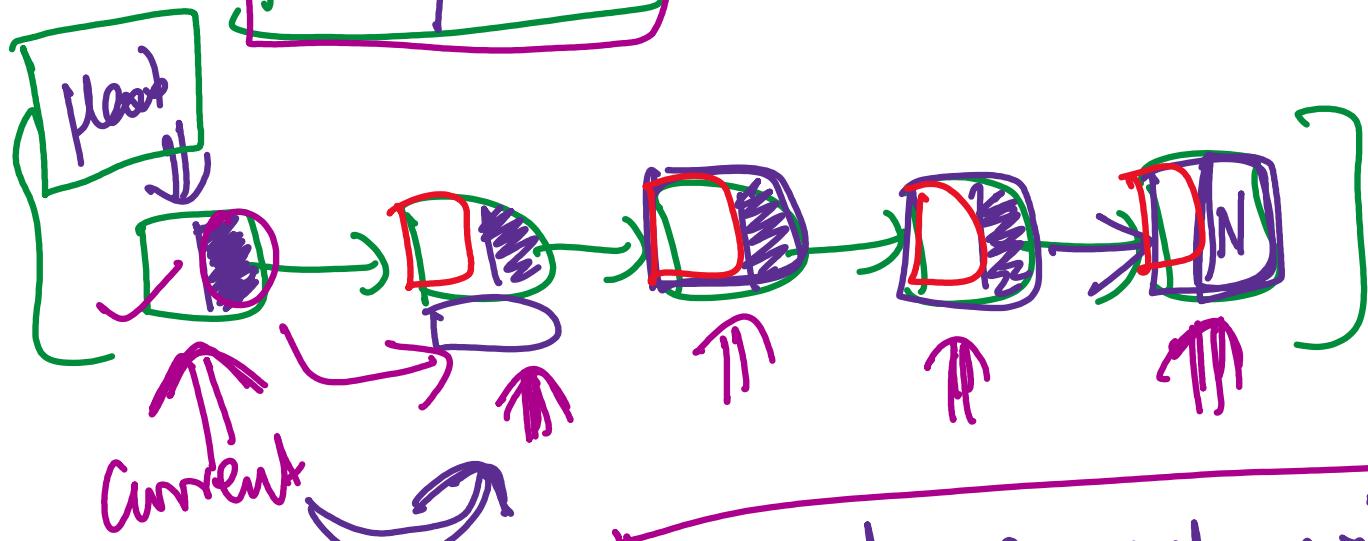
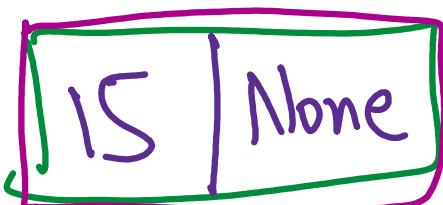
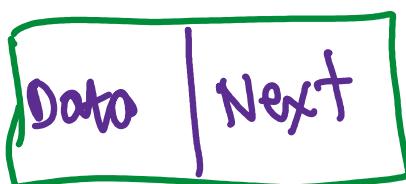
### 4.5.1 - SLL Initial Setup

Saturday, August 30, 2025

12:10 PM



Singly linked list



Current = Current.next

length = 0x2  
3 x 5

length  $\rightarrow$  0

head  $\rightarrow$  None  
current  $\uparrow$











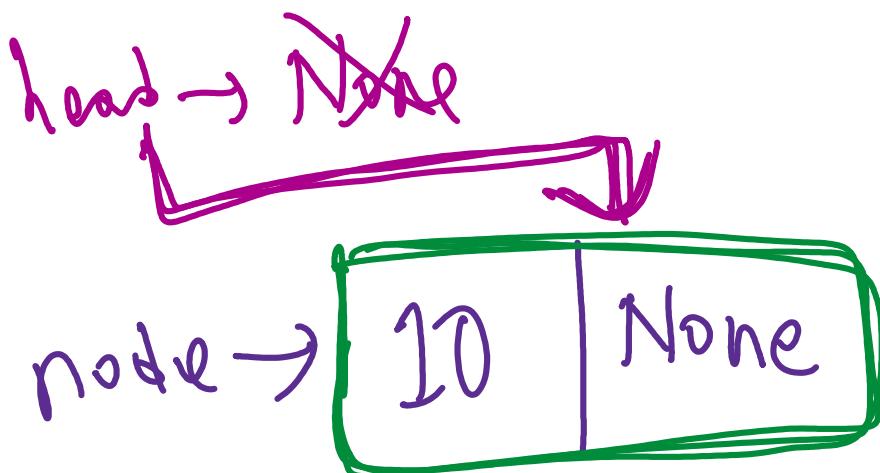
#### 4.5.2 - SLL Insertion

Saturday, August 30, 2025 12:31 PM

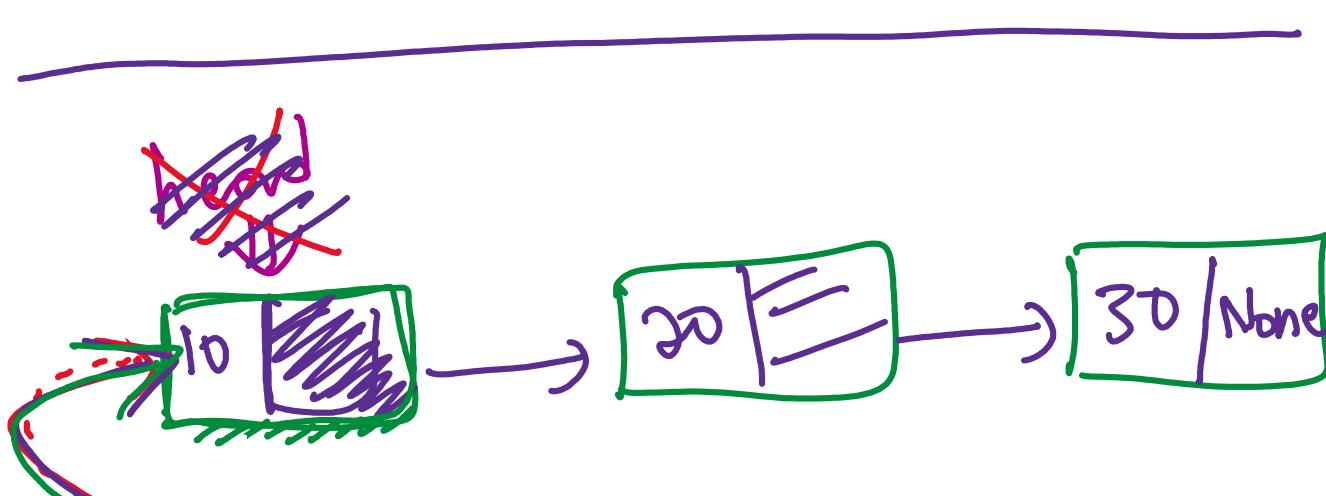
## 1. Insertion in SLL at beginning

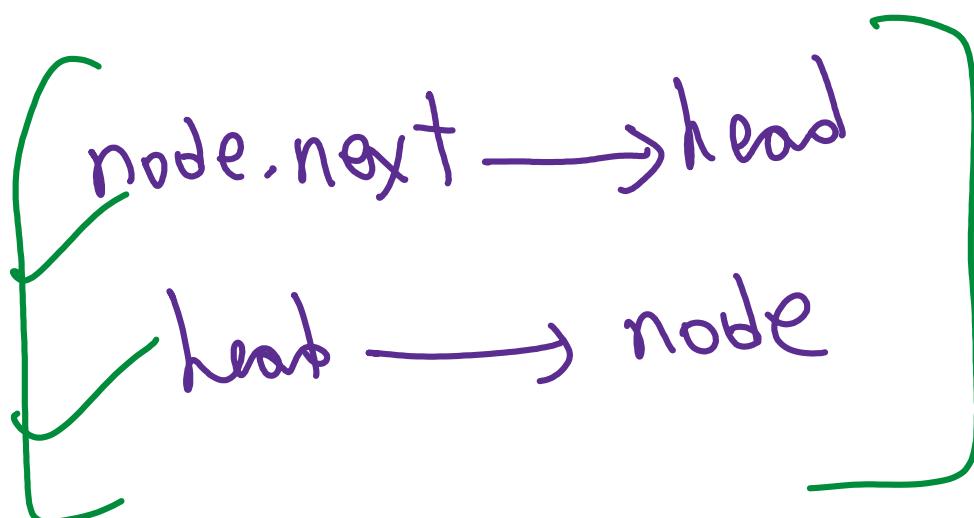
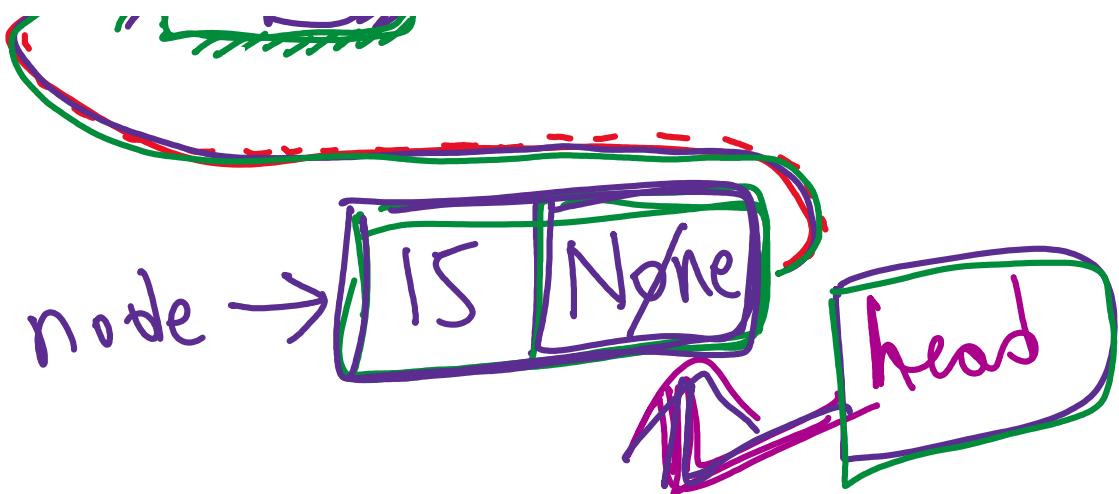
↳ list is empty

↳ list is not empty



head → node



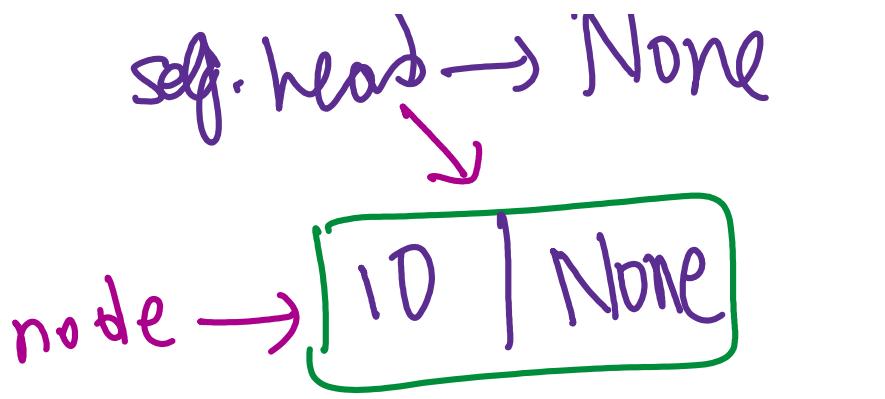


## 2. Insertion at End

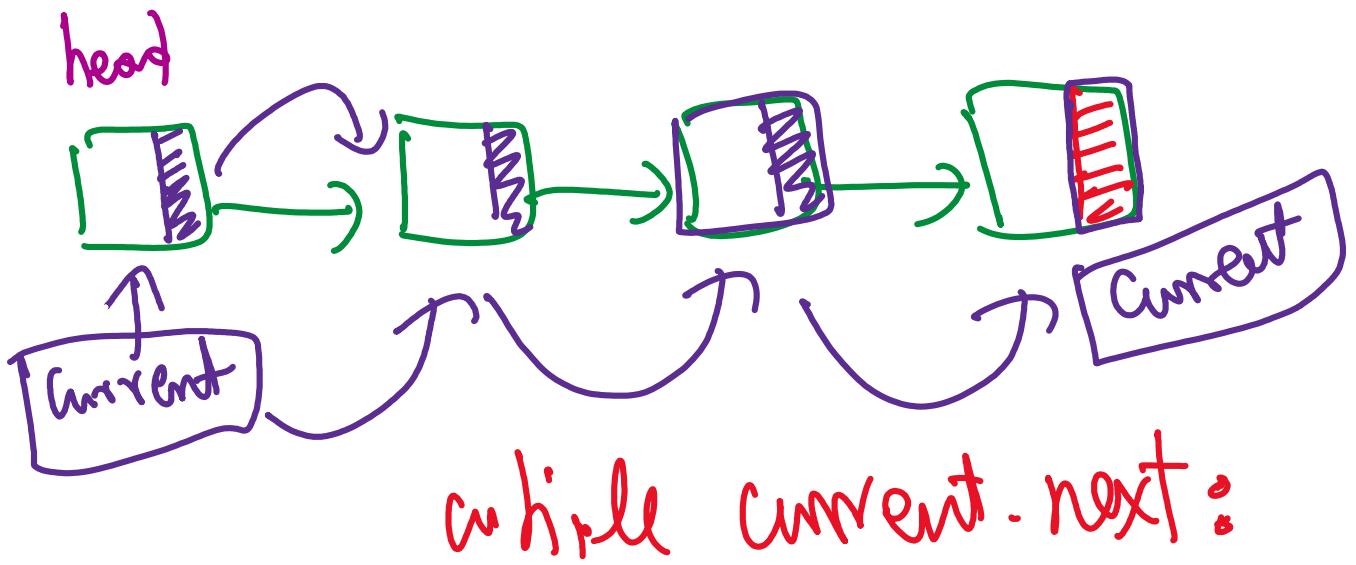
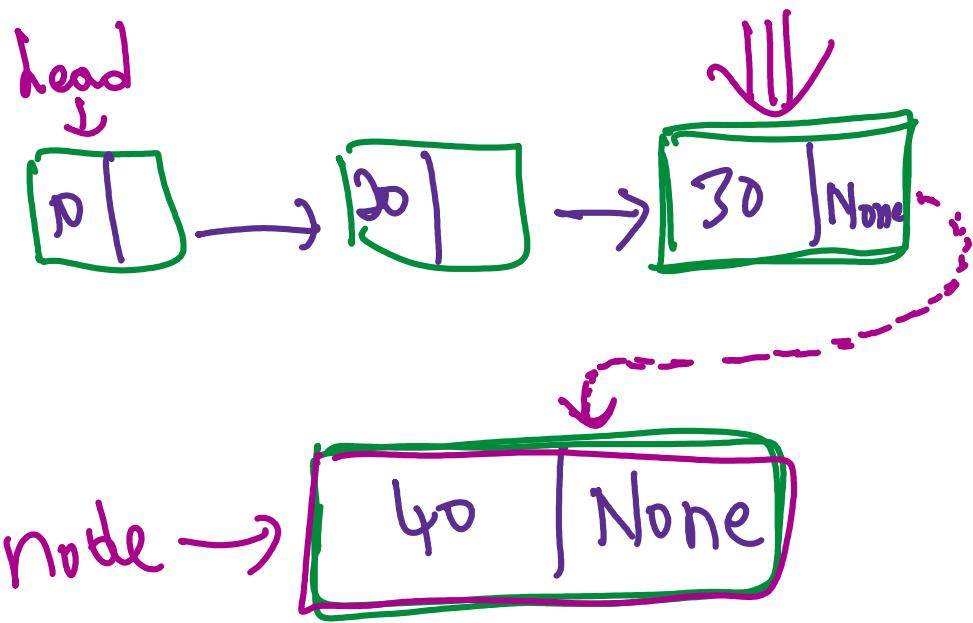
↳ if list empty

↳ if list not empty

$self.head \rightarrow None$



*head → node*



while current.next :

current = current.next

current.next = node

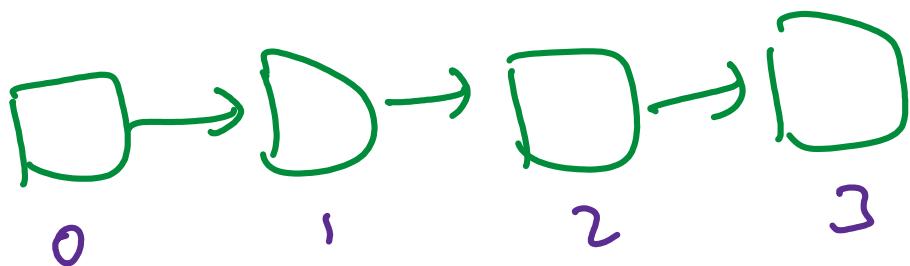
---

### 3. Insertion in the Middle

Value

position (0-indexed)

n=4

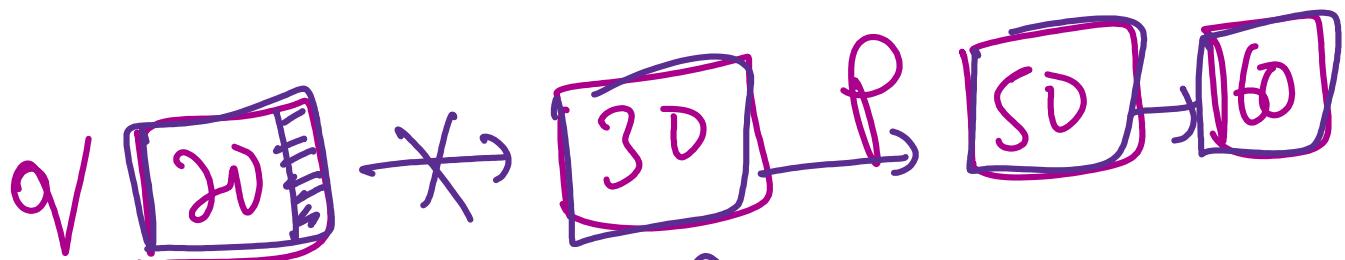
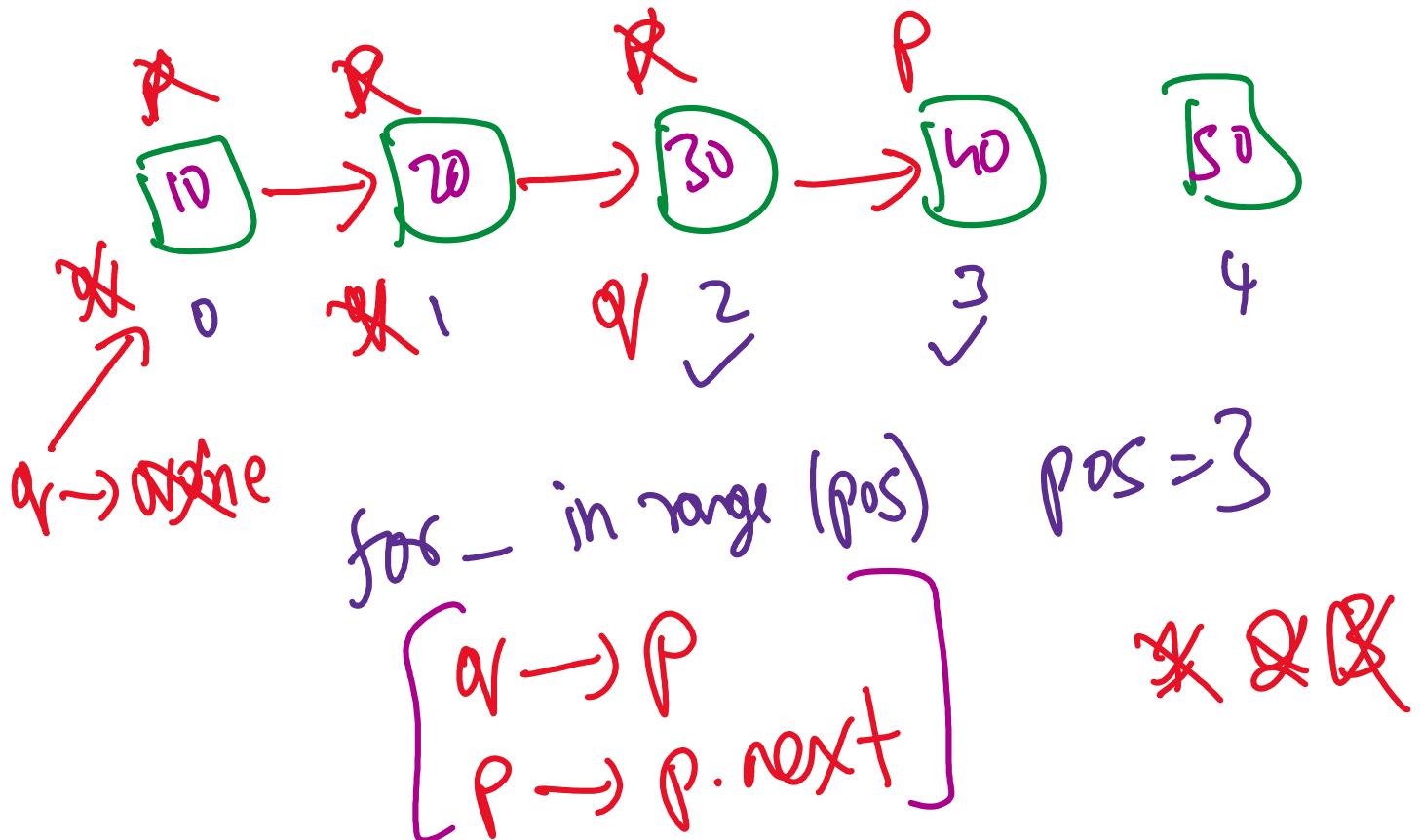
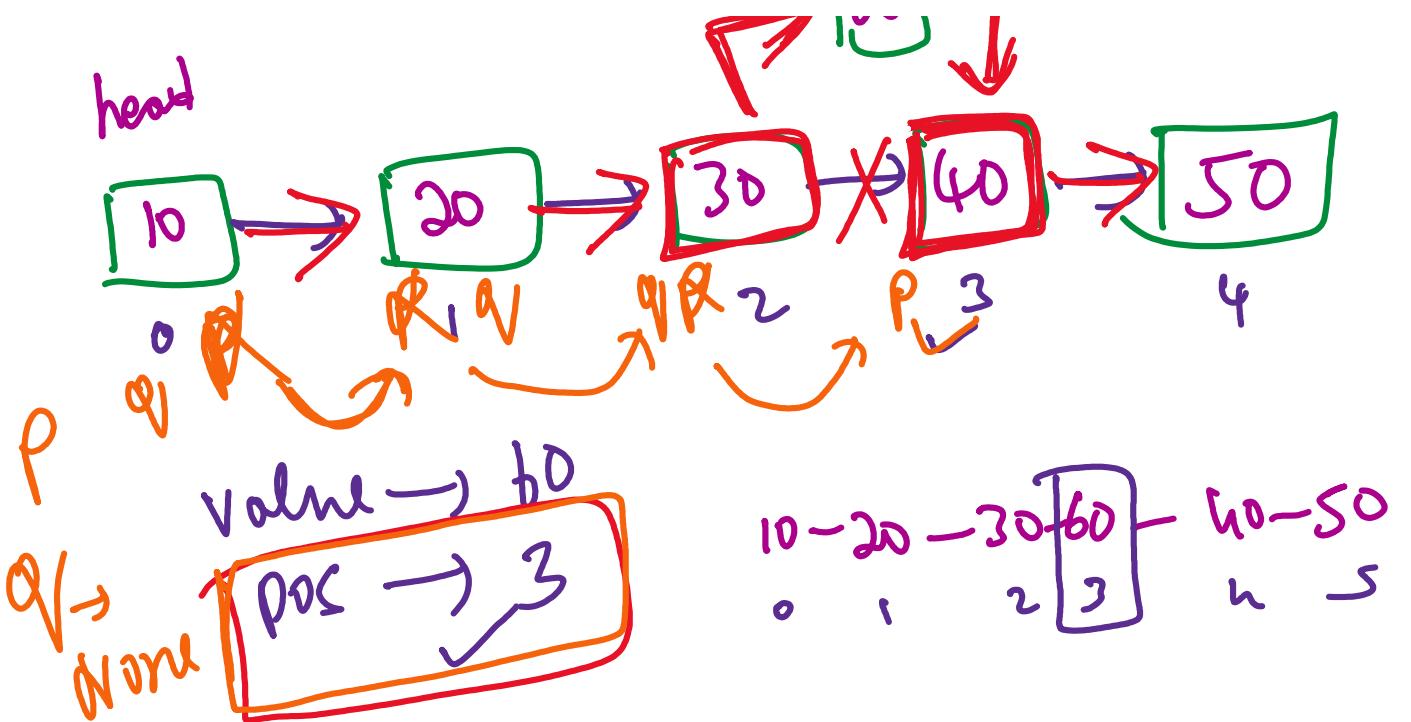


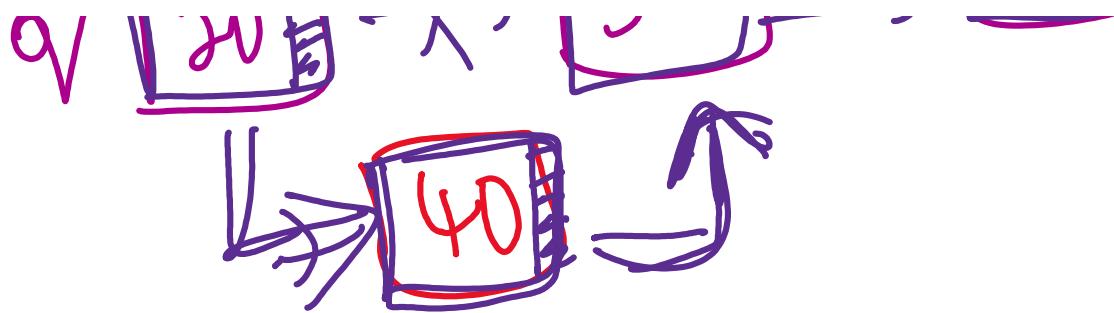
pos=0, insertion at beginning

pos=n, insertion at end

head







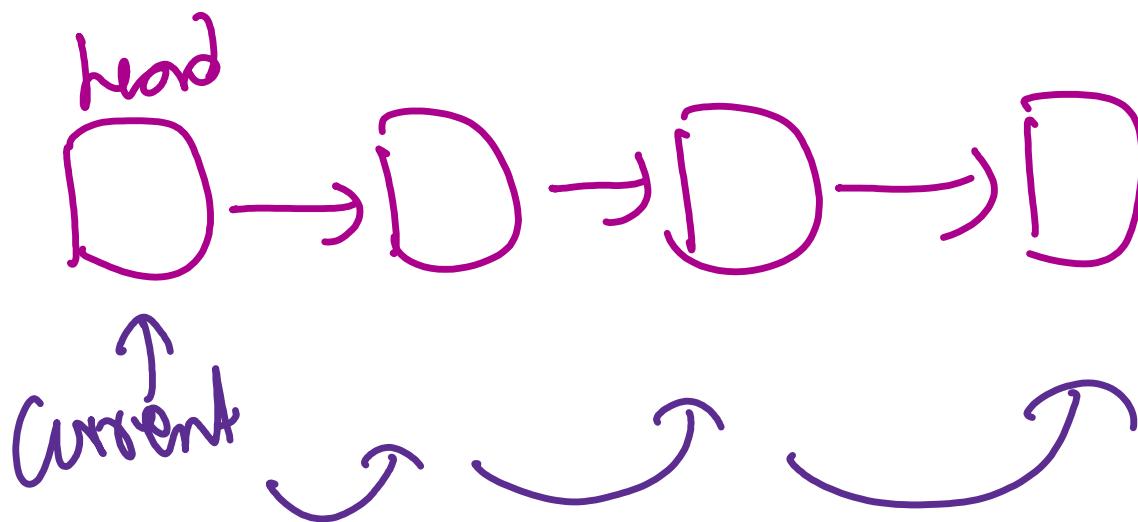
[  
q. next  $\rightarrow$  node  
node .next  $\rightarrow$  p  
]

#### 4.5.3 - SLL Search

Saturday, August 30, 2025 12:31 PM

pos = 0

pos ← pos + 1



current ← current.next



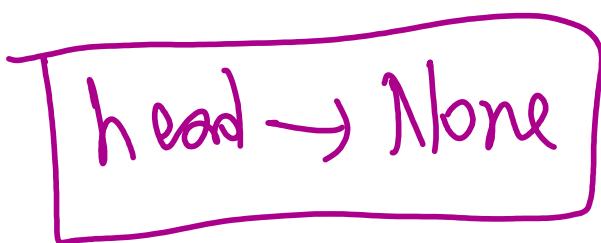
#### 4.5.4 - SLL Deletion

Saturday, August 30, 2025 12:31 PM

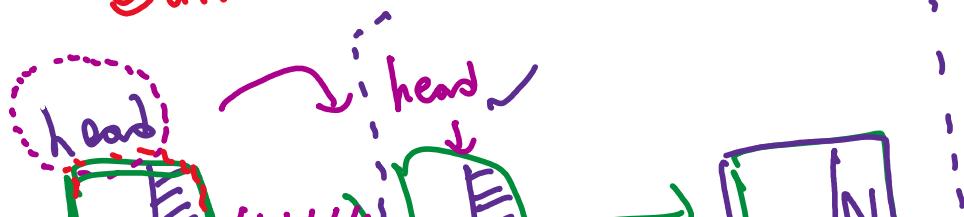
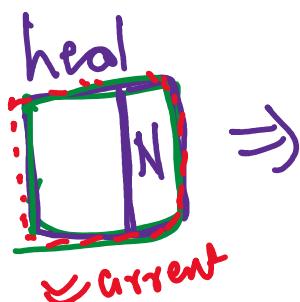
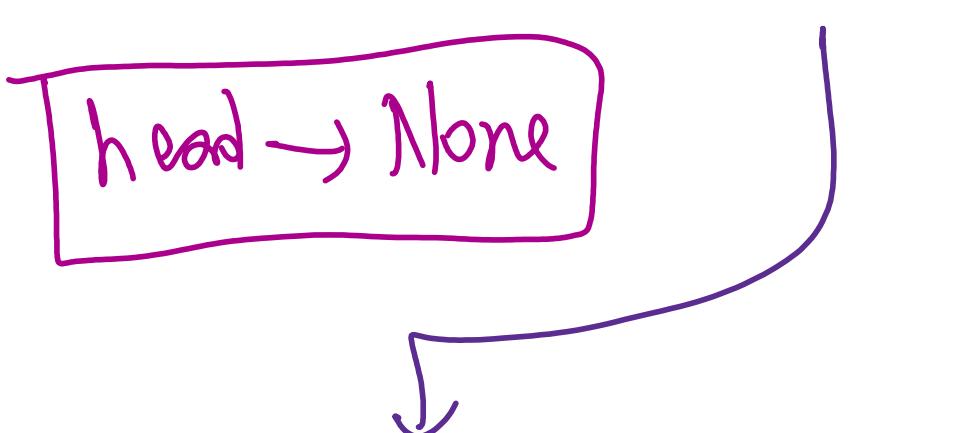
1. At begin
2. At end
3. At middle

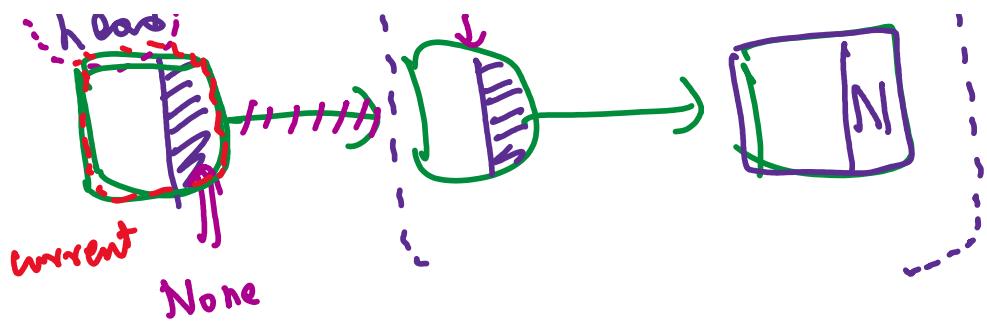
#### 1. Beginning

i) list is empty



ii) Not empty





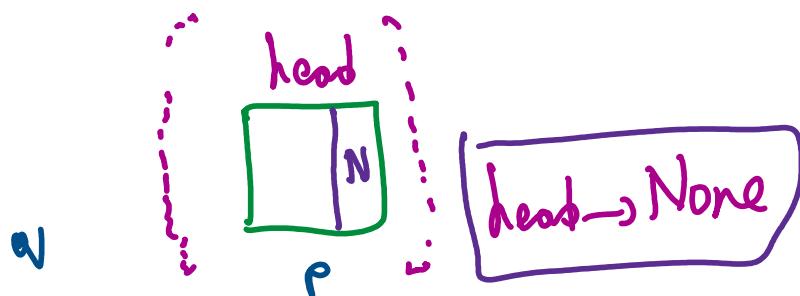
$\text{current} \leftarrow \text{head}$   
 $\text{head} \leftarrow \text{current.next}$   
 $\text{current.next} \leftarrow \text{None}$

## 2. At the end

i) Empty  $\rightarrow \text{print()}$

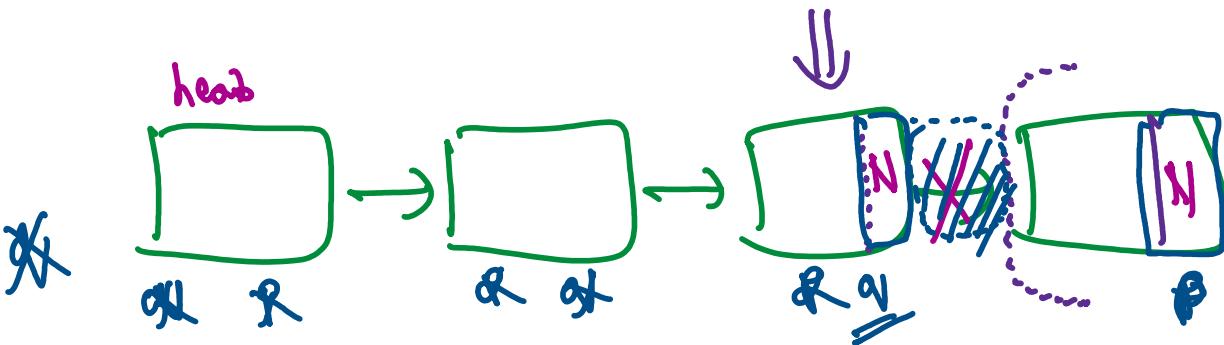
ii) Not Empty

Single Element



... Continue

L, Multiple Elements  
(next → None)



"next" attribute of 2<sup>nd</sup> last element

$p \rightarrow \text{head}$

$q \rightarrow \text{None}$

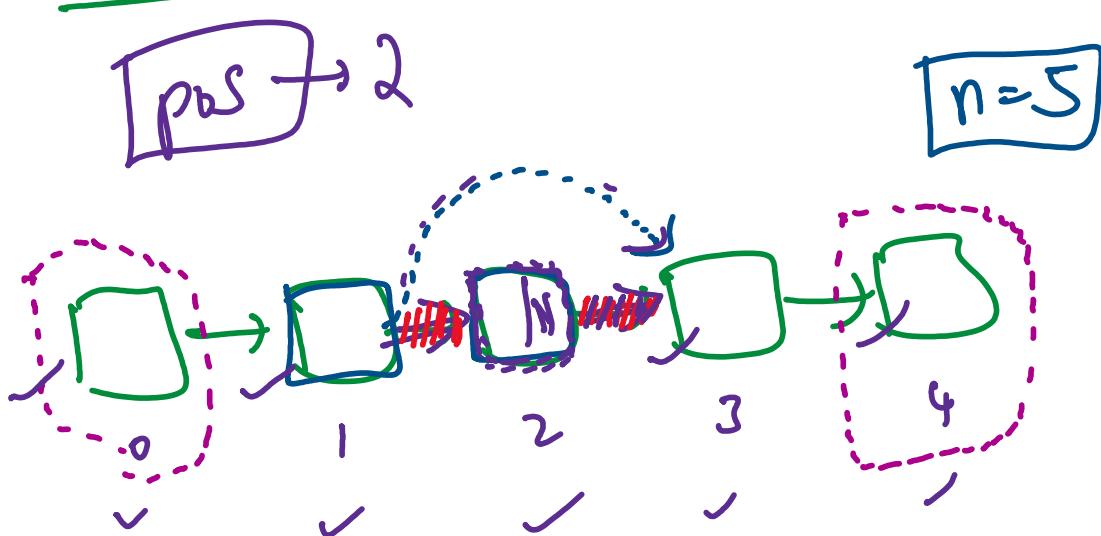
$[q \rightarrow p]$        $[p \rightarrow p.\text{next}]$  while  $p.\text{next} :$

$q.\text{next} = \text{None}$

3. In the middle o

$T_{n-2} \rightarrow 2$

$T_{n-1}$



$(pos < 0)$  or  $(pos \geq n)$

↳ Invalid position

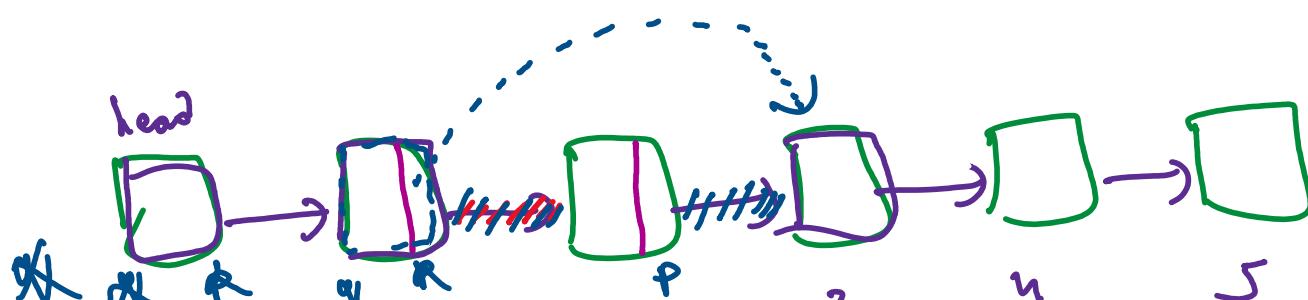
$pos \rightarrow 0$

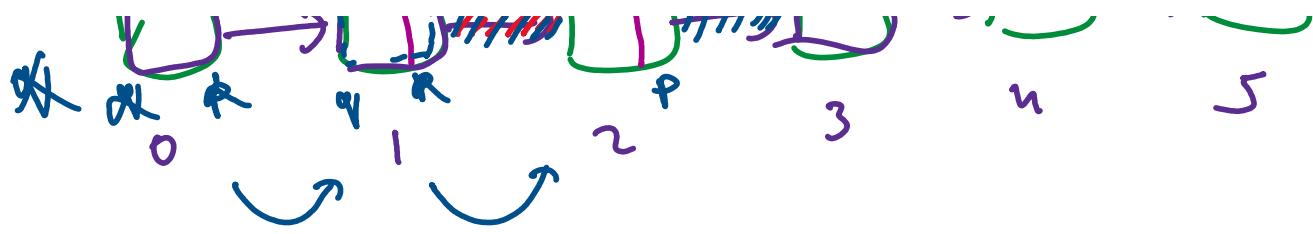
↳ `delete_at_begin()`

$pos \rightarrow n-1$

↳ `delete_at_end()`

else:





$\text{pos} \leftarrow 2$

$p \rightarrow \text{head}, q = \text{None}$

$\begin{bmatrix} q \rightarrow p \\ p \rightarrow p.\text{next} \end{bmatrix}$

for range(pos):

$\text{range}(2)$   
 $\hookrightarrow 0, 1$

$\begin{bmatrix} q.\text{next} \rightarrow p.\text{next} \\ p.\text{next} \rightarrow \text{None} \end{bmatrix}$



















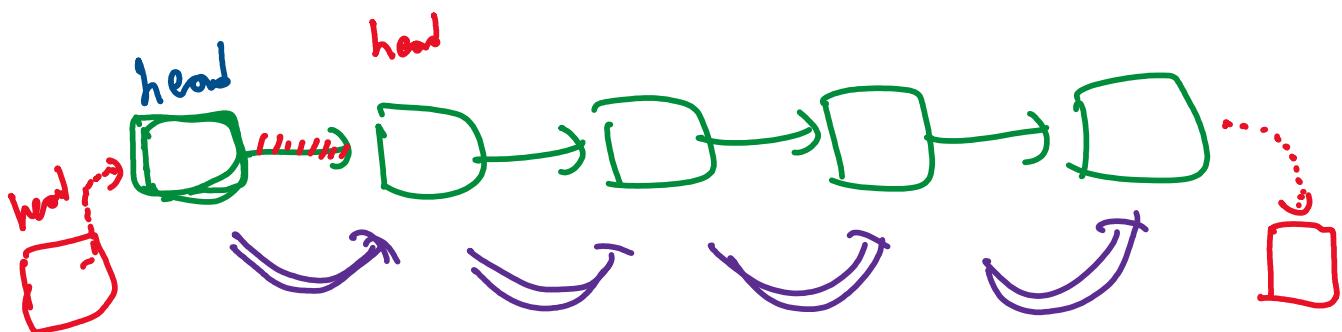




#### 4.5.5 - SLL Complexity Analysis

Saturday, August 30, 2025 12:32 PM

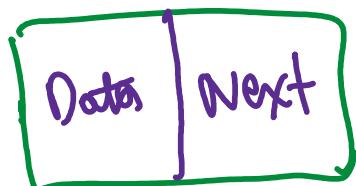
OPERATION	WORST CASE	REASONING
Traversal	$O(n)$ ✓	Must visit each node once
Search	$O(n)$ ✓	May need to scan the entire list
Insert at Beginning	$O(1)$ ✗	Just adjust <b>head</b> pointer.
Insert at End	$O(n)$ ✗	Need to traverse till last node
Insert at Middle	$O(n)$ ✓	Need to traverse to position
Delete at Beginning	$O(1)$ ✗	Just move <b>head</b> to next node
Delete at End	$O(n)$ ✓	Need to traverse till last node
Delete at Middle	$O(n)$ ✗	Traverse till position, then unlink node
Space per Node	$O(1)$ ✗	Each node stores data + one pointer ( <b>next</b> )
Overall Space Usage	$O(n)$ ✗	Needs memory for all nodes



$$5 \rightarrow 4$$

$$n \rightarrow n-1$$

$$\boxed{O(n)}$$



Node → 2 units

list  $\rightarrow$  "n" nodes

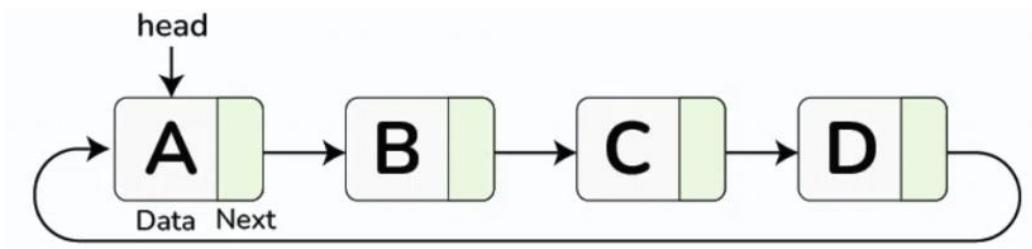
$\rightarrow 2n$

$\rightarrow n$   
 $\rightarrow O(n)$

## 4.6 - Circular Singly Linked List

Tuesday, September 9, 2025 9:27 AM

- A **Circular Singly Linked List** is a variation of a singly linked list
- Each node has two fields:
  - **Data**: stores the element
  - **Next**: pointer/reference to the next node
- The first node is called the **head**
- Unlike a normal singly linked list, the last node's **next** points back to the **head** node, **forming a circular structure**
- Traversal is only in one direction (forward), but since it is circular, starting from any node allows you to reach all other nodes

