EDYODA

Data Structures & Algorithms

What is a Data Structure?

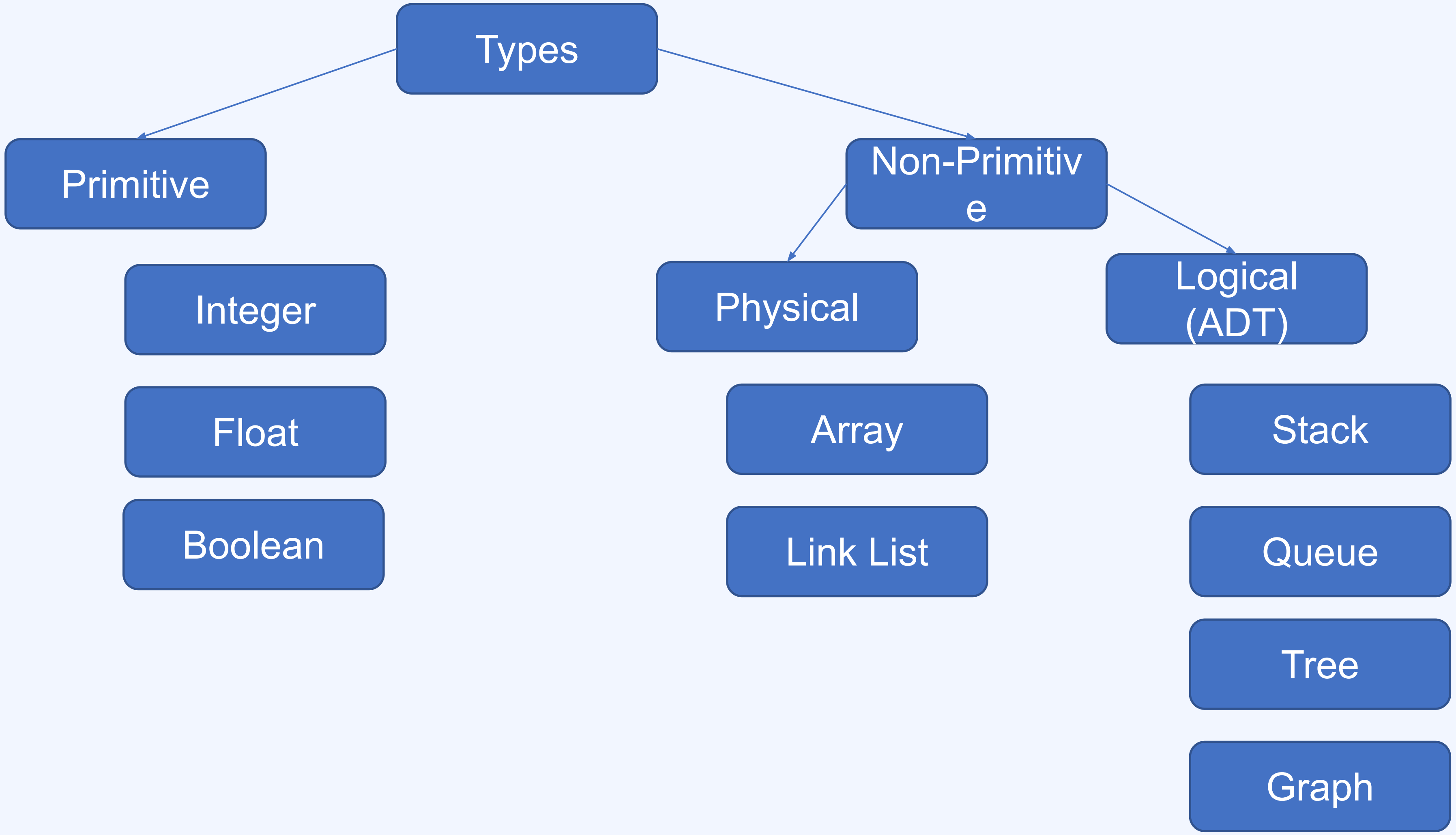
- It is a way to organize data that enables it to be processed in an efficient time.
- Some of the more commonly used data structures include – linked lists, arrays, stacks, queues, heaps, trees, and graphs.

What is an algorithm?

- Is a set of rules to be followed to solve a problem.
- To solve that problem we will be using some data structure.
- Example you want to prepare something from the data.

Types of Data Structure (DS)

ADT : Abstract Data Types



Linear and Non-Linear DS (Non – Primitive)

- Linear Data Structure:
  Data structure where data elements are arranged sequentially or linearly where the elements are attached to its previous and next adjacent in what is called a linear data structure. In linear data structure, single level is involved. Therefore, we can traverse all the elements in single run only. Linear data structures are easy to implement because computer memory is arranged in a linear way. Its examples are array, stack, queue, linked list, etc.

- Non-linear Data Structure:
  Data structures where data elements are not arranged sequentially or linearly are called non-linear data structures. In a non-linear data structure, single level is not involved. Therefore, we can't traverse all the elements in single run only. Non-linear data structures are not easy to implement in comparison to linear data structure. It utilizes computer memory efficiently in comparison to a linear data structure. Its examples are trees and graphs.

Why so many DS?