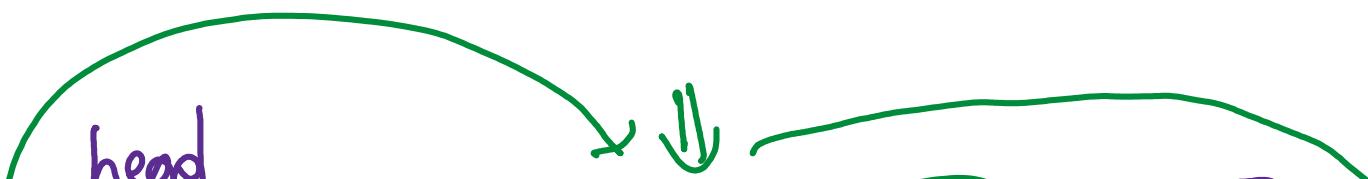


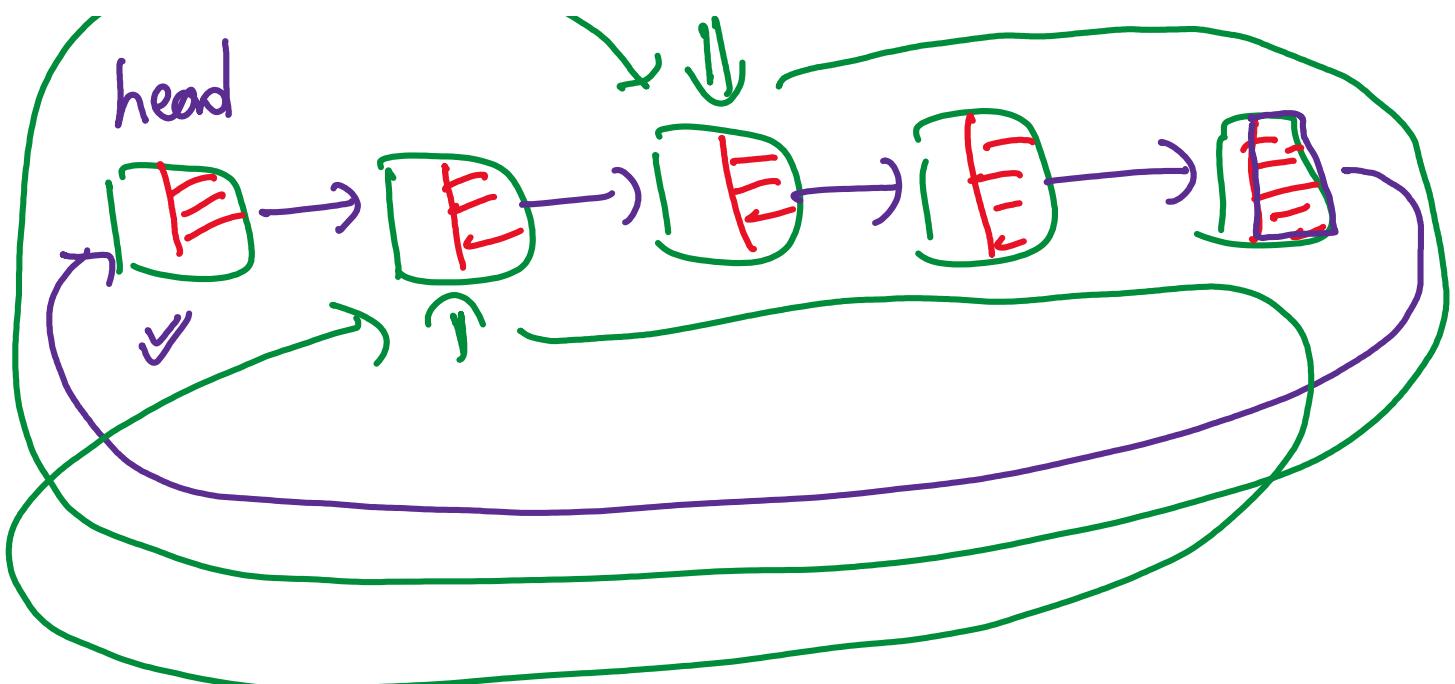
### Pros:

- Efficient for circular traversal (e.g., implementing round-robin scheduling)
- Can start traversal from any node and still access the entire list
- No **None** at the end, so memory usage is more consistent

### Cons:

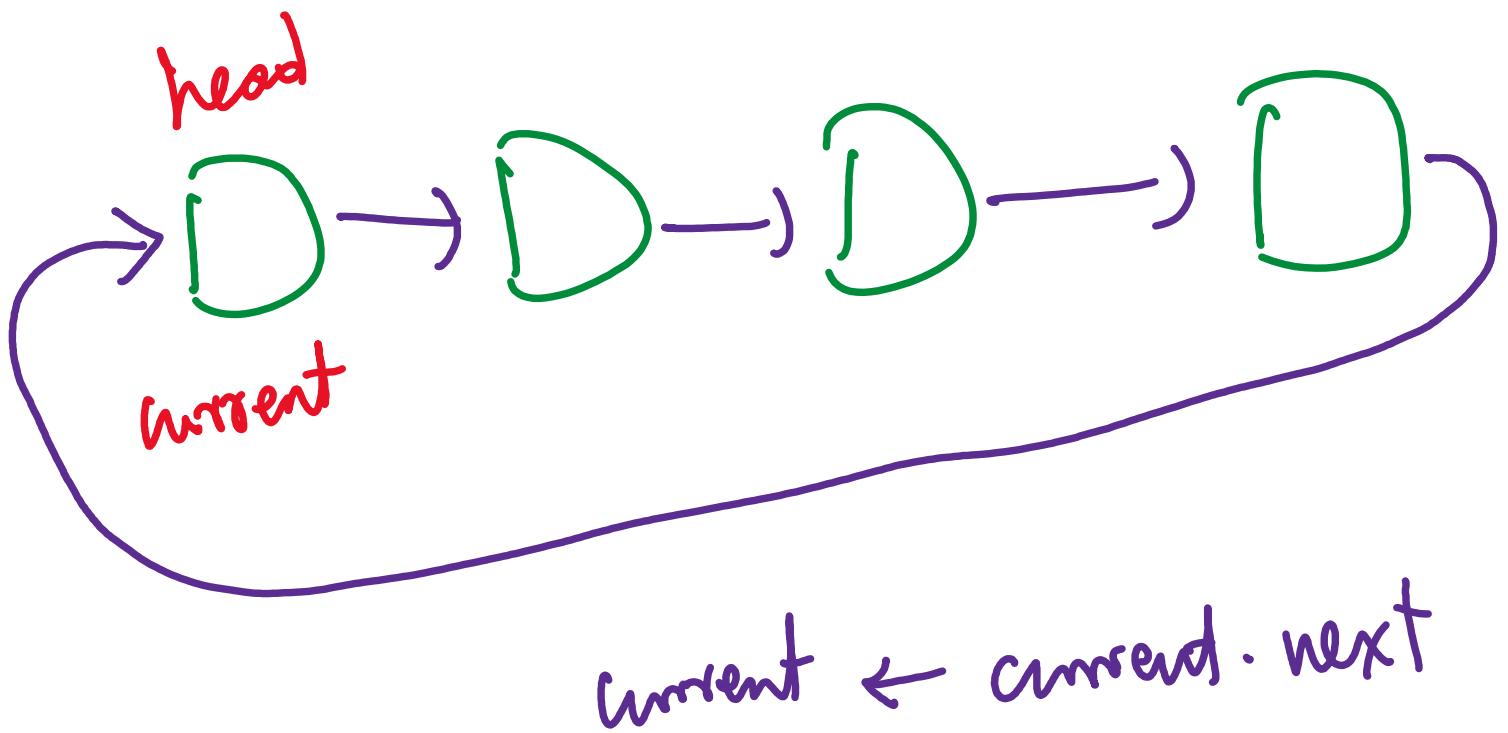
- More complex insertion and deletion logic compared to a simple singly linked list
- Risk of infinite loops if traversal stopping condition isn't handled properly
- Slightly harder to debug and implement than a standard singly linked list





#### 4.6.1 - CSLL Initial Setup

Tuesday, September 9, 2025 9:34 AM

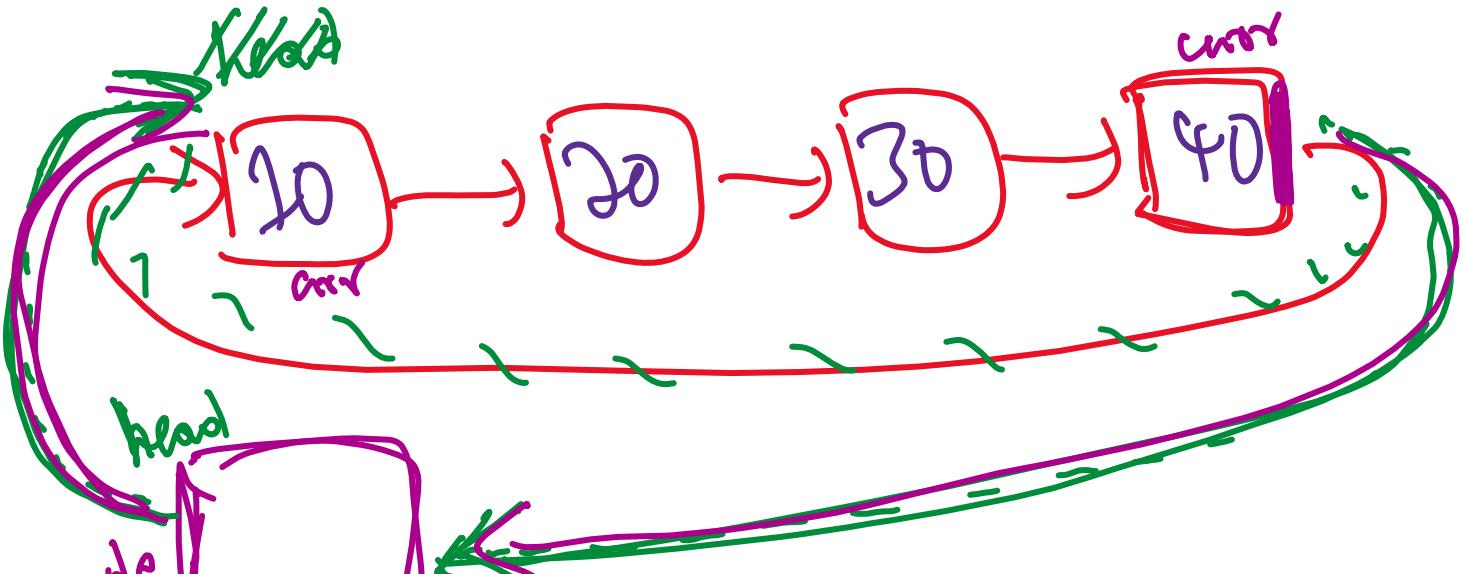
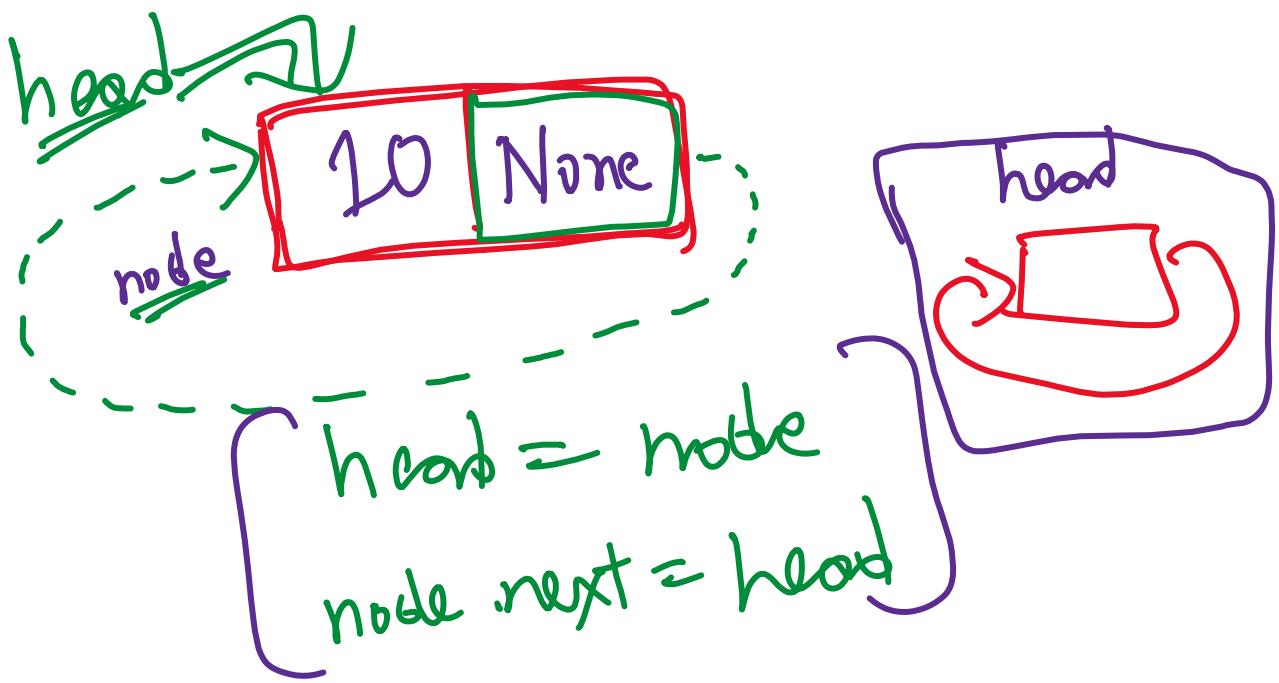


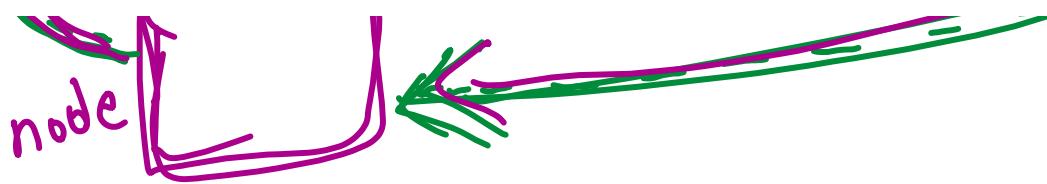
#### 4.6.2 - CSLL Insertion

Tuesday, September 9, 2025 9:34 AM

## 1. Insertion at Beginning

head → None





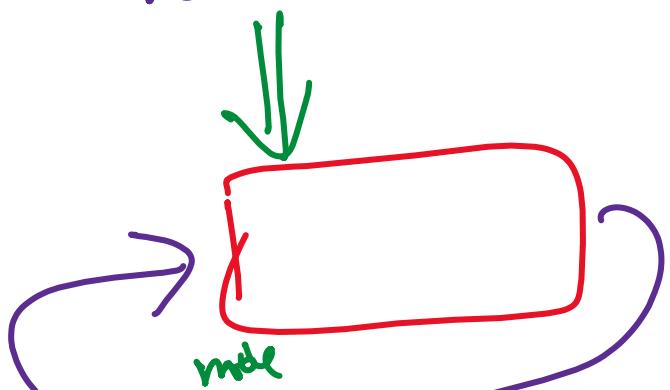
current.next = node

node.next = head

head = node

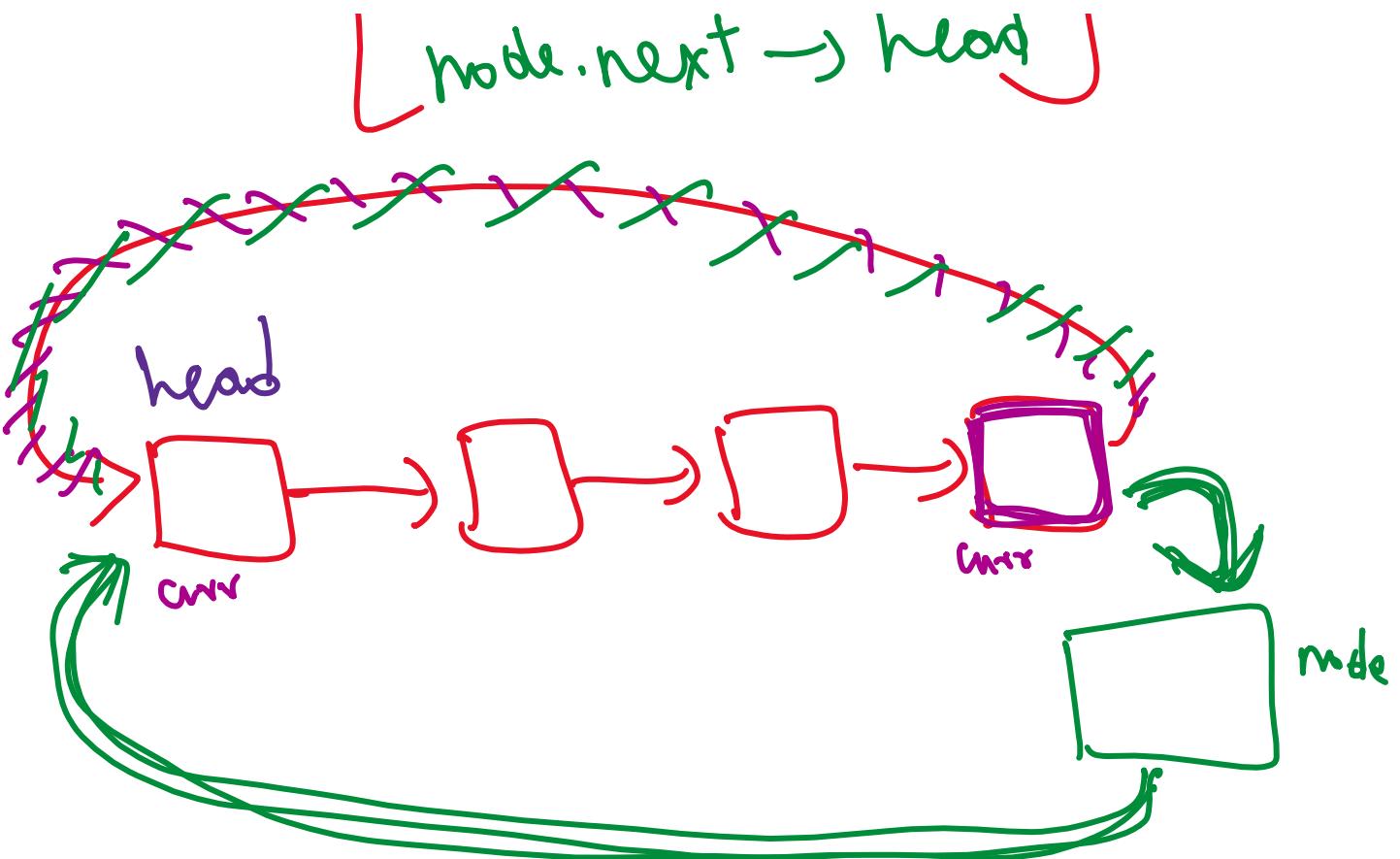
## 2. Insertion at End

head → None



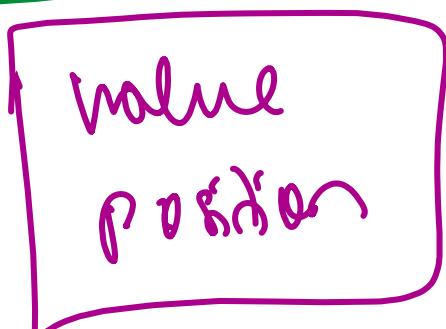
head → node

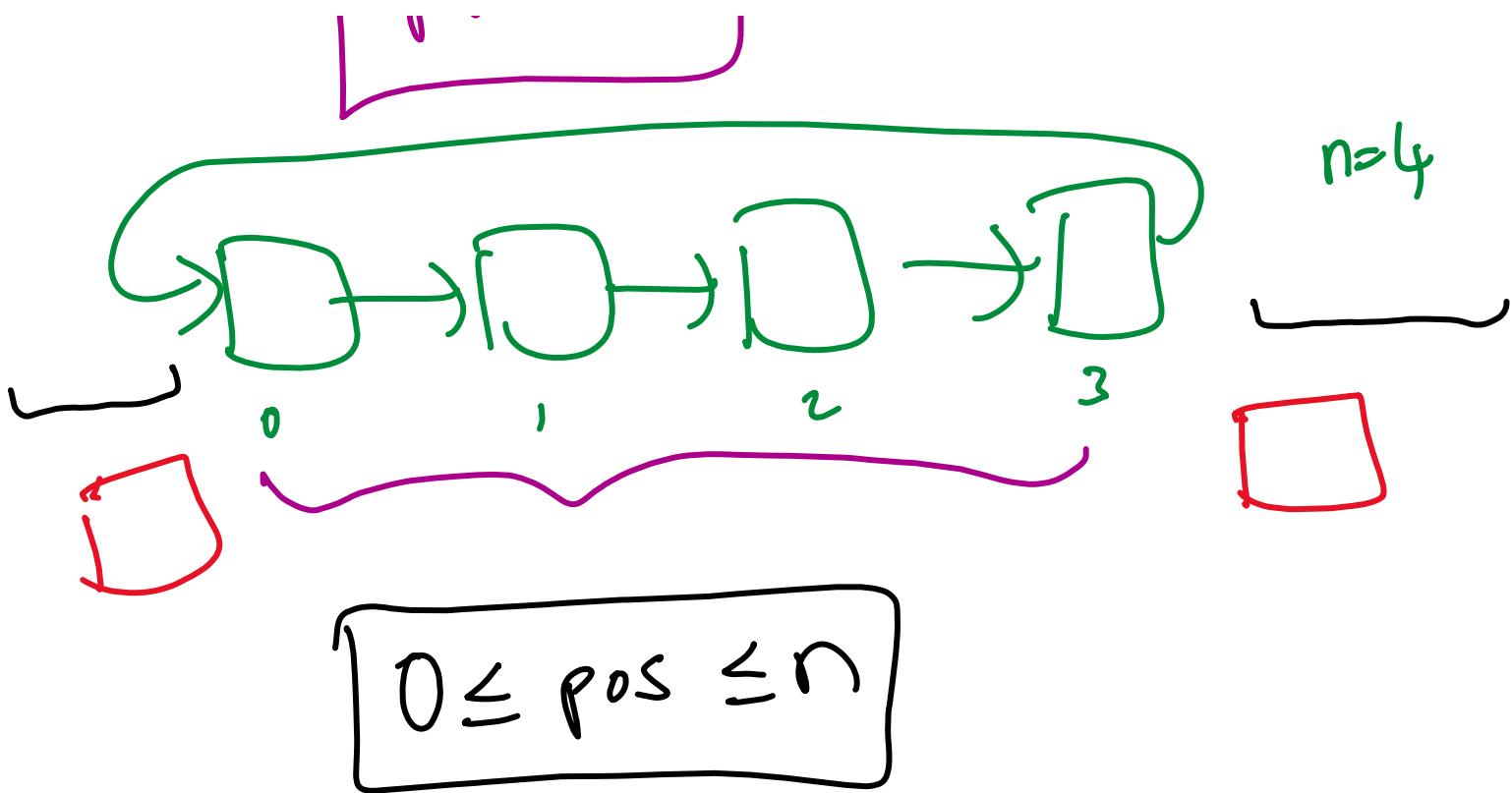
node.next → head



$\text{current.next} = \text{node}$   
 $\text{node.next} = \text{head}$

### 3. Insertion in Middle

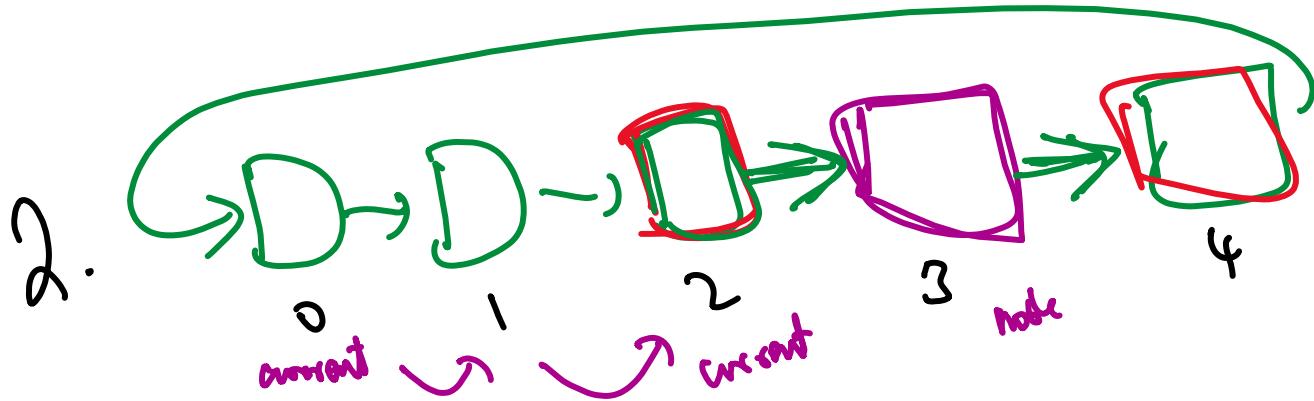
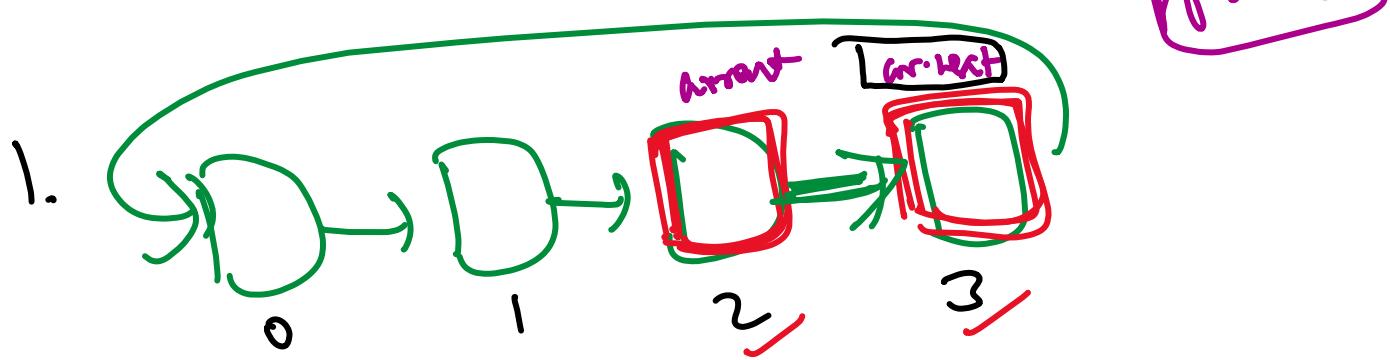




1.  $(\text{pos} < 0) \text{ or } (\text{pos} > n)$ 
  - ↳ Invalid index
2.  $\text{pos} = 0$ 
  - ↳ insert-beg
3.  $\text{pos} = n$ 
  - ↳ insert-end
4. else:
 

$\text{pos} = ?$

#### 4. Chs:



pos, pos-1

current  $\rightarrow$  pos-1  
current.next  $\rightarrow$  pos

range (pos-1)

node.next = current.next  
1 + - node

current.next = node











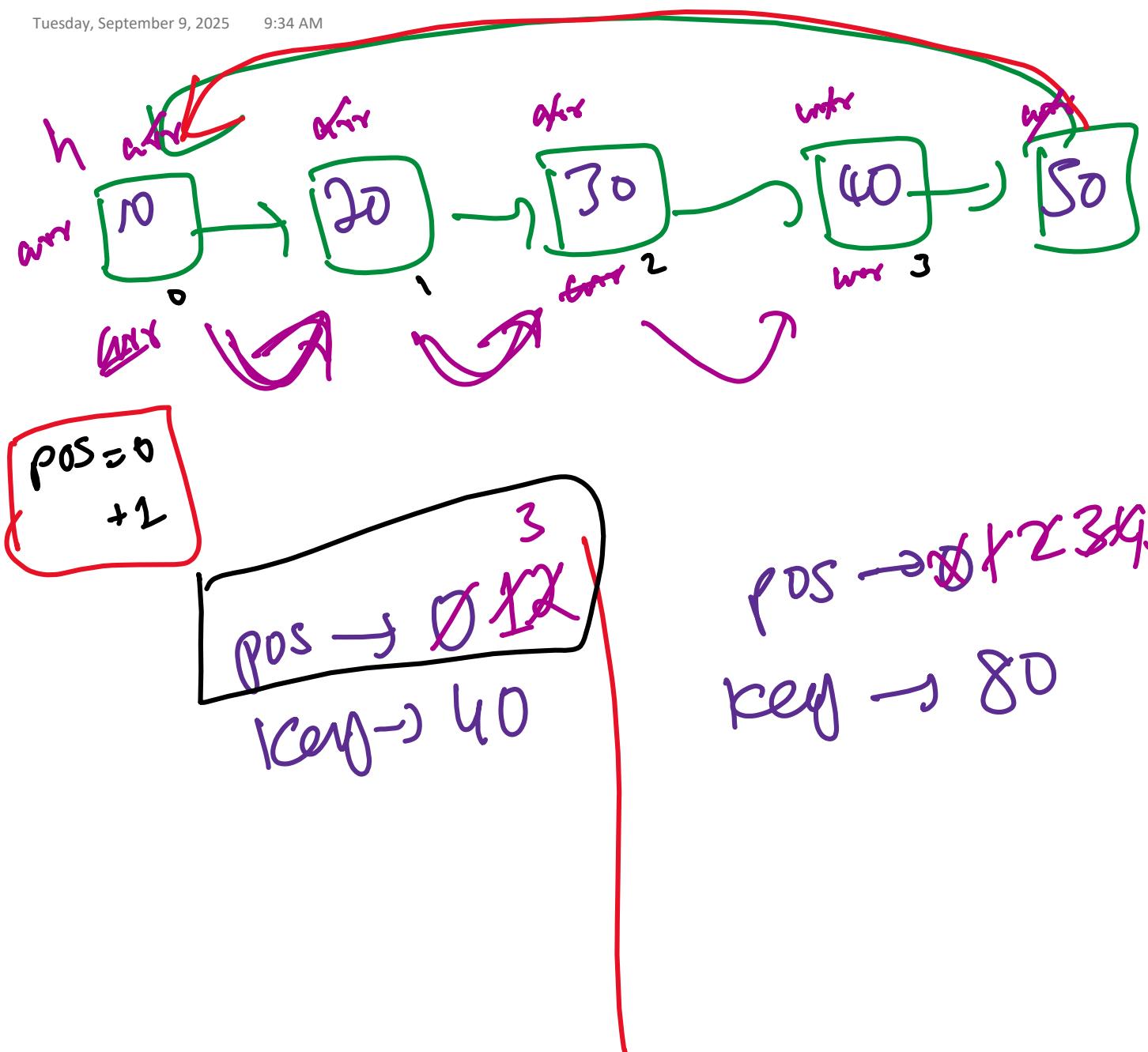






### 4.6.3 - CSLL Search

Tuesday, September 9, 2025 9:34 AM







#### 4.6.4 - CSLL Deletion

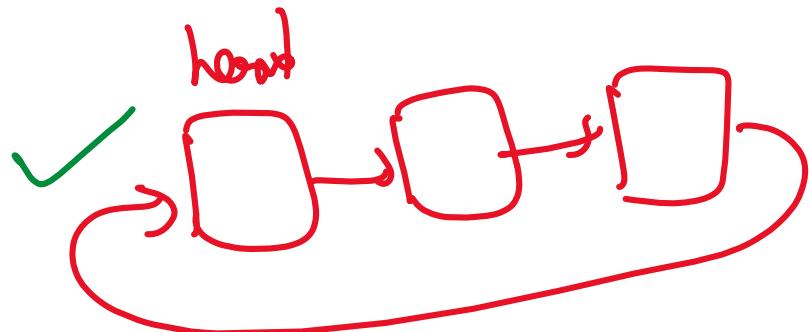
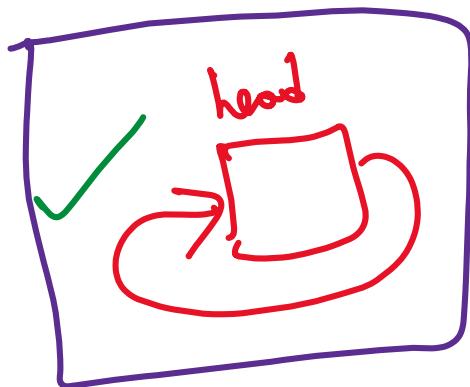
Tuesday, September 9, 2025 9:34 AM

## I. Beginning

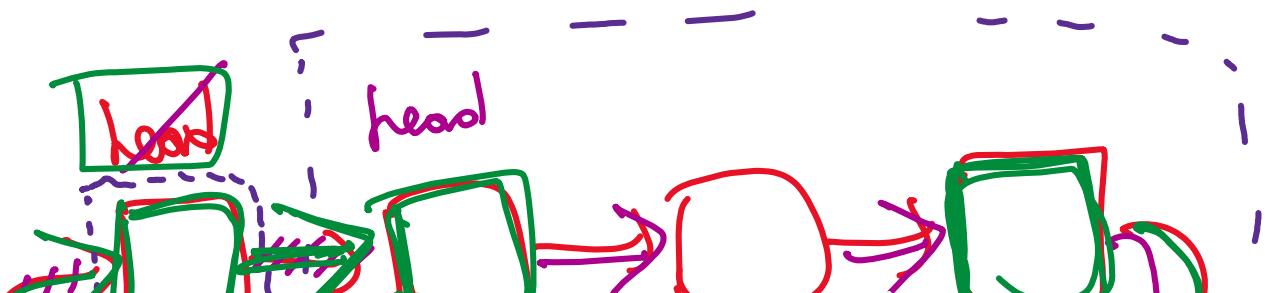
### i) Empty

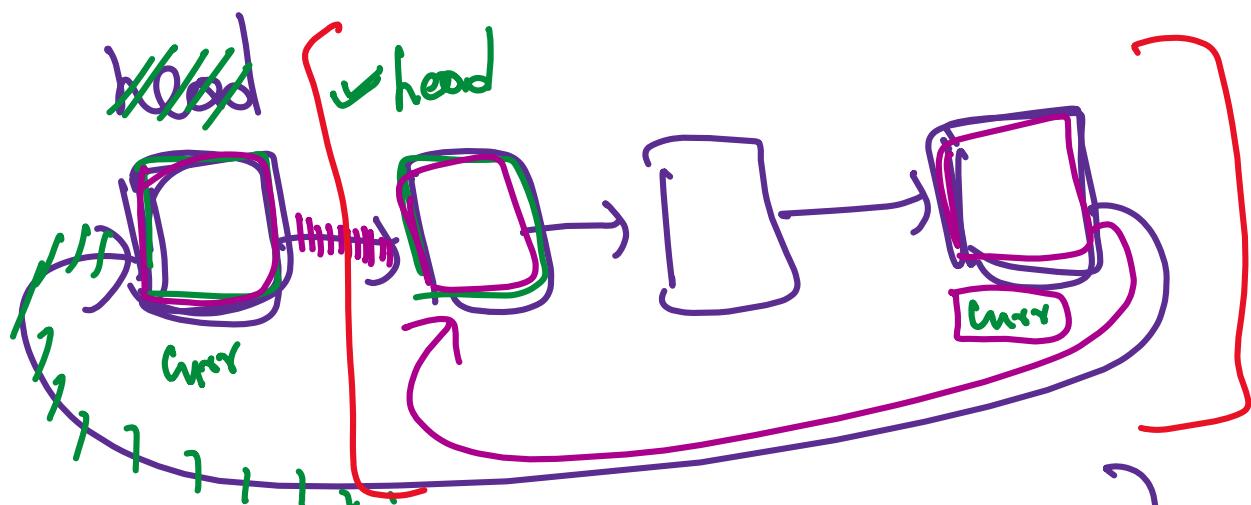
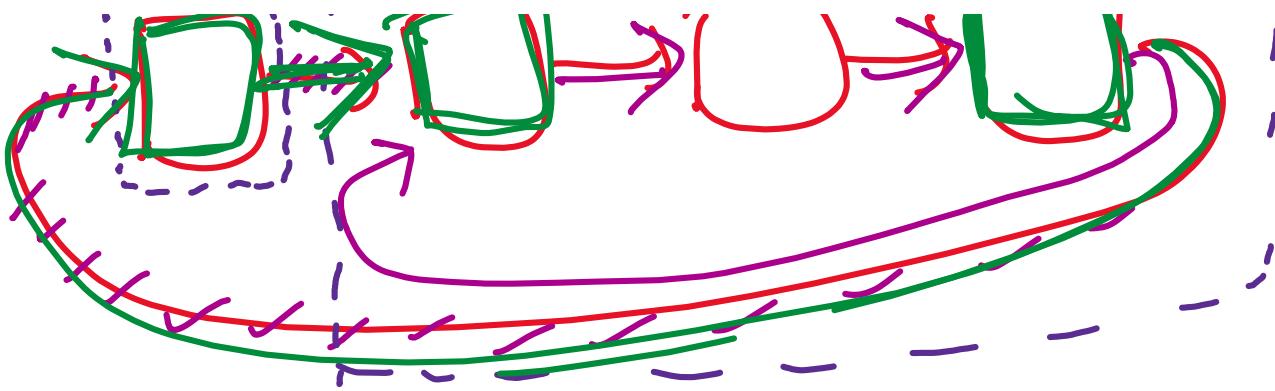
↳ cannot delete

### ii) Not Empty



↳ **head → None**





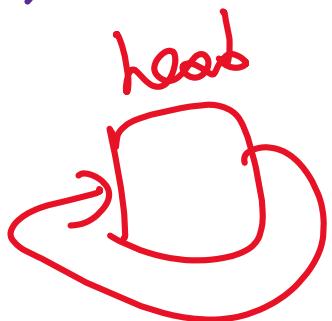
$\left[ \begin{array}{l} \text{while } \text{curr}.\text{next} \neq \text{head} \\ \quad \text{curr} = \text{curr}.\text{next} \end{array} \right]$

$\text{head} \rightarrow \text{head}.\text{next}$   
 $\text{curr}.\text{next} \rightarrow \text{head}$

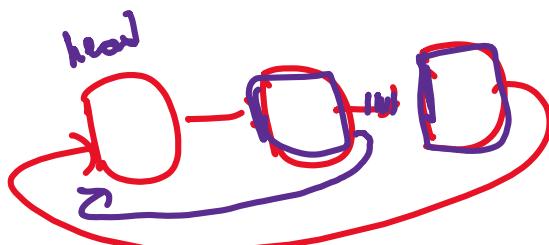
$\text{length} = \text{length} - 1$

## 2. End

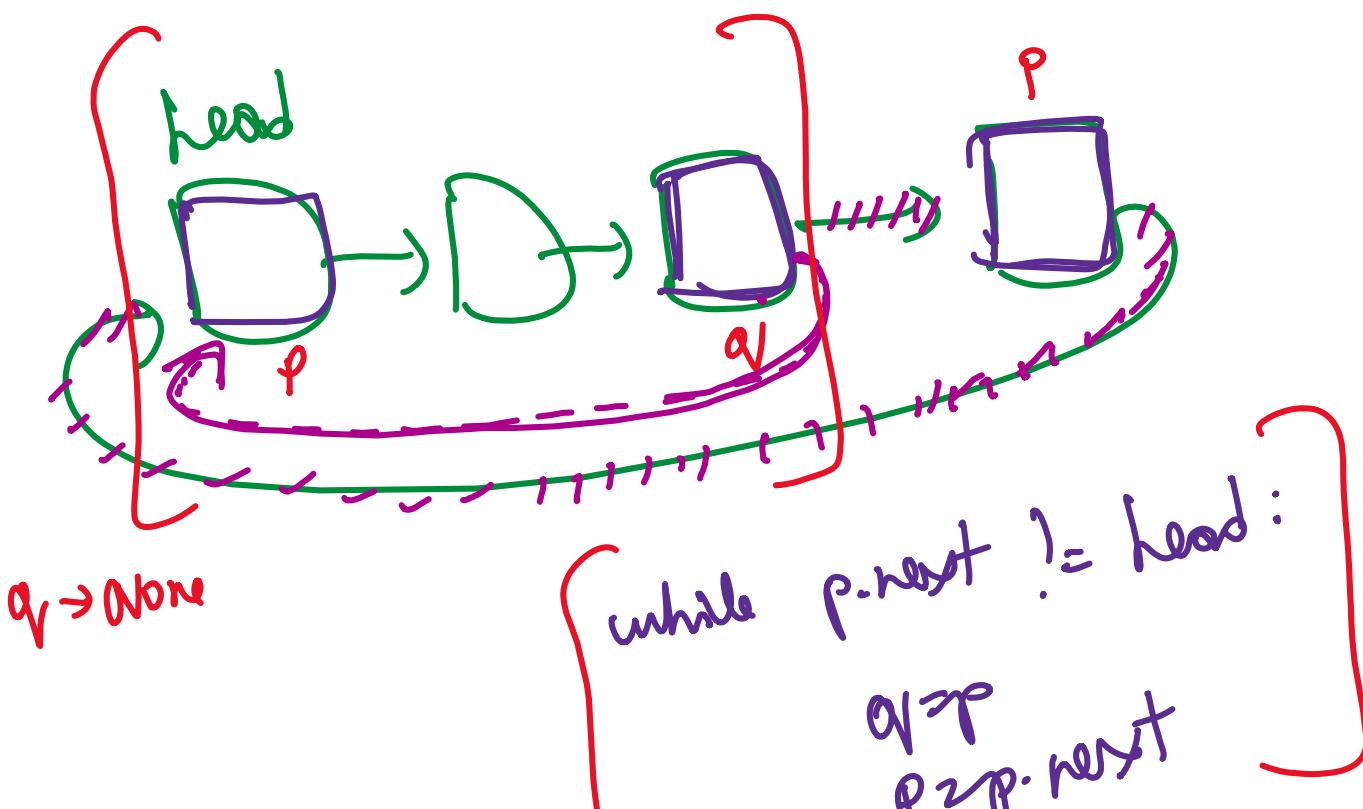
i) Single



ii) Multiple



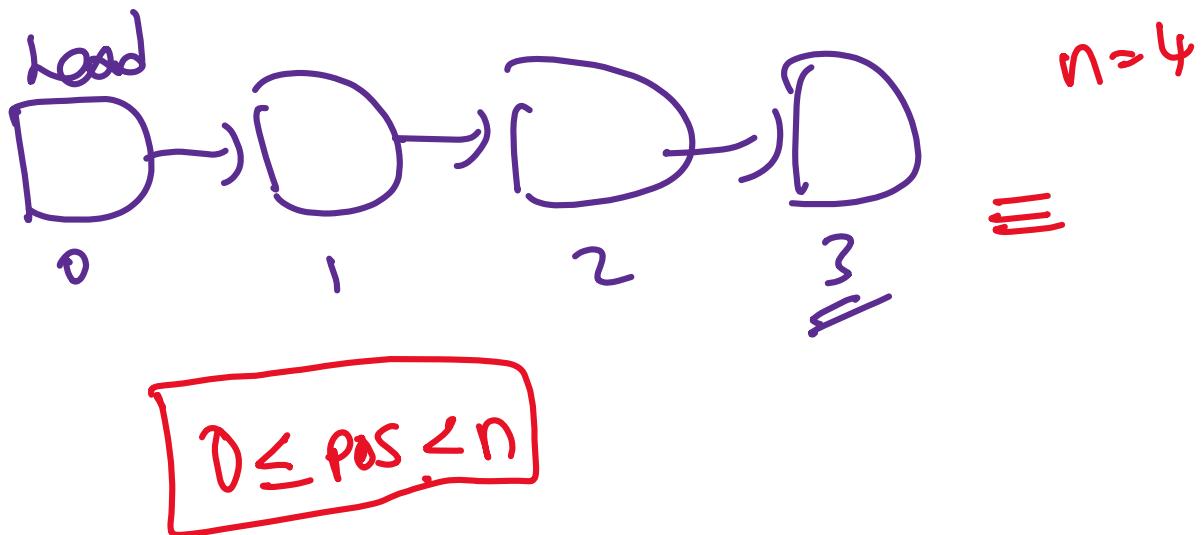
[ $\text{head} \rightarrow \text{None}$ ]



$L$        $\overset{q \leftarrow p \text{. next}}{\sim}$

$q \cdot \text{next} \rightarrow \text{head}$   
 $p \cdot \text{next} \rightarrow \text{None}$

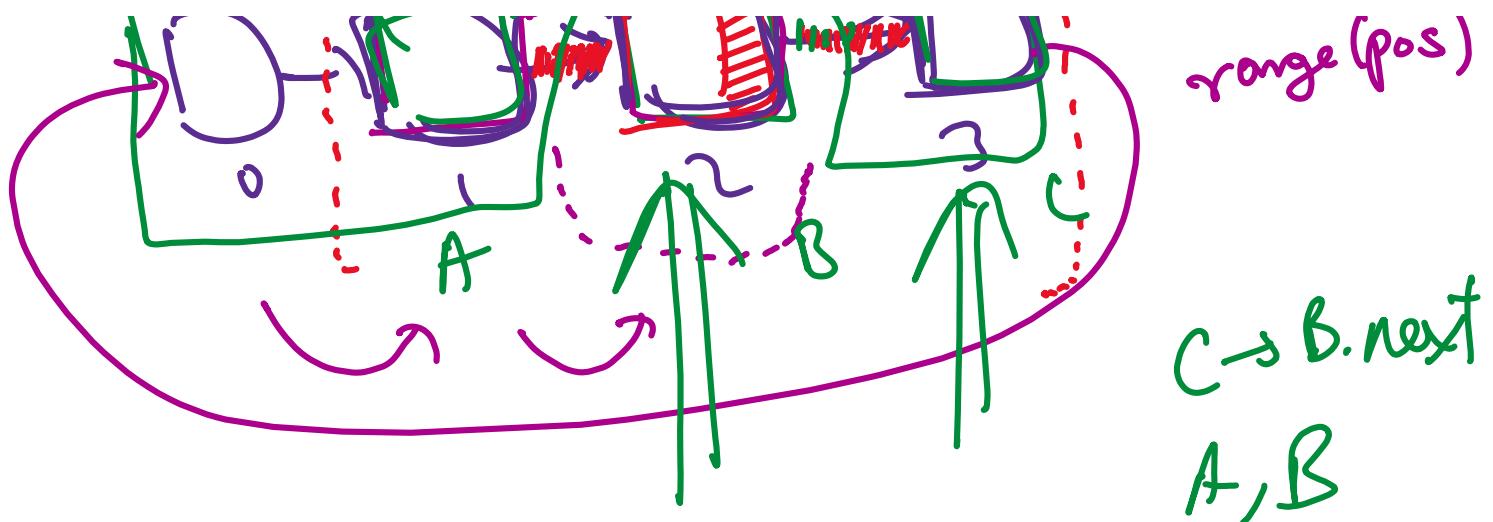
### 3. Middle:



$\text{pos} = 0 \rightarrow \text{beginning}$   
 $\text{pos} = n - 1 \rightarrow \text{end}$

$\hookrightarrow$  middle





loop over range(pos)

$q = p$   
 $p = p.\text{next}$

$q.\text{next} \rightarrow p.\text{next}$   
 $p.\text{next} \rightarrow \text{None}$







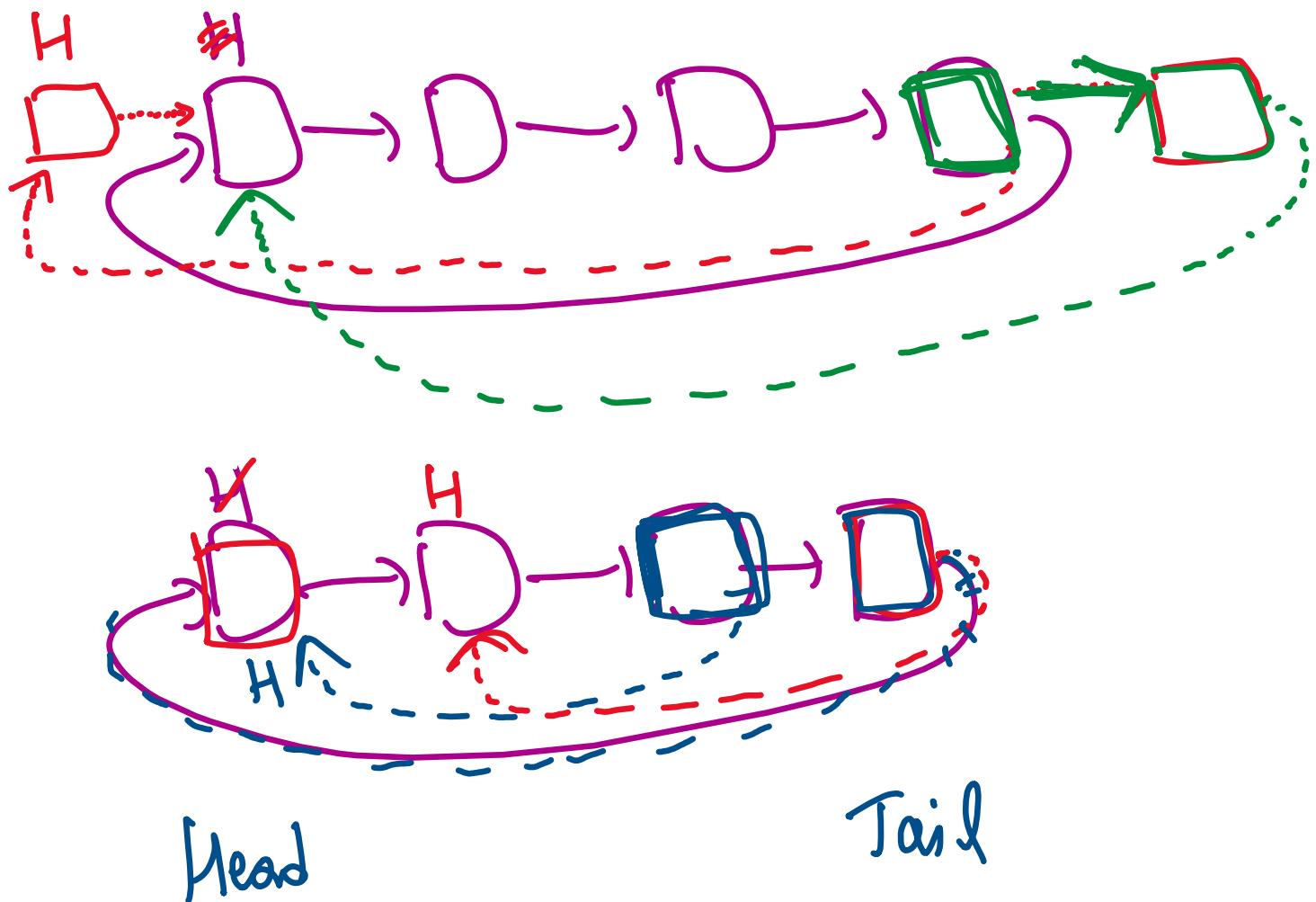


## 4.6.5 - CSLL Complexity Analysis

Tuesday, September 9, 2025 9:34 AM

OPERATION	
Traversal	$O(n)$
Search	$O(n)$
Insert at Beginning	$O(n)$
Insert at End	$O(n)$
Insert at Middle	$O(n)$
Delete at Beginning	$O(n)$
Delete at End	$O(n)$
Delete at Middle	$O(n)$
Space per Node	$O(1)$
Overall Space Usage	$O(n)$

$2 \times n \Rightarrow 2n \Rightarrow n \Rightarrow O(n)$



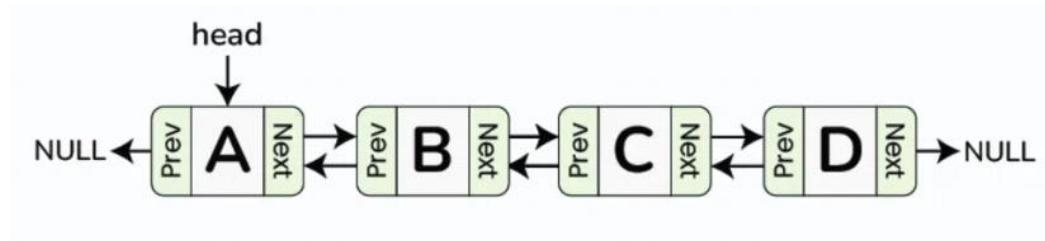
Head

Tail

## 4.7 - Doubly Linked List

Saturday, September 20, 2025 12:35 PM

- A **Doubly Linked List** is a type of linked list in which the nodes are linked in both directions
- Each node has three fields:
  - **Data:** stores the element
  - **Prev:** pointer/reference to the previous node
  - **Next:** pointer/reference to the next node
- The first node is called the **head** (its **prev** is None)
- The last node is called the **tail** (its **next** is None)
- Traversal is possible in both directions: forward (using **next**) and backward (using **prev**)



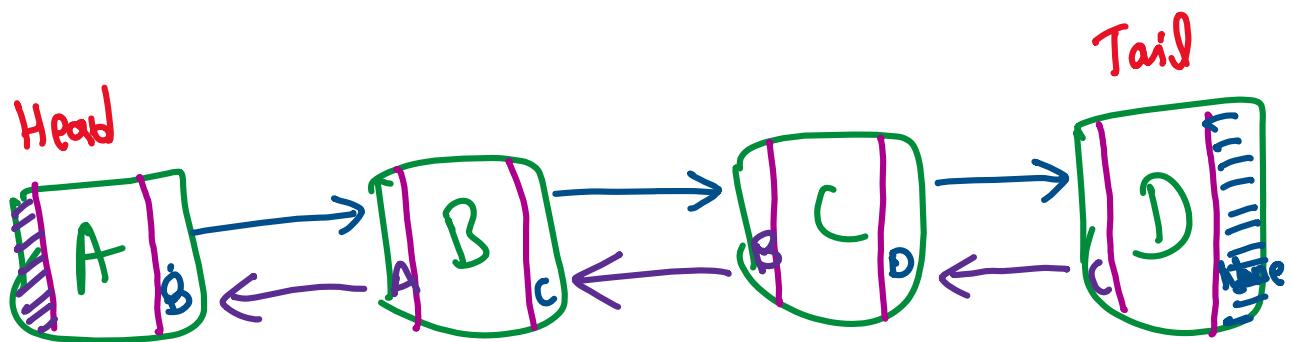
### Pros:

- Allows bidirectional traversal
- Easier deletion and insertion at both ends (head and tail)
- Can implement more complex data structures like deque and doubly-ended queues

### Cons:

- Requires extra memory for storing the **prev** pointer
- More complex implementation compared to a singly linked list
- Insertion and deletion involve more pointer updates compared to singly linked lists
- Slower due to additional pointer handling
- Extra pointer can increase chances of pointer errors

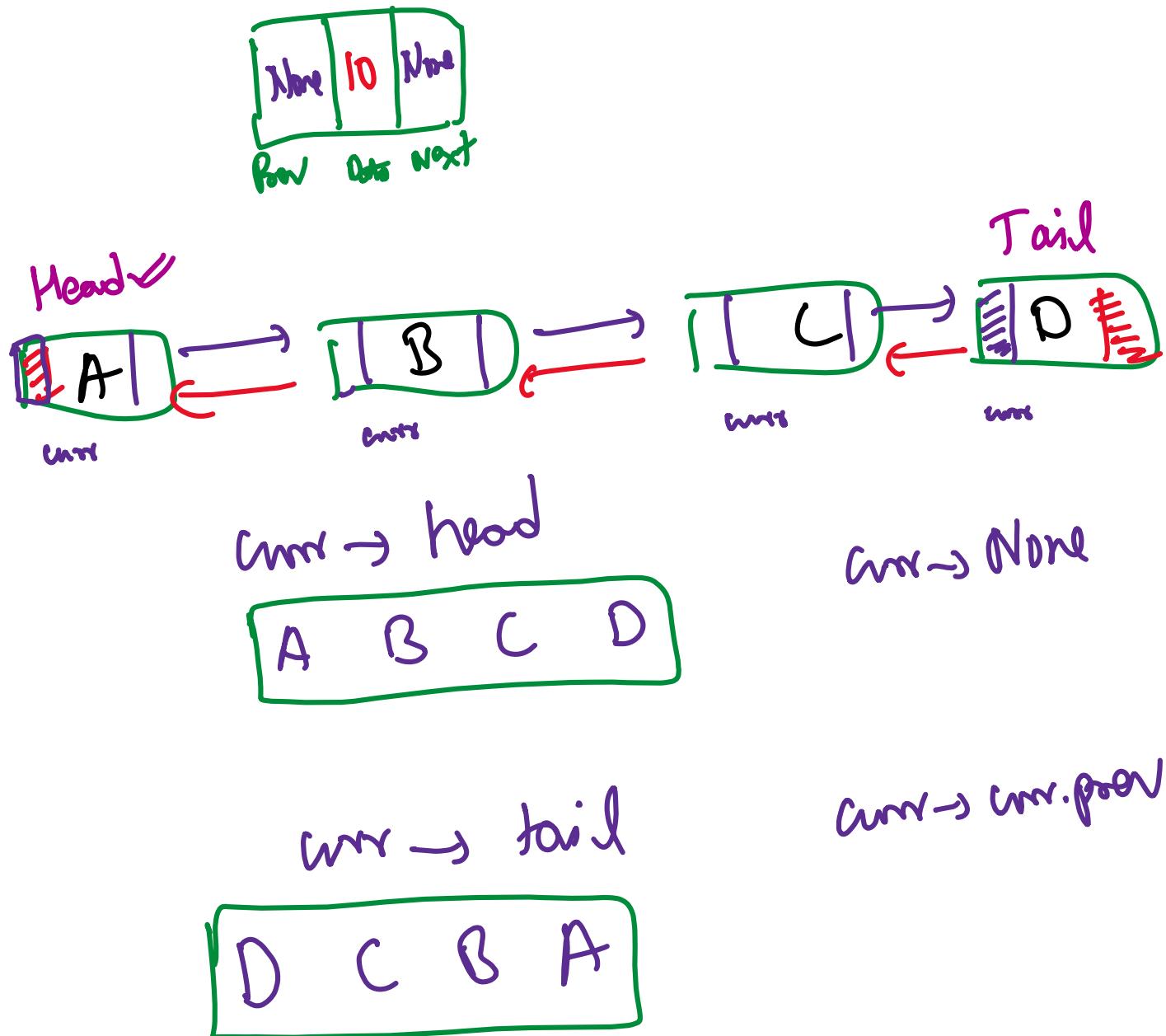
node





#### 4.7.1 - DLL Initial Setup

Saturday, September 20, 2025 12:36 PM

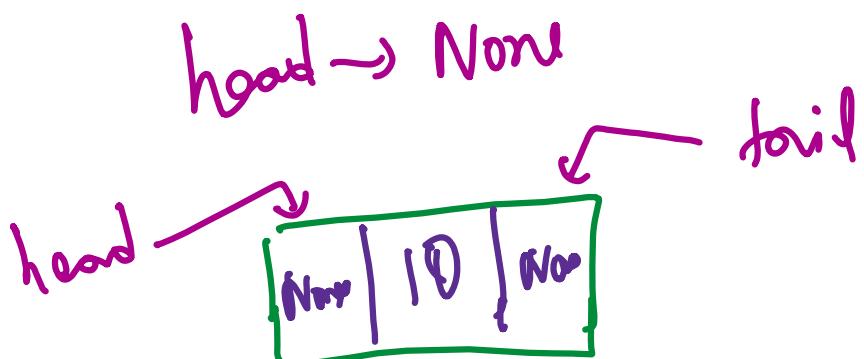


#### 4.7.2 - DLL Insertion

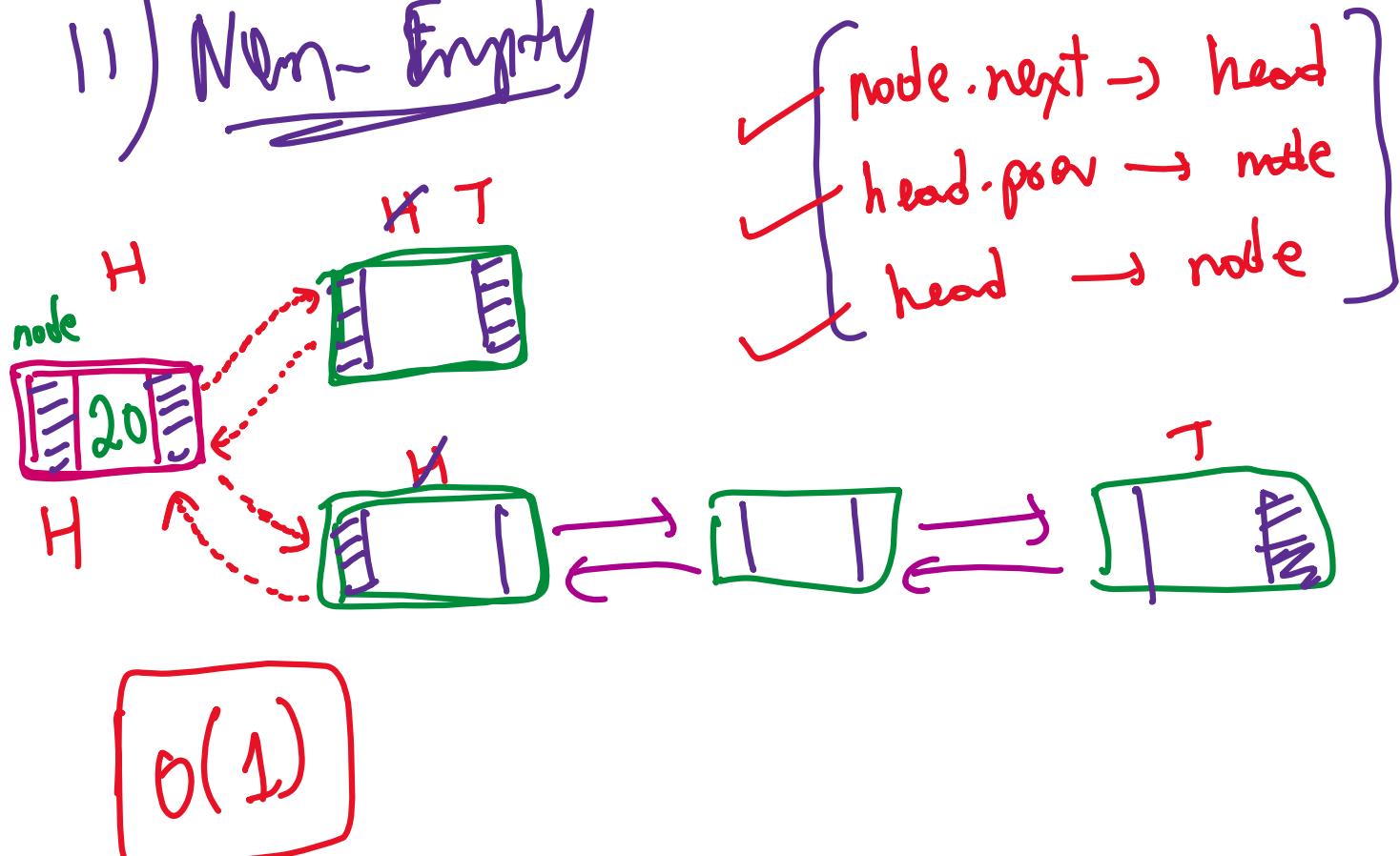
Saturday, September 20, 2025 12:36 PM

## 1. Beginning:

### i) Empty



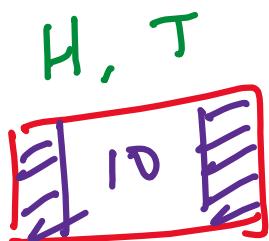
### ii) Non-Empty



10 11

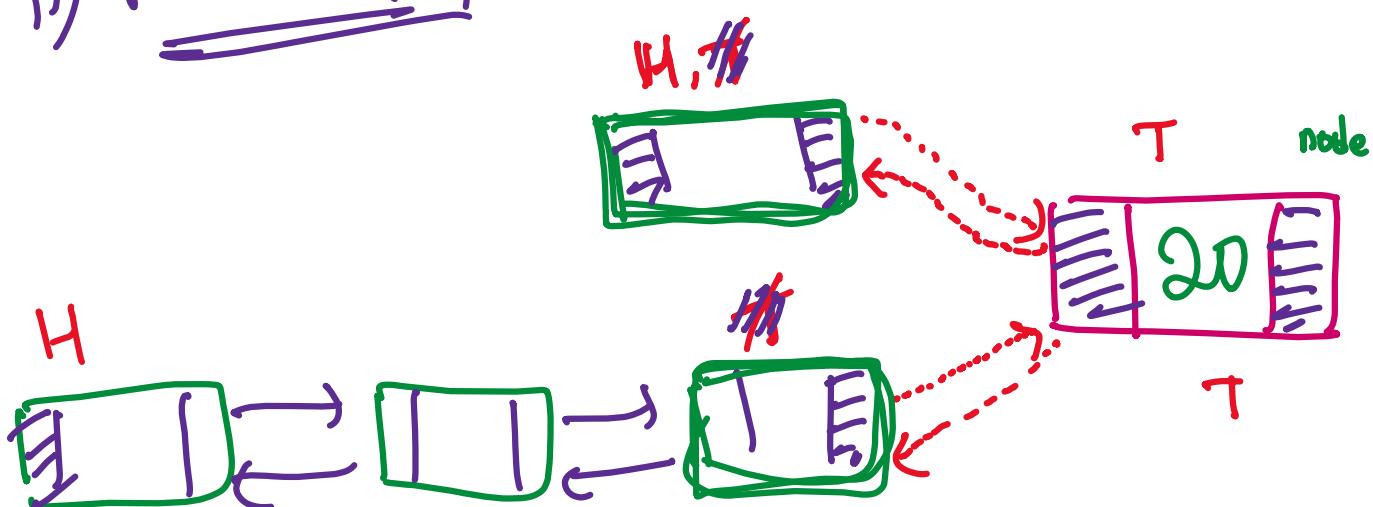
## 2. End:

i) Empty:



{ head → node  
tail → node }

ii) Non-empty:



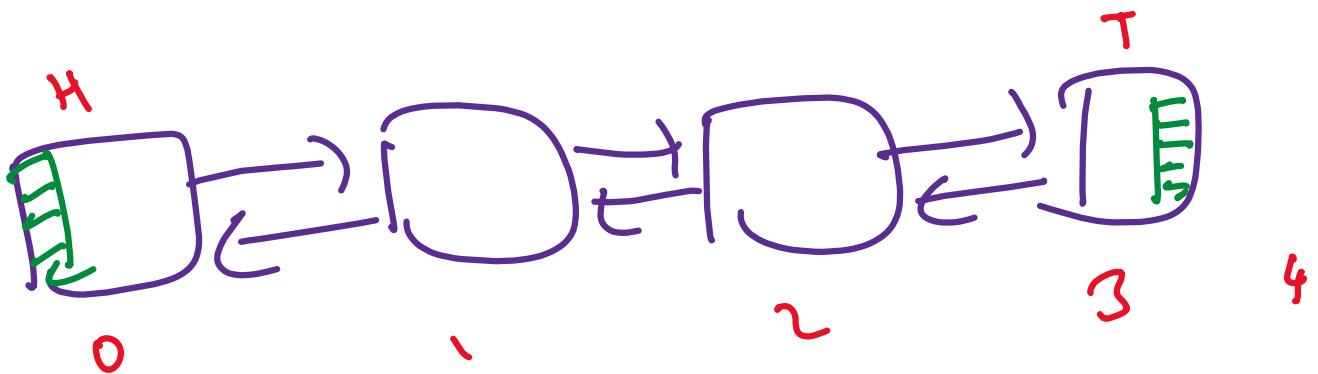
{ tail.next → node  
node.next → tail }

$\text{node}.\text{prev} \rightarrow \text{tail}$

$\text{tail} \rightarrow \text{node}$

$O(1)$

### 3. Middle :



$\Leftarrow$  pos  
data

$0 \leq \text{pos} \leq 3$

$n=4$

$0 \leq \text{pos} \leq n-1$

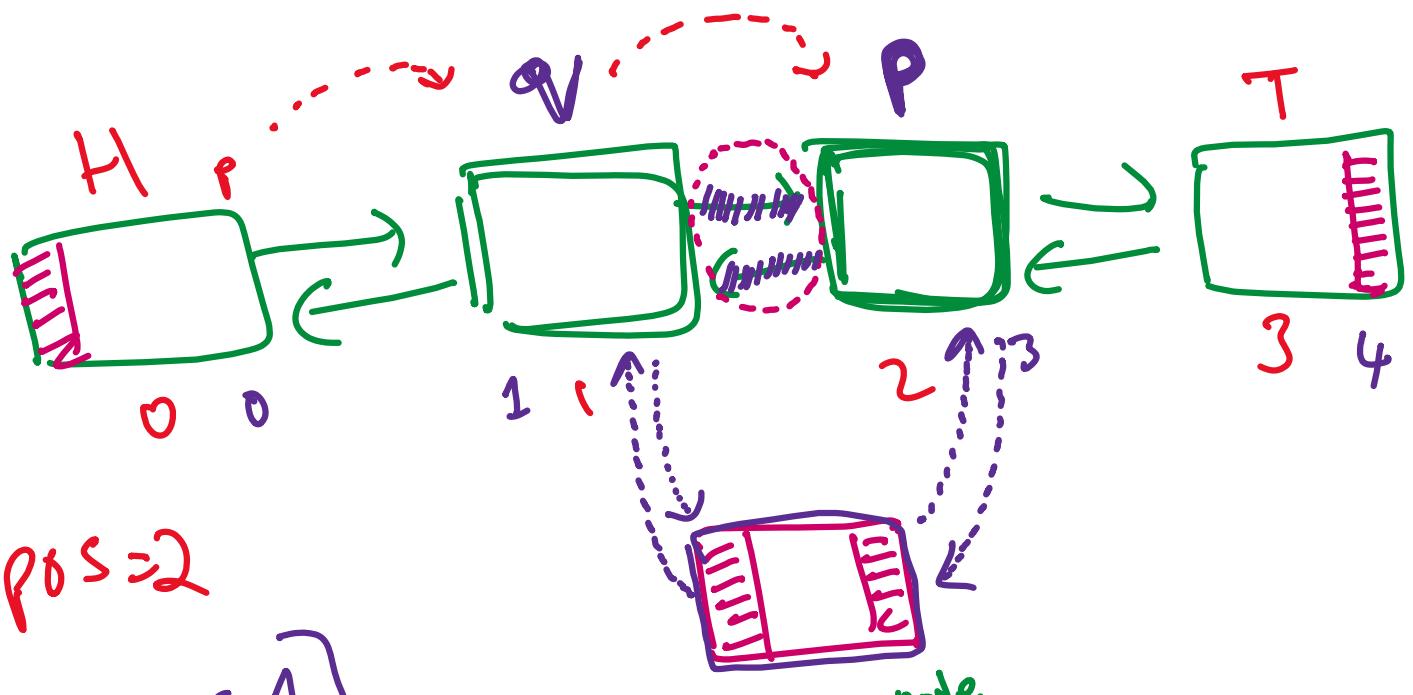
$\Leftarrow (\text{pos} < 0), (\text{pos} > n)$

✓ ✓

$\text{pos} \rightarrow D$

$\text{pos} \rightarrow R$

$\text{pos} \in (0, n)$



$p \rightarrow \text{head}$

for  $\text{range}(\text{pos})$ :

$p \rightarrow p.\text{next}$ ,

$P \rightarrow P \cdot \text{new}$

$Q = P \cdot \text{new} \quad \checkmark$

$Q \cdot \text{next} \rightarrow \text{node}$

$\text{node} \cdot \text{prev} \rightarrow Q$

$\text{node} \cdot \text{next} \rightarrow P$

$P \cdot \text{prev} \rightarrow \text{node}$



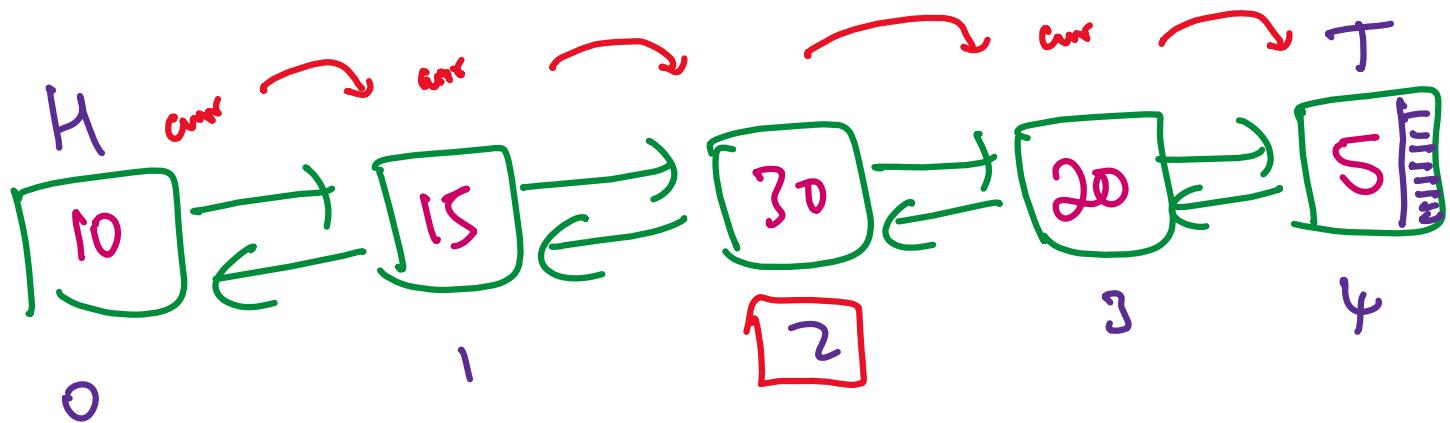






#### 4.7.3 - DLL Search

Saturday, September 20, 2025 12:36 PM



list → empty ✓

Key = 30  
= 60

pos = 0

while current:

{ current.data == key  
  ↳ pos

{ pos += 1  
  current → current.next

curr → None

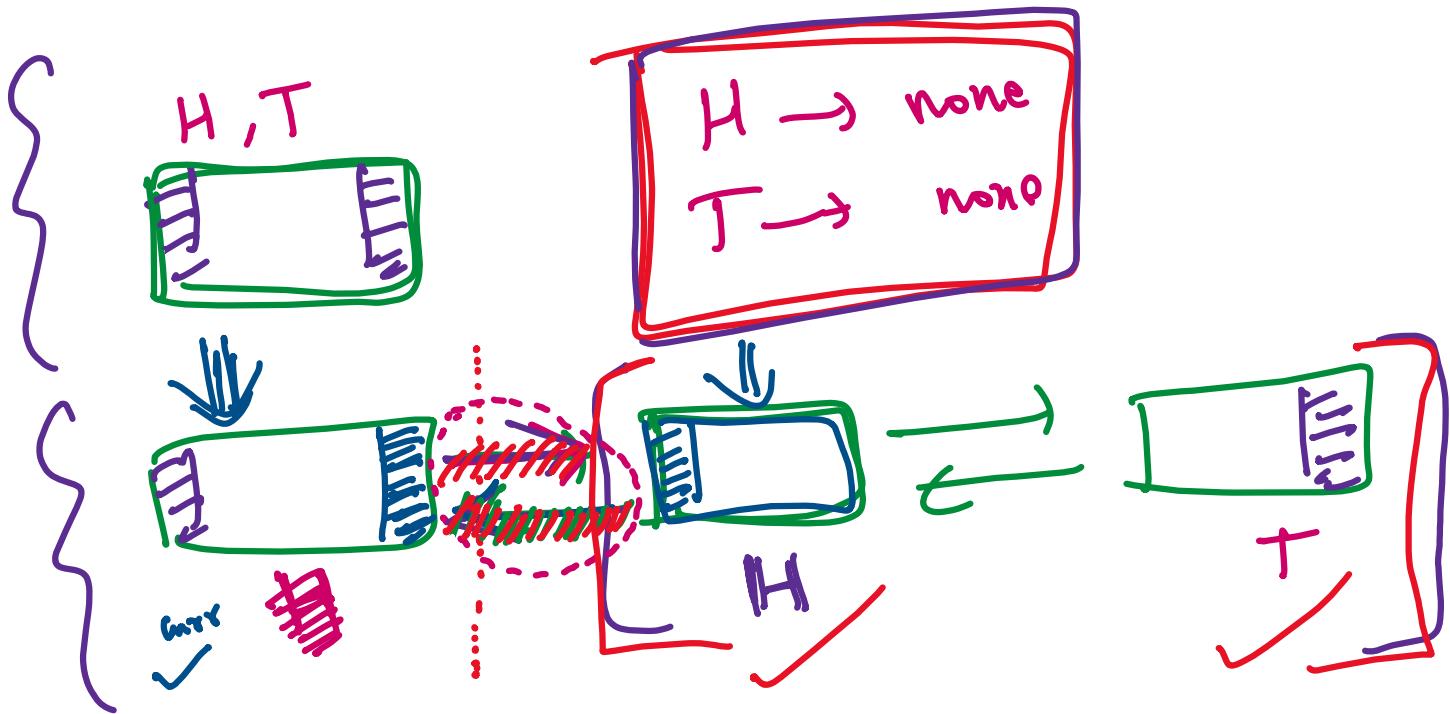
www → None

#### 4.7.4 - DLL Deletion

Saturday, September 20, 2025 12:36 PM

## 1. Beginning

list → empty



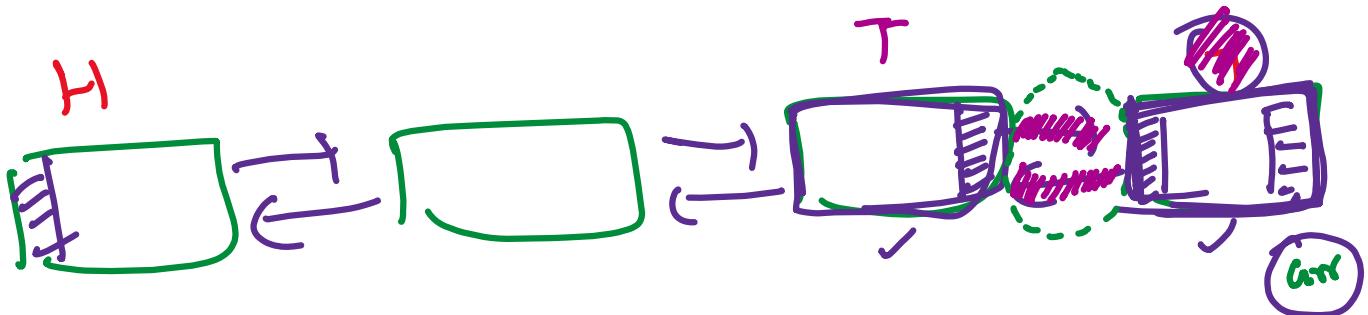
curr → head  
head → head.next  
head.prev → None  
+ .  $\nwarrow$  line

curr.next → None

2. End

list → empty

H, T



[  
head → None  
tail → None

[  
curr → tail  
... , tail, None

$\text{tail} \rightarrow \text{tail.prev}$   
 $\text{tail.next} \rightarrow \text{None}$   
 $\text{curr.prev} \rightarrow \text{None}$

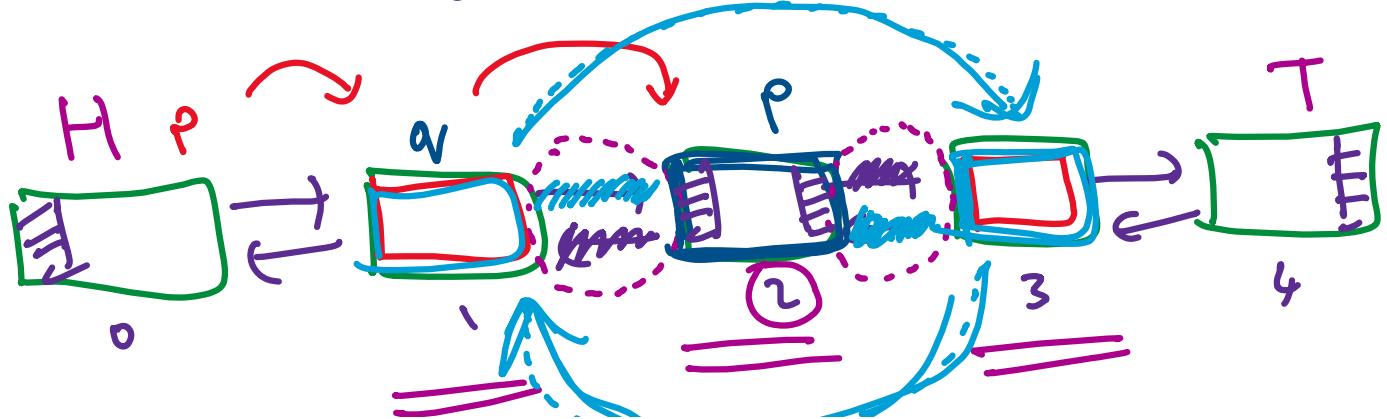
### 3. Middle:

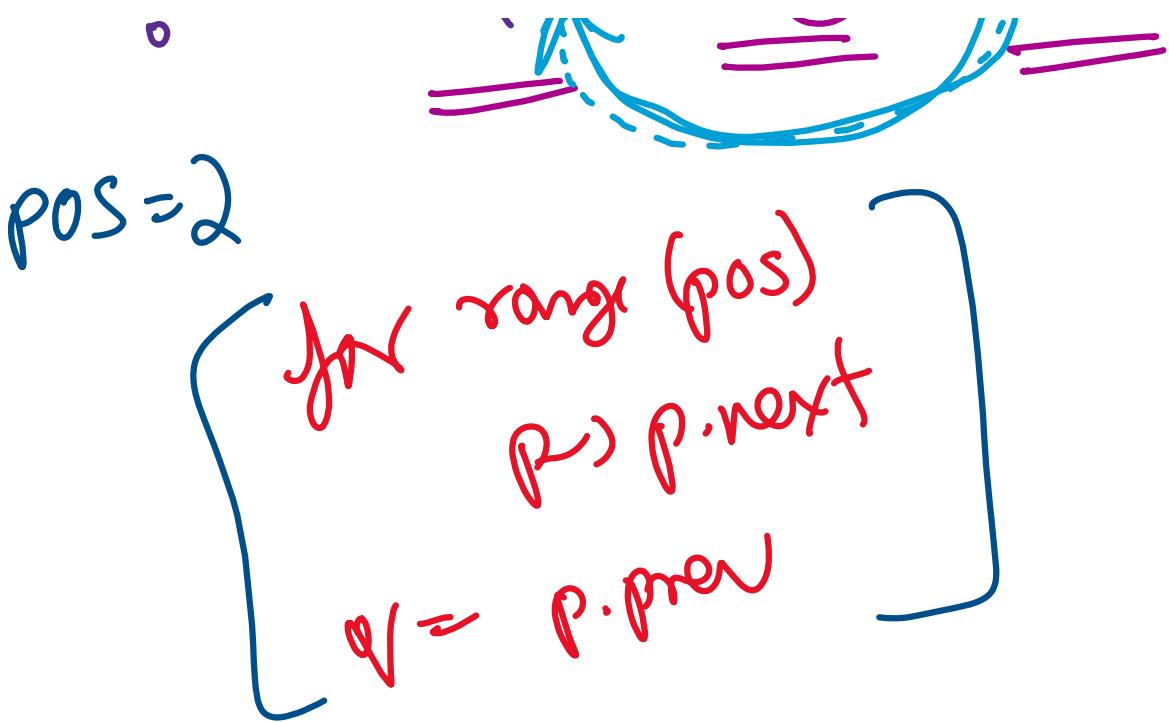
$n=4$



$$0 \leq \text{pos} \leq n-1$$

$$0 \leq \text{pos} < n$$





$q.\text{next} \rightarrow p.\text{next}$

$p.\text{next}.\text{prev} \rightarrow q$

$p.\text{next} \rightarrow \text{None}$

$p.\text{prev} \rightarrow \text{None}$











#### 4.7.5 - DLL Complexity Analysis

Saturday, September 20, 2025 12:36 PM

OPERATION	
✓ Traversal	$O(n)$
✓ Search	$O(n)$
✓ Insert at Beginning	$O(1)$
✓ Insert at End	$O(1)$
✓ Insert at Middle	$O(n)$
✓ Delete at Beginning	$O(1)$
✓ Delete at End	$O(1)$
✓ Delete at Middle	$O(n)$
✓ Space per Node	$O(1)$
✓ Overall Space Usage	$O(n)$

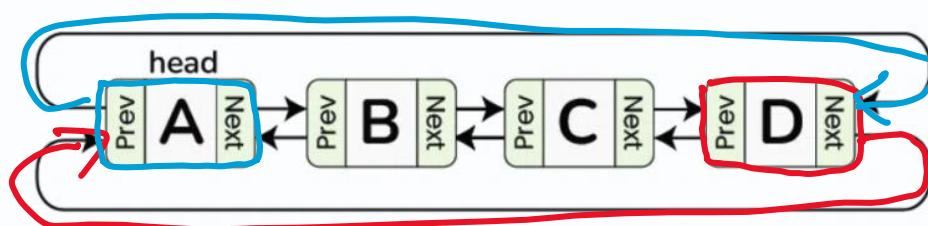
Handwritten annotations:

- $\rightarrow n$
- $\rightarrow 3$
- $\rightarrow \cancel{O(n)}$
- $\rightarrow O(n)$

## 4.8 - Circular Doubly Linked List

Saturday, October 4, 2025 2:19 PM

- A **Circular Doubly Linked List** is a variation of the doubly linked list in which the last node is connected back to the first node, forming a circular structure
- Each node has three fields:
  - **Data** – stores the element
  - **Prev** – pointer/reference to the previous node
  - **Next** – pointer/reference to the next node
- The first node's **prev** points to the **last node**, and the last node's **next** points to the first node
- There is no **NULL** (or **None**) reference in the list since it is circular
- Traversal is possible in both directions:
  - Forward using the **next** pointer
  - Backward using the **prev** pointer
- Circular structure makes it easy to move continuously through the list without needing to reset at the **head or tail**



### Pros:

- Can be traversed forward and backward easily using **next** and **prev** pointers
- No need to check for **None** — can loop through the list endlessly
- Nodes can be inserted or removed efficiently from both ends or any position

### Cons:

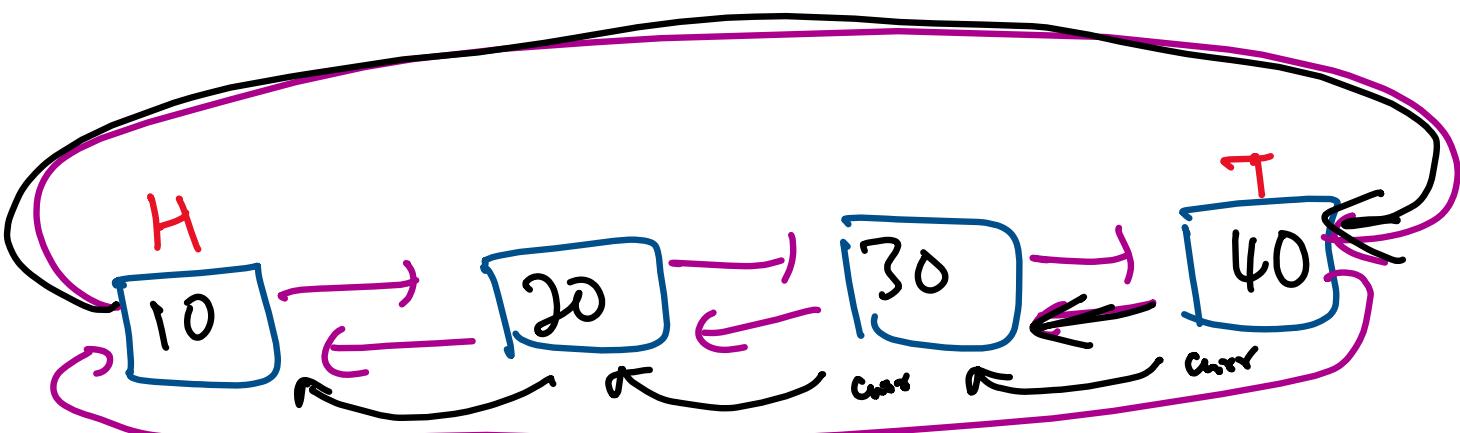
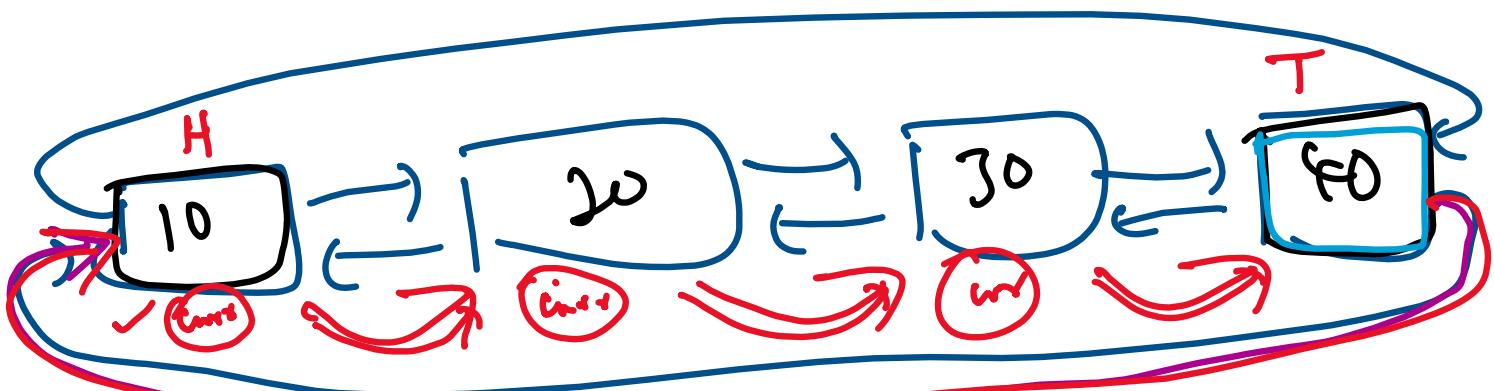
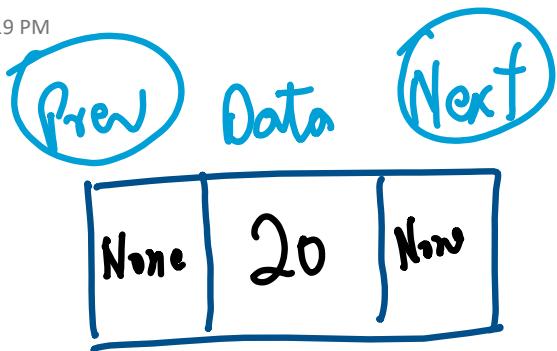
- Requires two pointers (**prev** and **next**) per node, increasing memory usage overall

- More pointer adjustments are needed during insertion and deletion
- If termination conditions are not handled properly, traversals may become infinite
- Implementation is complex among all other types

#### 4.8.1 - CDLL Initial Setup

Saturday, October 4, 2025

2:19 PM



blue pink black currant currant

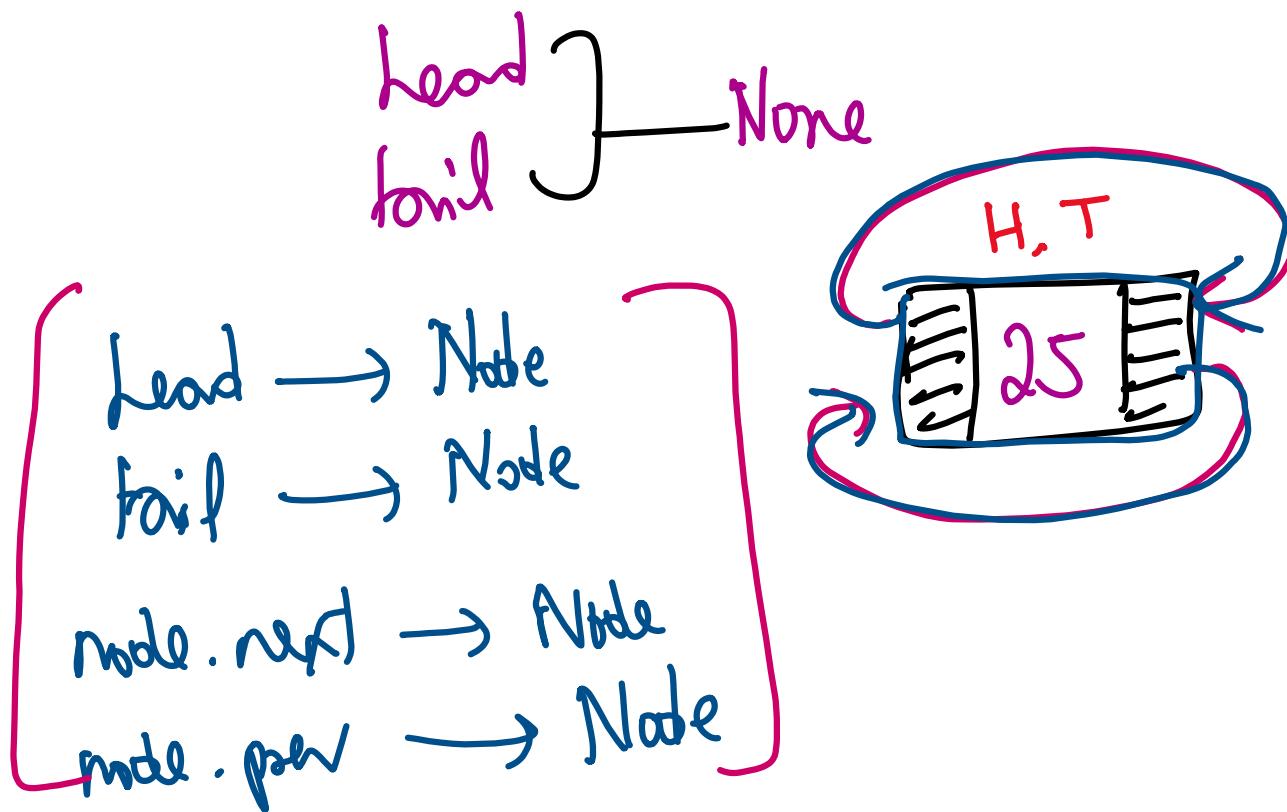
40 30 20 10

#### 4.8.2 - CDLL Insertion

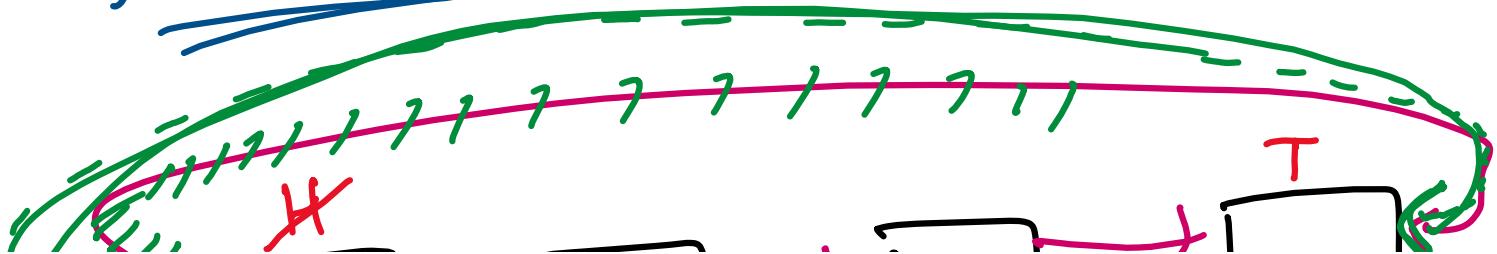
Saturday, October 4, 2025 2:19 PM

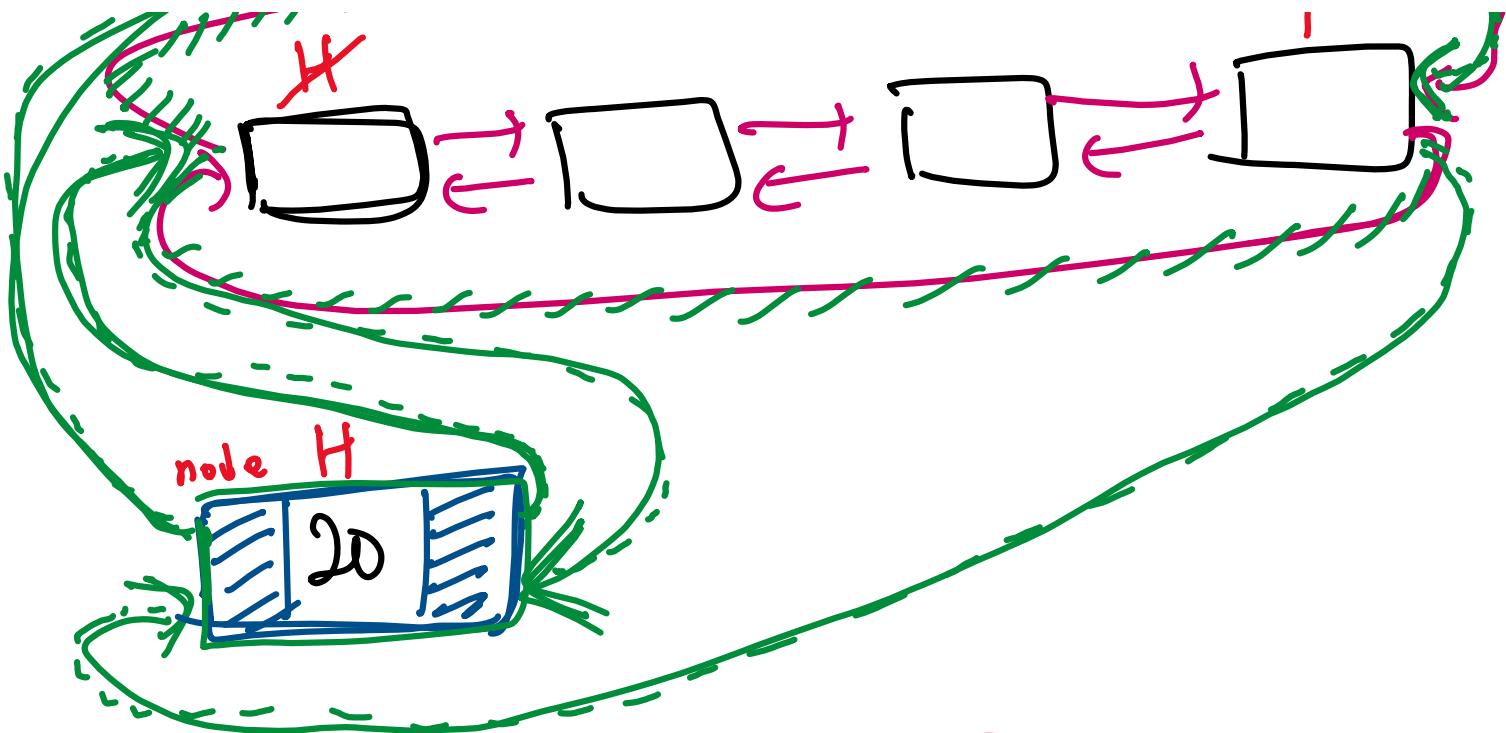
##### 1. Beginning

###### i) Empty



###### ii) Non-empty:



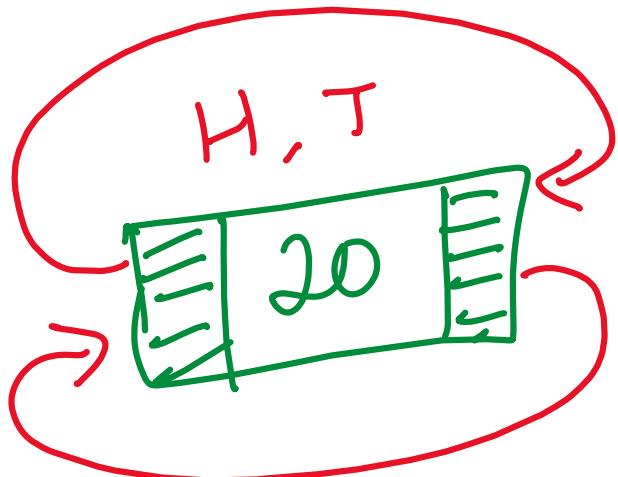


$\text{node.next} \rightarrow \text{head}$   
 $\text{node.prev} \rightarrow \text{tail}$   
 $\text{head.prev} \rightarrow \text{node}$   
 $\text{tail.next} \rightarrow \text{node}$   
 $\text{head} \rightarrow \text{Node}$

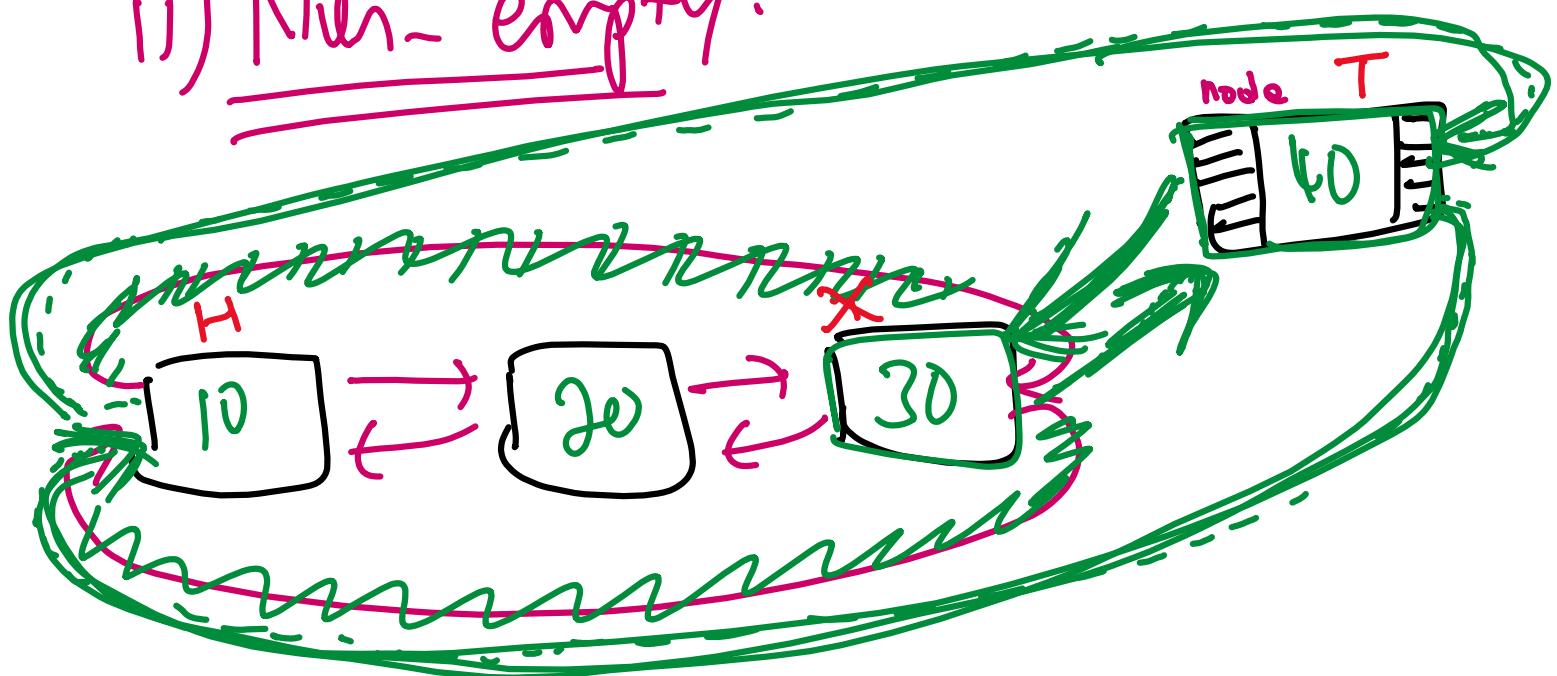
2. End:

• ↴ End ↴

i) Empty



ii) Non-empty:



node.next → head

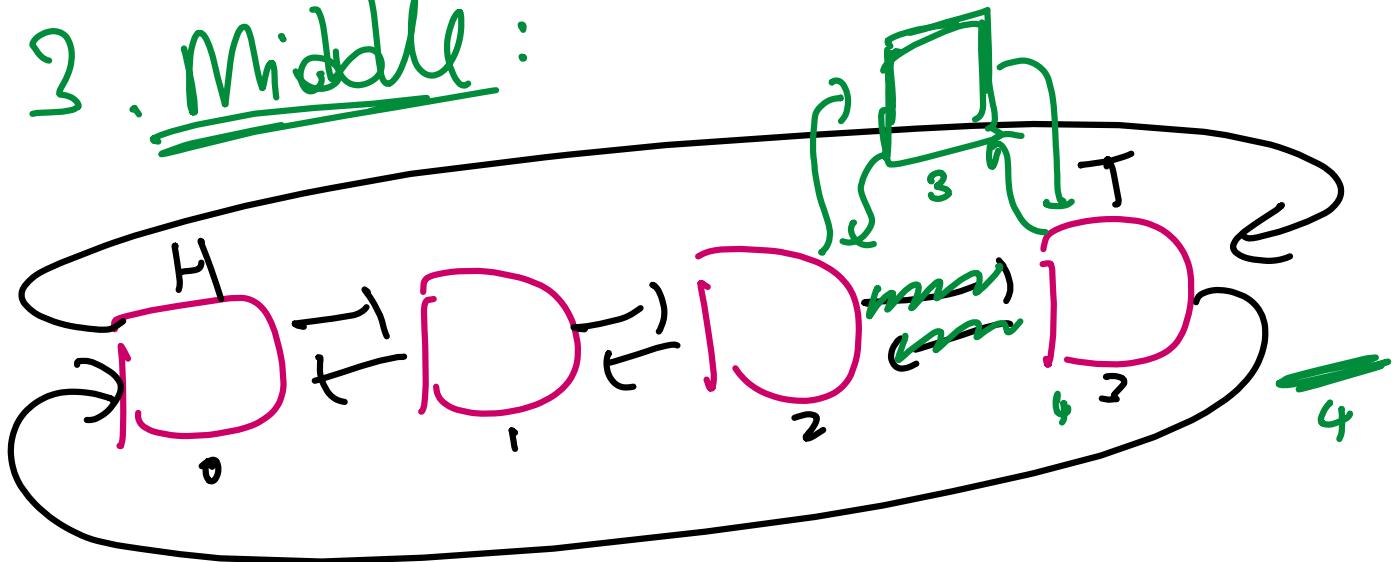
node.prev → tail

head.prev → node

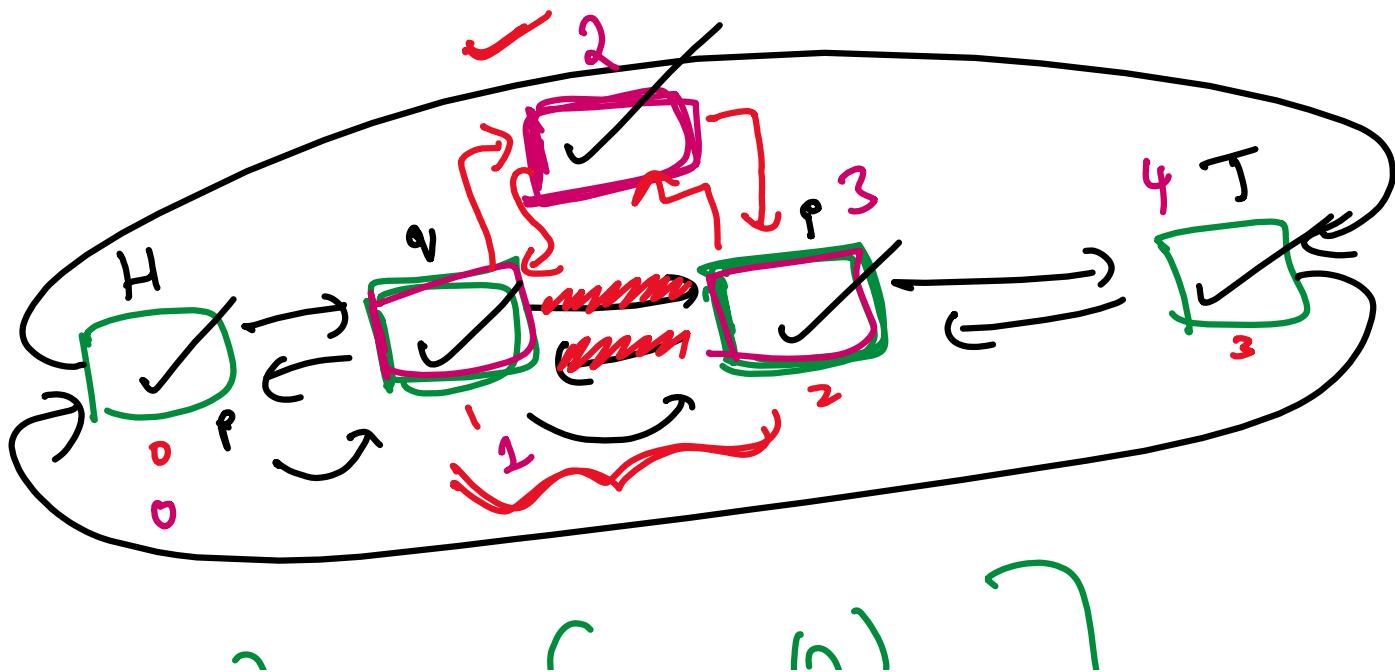
tail.next → node

tail. next = ...  
 tail → node

### 3. Middle:



$$0 \leq \text{pos} \leq n$$



$pos = 2$

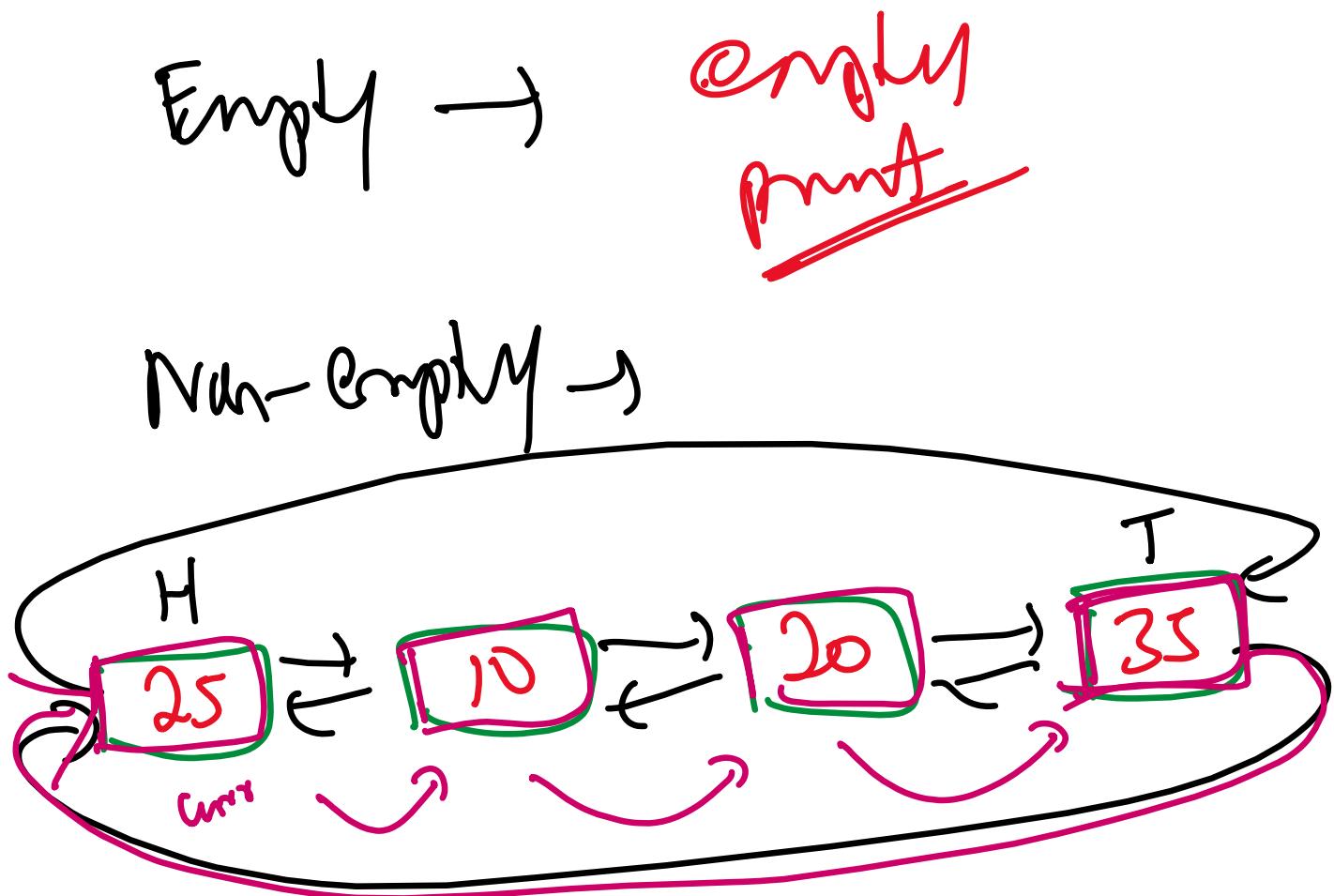
range(2)  
↳ 0, 1  
↳ 2 times

$q.next \rightarrow node$   
 $node.prev \rightarrow q$   
 $node.next \rightarrow p$   
 $p.prev \rightarrow node$



#### 4.8.3 - CDLL Search

Saturday, October 4, 2025 2:19 PM



key curr → head

while True:

if curr.data == key:

print("found")

break

→ curr.next

$\text{curr} \rightarrow \text{curr.next}$

if  $\text{curr} == \text{head}$ :

print("not found")

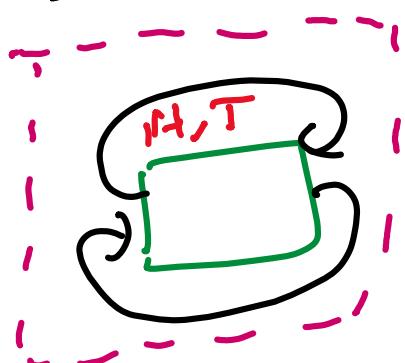
break



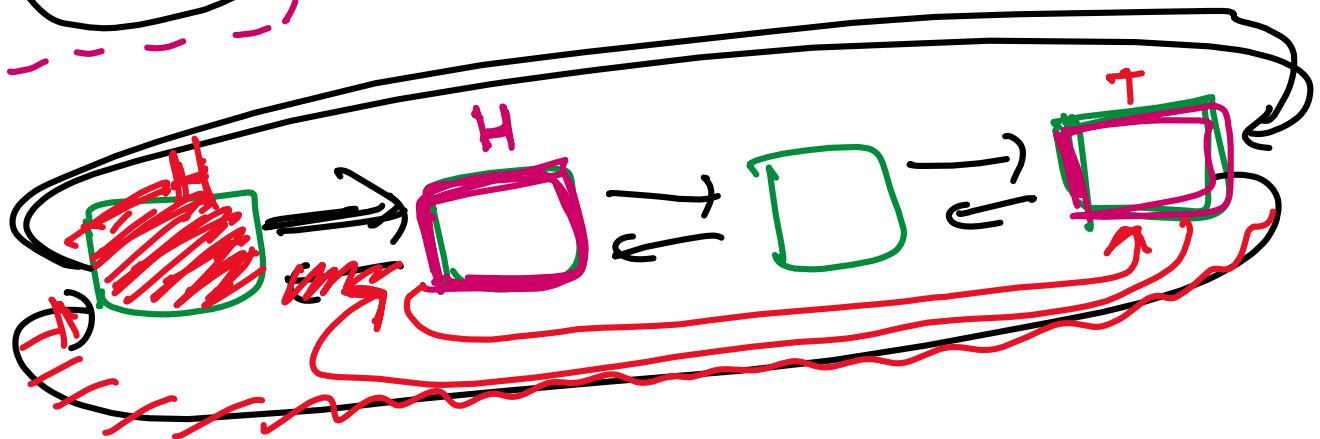
#### 4.8.4 - CDLL Deletion

Saturday, October 4, 2025 2:19 PM

1. Beginning:



head  
tail }  
→ None



head → head.next

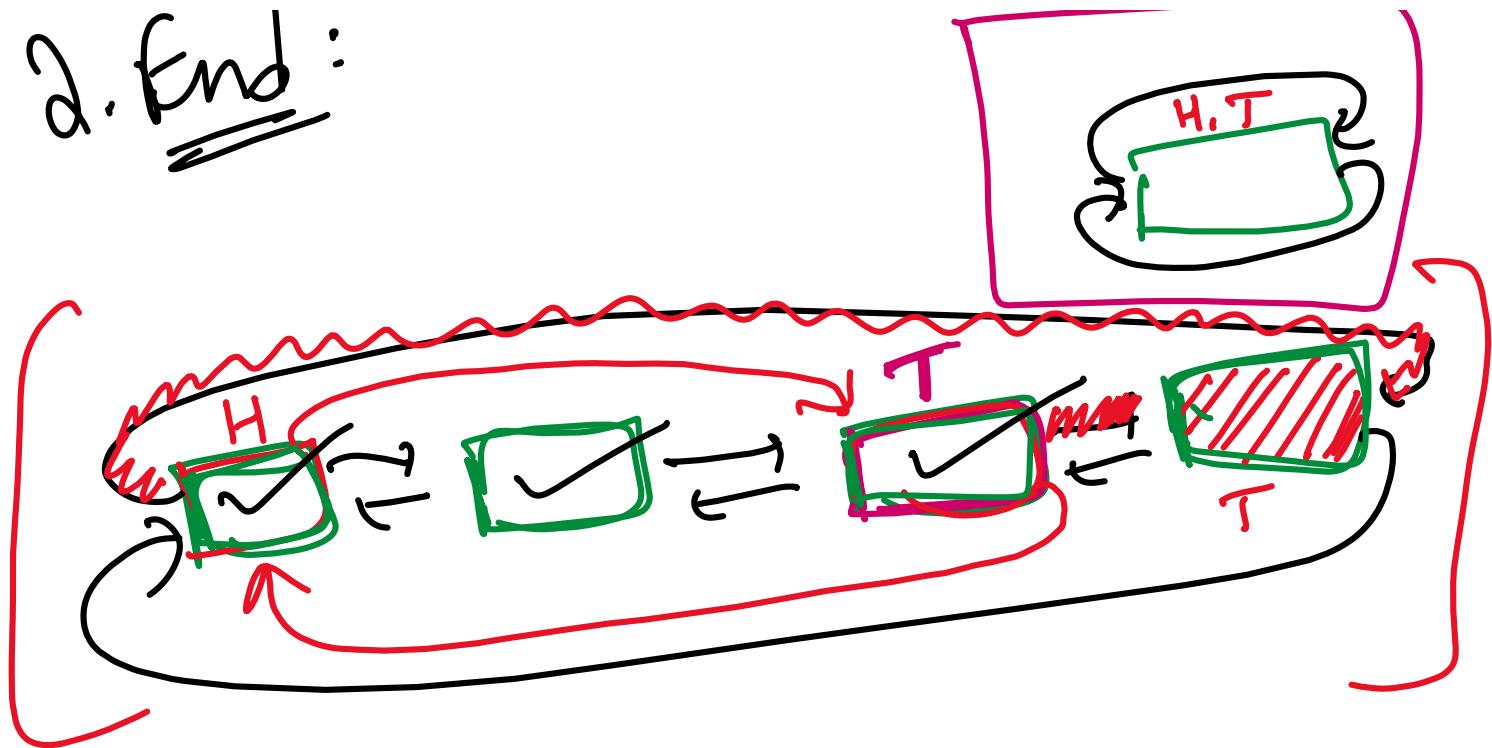
head.prev → tail

tail.next → head

2. End:



2. End:

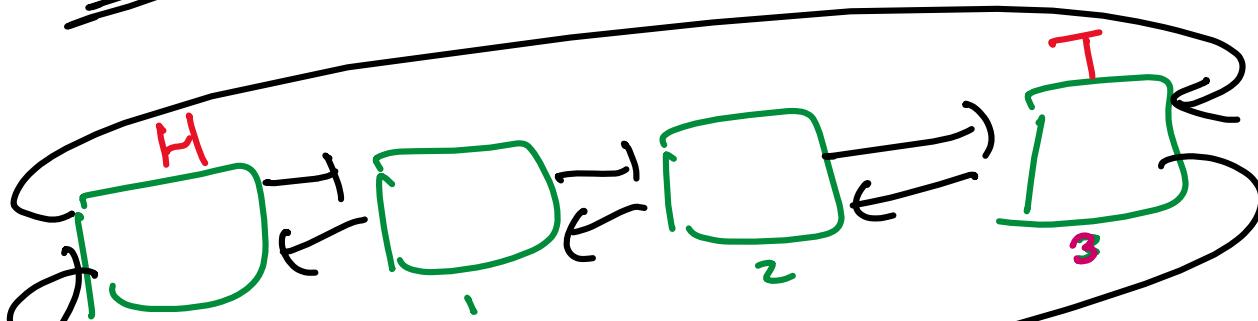


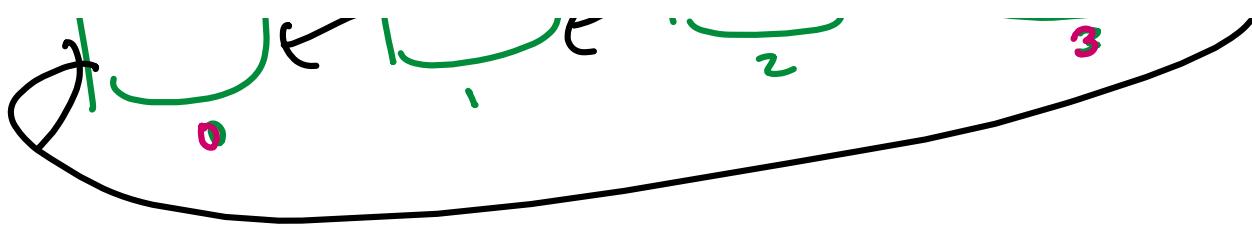
tail  $\rightarrow$  tail.prev

head.prev  $\rightarrow$  tail

tail.next  $\rightarrow$  head

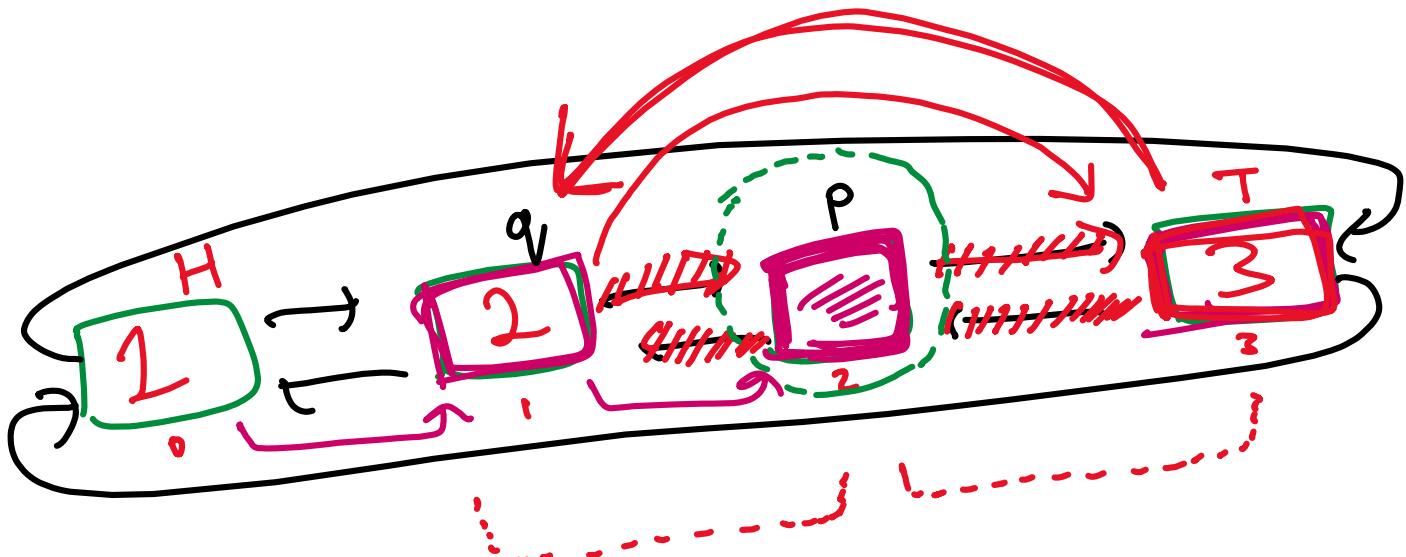
3. Middle:





$pos$

$$0 \leq pos < n$$



$$pos \rightarrow 2$$

$q.next \rightarrow p.next$   
 $p.next.prev \rightarrow q$   
 $p.next \rightarrow \text{None}$   
 $q.next \rightarrow \text{None}$

$\lfloor$   $p.\text{prev} \rightarrow \text{None}$   $\rfloor$





























## 4.8.5 - CDLL Complexity Analysis

Saturday, October 4, 2025 2:19 PM

OPERATION	
Traversal	$\rightarrow O(n)$
Search	$\rightarrow O(n)$
Insert at Beginning	$\rightarrow O(1)$
Insert at End	$\rightarrow O(1)$
Insert at Middle	$\rightarrow O(n)$
Delete at Beginning	$\rightarrow O(1)$
Delete at End	$\rightarrow O(1)$
Delete at Middle	$\rightarrow O(n)$
Space per Node	$\rightarrow O(1)$
Overall Space Usage	$\rightarrow O(n)$

Node  $\rightarrow 3$  units

List  $\rightarrow "n"$  nodes

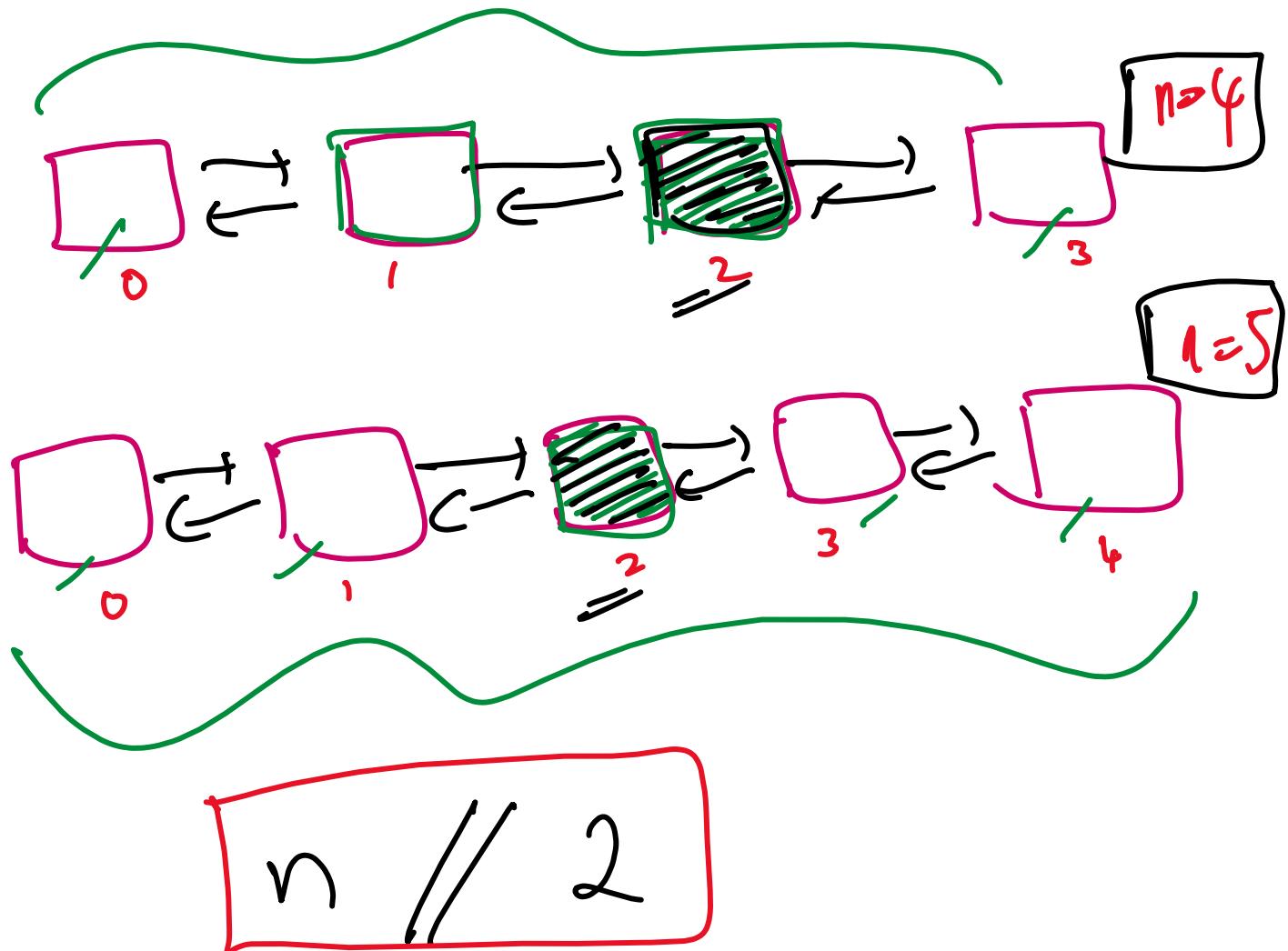
$\rightarrow 3 \times n$

$\rightarrow 3n$

$\rightarrow n$

## 4.9 - Coding Problems

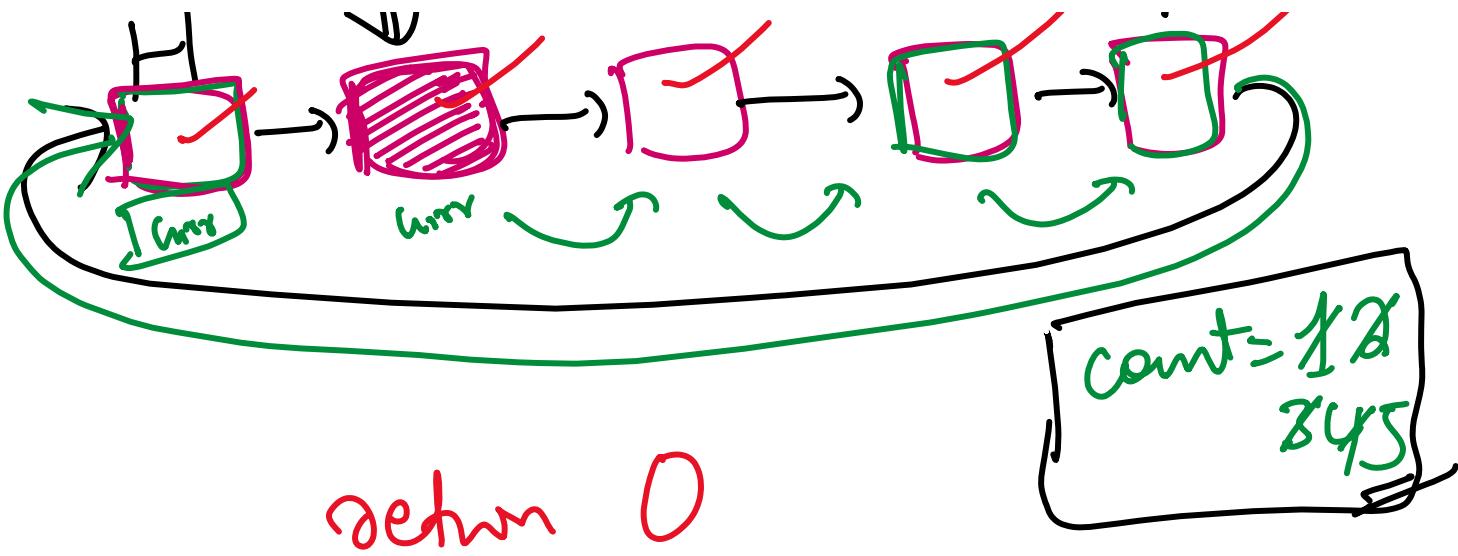
Saturday, October 18, 2025 11:24 AM



$$4 // 2 \rightarrow 2$$

$$5 // 2 \rightarrow 2$$

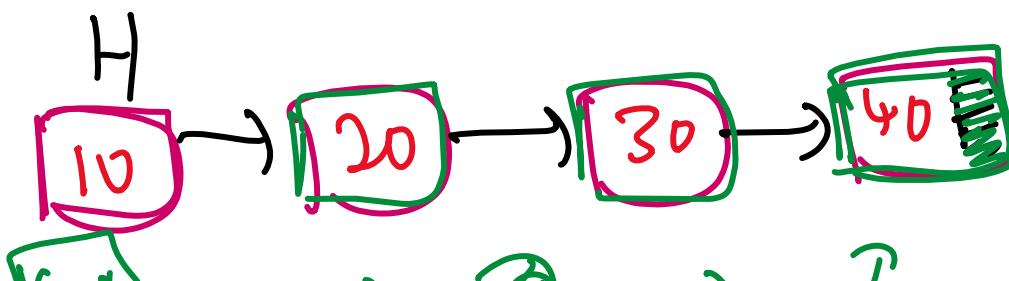
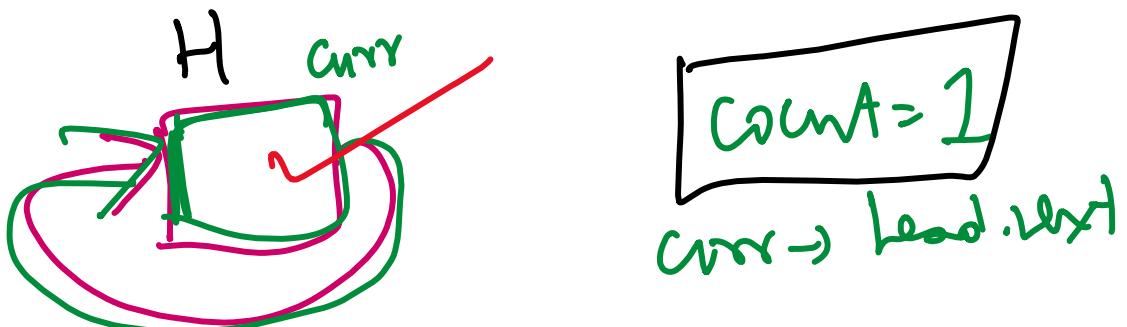
~~H~~ ~~T~~ ~~F~~ ~~T~~

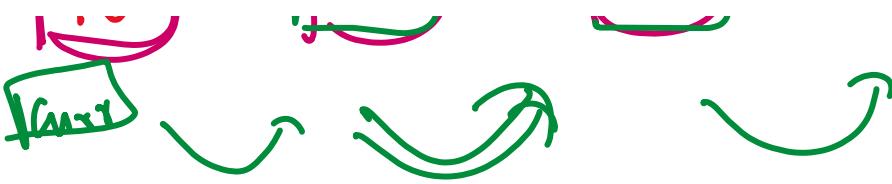


$curr \rightarrow head$

$count = 1$

$curr \rightarrow head.next$





~~total = 0 / 10~~

30 60

100

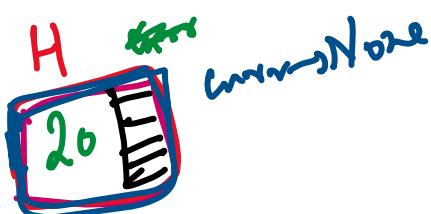
while curr:

total = total + curr.data

curr = curr.next

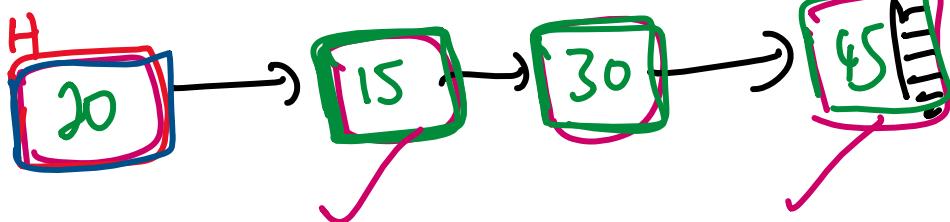
return total

None



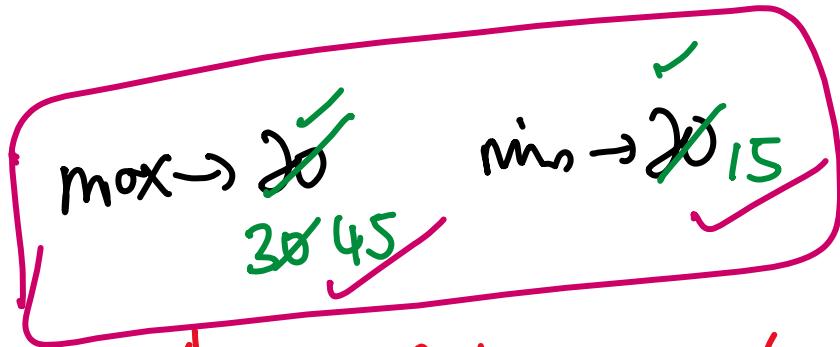
max → 20, min → 20

curr → head.next



max  
min }  $\rightarrow$  Lead. Data

n  
2n }  $O(n)$



5s, 10s

if curr. data > max

max  $\rightarrow$  curr. data

elif curr. data < min

min  $\rightarrow$  curr. data

curr  $\rightarrow$  curr.next



















# 5. Stacks

Saturday, October 18, 2025 11:24 AM

- **Introduction**
- **Common Operations**
- **Implementation**
- **Complexity Analysis**
- **Applications**
- **Pros & Cons**
- **Coding Problems**

## 5.1 - Introduction

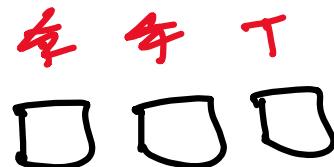
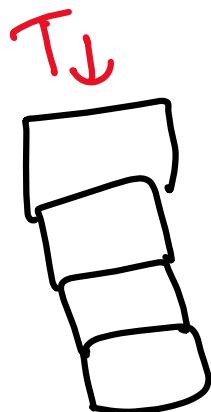
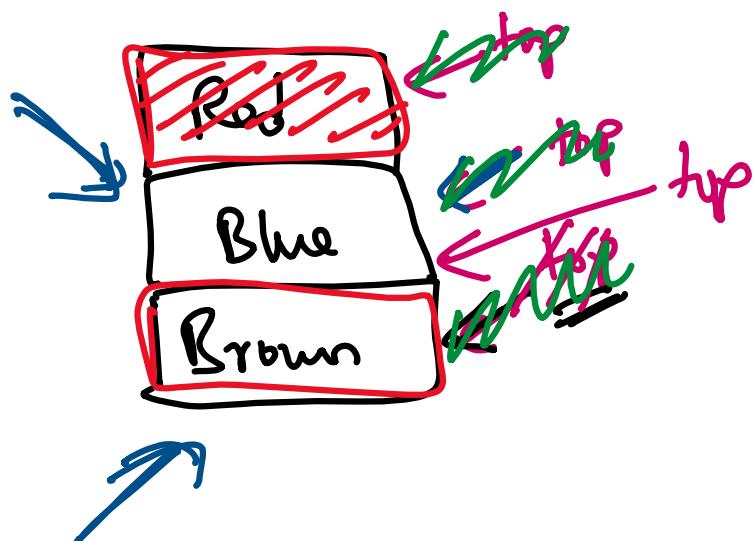
Saturday, October 18, 2025 11:24 AM

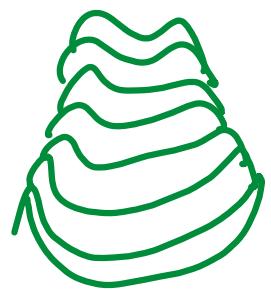
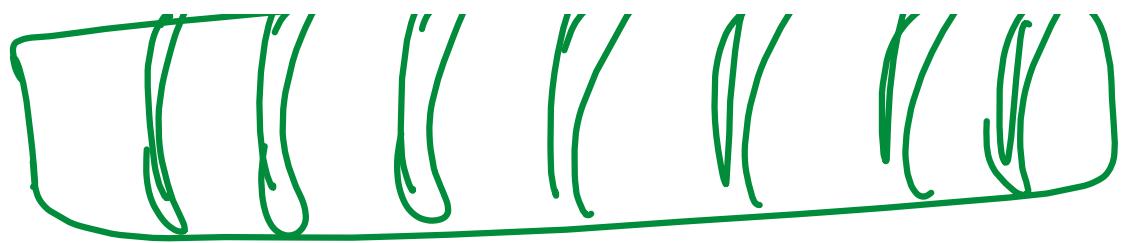
### What is a Stack?

- A Stack is a linear data structure that stores elements in a sequential order
- Insertion and Deletion of elements happen only from one end, called the top of the stack
- Follows the LIFO (Last In, First Out) principle:
  - When an element is inserted, it is the last element in the stack
  - When an element is removed, it is the first one to be removed from the stack



top → None

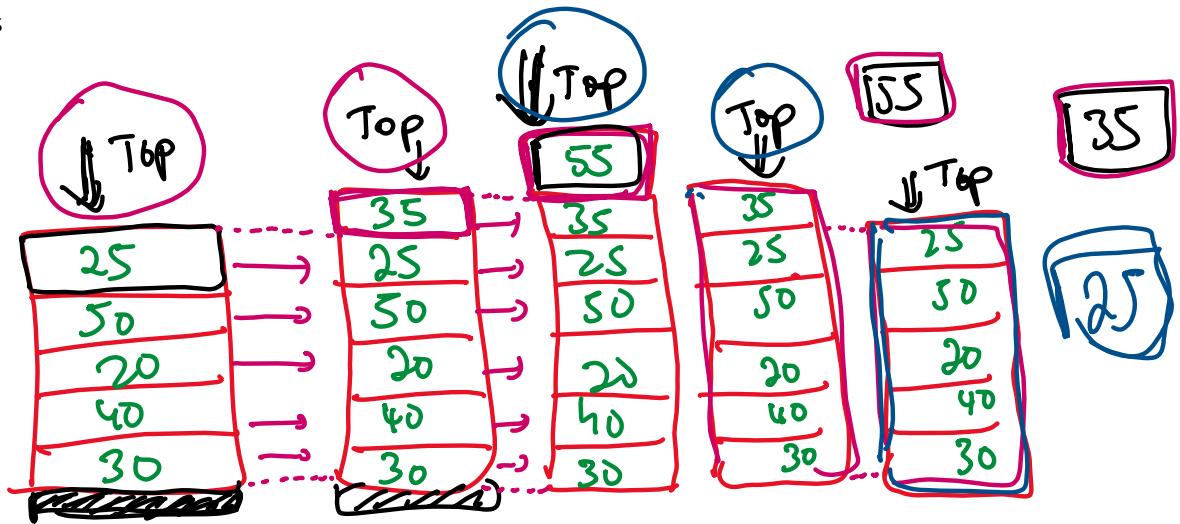




## 5.2 - Common Operations

Saturday, October 18, 2025 11:24 AM

- Push (Insert)
- Pop (Delete)
- Peek (Access)
- isEmpty









## 5.3 - Implementation

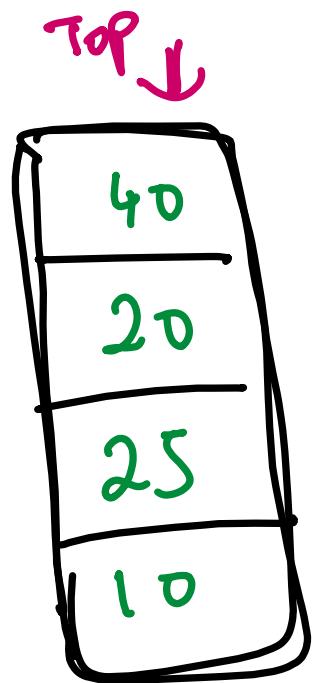
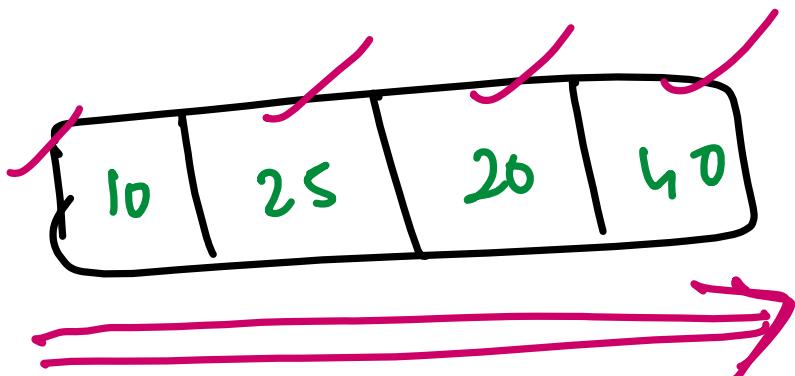
Saturday, October 18, 2025 12:08 PM

In Python, Stacks can be implemented in 3 different ways:



### 5.3.1 - Using Lists

Saturday, October 18, 2025 12:27 PM















### 5.3.2 - Using collections

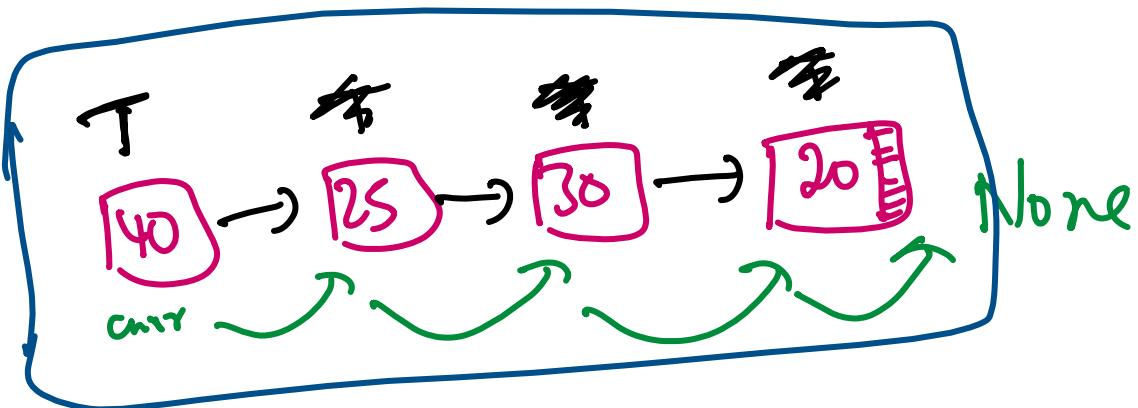
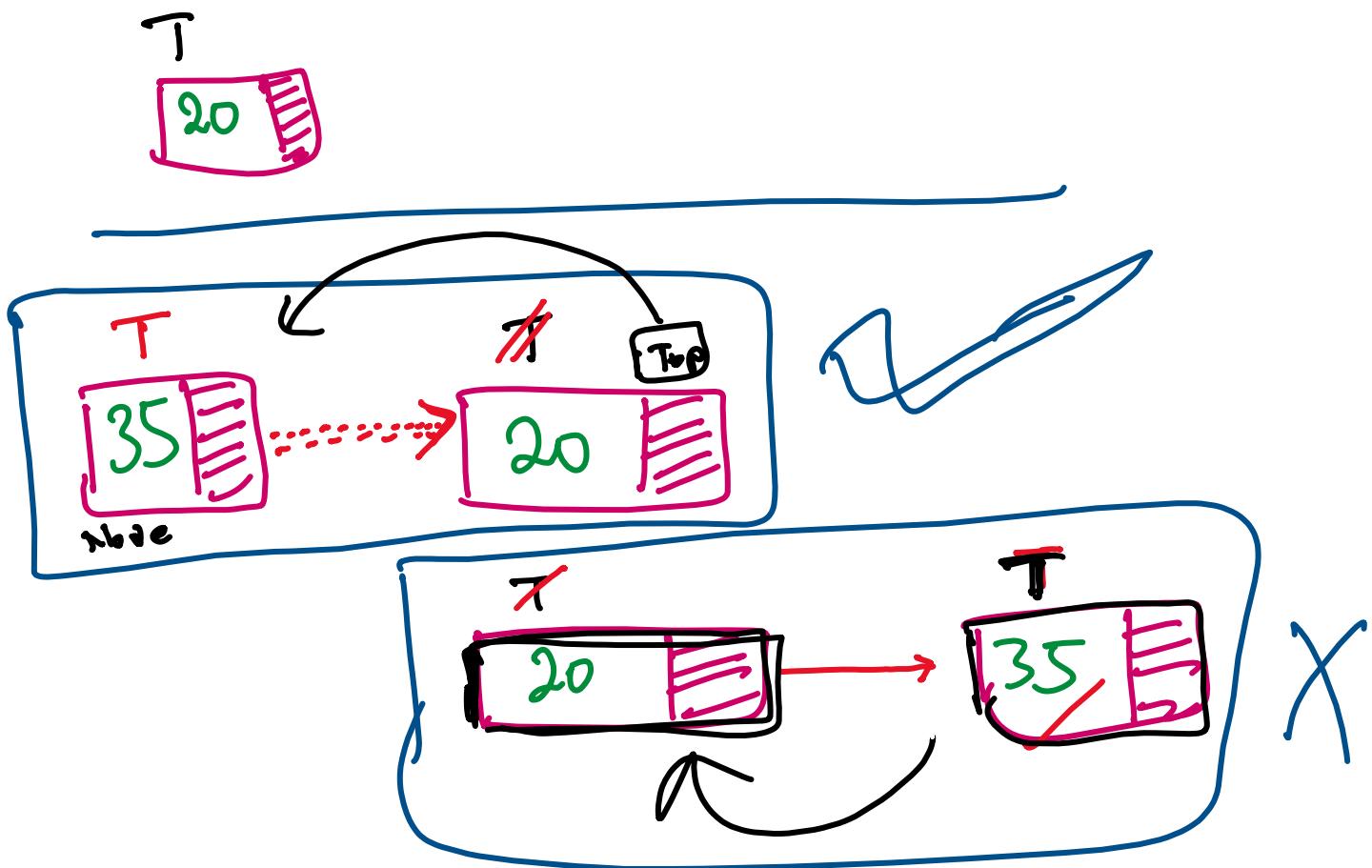
Saturday, October 18, 2025 12:27 PM

- Implemented using a high-performance data structure called **deque** from Python's built-in **collections** module
- A deque allows fast insertion and deletion of elements in O(1) time
- It's implemented as a **doubly-linked list** under the hood

### 5.3.3 - Using Linked Lists

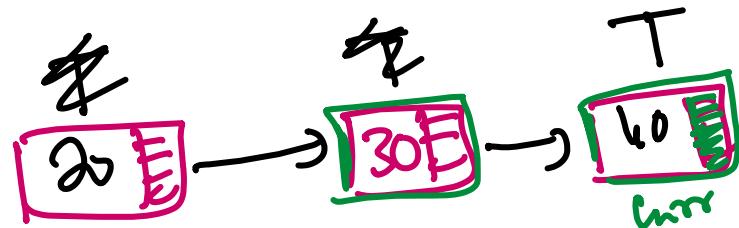
Saturday, October 18, 2025 12:32 PM

$\text{top} \rightarrow \text{None}$

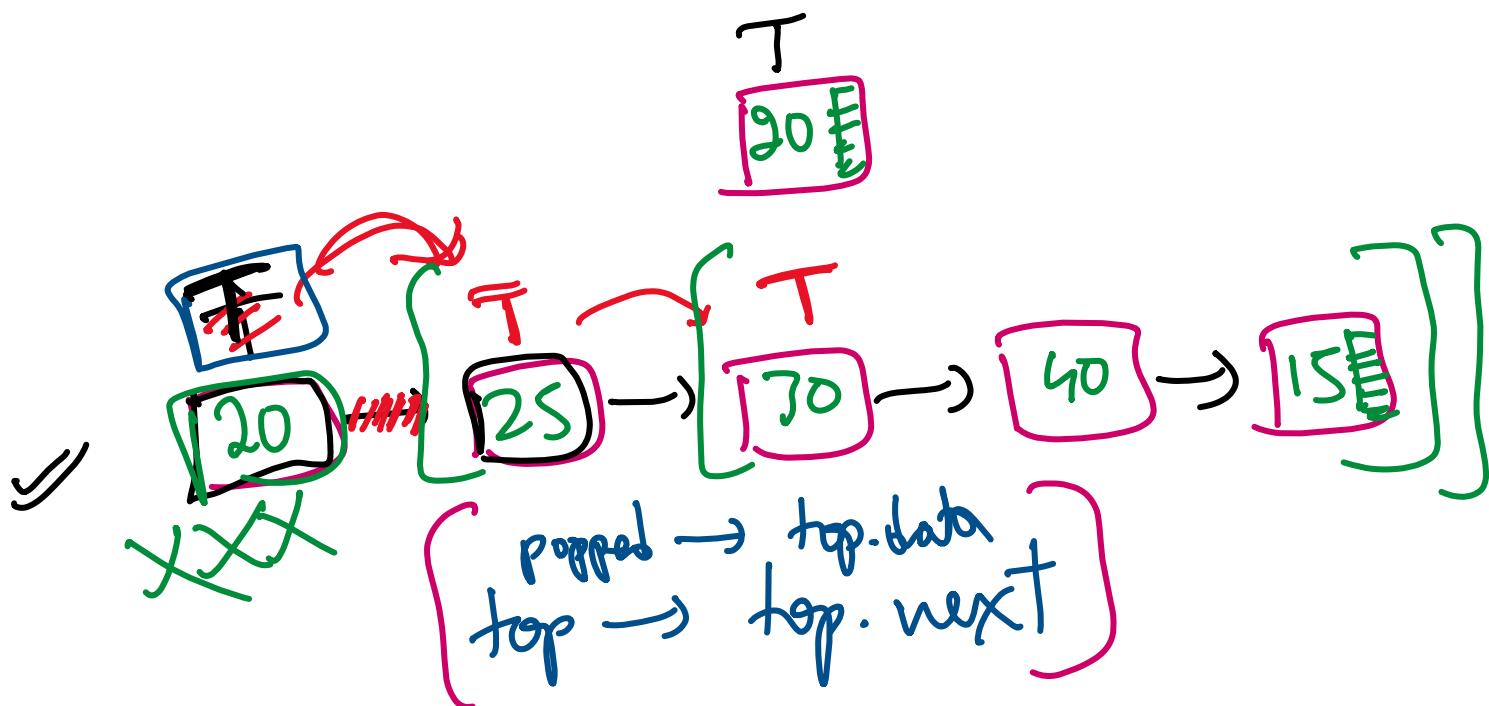


40  
25 } while curr  
30  
20

25  
30  
20 } min max val.



40



25











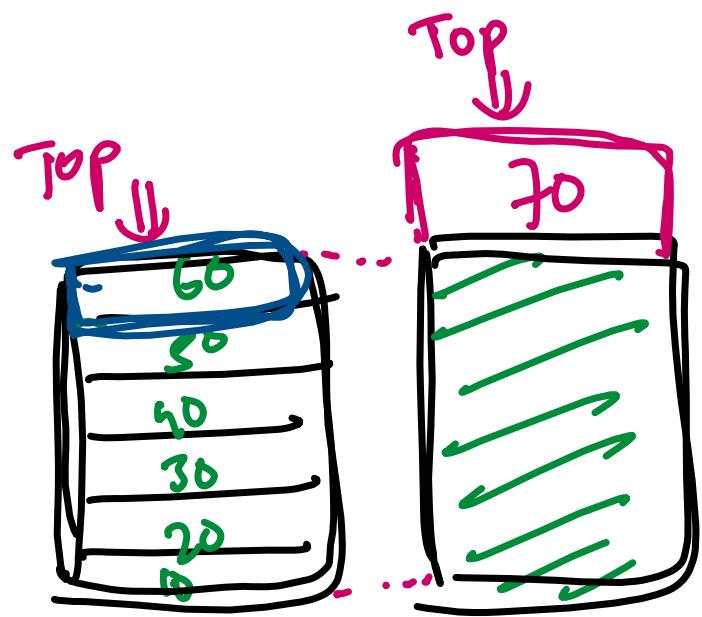
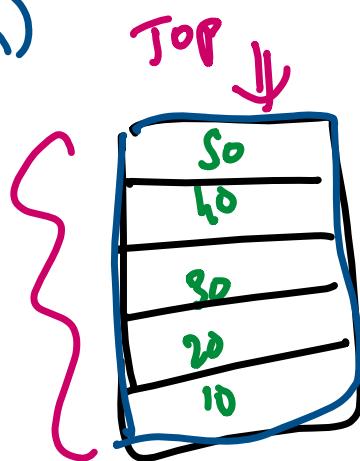
## 5.4 - Complexity Analysis

Saturday, October 18, 2025 12:08 PM

### Operations:

- Push  $\rightarrow O(1)$
- Pop  $\rightarrow O(1)$
- Peek  $\rightarrow O(1)$
- isEmpty  $\rightarrow O(1)$

Space  $\rightarrow O(n)$



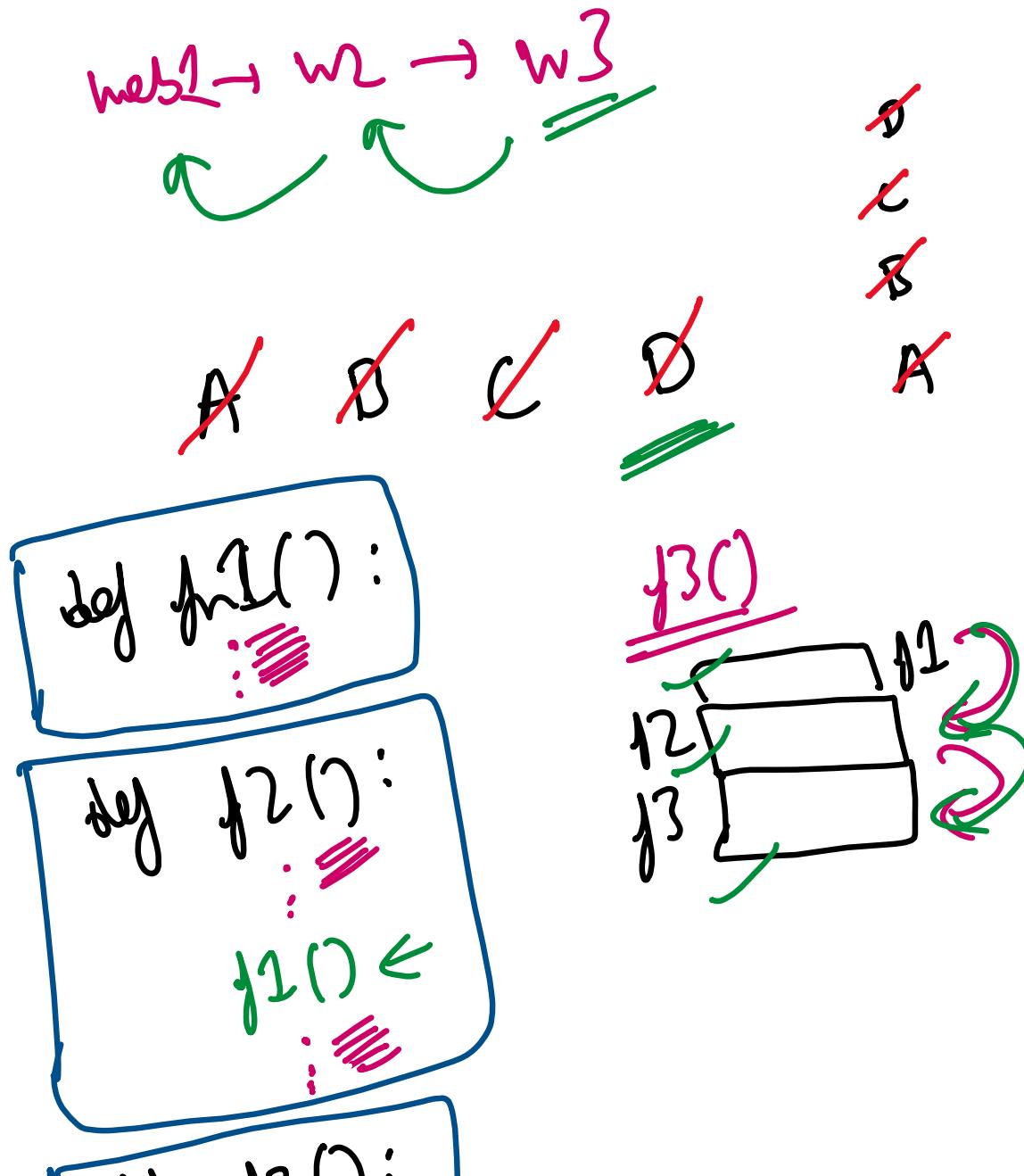




## 5.5 - Applications

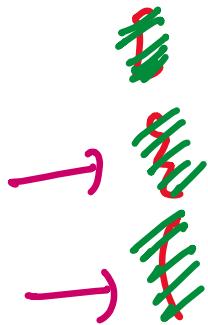
Saturday, October 18, 2025 12:08 PM

- Undo/Redo functionality in Text Editors
- Browser navigation (forward/backward)
- Function calls in programming → recursive
- Parentheses/Bracket matching



```
def f3():
    :
    f2() *
```

$$(2 + 4 \times \{5 + [1 + 1] - 4\}^3 + 3))$$



Tree-based RNNs

## 5.6 - Pros & Cons

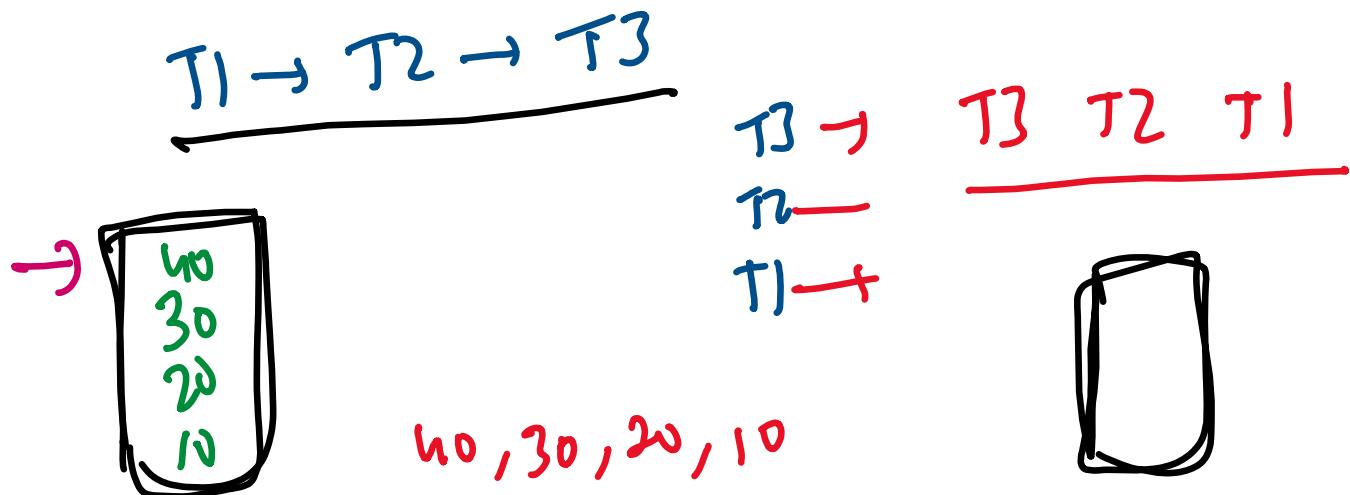
Saturday, October 18, 2025 12:11 PM

### Advantages:

- Efficient Memory Management: Since data is added and removed only from one end, **top**
- Useful in Backtracking Problems: Allows navigation to previous states
- Prevents Data Inconsistency
- Ideal for Reversing Data
- Helps in Expression Evaluation & Syntax Parsing: Compilers use stacks to manage operator precedence and parenthesis matching

### Disadvantages:

- Limited Access: Can't reach middle or lower elements since only **top** is available
- Limited usability: Not suitable for random access or sorting
- Difficult to Traverse/Search: Since only the **top** element is readily accessible

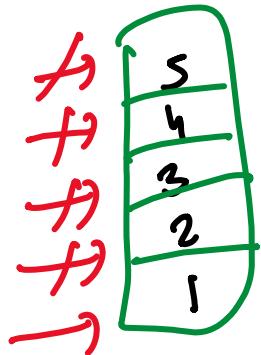


## 5.7 - Coding Problems

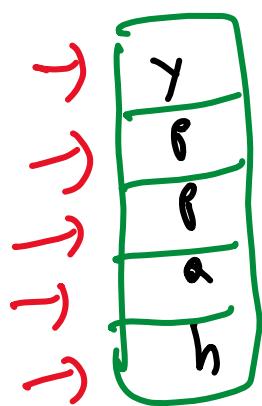
Saturday, October 18, 2025

12:09 PM

1 → 2 → 3 → 4 → 5



5 4 3 2 1



" happy "

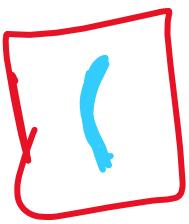
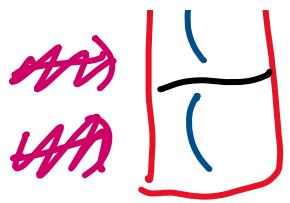
y p p a h

" happy "

y p p a h

" ( ( { [ ( ) ] } ) ) ) " ←

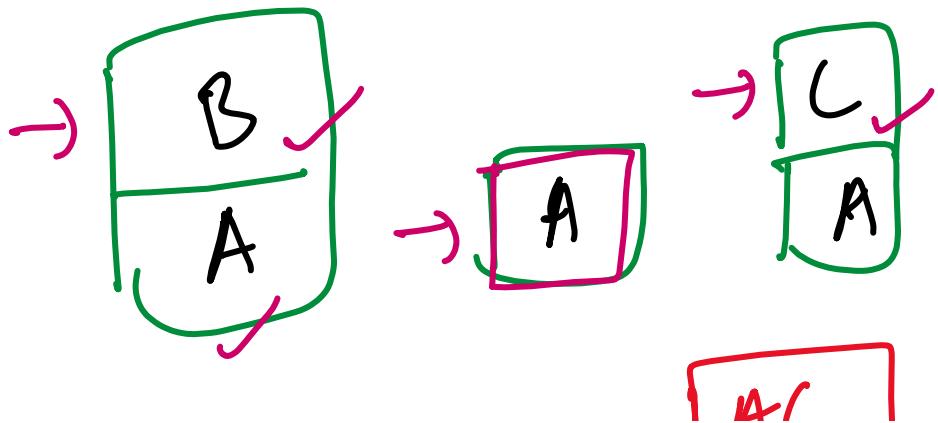
Diagram illustrating a stack or queue structure. On the left, there is a vertical stack of four boxes, each containing a different character: '(', '[', '{', and ')'. Above the stack, the string "( ( { [ ( ) ] } ) ) ) ) " is shown with arrows indicating the path of characters into the stack. The arrows point from the outermost characters inward, with some arrows being dashed to indicate they pass over already stacked characters.



## Imbalance

1. If recent closing bracket & popped bracket not match
2. String transd, stack not empty
3. Stack empty, still closing bracket

type A  
type B  
type C

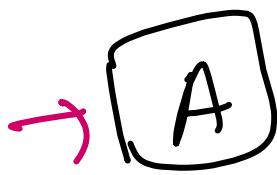


7 70 ~

~

AC

Type A  
emb  
emb

















## 5.8 - DS Context

Tuesday, December 16, 2025 2:00 AM

### Data Science / ML Correlation:

- Neural network forward/backward pass

### Libraries & Usage:

- PyTorch *autograd*
- TensorFlow graph execution

# 6. Queues

Saturday, October 18, 2025 3:01 PM

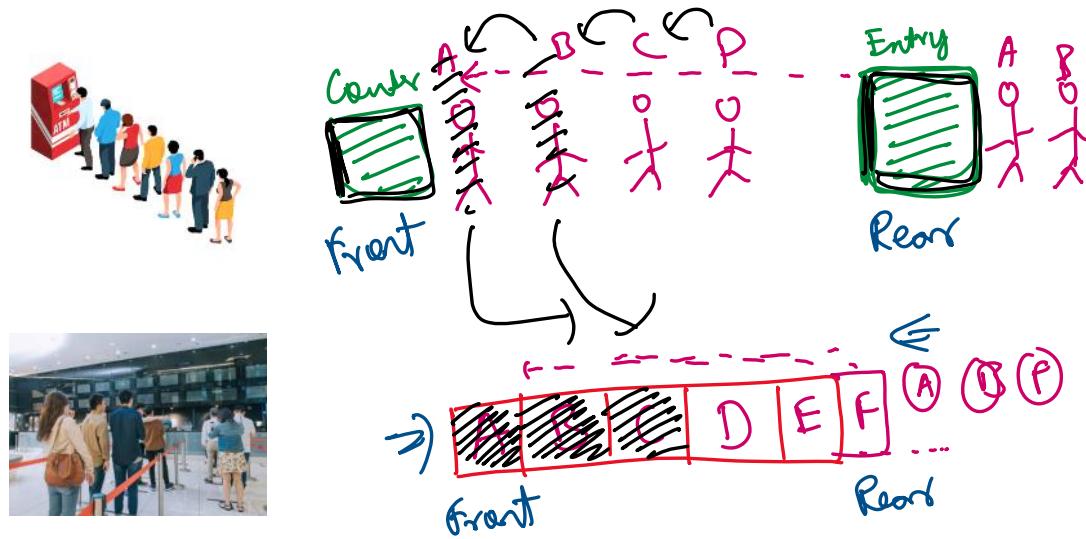
- **Introduction**
- **Common Operations**
- **Implementation**
- **Complexity Analysis**
- **Coding Problems**

## 6.1 - Introduction

Saturday, October 18, 2025 3:10 PM

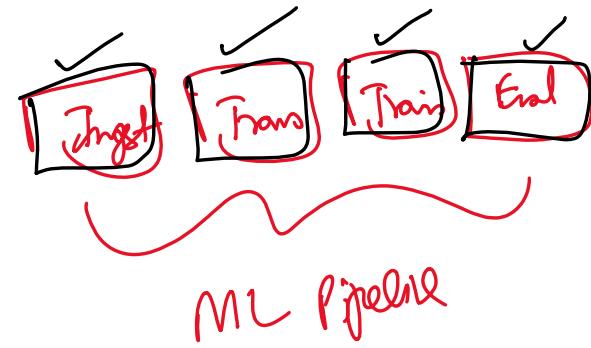
### What is a Queue?

- A Queue is a linear data structure that follows the FIFO (First In, First Out) principle — the first element added to the queue is the first one to be removed



### Real-World Applications:

- Call Center Systems:**
  - Incoming calls are kept in a queue until an agent is free
  - The first caller waits the least, while others wait behind in line
  - Ensures fairness and orderly service
- CPU / Process Scheduling:**
  - Operating systems use **ready queues** to manage processes waiting for CPU time
  - The process that enters first gets executed first
  - To manage multiple processes efficiently without conflicts
- Data Preprocessing:**
  - During ML training, large datasets are processed in stages
  - Each stage can push processed data into a queue, from which the next stage consumes it
  - In TensorFlow, `tf.data.Dataset` uses iterators and queues internally to manage data pipelines efficiently
- Streaming Data & Online Learning:**
  - In **real-time ML** or **stream processing**, incoming data is buffered using a queue
  - Data from the queue is then processed or fed into an online learning model
- Model Serving & Inference:**
  - Deployed ML models receive multiple inference requests simultaneously
  - A queue manages the incoming prediction requests, ensuring FIFO (first-in-first-out) and system stability
  - Ex: **Redis Queue**, **RabbitMQ**



### Advantages:

- Maintains Order via the FIFO Principle
- Efficient for Sequential Processing:** Ideal when data needs to be handled one at a time
- Prevents Overload in Systems:** By queuing requests, servers and routers can handle high loads gracefully instead of crashing
- Wide-level Usage:** Essential for algorithms like **Breadth-First Search (BFS)**, **Level Order Traversal**, and **CPU scheduling**

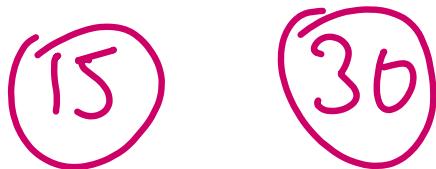
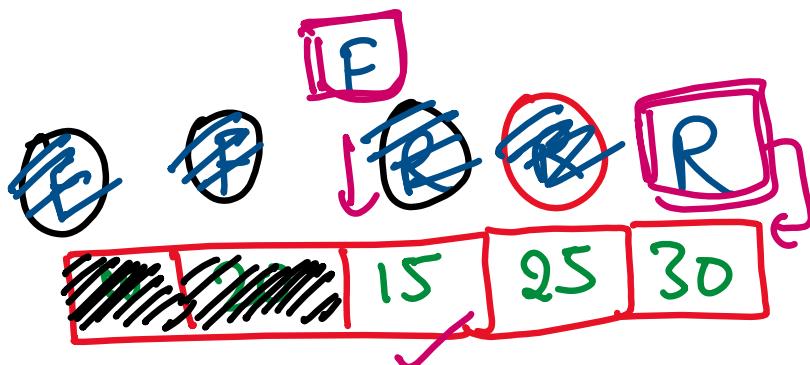
### Disadvantages:

- **Limited Access:**
  - Only the **front** and **rear** elements can be accessed directly
  - Cannot access or modify elements in the middle easily
- **Inefficient in List-Based Queues:**
  - If the queue is large, deletion can take  $O(n)$  time
- **Not Suitable for Random Access:**
  - Queues don't support random access or search operations efficiently

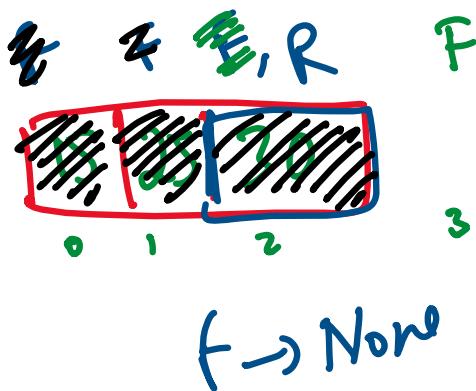
## 6.2 - Common Operations

Saturday, October 18, 2025 3:13 PM

- **Enqueue:** Insert element at **rear**
- **Dequeue:** Remove element from **front**
- **FrontPeek:** View first element
- **RearPeek:** View last element
- **isEmpty:** Check if queue is empty



enqueue(25)  
enqueue(30)



frontPeek()

dequeue()  
dequeue()

## 6.3 - Implementation

Saturday, October 18, 2025 3:13 PM

**In Python, Queues can be implemented in multiple ways:**

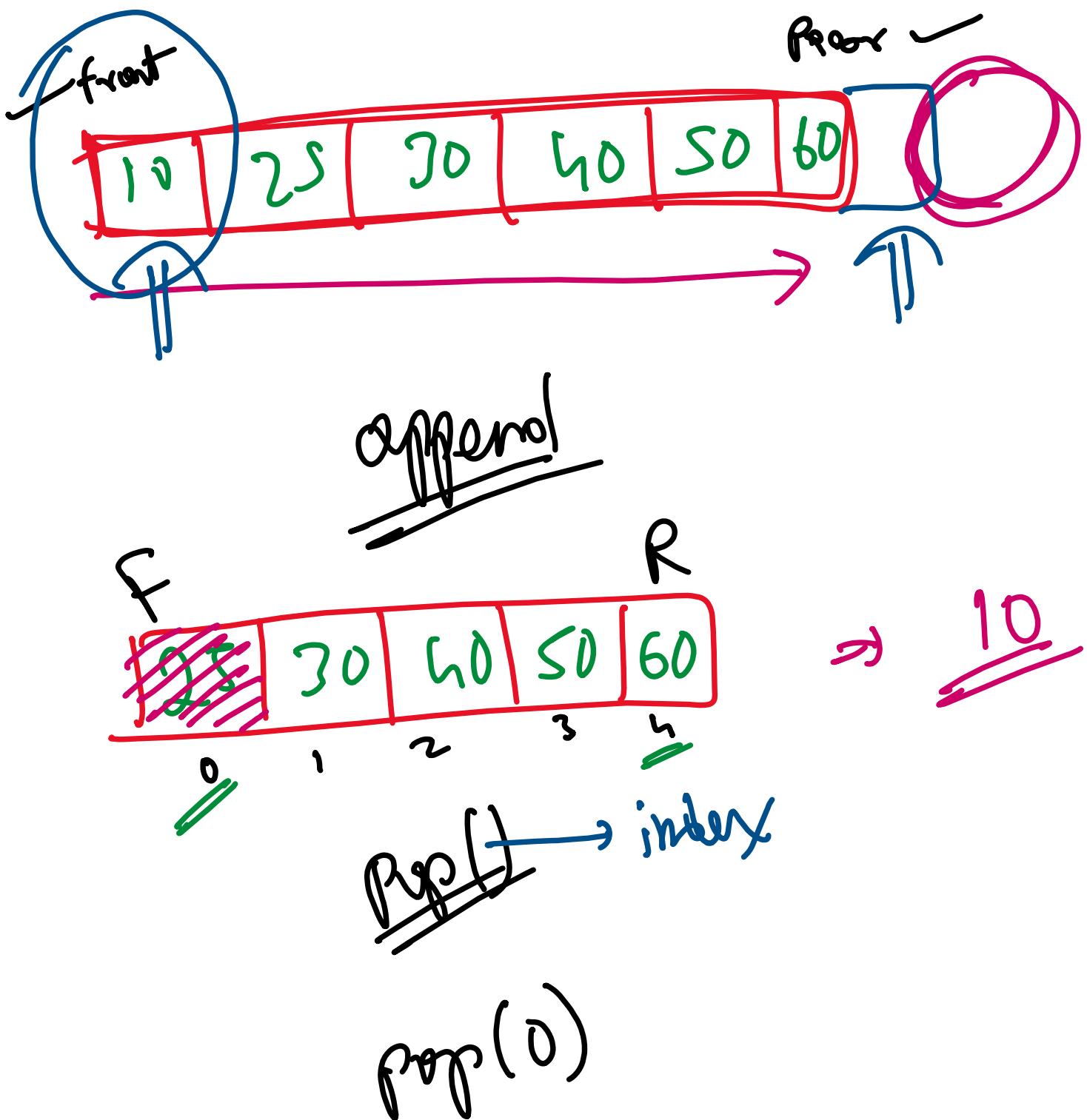
- **Lists**
- **Linked List**
- ***collections* → deque**
- ***heapq***: Used for implementing **Priority Queues**
- ***queue***: Handles locking and blocking; suitable for cross-thread communication
- ***asyncio***: Suitable for **async/await** contexts

**NOTE:** The modules - ***queue*** and ***asyncio*** implementations aren't much used in DSA/CP context



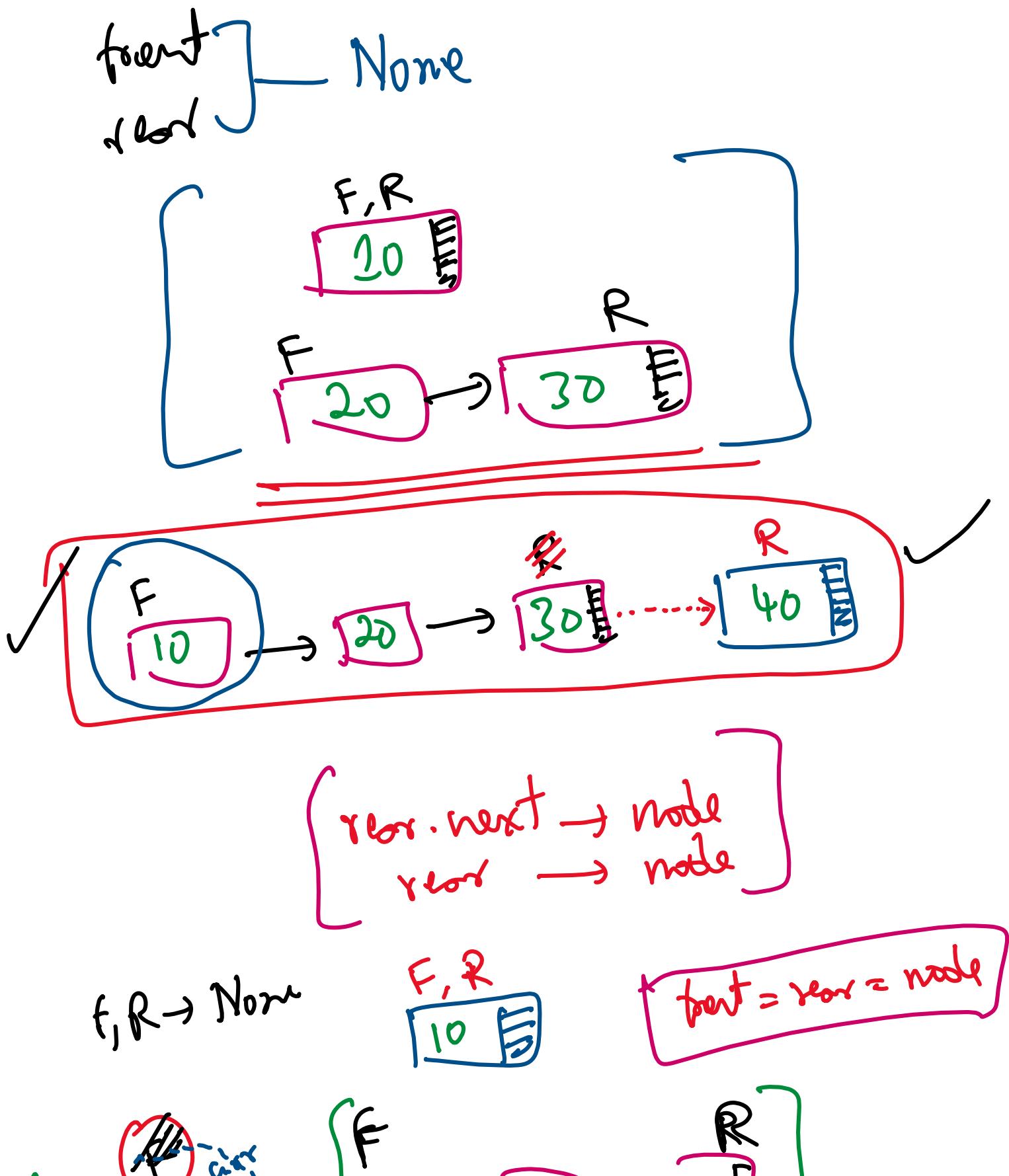
### 6.3.1 - Using List

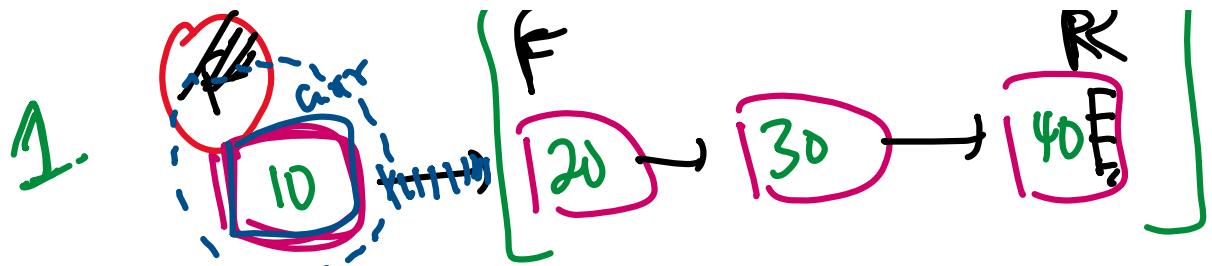
Wednesday, October 22, 2025 9:19 PM



### 6.3.2 - Using Linked List

Wednesday, October 22, 2025 9:19 PM





*curr = front  
front → front.next*

*front → None*

*front → None*

*return curr.data*

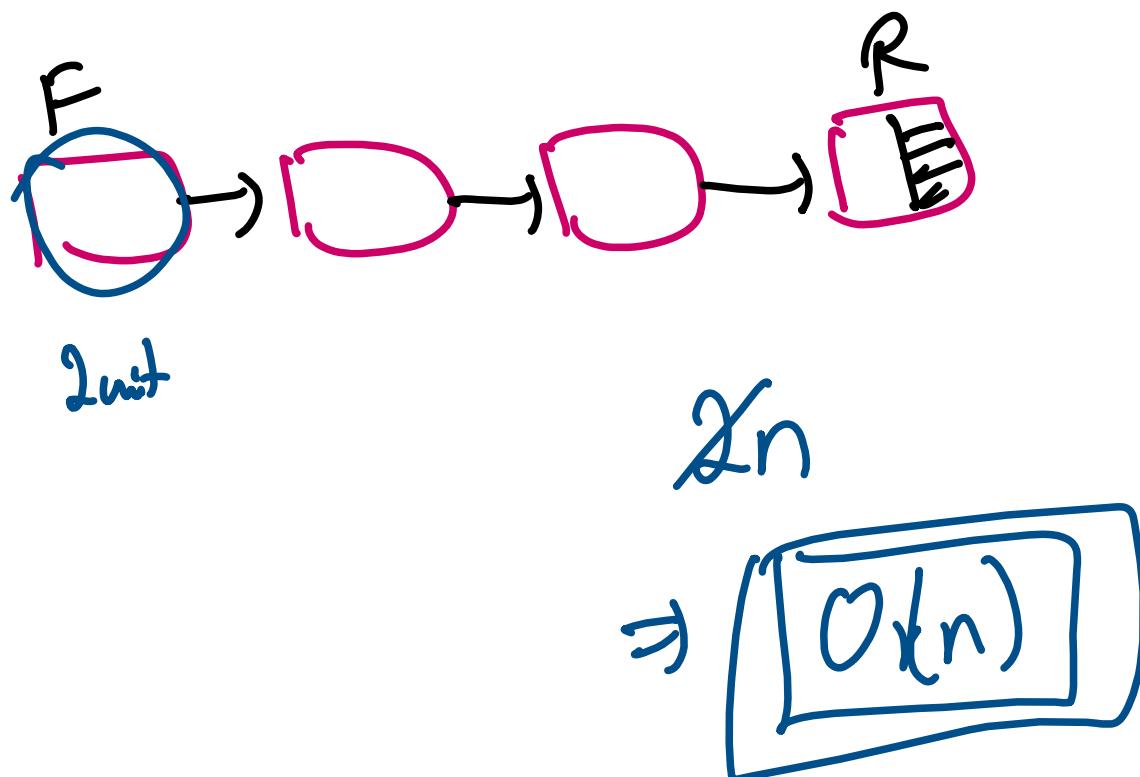
### 6.3.3 - Using collections

Wednesday, October 22, 2025 9:19 PM

## 6.4 - Complexity Analysis

Saturday, October 18, 2025 3:24 PM

- Enqueue:  $O(1)$
- Dequeue:  $O(1)$
- FrontPeek:  $O(1)$
- RearPeek:  $O(1)$
- isEmpty:  $O(1)$

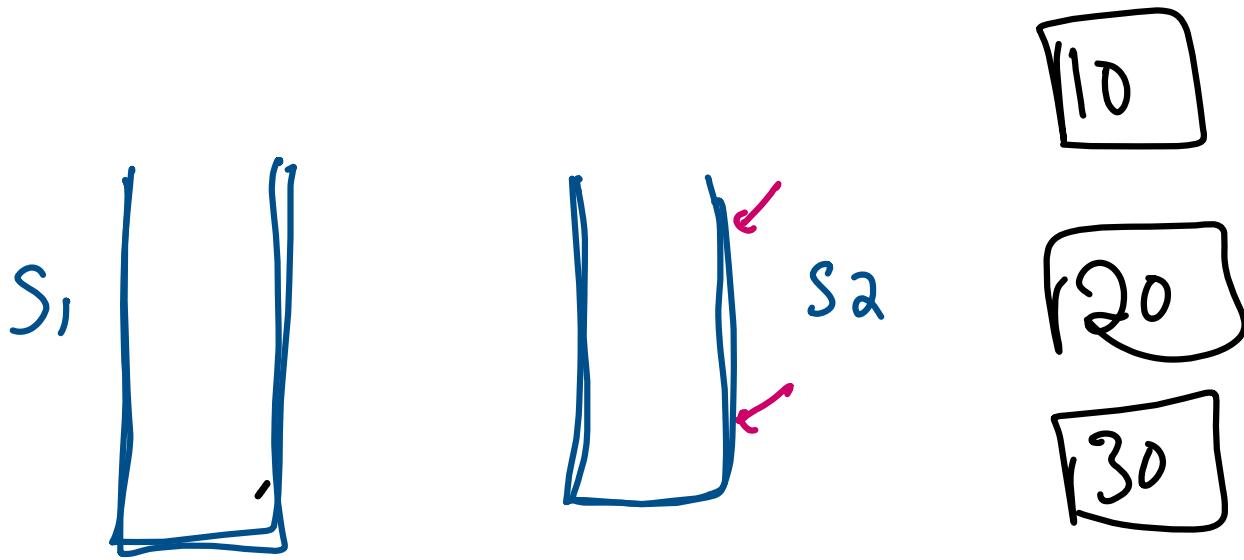






## 6.5 - Coding Problems

Saturday, October 18, 2025 3:24 PM



Enqueue :

S1 → push

Dequeue :

[ S1.pop() → S2.push() ]

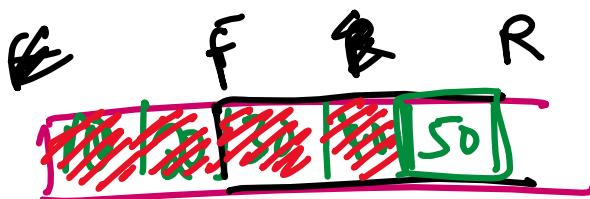
S2.pop()

S1 → empty  
S2 → empty

Queue  
empty



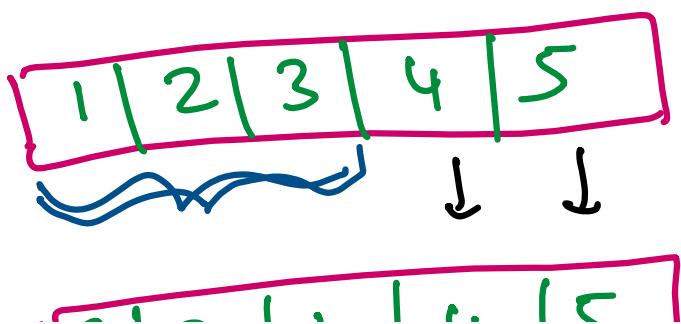
Cannot  
Dequeue



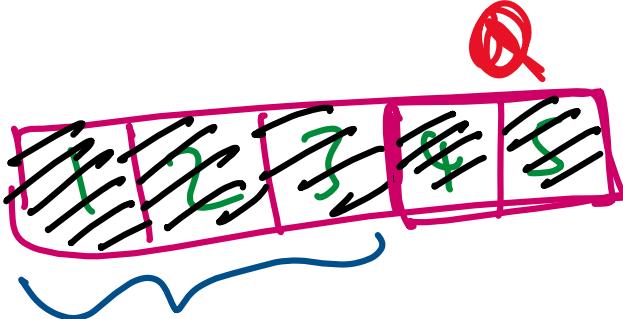
q → Queue

k → integer

k = 3



3 | 2 | 1 | 4 | 5

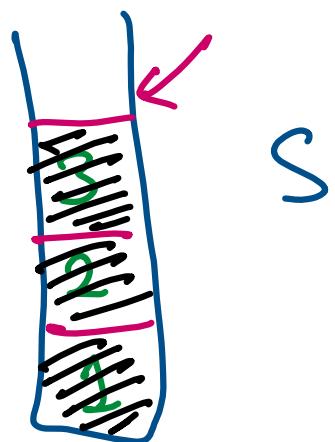
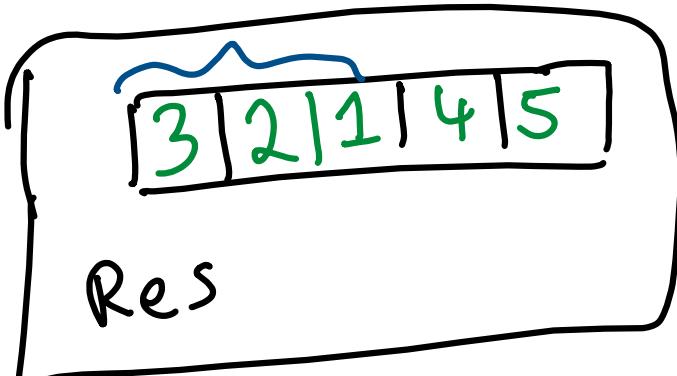


$$k=3$$

for  $i \leq k$

$S.push(q.dequeue())$

$res.enqueue(S.pop())$

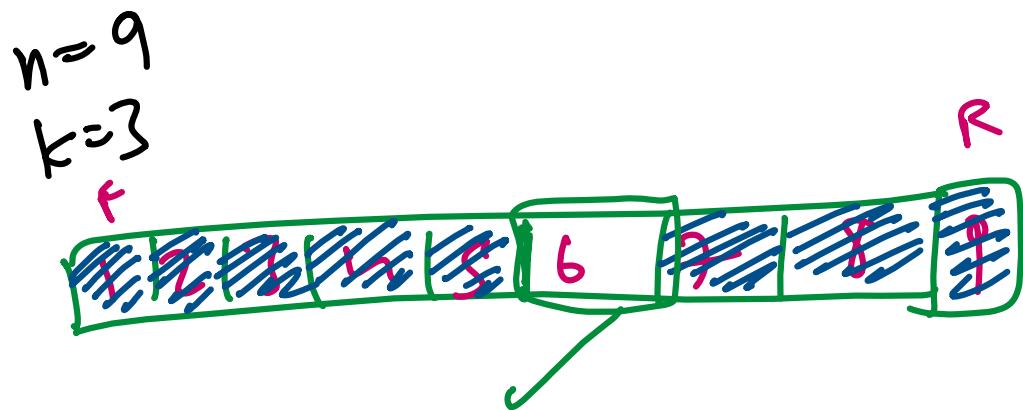
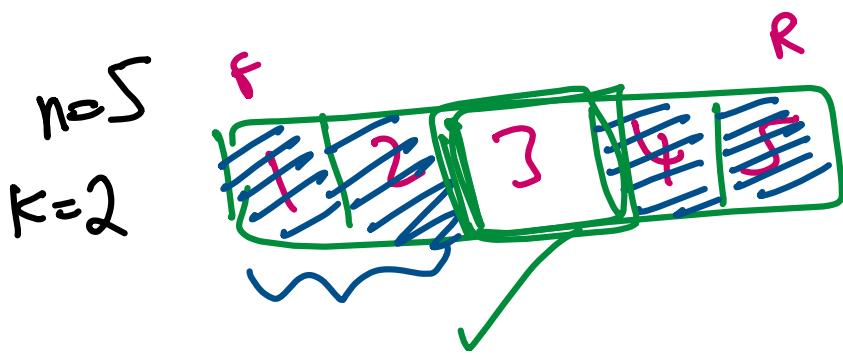


$res.enqueue(q.dequeue())$

$n$   
 $K$

$K \leq n$

$\leftarrow$   $\nwarrow$   $\xrightarrow{\quad}$   $R$



if  $n \rightarrow 1$ :

returns  $n$

$q = \text{degree}()$   
 $\text{for } x \in \text{range}(n)$   
 $q.append(x)$

create  
array

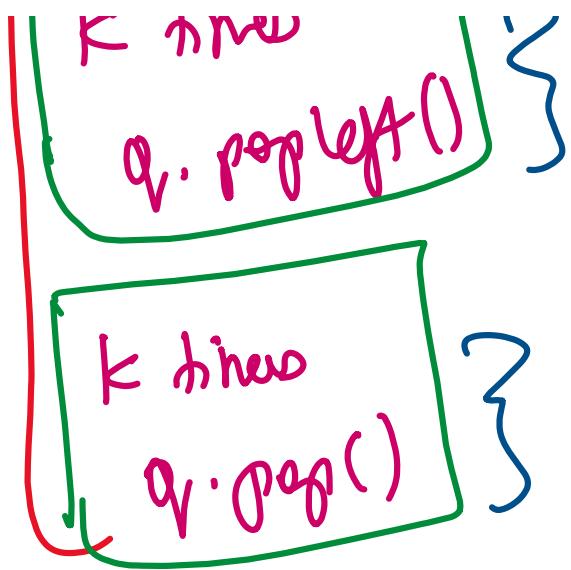
while True

$f$  k times

... n

1. 1 1 1 1

unvi



last "K"



$k=1$



















## 6.6 - DS Context

Tuesday, December 16, 2025 2:06 AM

### Data Science / ML Correlation:

- Mini-batch training
- Stochastic Gradient Descent (SGD) batches

### Libraries & Usage:

- TensorFlow Data API (*tf.data.Dataset*)
- PyTorch *DataLoader*

## 7. Searching

Wednesday, October 29, 2025 10:09 PM

- **Linear Search**
- **Binary Search**
- **Exponential Search**

## 7.1 - Linear Search

Monday, November 3, 2025 9:30 PM

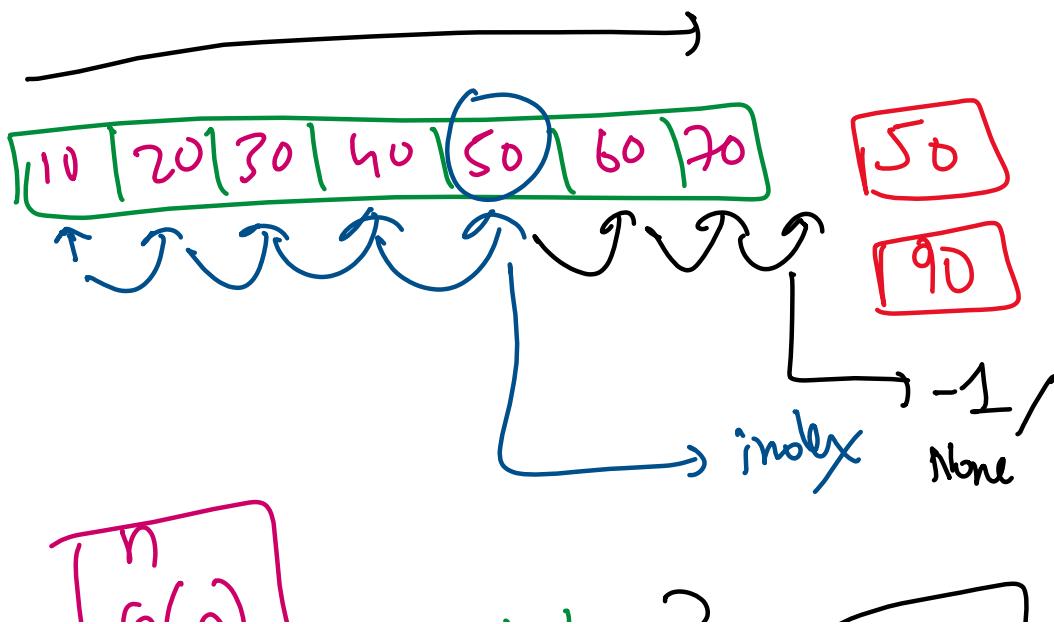
- Linear search, also known as sequential search, is a fundamental search algorithm used to find a specific element within a data structure (array, linked list, etc.)
- It operates by sequentially examining each element in the collection until either the target element is found or the end of the collection is reached

### Working:

- The search begins at the first element of the data structure
- Each element is compared to the target value
- If a match is found, the index of the element is returned, and the search terminates
- If no match is found after checking all elements, the algorithm indicates that the element is not present

### Characteristics:

- **Simplicity:** Linear search is easy to understand and implement
- **No Sorting Required:** It does not require the data to be sorted, making it suitable for unordered lists
- **Time Complexity:** In the worst-case scenario (element not present or at the very end), linear search has a time complexity of  $O(n)$
- **Space Complexity:** It has a space complexity of  $O(1)$  as it only requires a constant amount of extra space



$n$   
 $O(n)$

index  
temp

$O(1)$

## 7.2 - Binary Search

Monday, November 3, 2025 9:30 PM

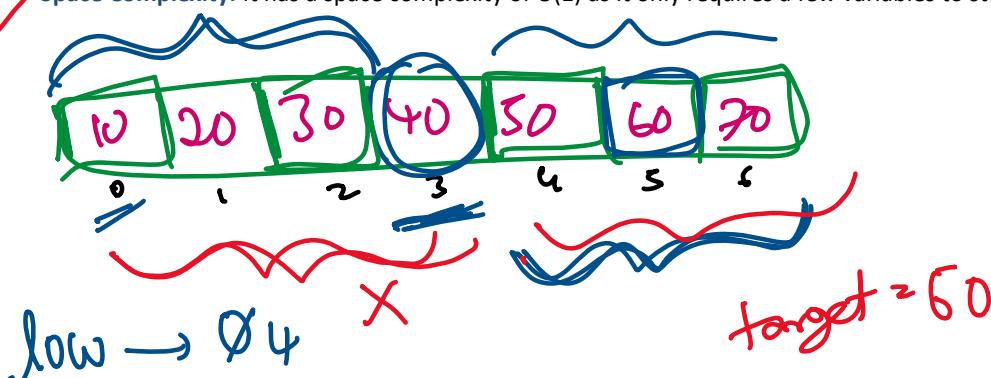
- Binary Search is an efficient algorithm used to find the position of a target value within a sorted array or list
- It works on the principle of "divide and conquer," repeatedly dividing the search interval in half

### Working:

- **Initialize Pointers:** Set a low pointer to the beginning of the array and a high pointer to the end of the array
- **Calculate Midpoint:** Calculate the middle index mid as  $\text{low} + (\text{high} - \text{low}) / 2$  (to prevent potential integer overflow)
- **Compare and Adjust:**
  - If the element at **mid** is equal to the target value, the search is successful, and **mid** is returned
  - If the element at **mid** is less than the target value, the target must be in the right half of the array; Update **low** to **mid + 1**
  - If the element at **mid** is greater than the target value, the target must be in the left half of the array; Update **high** to **mid - 1**
- **Repeat:** Continue steps 2 and 3 until the target is found or the low pointer crosses the **high** pointer (indicating the target is not in the array)

### Characteristics:

- ✓ **Sorted Data Requirement:** Binary Search is only applicable to data structures that are sorted (ascending or descending)
- ✓ **Time Complexity:** It has a time complexity of  $O(\log N)$ , making it significantly faster than linear search ( $O(N)$ ) for large datasets
- ✓ **Space Complexity:** It has a space complexity of  $O(1)$  as it only requires a few variables to store pointers



$$\begin{aligned} \text{high} &\rightarrow n-1 \rightarrow 6 \\ \text{mid} &\rightarrow \text{low} + (\text{high}-\text{low})/2 \end{aligned}$$

$$\frac{6-0}{2} \Rightarrow 3 \Rightarrow 3$$

while  $\text{low} \leq \text{high}$  :

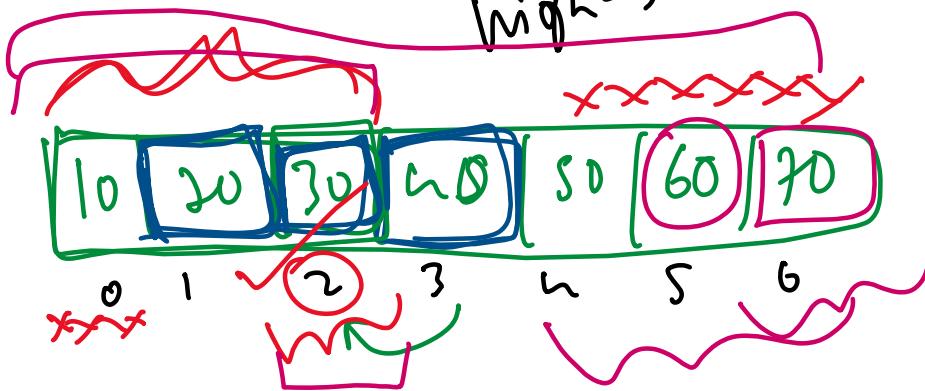
$$\begin{aligned} \text{mid} &= \text{low} + (\text{high}-\text{low})/2 \\ [\text{mid}] &\approx \underline{\text{target}} \end{aligned}$$

$$\begin{aligned} \text{mid} &\rightarrow 3 \\ 4 + \frac{(6-4)}{2} &\rightarrow 5 \end{aligned}$$

$\text{mid} = \dots$   
 if  $\text{arr}[\text{mid}] == \text{target}$ :  
 return  $\text{mid}$

if  $\text{arr}[\text{mid}] < \text{target}$ :  
 $\text{low} \rightarrow \text{mid} + 1$

else:  
 $\text{high} \rightarrow \text{mid} - 1$



$\text{low} \rightarrow 2$

$$\text{mid} \rightarrow 0 + \frac{(6-0)}{2} \rightarrow 0 + \frac{6}{2} \rightarrow 3$$

$\text{high} \rightarrow \frac{6}{2}$

$$0 + \frac{(2-0)}{2} \rightarrow \frac{2}{2} \rightarrow 1$$

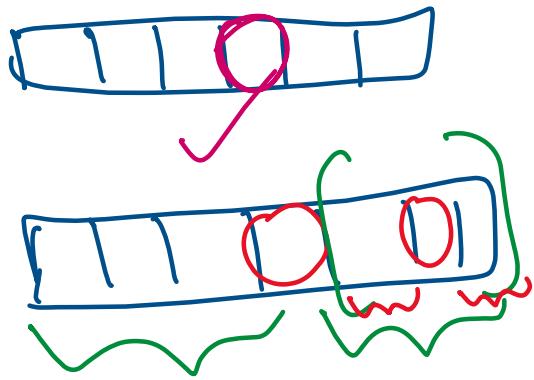
$$2 + \frac{(2-2)}{2} \rightarrow 2 + \frac{0}{2} \rightarrow 2$$



$$h \sim 10^{31} \quad l \sim 10^{31}$$

$L \sim d'$

↳ numerical  
overflow



$$\checkmark l \sim O^n$$

$$\frac{n}{2} \rightarrow O(n)$$

$$\log(n)$$

## 7.3 - Exponential Search

Monday, November 3, 2025 9:30 PM

- Exponential Search, also known as Doubling Search or Galloping Search, is a search algorithm particularly efficient for sorted arrays
- Most useful when the array is very large or its size is unknown (unbounded arrays)

Working: Operates in 2 main stages

### 1. Finding the Range:

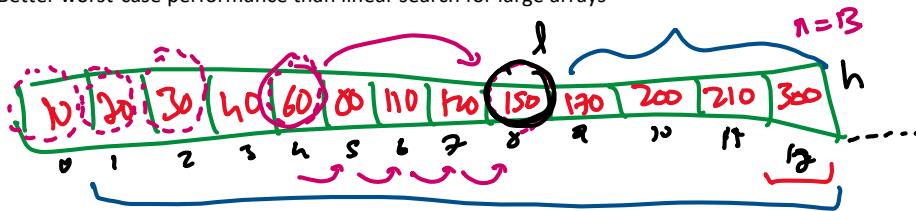
- The algorithm starts by checking the first element of the array; If it's the target, the search ends
- Otherwise, it iteratively doubles the index (**bound = 1, then 2, 4, 8, etc.**) as long as the element at the current bound index is less than the target value and the **bound** remains within the array limits
- This stage aims to find a range [**bound/2, min(bound, n-1)**] where n is the array size, such that the target element, if present, must lie within this range

### 2. Binary Search within the Range:

- Once the appropriate range is identified, a standard Binary Search algorithm is applied within this narrowed-down range to locate the exact position of the target element

### Characteristics:

- Efficient for very large or unbounded sorted arrays, especially when the target is near the beginning
- Better worst-case performance than linear search for large arrays



Target  $\rightarrow 120$

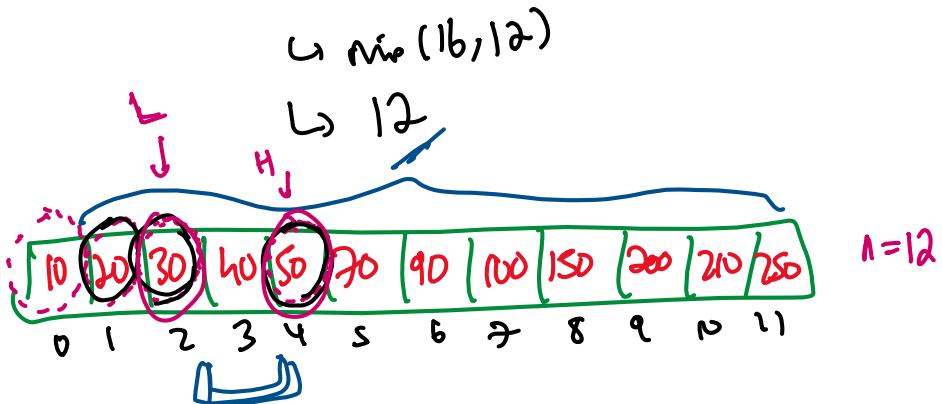
$i=1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64, \dots$

while ( $i < n$ ) :  
 $i *= 2$

$i=1 \xrightarrow{2} \xrightarrow{4} \xrightarrow{8} 16 \times$

low  $\rightarrow i // 2 \rightarrow 8$

high  $\rightarrow \min(i, n-1)$   
 $\hookrightarrow \min(16, 13-1)$   
 $\hookrightarrow \min(16, 12)$



target  $\rightarrow 40$

$$50 < 40 \times$$

$i = l \times 4$

$(i < n) \& (\text{arr}[i] \geq \text{target})$

$\boxed{\text{low} \rightarrow 2 \ (i // 2)}$

$\boxed{\text{high} \rightarrow \min(4, 11) \rightarrow 4}$

$\boxed{\min(i, n-1)}$



## 7.4 - DS Context

Tuesday, December 16, 2025 4:15 AM

### Data Science / ML Correlation:

- Hyperparameter tuning
- Threshold optimization
- Nearest neighbor search

### Libraries & Usage:

- **scikit-learn:** *GridSearchCV*
- **FAISS:** similarity search
- **scipy:** optimization



# 8. Sorting

Wednesday, November 5, 2025 10:11 PM

- **Introduction**
- **Bubble Sort**
- **Selection Sort**
- **Insertion Sort**
- **Merge Sort**
- **Quick Sort**
- **Counting Sort**
- **Complexity Analysis**

## 8.1 - Introduction

Tuesday, November 11, 2025 10:53 PM

### What is Sorting?

Sorting is the process of arranging data in a particular format or order (ascending or descending)

### Why Sorting?

- Helps in efficient searching (e.g., Binary Search)
- Simplifies data visualization and analysis
- Essential in data processing, ranking, and optimization problems

### Classifications of Sorting:

#### 1. Based on Implementation:

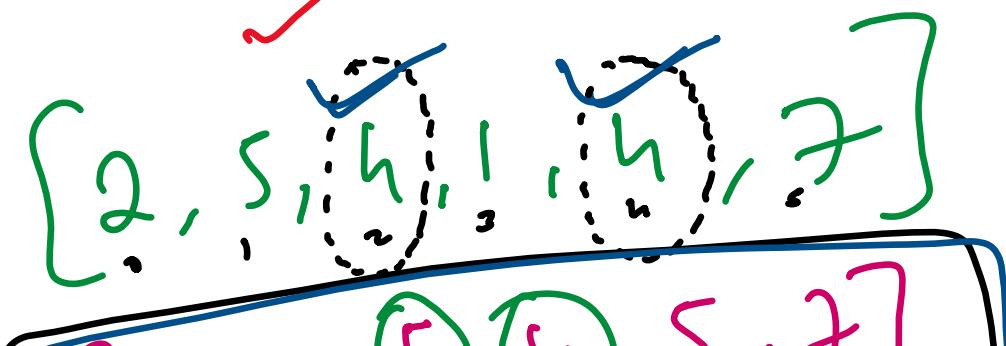
- **Internal Sorting:** Entire data fits in main memory (e.g., Quick Sort, Merge Sort)
- **External Sorting:** Data stored in external memory (e.g., External Merge Sort)

#### 2. Based on Stability:

- **Stable:** Equal elements maintain their relative order (e.g., Merge Sort, Insertion Sort)
- **Unstable:** Equal elements may change order (e.g., Quick Sort, Heap Sort)

#### 3. Based on Time Complexity:

- **Quadratic Time ( $O(n^2)$ ):** Bubble, Selection, Insertion
- **Log-linear Time ( $O(n \log n)$ ):** Merge, Quick, Heap



[1, 2, 4, 4, 5, 2]







## 8.2 - Bubble Sort

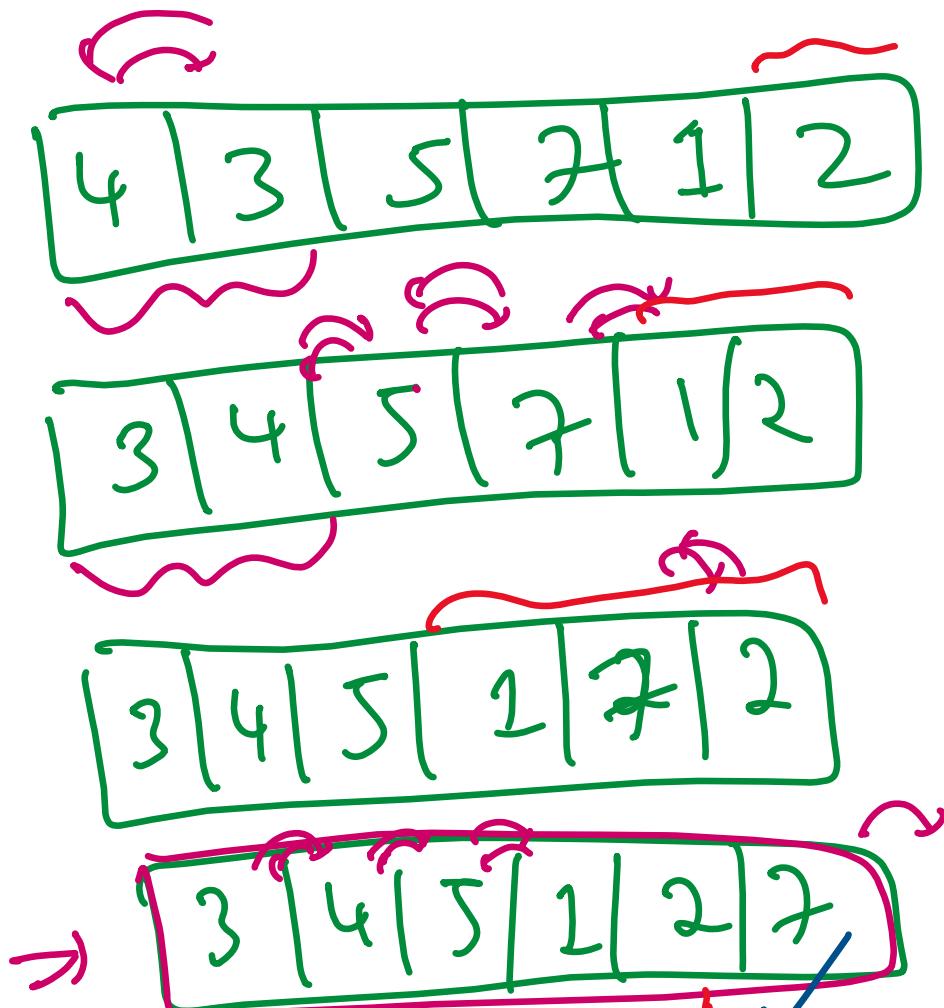
Tuesday, November 11, 2025 11:00 PM

### Main Ideas:

- Compare **neighboring pairs**
- Swap if they're out of order
- Keep repeating until no swaps are needed
- After each pass, the largest element moves to the end

### Key Points:

- Very easy to understand and implement
- Inefficient for large lists



$\rightarrow$   $3 \boxed{1} 4 \boxed{2} 1 \boxed{4} 5 \boxed{6} 7$

$3 \boxed{4} 1 \boxed{2} \cancel{8} \cancel{5} \cancel{6} \boxed{7}$

$3 \boxed{4} 1 \boxed{2} \cancel{5} \cancel{6} \boxed{7}$

$3 \boxed{1} 4 \cancel{2} \cancel{5} \cancel{6} \boxed{7}$

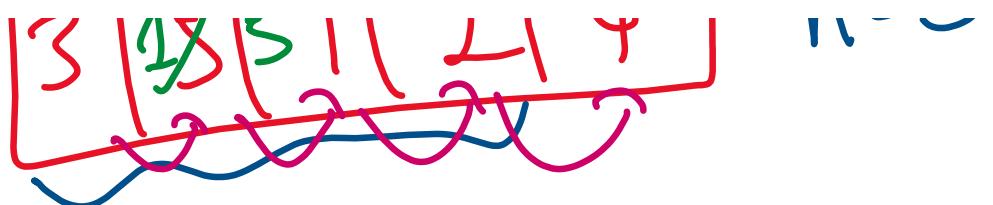
$3 \boxed{1} \cancel{2} \cancel{4} \cancel{5} \cancel{6} \boxed{7}$

$1 \boxed{3} \cancel{2} \cancel{4} \cancel{5} \cancel{6} \boxed{7}$

$1 \cancel{2} \cancel{3} \cancel{4} \cancel{5} \cancel{6} \boxed{7}$

$0 \quad 1 \quad 2 \quad 3 \quad 4$   
 $3 \boxed{1} \cancel{5} \cancel{4} \cancel{2} \cancel{6} \boxed{7}$

$n=5$

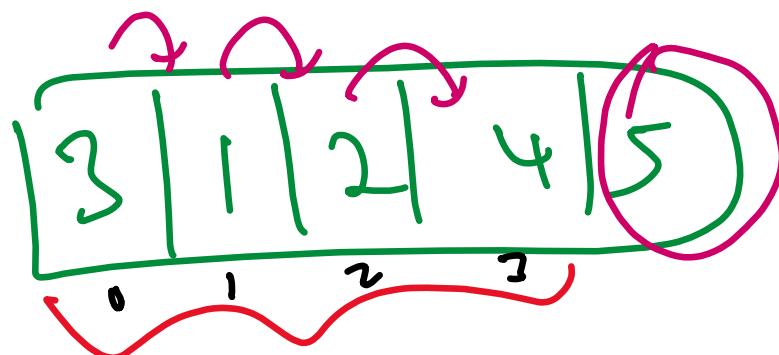
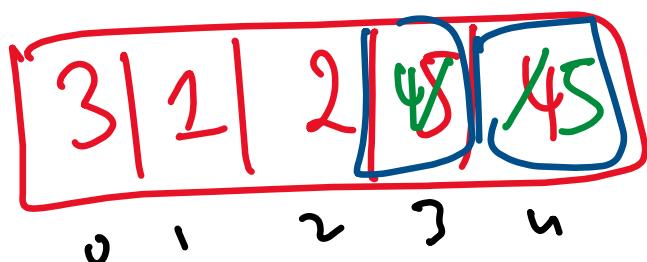


$i = \emptyset 1$

swap  $\rightarrow f$

$$\begin{aligned} j \rightarrow 0 &\rightarrow n-1-i \\ &\rightarrow 5-1-0 \rightarrow 4 \end{aligned}$$

$\rightarrow 8 \neq 3$



$$\begin{aligned} j \rightarrow 5-1-1 \\ \rightarrow 3 \end{aligned}$$

$$\begin{aligned} \rightarrow j(0, 3) \\ \rightarrow (0, 1, 2) \end{aligned}$$

$\rightarrow (0, 1, 2)$

### Applications:

- **Data Validation:** Quickly detect whether data is already sorted
- **Test Case Generation:** Verify correctness of complex sorting implementations





## 8.3 - Selection Sort

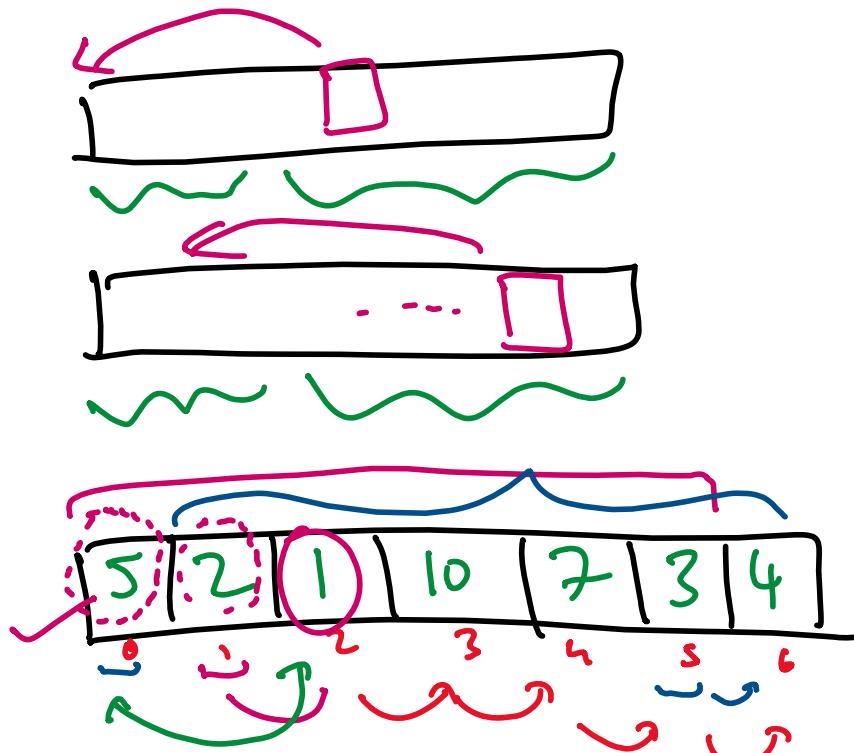
Tuesday, November 11, 2025 11:14 PM

### Main Ideas:

- Find the smallest element in the unsorted part of the list
- Swap it with the first element of the unsorted section
- Move the boundary between sorted and unsorted parts one step forward and repeat until the list is sorted

### Key Points:

- Makes fewer swaps than Bubble Sort
- Inefficient for large lists since it needs  $O(n^2)$  comparisons



$$j=0, j < n-1$$

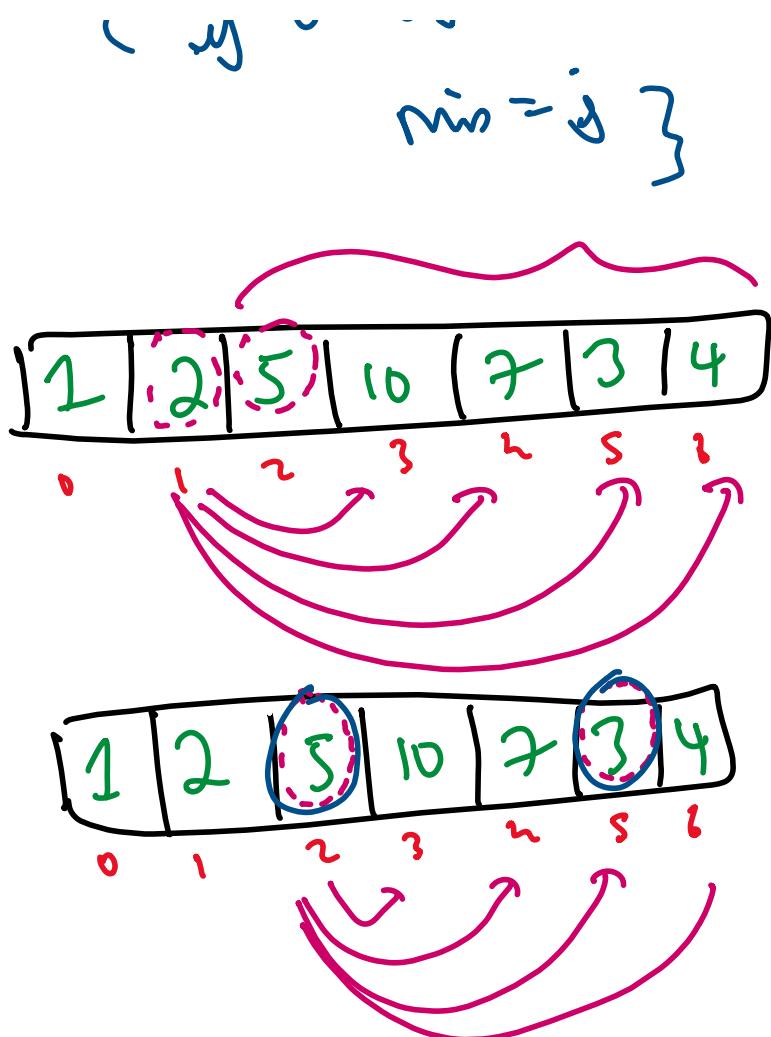
$$j \neq 1$$

$$j = +2$$

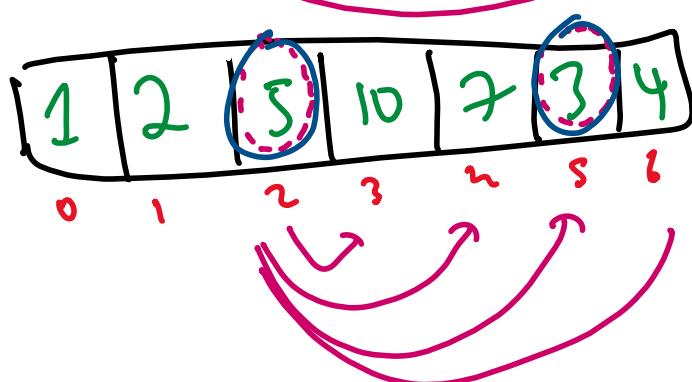
$$\min = i \quad j = i+1, j < n$$

$$\min = 12$$

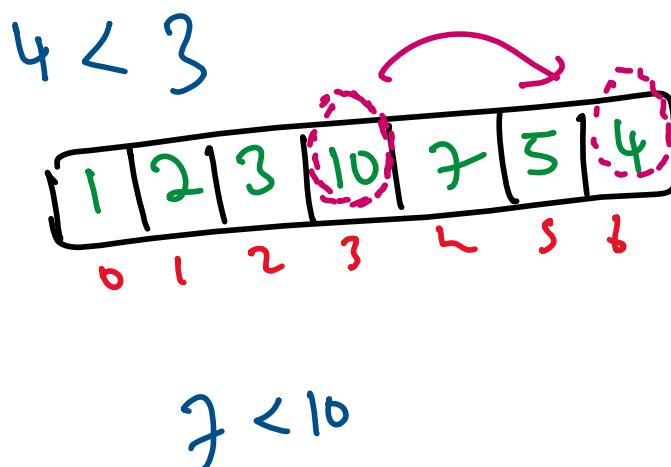
$$\left\{ \begin{array}{l} \text{if } \text{arr}(j) < \text{arr}(\min) \\ \min = j \end{array} \right.$$



$i = 1$   
 $\min = 1$   
 $j = 2$

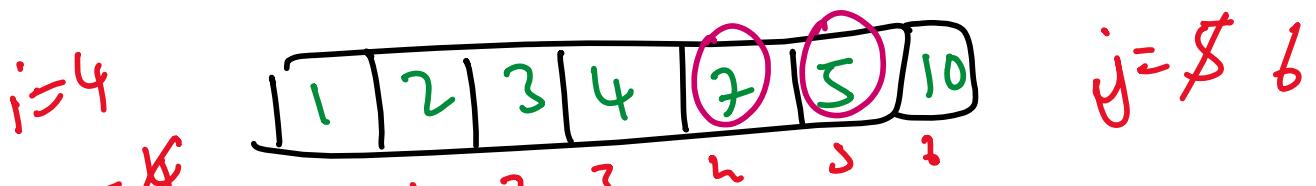


$i = 2$   
 $\min = 2 \neq 5$   
 $j = 3$

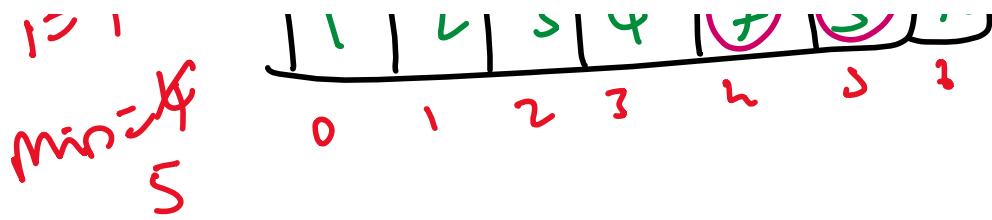


$i = 3$   
 $\min = 3 \neq 8$   
 $j = 486$

$7 < 10$   
 $5 < 7$        $4 < 5$



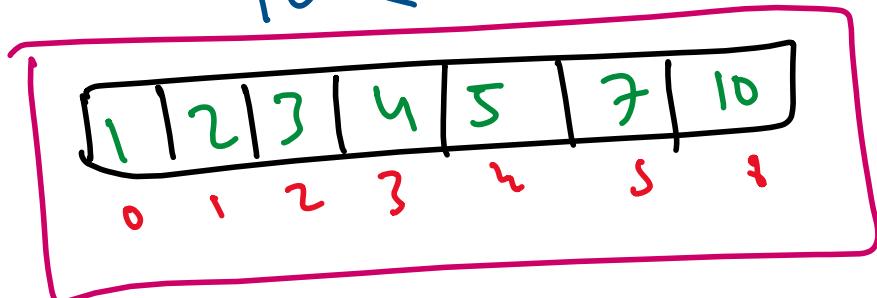
$j = \$ 6$



$$5 < 7$$

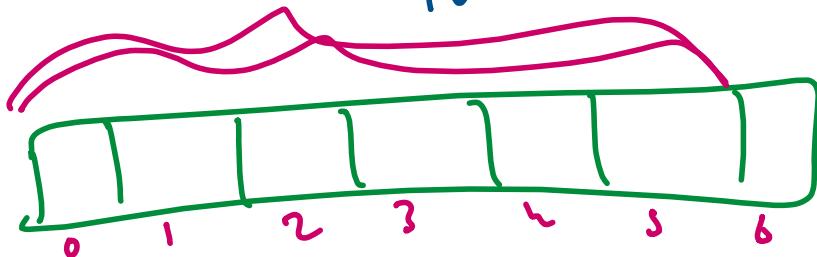
$$10 < 5$$

$i = 5$   
 $\text{min} = 5$



$i = 6$

$$10 < 7$$



$i \rightarrow$  index deret  
 $j \rightarrow$  "min" eleret in unsorted section

min-index

$$\text{arr}[i] \rightarrow \text{arr}[\text{min}]$$

Applications:

- **Flash Memory:** Writes degrade hardware lifespan
- **Firmware & IoT systems:** Sorting configuration tables





## 8.4 - Insertion Sort

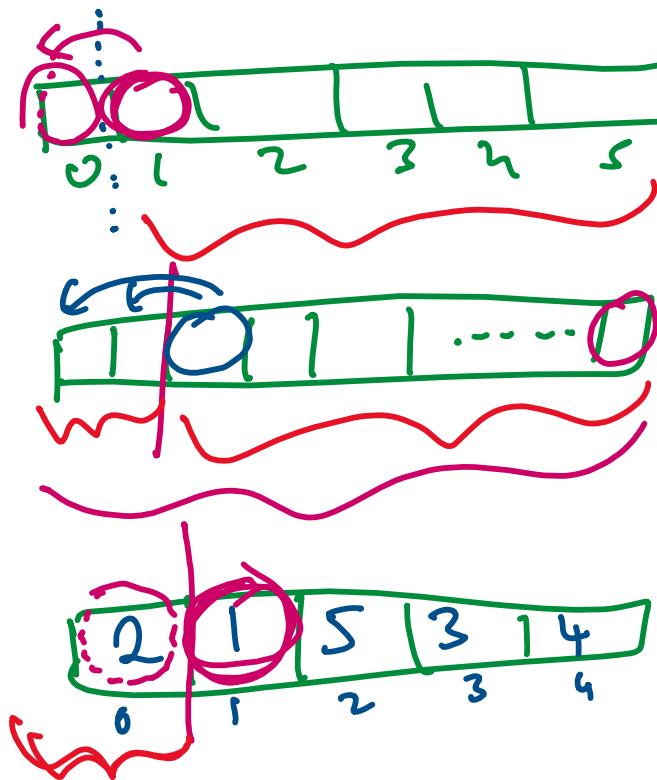
Tuesday, November 11, 2025 11:21 PM

### Main Ideas:

- Start from the second element and compare it with the elements before it
- Insert the current element into its correct position in the already sorted part of the list
- Repeat this for all elements until the entire list is sorted

### Key Points:

- Works well for small lists or lists that are almost sorted
- Sorting happens in-place (no extra memory needed)
- Inefficient for large lists



$i=1, i \neq n, i++$

$i=1 \quad \text{key} \rightarrow arr[i]$

$j=i-1$

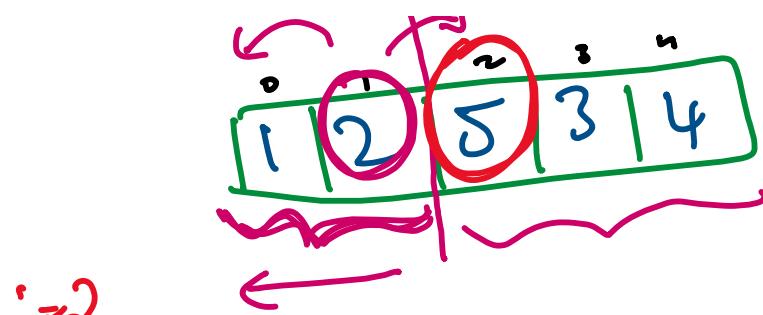
$(j \geq 0 \quad \& \quad arr[j] > \text{key})$

$arr[j+1] = arr[i]$

$$\begin{aligned} j &= i-1 \\ &= 0 \end{aligned}$$

$$\begin{aligned} arr(i) &= 1 \\ arr(j) &= 2 \end{aligned}$$



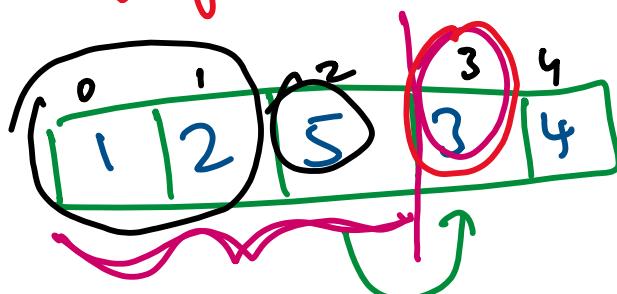


$i = 2$

$key = 5$

$j = 1$

$$\alpha(i) = 2$$



$j = 3$

$key = 3$

$j = 2$

$$\alpha(i) = 5$$

$$\alpha(i) = 2$$

$$\alpha(j+1) = key$$

3  
↓



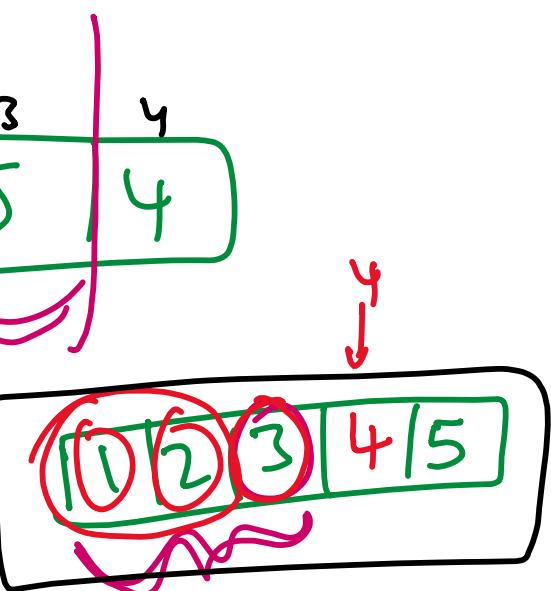
$i = 4$

$key = 4$

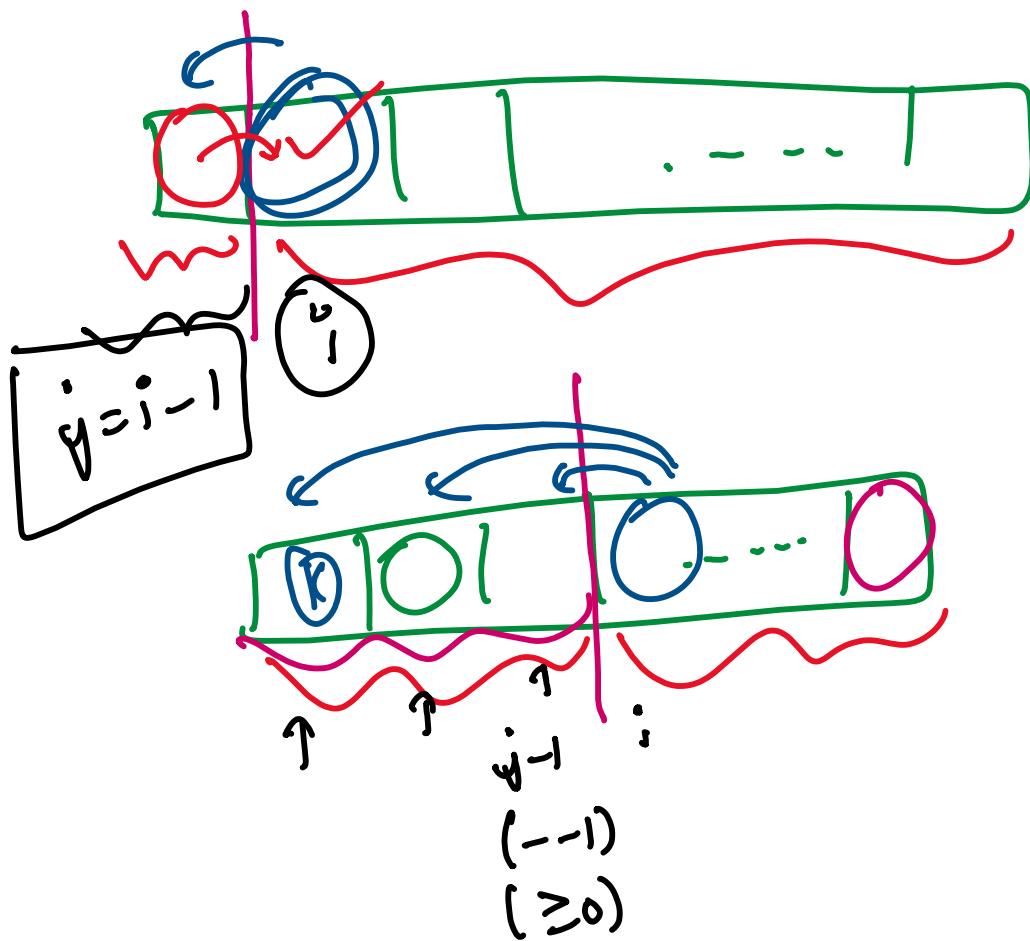
$j = 2$

$$\alpha(i) = 5$$

$$\alpha(i) = 3$$



$$g = \frac{P}{2} \quad \alpha(lj) = j$$



### Applications:

- **Real-time streaming systems:** New data arrives continuously
- **Maintaining sorted order incrementally**









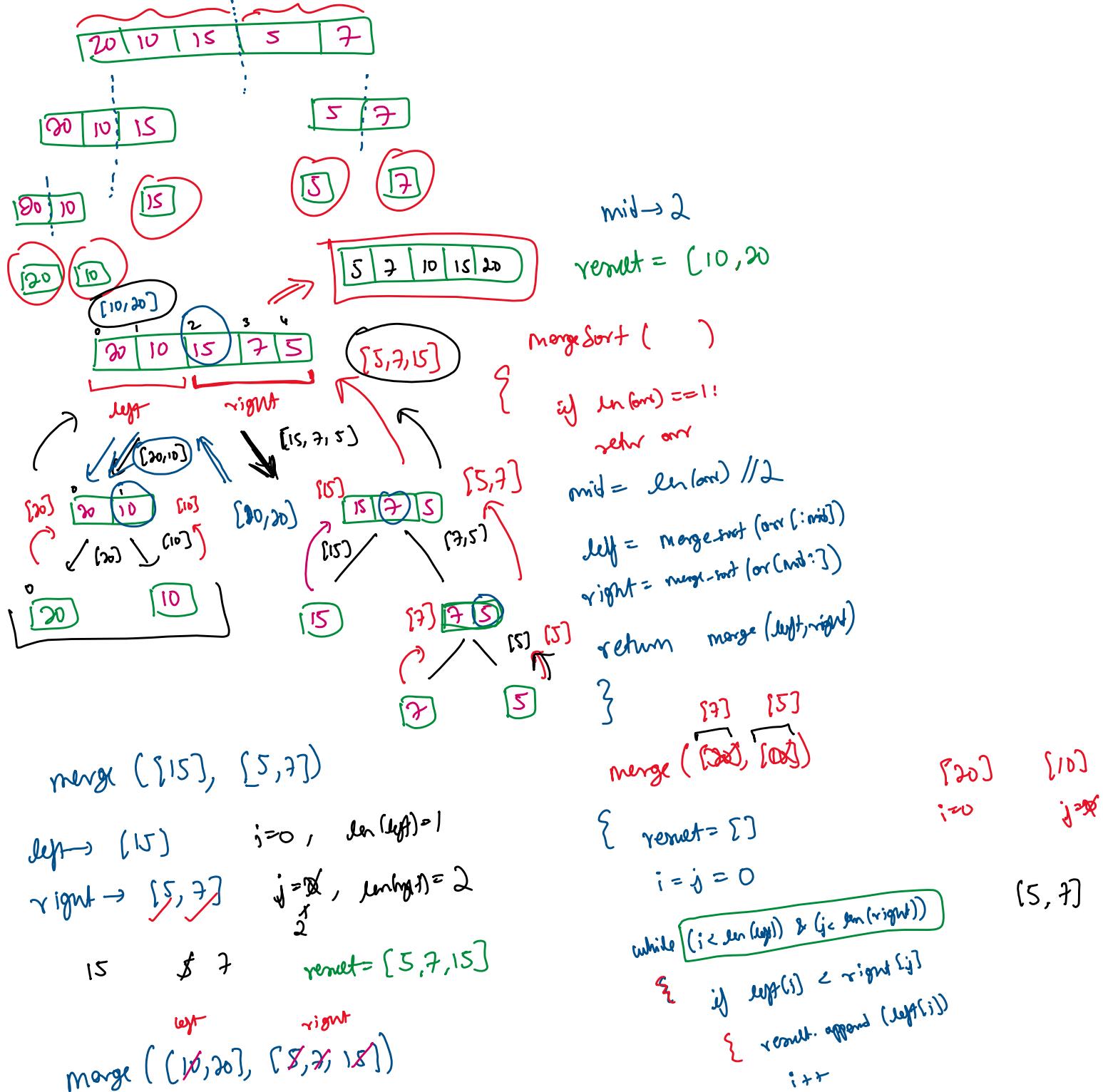


Main Ideas:

- Divide the list into two halves until each sublist has only one element
- Merge the sublists by comparing elements and combining them in sorted order
- Repeat merging until one fully sorted list is formed

Key Points:

- Based on the Divide and Conquer strategy
- Very efficient for large datasets, with time complexity  $O(n \log n)$
- Requires extra space for merging (not an in-place algorithm)



merge ( $\{10, 20\}$ ,  $\{8, 12, 18\}$ )

$i \rightarrow \alpha_1$

20 ~~10~~ ~~8~~ 15

$j \rightarrow \alpha_2 \alpha_3$

result  $\rightarrow [5, 7, 10, 15, 20]$

{ result = []

$i++$

}

else

{

}

$j++$

}

if ( $i < j$ )

result.append (right[i:j])

if ( $i < j$ )

result.append (right[i:j])

if ( $i < j$ )

result.append (right[i:j])

return result

}

#### Applications:

- External sorting (data > RAM)
- Distributed computing frameworks
- Stable sorting requirements





## 8.6 - Quick Sort

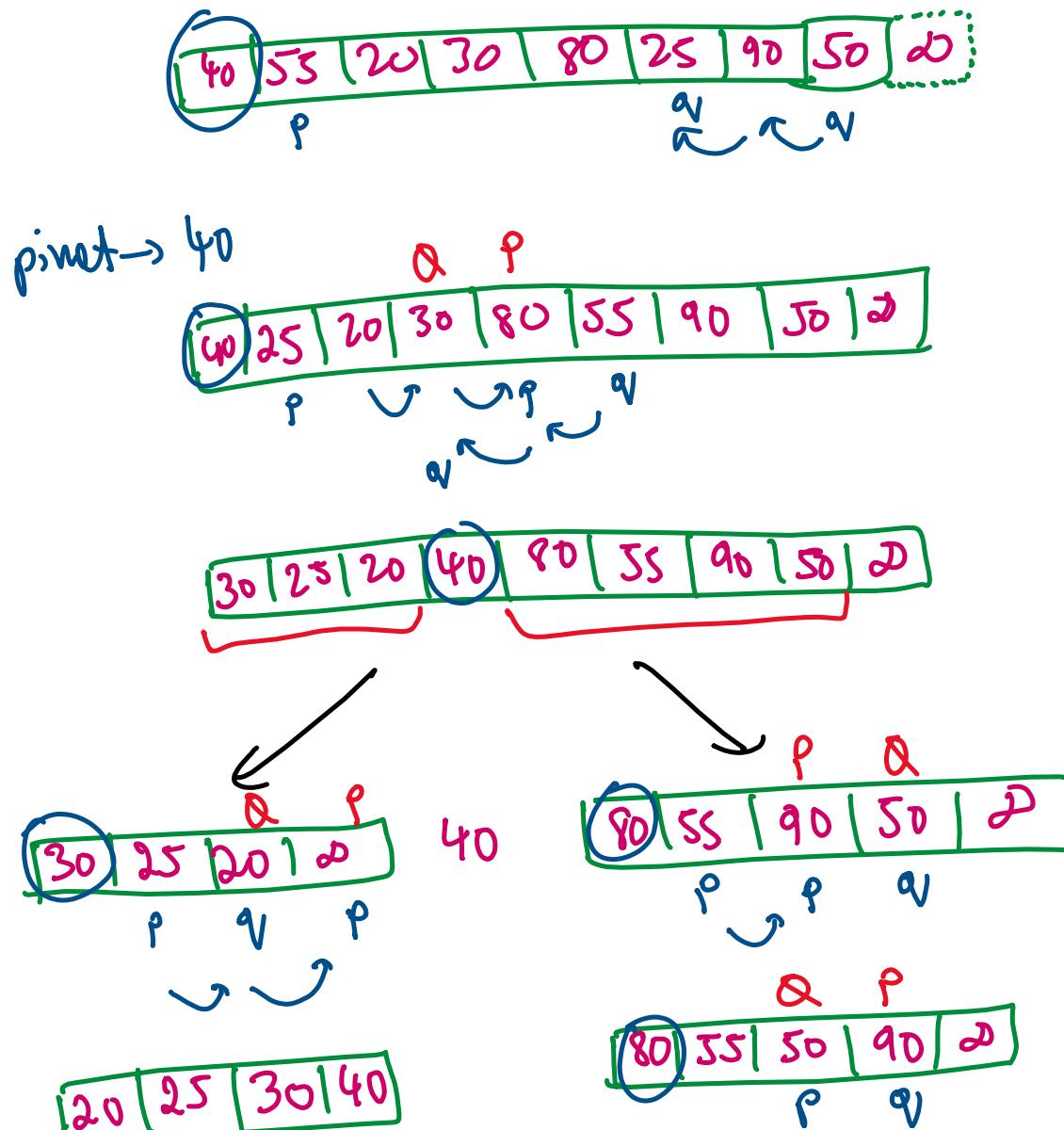
Tuesday, November 11, 2025 11:47 PM

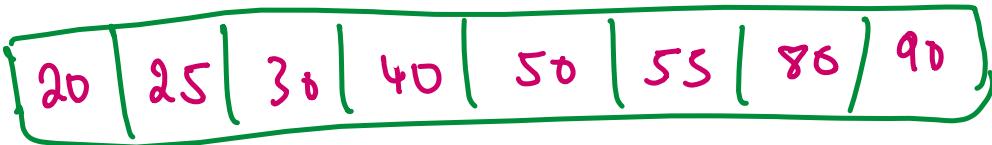
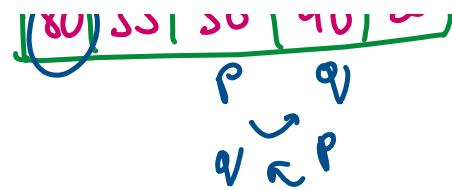
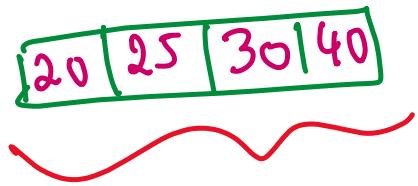
### Main Ideas:

- Choose a pivot element from the list
- Rearrange (partition) the list so that all elements smaller than the pivot are on its left, and all greater elements are on its right
- Recursively apply the same process to the left and right sublists until the list is sorted

### Key Points:

- Based on the Divide and Conquer strategy
- Faster than Merge Sort in practice for many datasets (average case:  $O(n \log n)$ )
- In-place sorting algorithm (needs little extra space)





### Applications:

- Better Applications
- System libraries
- Low-latency applications
- Real-time analytics















## 8.7 - Counting Sort

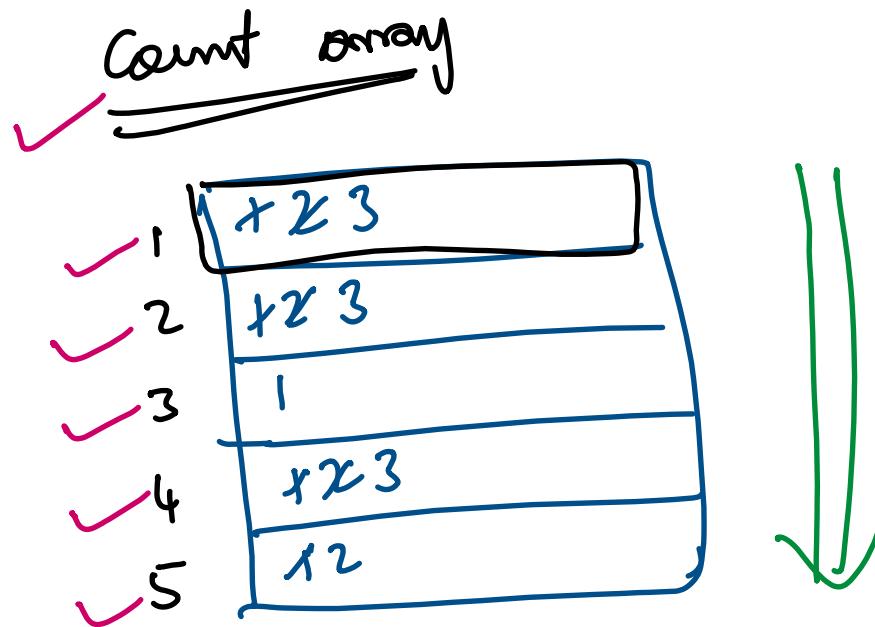
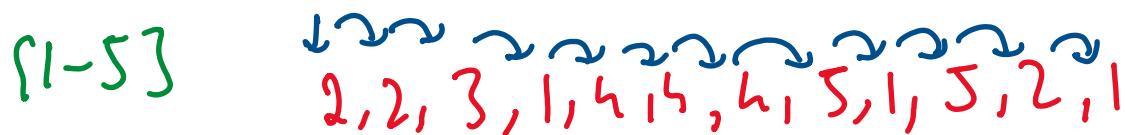
Tuesday, November 11, 2025 11:23 PM

## Main Ideas:

- Count the frequency of each unique element in the input list
  - Store these counts in an auxiliary array (called the count array)
  - Use the count array to place each element directly in its correct sorted position in the output array

## **Key Points:**

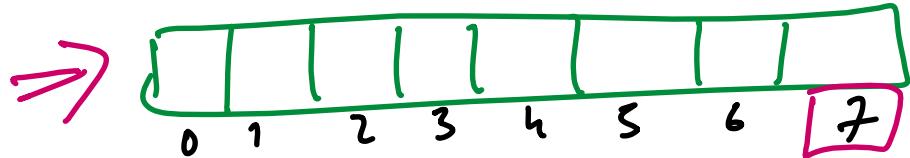
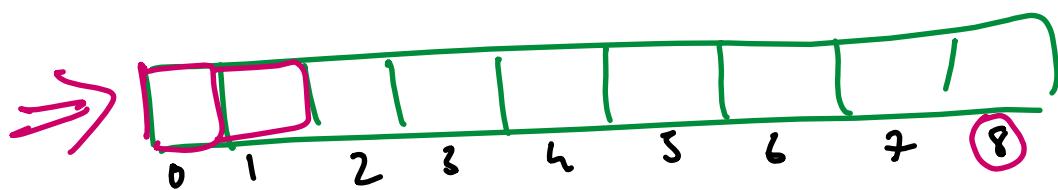
- Works best for integers or discrete values within a small range
  - Very fast when the range of input values is not much larger than the number of elements (time complexity  $O(n + k)$ )
  - Not comparison-based and stable (if implemented carefully)
  - Requires extra space for the count array, so not ideal for large ranges



1, 1, 1, 2, 2, 2, 3, 4, 4, 4, 5, 5

min  $\rightarrow$  2

max  $\rightarrow$  8



### Applications:

- Better Applications
- Voting systems
- Telemetry & sensor data
- Categorical data processing
- Image processing





## 8.8 - Complexity Analysis

Tuesday, November 11, 2025 11:49 PM

ALGORITHM	TIME	SPACE	STABLE	TYPE
Bubble Sort	$O(n^2)$ ✓	$O(1)$ ✓	✓	Comparison
Selection Sort	$O(n^2)$ ✓	$O(1)$ ✓	✗	Comparison
Insertion Sort	$O(n^2)$ ✓	$O(1)$ ✓	✓	Comparison
Merge Sort	$O(n \log n)$ ✓	$O(n)$ ✓	✓	Comparison
Quick Sort	$O(n \log n)$ ✓	$O(\log n)$ ✓	✗	Comparison
Counting Sort	$O(n + k)$ ✓	$O(k)$ ✓	✓	Non-Comparison

$$K \rightarrow [n + K]$$



## 8.9 - DS Context

Tuesday, December 16, 2025 4:21 AM

### **Data Science / ML Correlation:**

- Ranking predictions
- Feature importance ordering
- Evaluation metrics (Top-K)

### **Libraries & Usage:**

- `numpy.argsort()`
- `pandas.sort_values()`
- Recommendation systems

# 9. Recursion

Sunday, November 23, 2025 5:51 PM

- **Introduction to Recursion**
- **Internal Working of Recursion**
- **Common Mistakes**
- **Recursion vs Iteration**
- **Coding Problems**

What is Recursion?

- Recursion is a programming technique where a function calls itself to solve smaller subproblems of the original problem
- This process continues until a specific base case (or stopping condition) is met
- At this point the recursion terminates and the results are combined back up the chain of calls to solve the original problem

Structure of Recursive Functions:

A recursive solution typically has two components -

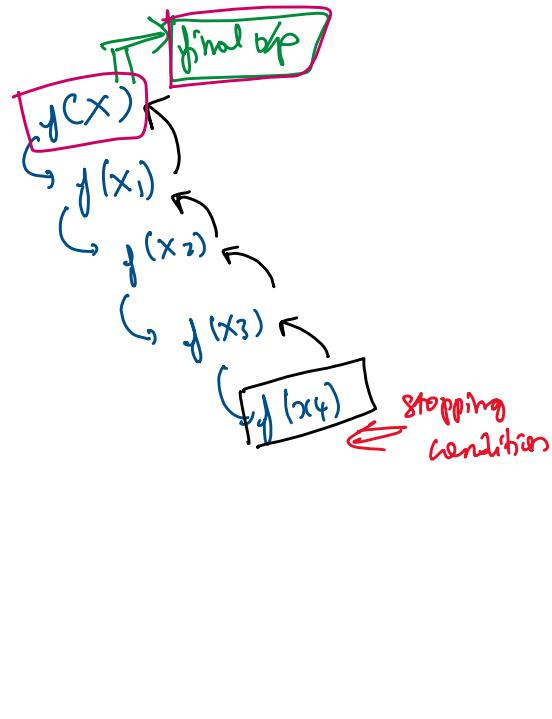
1. Base Case:

- Stops recursion from continuing infinitely
- Should always be reachable
- Returns the answer for the smallest version of the problem

2. Recursive Case:

- Divides the problem into smaller versions
- Should always progress toward base case

```
def recursive_function(parameters):
    if base_condition:
        return base_value
    else:
        return recursive_function(smaller_parameters)
```

Applications of Recursion:

- **Tree Traversals:** Algorithms like in-order, pre-order, and post-order traversal are inherently recursive
- **Graph Algorithms:** Depth-First Search (DFS) uses recursion to explore all possible paths in a graph
- **Sorting Algorithms:** Quick Sort and Merge Sort are divide-and-conquer algorithms that use recursion to sort subarrays
- **Mathematical Problems:** Calculating factorials, Fibonacci sequences, and the Tower of Hanoi puzzle are classic examples solved with recursion
- **Backtracking:** Problems like solving a Sudoku puzzle or the N-Queens problem often rely heavily on recursion to explore different possibilities

Advantages:

- ✓ Leads to shorter, cleaner, and more intuitive code
- ✓ Ideal for Divide and Conquer algorithms
- ✓ Good for Backtracking
- ✓ Helps model Hierarchical Structures (file system, XML, JSON)

Disadvantages:

- ✓ High memory usage
- ✓ Risk of **Infinite Recursion**
- ✓ Harder to debug
- ✓ Not always efficient
- ✓ Limited by **Maximum Recursion Depth** (Python ~ 1,000 calls)

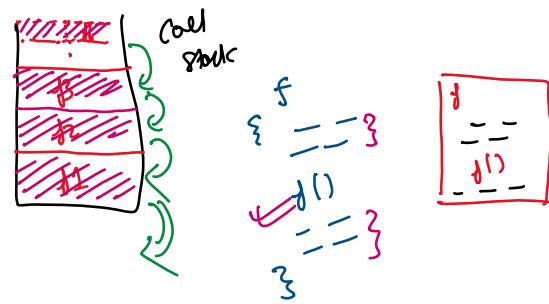
When to Use Recursion?

- Problem is naturally repetitive
- Tree/Graph traversal is needed
- Need for Divide and Conquer algorithm
- Backtracking/Search problems
- Code clarity is more important than raw speed



**Recursion is managed using a Call Stack, which is a Last-In, First-Out (LIFO) data structure:**

1. When a function is called, a new stack frame (also known as an activation record) is created and pushed onto the call stack
  - o This frame stores the function's local variables, parameters, and the return address
2. As the function recursively calls itself, new stack frames accumulate on top of the previous ones
3. Once the base case is reached, the functions start returning their results
  - o The top stack frame is popped off, and control returns to the function below it in the stack
  - o The function uses the returned result to perform its remaining calculations and return its own result
  - o This process is called **stack unwinding** or the **ascending phase**



**Examples:**

- Factorial
- Fibonacci Numbers

$$\begin{aligned} 5! &= 5 \times 4 \times 3 \times 2 \times 1 \\ 3! &= 3 \times 2 \times 1 \\ 2! &= 1 \\ 0! &= 1 \end{aligned}$$

$$\begin{aligned} 5! &= 5 \times 4 \times 3 \times 2 \times 1 \\ &= 5 \times [4 \times 3 \times 2 \times 1] \\ &= 5 \times 4! \\ 3! &= 3 \times 2 \times 1 \\ &= 3 \times 2! \\ 1! &= 1 \times 0! \end{aligned}$$

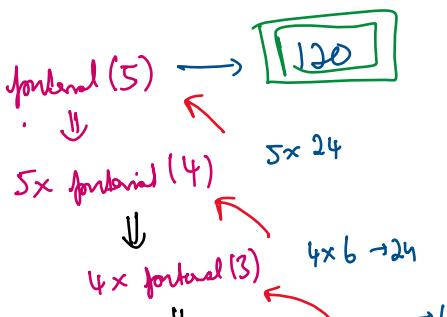
$\downarrow$        $\downarrow$        $\downarrow$        $\downarrow$

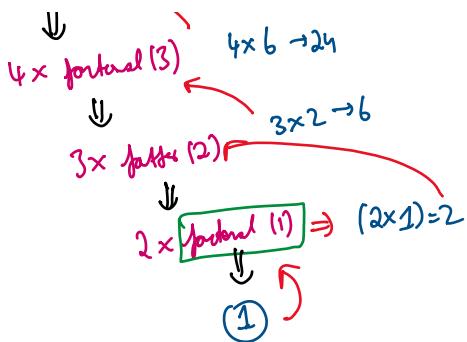
$$fact(n) = n \times fact(n-1)$$

$n > n-1 \Rightarrow$  repetitive

$$fact(n-1) = (n-1) \times fact(n-2)$$

```
def factorial(n):
    # base condition
    if (n == 0) or (n == 1):
        return 1
    # recursive call
    return n * factorial(n-1)
```





0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Let  $F \rightarrow \text{fibonacci}$

$$\begin{cases} f(0) = 0 \\ f(1) = 1 \end{cases} \Rightarrow \text{Base condition}$$

$$f(2) = f(1) + f(0) = 0 + 1 = 1$$

$$f(3) = f(2) + f(1) = 1 + 1 = 2$$

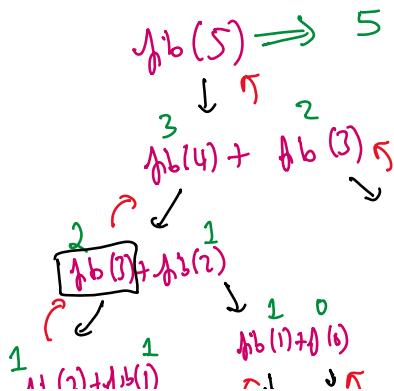
$$f(4) = f(3) + f(2) = 2 + 1 = 3$$

$$f(5) = f(4) + f(3) = 3 + 2 = 5$$

$$f(n) = f(n-1) + f(n-2)$$

$n \geq 2$

```
def fib(n):
    # base cond.
    if (n == 0) or (n == 1):
        return n
    # recursive call
    return fib(n-1) + fib(n-2)
```



$$\begin{array}{r} \overline{P} \\ \times \quad \overline{Q} \\ \hline \end{array}$$

Handwritten annotations:

- Top row:  $1 \downarrow$  (under  $\overline{P}$ ) and  $1 \downarrow$  (under  $\overline{Q}$ )
- Middle row:  $1 \downarrow$  (under  $\overline{P}$ ) and  $1 \downarrow$  (under  $\overline{Q}$ )
- Bottom row:  $1 \downarrow$  (under  $\overline{P}$ ) and  $0 \downarrow$  (under  $\overline{Q}$ )

Red arrows point from the bottom row annotations to the middle row annotations.





## 9.3 - Common Mistakes

Monday, November 24, 2025 8:53 PM

- **Forgetting the Base Case:**

- Leads to infinite recursion
- Overflow of the memory stack



Call Stack

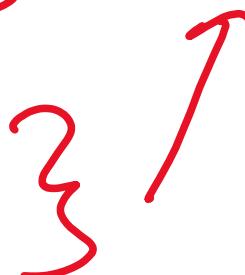
- **Reducing the problem incorrectly:**

- May skip base case
- Could lead to infinite loop



- **Doing extra work in recursive return:**

- Inefficient code



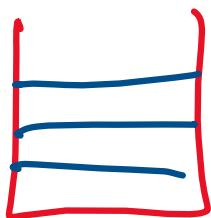




## 9.4 - Recursion vs Iteration

Monday, November 24, 2025 9:15 PM

RECURSION	ITERATION
Uses call stack ✓	Uses loops ✓
Elegant and mathematical ✓	Faster & memory efficient ✓
May be slower due to repeated calls ✓	No overhead of stack frames ✓
Easy for tree/graph and backtracking ✓	Hard for problems like DFS ✓



for, while

$$f(n) = n \times f(n-1)$$

↓

Remove





















## 9.5 - Coding Problems

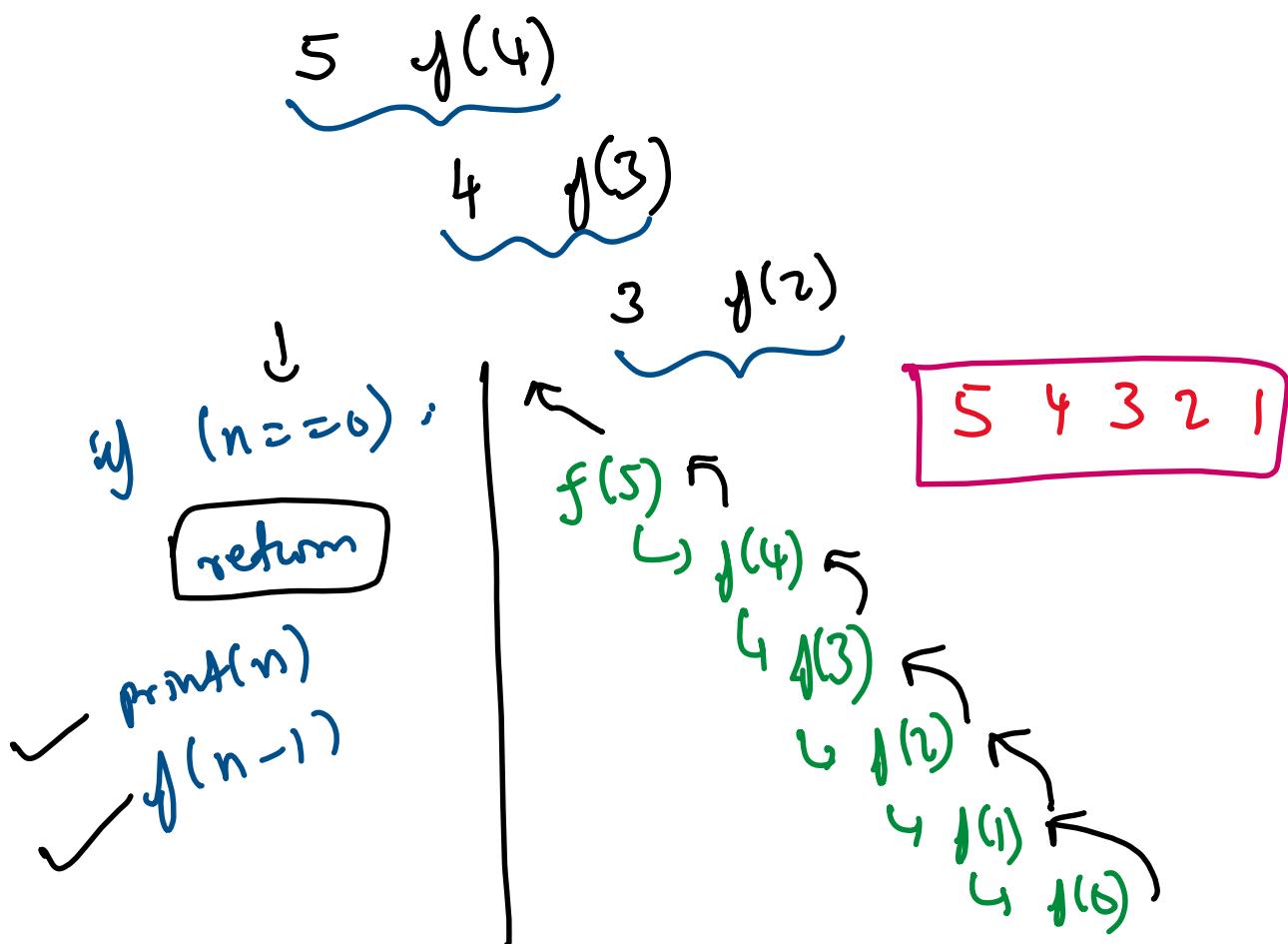
Monday, November 24, 2025 9:31 PM

$n \rightarrow 1$

$n=5$   
 $O/P \Rightarrow 5 [4 [3 [2 [1]]]]$

$f(5) \rightarrow ?$

$f(4) \Rightarrow$



$\overbrace{1 \dots 7} n=10, \overbrace{5 \dots 11} \rightarrow 55$

$$f \Rightarrow \boxed{\frac{n(n+1)}{2}}$$

$n=10,$

$$\frac{10(11)}{2} \rightarrow \underline{\underline{55}}$$

$$n=5, \text{ op: } 5+4+3+2+1 \\ = 15$$

$$n=7, \text{ op: } 2+6+\boxed{5+4+3+2+1} \\ \approx 28$$

$$n=6, \text{ op: } 6+\boxed{5+4+3+2+1}$$

$$\underline{\underline{f(6)}} = \underline{\underline{6}} + \underline{\underline{f(5)}}$$

$$\underline{\underline{f(7)}} = \underline{\underline{7}} + \underline{\underline{f(6)}}$$

$$\underline{\underline{f(3)}} = \underline{\underline{3}} + \underline{\underline{f(2)}}$$

$$\underline{\underline{f(2)}} = \underline{\underline{2}} + \underline{\underline{f(1)}}$$

$$\underline{\underline{f(1)}} = \underline{\underline{1}} + \underline{\underline{f(0)}}$$

$$\boxed{f(0) = 0}$$

def func(n):

if  $n==0:$   
return 0

return  $n + f(n-1)$

$$f(4) \xrightarrow{10} \boxed{10}$$

$$\underline{4} + \underline{\underline{f(3)}} \xrightarrow{6}$$

$$\underline{3} + \underline{\underline{f(2)}} \xleftarrow{3}$$

$$\begin{array}{r}
 3 + f(2) \\
 \downarrow \\
 2 + f(1) \\
 \downarrow \\
 1 + f(0)
 \end{array}$$

$n = 4326$

$0_{10} \rightarrow 4$

$n // 10$

$\text{count} = 0 + 2 = 34$

$\boxed{n = 4326}$

$\uparrow$

$n \rightarrow 4326 // 10 \rightarrow 432$

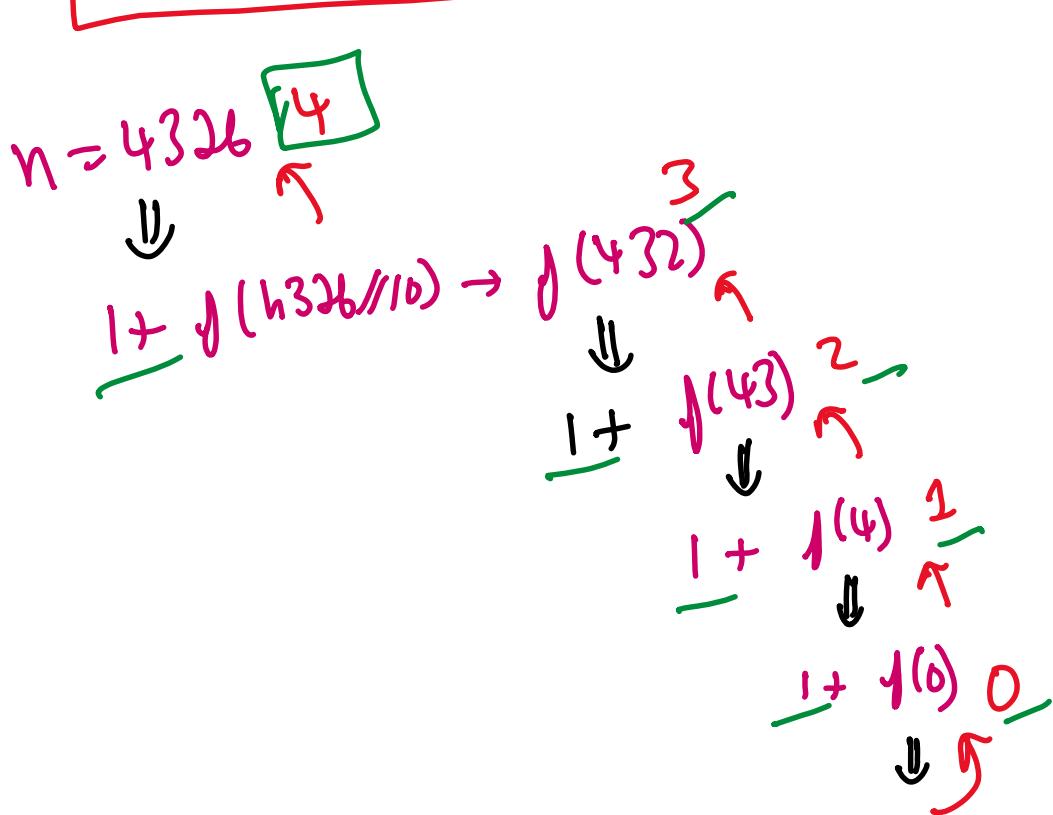
$n \rightarrow 432 // 10 \rightarrow 43$

$n \rightarrow 43 // 10 \rightarrow 4$

$n \rightarrow 4 // 10 \rightarrow 0$

$\boxed{1 + f(n // 10)}$

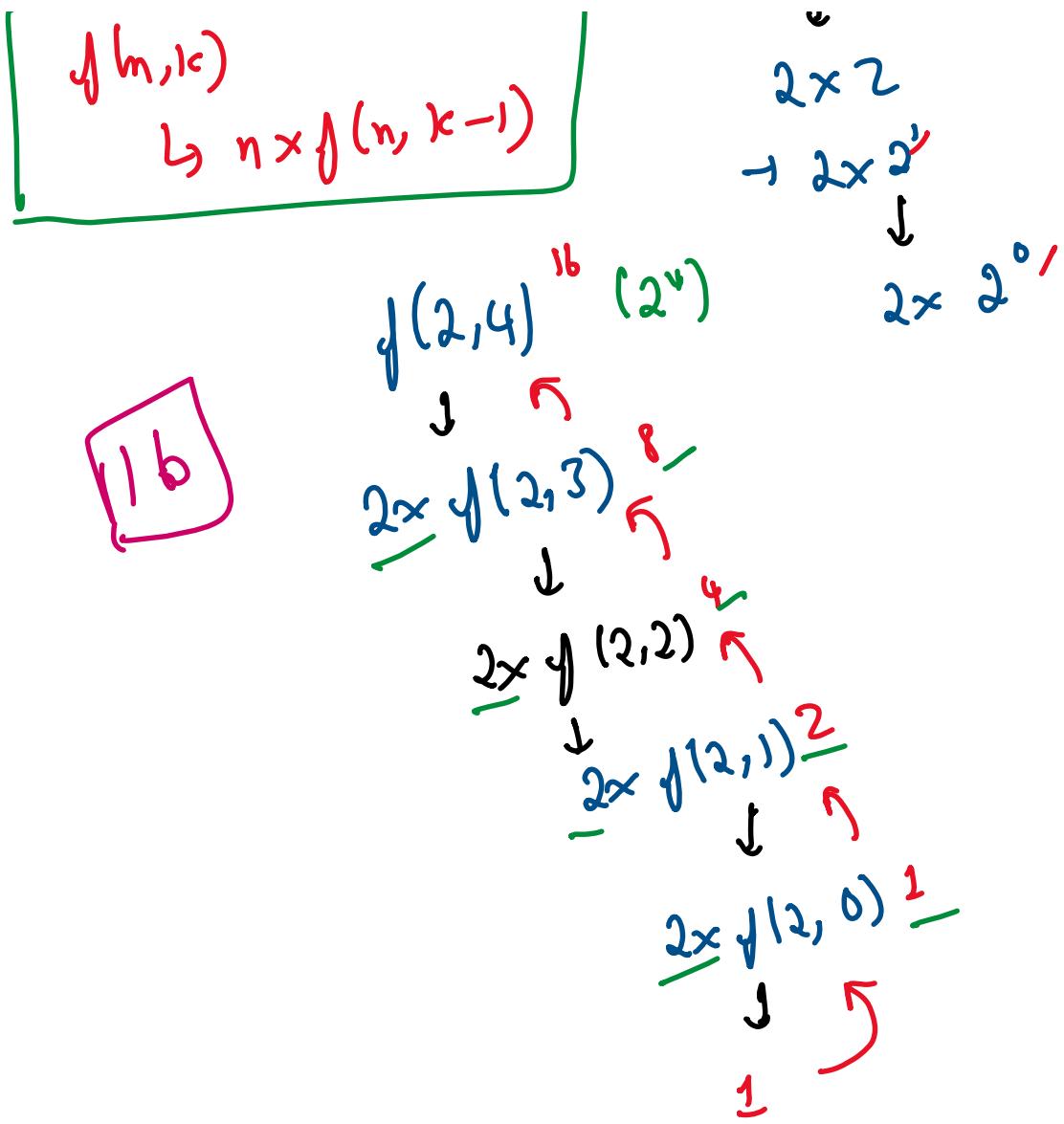
$\swarrow$



$$\begin{aligned}
 \mathfrak{f}(5,3) &\rightarrow 5^3 \\
 &\rightarrow 5 \times 5 \times 5 \\
 &\rightarrow 5 \times 5^2
 \end{aligned}$$

$$\begin{aligned}
 \mathfrak{f}(2,4) &\rightarrow 2 \times 2 \times 2 \times 2 \\
 &\rightarrow 2 \times 2^3 // \\
 &\quad \downarrow \\
 &\quad 2 \times 2 \times 2 \\
 &\rightarrow 2 \times 2^2 // \\
 &\quad \downarrow \\
 &\quad 2 \times 2
 \end{aligned}$$

$\mathfrak{f}(n,1c)$

















## 9.6 - DS Context

Tuesday, December 16, 2025 4:27 AM

### **Data Science / ML Correlation:**

- Tree-based models
- Hierarchical modeling

### **Libraries & Usage:**

- **scikit-learn:** *DecisionTree, RandomForest*
- Recursive splitting in trees

# 10 - Hashing

Wednesday, November 26, 2025 11:51 PM

- **Introduction**
- **Hash Functions**
- **Hash Tables**
- **Hash Collisions & Collision Resolution**
- **Load Factor**
- **Common Operations**
- **Complexity Analysis**
- **Coding Problems**

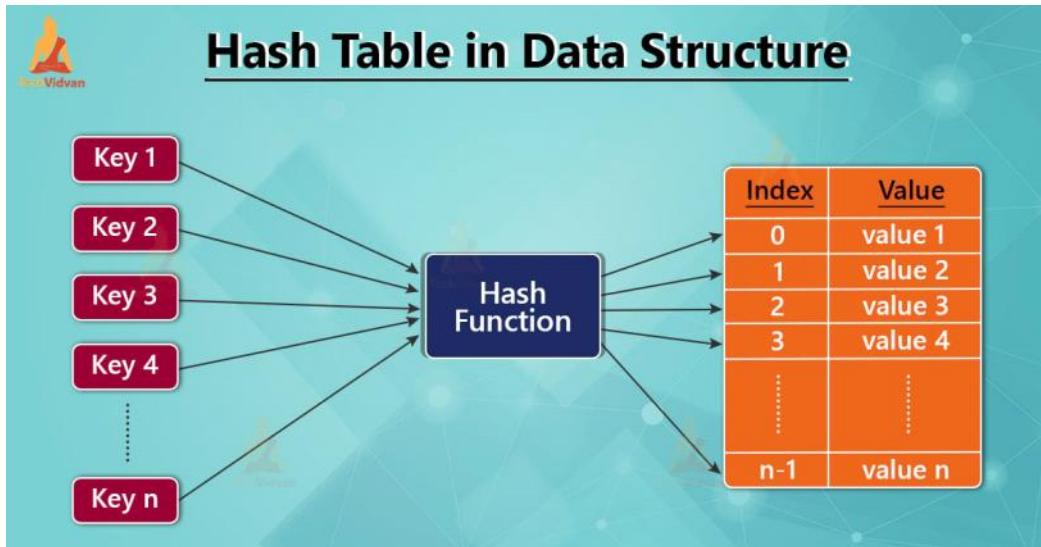


## 10.1 - Introduction

Wednesday, November 26, 2025 11:52 PM

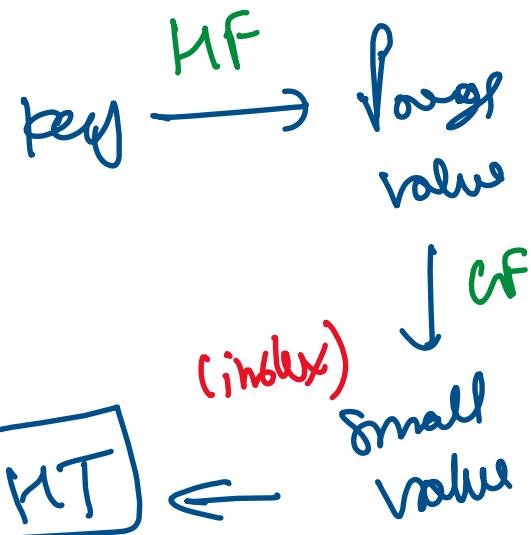
### What is Hashing?

- Hashing is a technique that converts a key (string, number, object) into a fixed-size number called a **hash value**
- Makes use of a **hash function** to map data keys to a specific index in an array, known as a **hash table**



### Major Components in Hashing:

- Key
- Hash Function
- Hash Table



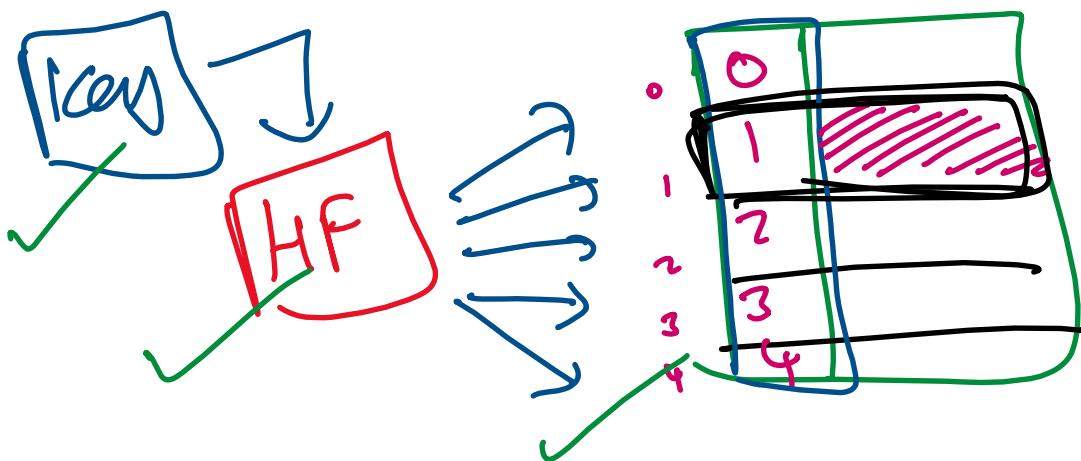
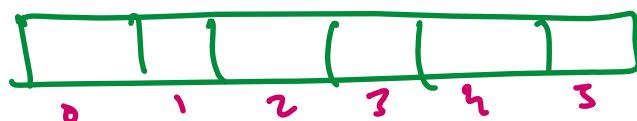
### Common Workflow:

- Convert key into hash value (hash function)
- Map the hash value to a smaller integer (compression function)
- Insert the key into the index position in array (hash table)

### Applications:

APPLICATION	PURPOSE	EXAMPLE
Database Indexing	Fast lookup	MySQL hash index

APPLICATION	PURPOSE	EXAMPLE
Database Indexing	Fast lookup	MySQL hash index
Password Storage	Security	SHA-256, bcrypt
Data Deduplication	Avoid duplicates	Dropbox, cloud storage
Data Structures	Key-value storage	Python dict, Java HashMap
Data Integrity	Detect tampering	MD5/SHA checksums
Load Balancing	Distribute requests	Consistent hashing
Caching	Quick access	Web browsers
Plagiarism Detection	Detect duplicates	Document fingerprinting
Networking	P2P storage	BitTorrent DHT
Blockchain	Secure transactions	Bitcoin, Ethereum





## 10.2 - Hashing Function

Friday, November 28, 2025 7:44 AM

### What is a Hashing Function?

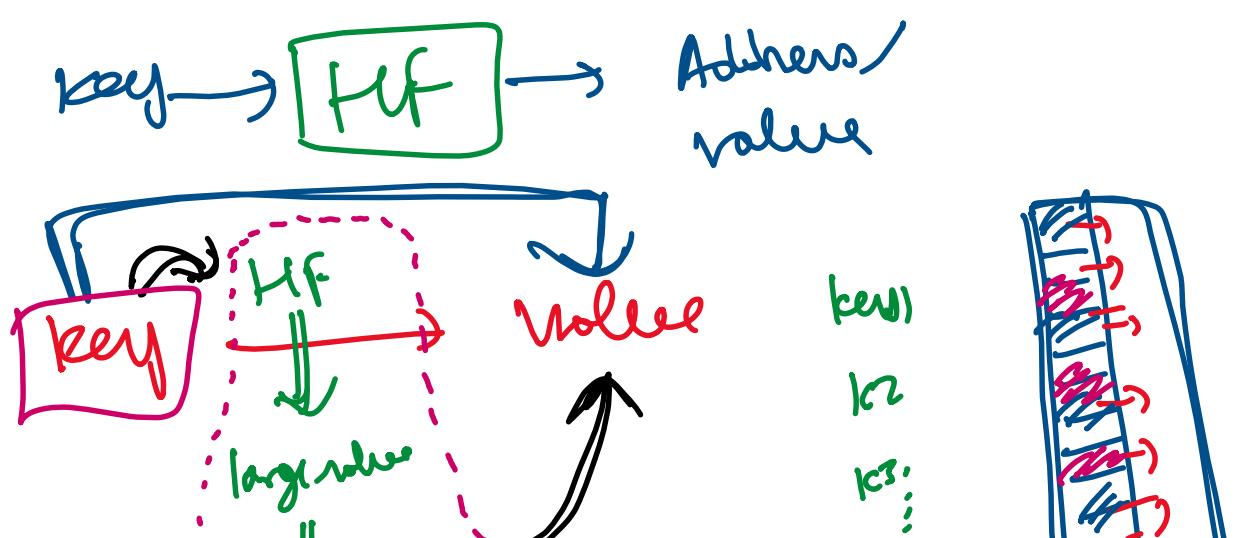
- A hash function is a function that maps keys to one of the values in the hash table
- Hash functions return the location/address where we can store the particular key
- We input the key to the hash function and the output is an address
- Internally, complex mathematical operations are implemented to generate the hash value

### Properties of Hashing Function:

- **Deterministic:** For a particular input, the output is the same
- **Uniform Distribution:** Keys should spread across the table and not cluster
- **Fast Computation:** A hash function must be extremely fast since it is used for every insertion, deletion, and search
- **Low Collision:** Two different keys shouldn't/rarely end up with the same hash value

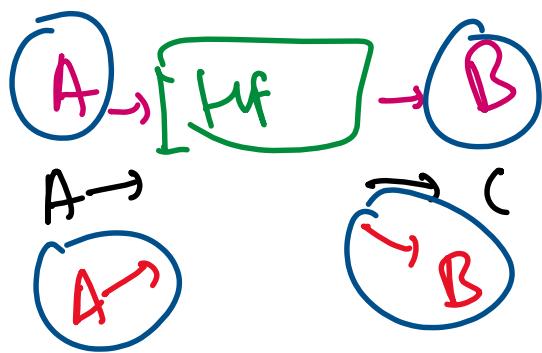
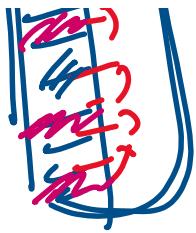
### Common Hashing Functions:

1. Simple Hash ✓
2. Multiplicative Hash ✓
3. DJB2 (very popular) ✓
4. Built-in **hash** function ✓



large value  
↓  
small value (index)

IC3:  
Kn



## 10.3 - Hash Table

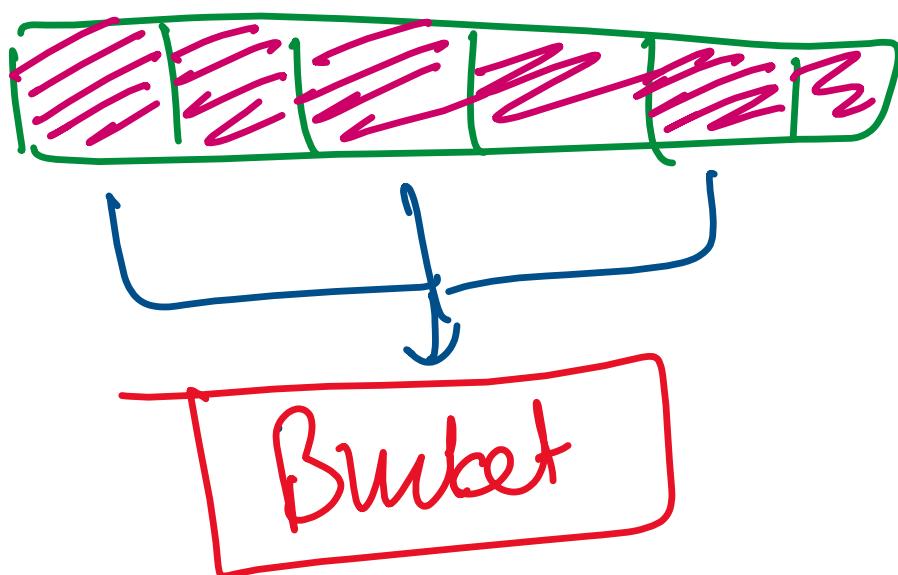
Friday, November 28, 2025 7:55 AM

### What is a Hash Table?

- Type of data structure that makes use of the **hash function** to map values to the key
- Stores values in an associative manner i.e. each key is mapped to a specific value
- **Hash tables** make use of array-like data structures for storing values
- Each slot/cell in the table is also called as **bucket**
- The process of mapping keys to the values is known as **Hashing**
- The purpose of using hash tables as a data structure is to enable faster access

### Python Implementation:

- **Sets (set)**
- **Dictionaries (dict)**



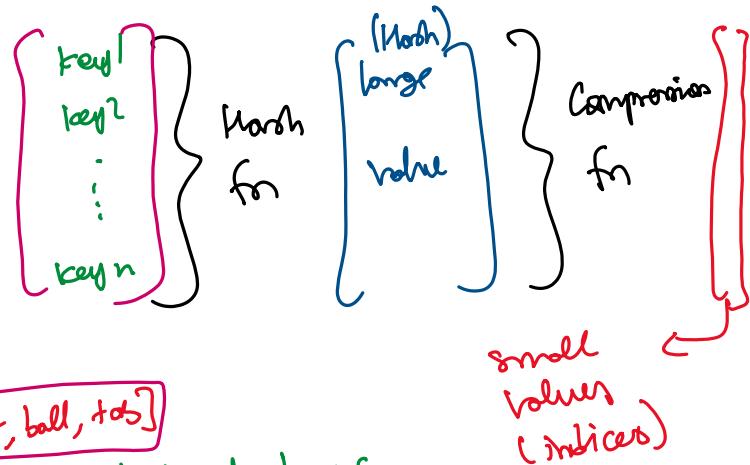
key → value index → value

## 10.4 - Hash Collisions

Friday, November 28, 2025 8:42 AM

### What is a Collision?

- Happens when two different keys produce the same table index
- This is the situation where multiple keys want the same index in the hash table
- Since, each index can typically hold only one key, this is called a **Collision**

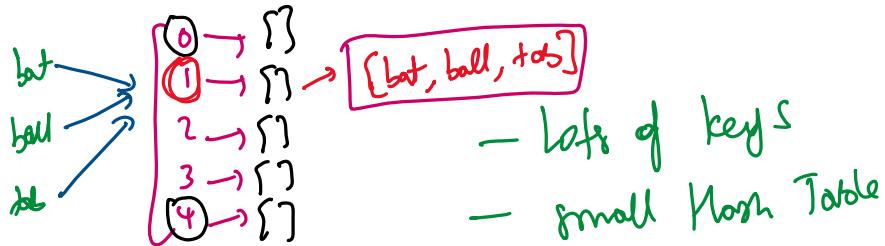


### Why do Collisions occur?

- Input space is infinite (all possible keys)
- Table size is finite (hash table)

### Example:

- Keys: ['bat', 'ball', 'tab']
- Hash Table size: 5
- Hashing Function: Simple ASCII
- Compression Function: modulus (%)



### Collision Resolution:

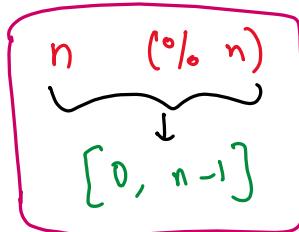
Two main types of approaches to handle collisions:-

#### 1. Chaining:

- Each bucket is a list / linked list / dynamic array
- On Insert, append them key into the bucket
- On Search, scan them keys in the bucket

#### 2. Open Addressing: If a bucket is occupied, we probe to find another slot; sequence must eventually visit empty buckets

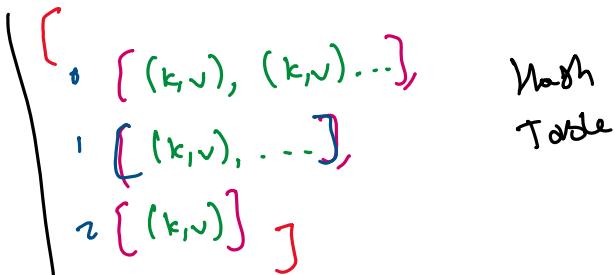
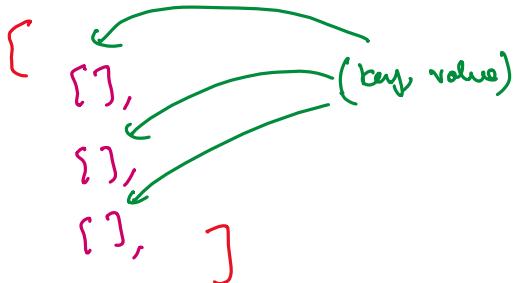
- Linear Probing: If index  $h$  is occupied, keep visiting  $(h + 1) \% \text{size}$  until an empty bucket is found
- Quadratic Probing:  $h, (h + 1^2) \% \text{size}, (h + 2^2) \% \text{size}, (h + 3^2) \% \text{size}$ , etc.
- Double Hashing: Uses 2 hashing functions



0	→	0
1	→	1
2	→	2
3	→	3
10	→	0
20	→	0
30	→	0
33	→	3
77	→	2
94	→	4

$$h \rightarrow (h+1) \% n, (h+2) \% n, (h+3) \% n, \dots$$

$$h \rightarrow (h+1^2) \% n, (h+2^2) \% n, (h+3^2) \% n, \dots$$



1 2 3 4

key → hash-fn → compare-fn → index  
(h)

$$\left(\begin{array}{c} h \\ \downarrow \\ h+0 \end{array}\right)^{\circ}/n, \left(\begin{array}{c} h+1 \\ \downarrow \\ h+1 \end{array}\right)^{\circ}/n, \left(\begin{array}{c} h+2 \\ \downarrow \\ h+2 \end{array}\right)^{\circ}/n, \dots$$

$n=5$

$\text{h} = 2,$

$(2+1) \cdot \%S \rightarrow 3 \cdot \%S \rightarrow 3$

$(2+2) \cdot \%S \rightarrow 4 \cdot \%S \rightarrow 4$

$(2+3) \cdot \%S \rightarrow 5 \cdot \%S \rightarrow 0$

$(2+4) \cdot \%S \rightarrow 6 \cdot \%S \rightarrow 1$

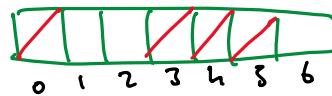
$\cancel{\text{X}} \quad (2+5) \cdot \%S \rightarrow 7 \cdot \%S \rightarrow 2$

[None, None, None, ..., None]  
0 1 2 ... n]

- i) index is empty  $\rightarrow$  insert value
  - ii) index is full, same key  $\rightarrow$  nothing
  - iii) index is full, different key  $\rightarrow$  next slot

[None, None, ----, None]

$$(h+0^2) \text{ on}, (h+1^2) \text{ on}, (h+2^2) \text{ on}, \dots$$



$$\begin{aligned}(3+4) \cdot 17 &\rightarrow 19 \cdot 17 \rightarrow 5 \\(3+5) \cdot 17 &\rightarrow 29 \cdot 17 \rightarrow 0 \\(3+1) \cdot 17 &\rightarrow 39 \cdot 17 \rightarrow 4\end{aligned}$$



## 10.5 - Load Factor

Friday, November 28, 2025 9:46 AM

### What is Load Factor?

- The **Load Factor** is one of the most important concepts in hashing
- Used to indicate how full the hash table is
- Mathematically, it is expressed as:

$$\text{Load Factor} = \frac{\text{Number of elements inserted}}{\text{Total number of buckets (table size)}}$$



### Interpretation of Load factor:

- 0.0** → completely empty
- 0.5** → half full
- 1.0** → completely full
- >1.0** → impossible in open addressing tables, possible in chaining tables

$$n=5 \quad a=4 \quad LF = \frac{4}{5} = 0.8$$

$$n=5 \quad a=5 \quad LF = \frac{5}{5} = 1$$

$$a=0, LF = \frac{0}{5} = 0$$

### Why do we need Load Factor?

- How crowded the table is:** A high LF means more keys are trying to fit into a limited number of buckets
- How likely collisions are:** More items → more chances two keys hash to the same index
- Speed of operations:**
  - Low load factor** → fewer collisions → faster operations
  - High load factor** → many collisions → slow operations
- When to resize the hash table:** Most hash table implementations **automatically resize and rehash** when LF crosses a threshold

$$\left[ \begin{array}{l} \{ v_1, v_2, \dots \}, \\ \{ v_1, v_2, v_3, \dots \}, \\ \{ v_1, \dots \}, \end{array} \right] \quad n=5$$
$$a \geq n \Rightarrow \boxed{LF \geq 1}$$

## 10.6 - Common Operations

Friday, November 28, 2025 10:19 AM

- Insert
- Search
- Delete
- Update
- Membership
- Iteration
- Handling Collisions

## 10.7 - Complexity Analysis

Friday, November 28, 2025 10:28 AM

### Common Operations:

OPERATION	DICTIONARY	SET	Avg Time (theta)	Worst Time (O)
}	✓	✓	O(1)	O(n) ✓
	✓	✓	O(1)	O(n) ✓
	✓	✓	O(1)	O(n) ✓
	✓	✗	O(1)	O(n) ✓
	✓	✓	O(1)	O(n) ✓
	✓	✓	O(n)	O(n) ✓

## 10.8 - Coding Problems

Friday, November 28, 2025 10:36 AM

seen (.....)

target

seen = ~~set()~~

for x in seq:

target - x

seen = {3, 1}

x → 3 ~~+ 2~~

y → target - x  
→ 5 - 3

→ 2 ~~+ 3~~

→ (3, 2)

  
↳ (3,2)



























## 10.9 - DS Context

Tuesday, December 16, 2025 4:34 AM

### Data Science / ML Correlation:

- Feature encoding
- Caching & memoization
- Deduplication

### Libraries & Usage:

- *sklearn.feature\_extraction.FeatureHasher*

# 11 - Problem Solving

Thursday, December 4, 2025 9:17 AM

- Sliding Window
- Two Pointers
- Fast & Slow Pointers
- Prefix Sum
- Hashing
- Binary Search Pattern
- Greedy Approach



# Topics

Sunday, July 20, 2025 10:08 AM

## 1. Mathematical Foundations

- *Discrete Mathematics*
- *Recurrences & Growth Functions*
- *Probability & Expected Value*

## 2. Trees

- *Binary Search Tree*
- *Tree DP*
- *N-ary Tree*

## 3. Graphs

- *Graph Theory*
- *Traversal Algorithms*
- *Shortest Path Algorithms*
- *Topological Sorting & DAG Problems*
- *Disjoint Set Union*

## 4. Heaps

- *Mathematical Foundation*
- *Implementation*

## 5. Priority Queues (PQ)

- **PQ Theory**
- **Graph & PQ Applications**

## 6. Tries

- **Foundations**
- **Advanced Problems**

## 7. Backtracking

- **Theory**
- **Classic Problems**
- **Advanced Problems & Optimizations**

## 8. Dynamic Programming (DP)

- **Mathematical Foundation**
- **1-dimensional DP**
- **2-dimensional DP**
- **String DP**
- **DP on Trees & Graphs**

## 9. Segment Trees & Fenwick Trees

- **Segment Tree Theory**
- **Fenwick Trees**

## **10. Advanced Algorithmic Patterns**