- Each DS has its own property
- This property is generally unique
- Based on the problem statement we select which DS to use

Runtime Analysis

- It is a study of runtime of a given algorithm, by identifying its behaviour as the input size for the algo increases.
- What is the performance of a given algorithm.
- Why do we need to measure the performance of the given algorithm? Basically, it tells us if our algo is good enough to give faster response on bigger data.

Notations in Runtime Analysis

- Consider an example of a car/bike, how do one tell about the mileage
  - Highway -> 20 km/l – Good (Best)
  - City Traffic -> 10 km/l – Poor (Worst)
  - Mixed Environment -> 15 km/l - Average
- Similarly algo runtimes may be defined as:
  - Best Case Scenario – Time Taken
  - Worst Case Scenario – Time Taken
  - Average Case Scenario – Time Taken

Types of Notations

- Omega Ω = 10sec [Best Case Scenario]
    - Gives the tighter lower bound for a given algorithm
    - Running time will not be less than this given time
- Big O = 100sec [Worst Case Scenario]
    - Gives the tighter upper bound for a given algorithm
    - Running time will not be greater than the given time
- Theta Θ – [Average Case Scenario]
    - Decides the upper bound and lower bound are same of not.
    - Running time of a given algorithm will be on an average be equal too this.

Time Complexity Examples

Say you have to write a program to search a given value in an array

[10, 5, 6, 11, 3,..... 26,....... 20,......... 30]

[1,1,1,,1,1...........m26....m20..........n]

Omega (Ω) – 10 search – Best case scenario Ω(1)

1 : Always guarantees that minimum time required for executing the algo.

Big O – Worst Case Scenario O(n)

n : Algo will not be more than the given time

Theta (Θ) *- Average Case Scenario* Θ(n/2)

n/2 : On an average it will take n/2 time on an average

What is an Array/List?

Array is a DS consisting of collection of elements each identified by an index. It is stored such that the position of each element can be computed from its index.

For example,

- List of Science marks for Class-XII: ***[89, 78, 95, 92, 91]***
- List of things to purchase: ***[“Apples”, “Notebook”, “Highlighters”, “Chocolates”]***

In programming languages like Python and JavaScript, arrays are heterogeneous which means you can store different data type values in the same array.

For example,

[“John”, “Wick”, 42, true]

Properties of Array

- Array can store values of any single data type. (int, Boolean, float, String)
- Every cell of an Array has an unique index.
- Index starts with 0.
- In programming languages like Python and JavaScript, you do NOT have to specify the length of the array. The length increases when you add more items. The length decreases when you remove items.

What is the need of an Array?

“We want to store 1 million data of some data types”

How do we maintain such list of variables?

We store it in an array!

Types of Arrays

- One Dimensional(1D) Array - Line
- Multi Dimensional Array
  - 2D - rows and columns, plane
  - 3D – depth, cube
  - 4D
  - ….
  - ND

Common Operations in an Array

- Declaring, instantiating, initializing an Array
- Inserting a value
- Traversing the array
- Accessing a given cell
- Searching a given value
- Deleting a given value

Declaring, Instantiating, Initializing

- Declaration: Create a reference ------- O(1)
  - arr = []
- Instantiating: Create and Array --- O(1), Space Complexity – O(n)
  - arr = [] * 5
- Initialization: Assign values to cells in Array ------- O(n)
  - arr[0]=10 ---- O(1)
  - arr[1]=20 ---- O(1)
  - arr[2]=30 ---- O(1)
  - arr[3]=40 ---- O(1)
  - arr[4]=50 ---- O(1)

Inserting New Item in an Array

```
insertElemAtIndex(arr, value, index) ------------ O(constant) = O(1)
    if(arr[index] is occupied) -------------- O (1)
        return error ----------------------- O (1)
    else ---------------------------------------- O (1)
        arr[index] = value --------------- O (1)
```

Time Complexity : O(1)

Space Complexity : O(1)

Traversal of an Array

```
traverse(arr):
    loop i=0 to len(arr) - 1 ------- O(n)
        print arr[i] ---------- O(1) | 20*O(1) | 100*O(1) | n * O(1)
```

Time Complexity : O (n)

Space Complexity : O (1)

Access an Item at Position in an Array

```
accessElemAtIndex(Arr, index)
    if index > len(arr)  ------------------- O(1)
        return exception ----------- O(1)
    else ------------------------------------- O(1)
        return arr[index] ----------- O(1)
```

Time Complexity : O(1)

Space Complexity : O(1)

Searching for an item in an Array

```
searchInArr(Arr, searchVal)
    loop i=0 to len(Arr) -1  ------------------------ O(n)
        if arr[i] == searchVal --------------- O(1)
            return i ---------------------- O(1)
    return error ---------------------------------- O(1)
```

Time complexity -> O(n)

Space -> O(1)

Delete an item from an Array

```
delete(Arr, location)
    if arr[location] is occupied ---------------------- O(1)
        arr[location] = Integer.Min_Value ---- O(1)
    else ---------------------------------------------------- O(1)
        return ------------------------------------------- O(1)
```

Time Complexity : O(1)

Space Complexity : O(1)

Uses of Arrays

- Create Hash Tables
- Create Stacks
- Create Queues

EDYODA

Stacks

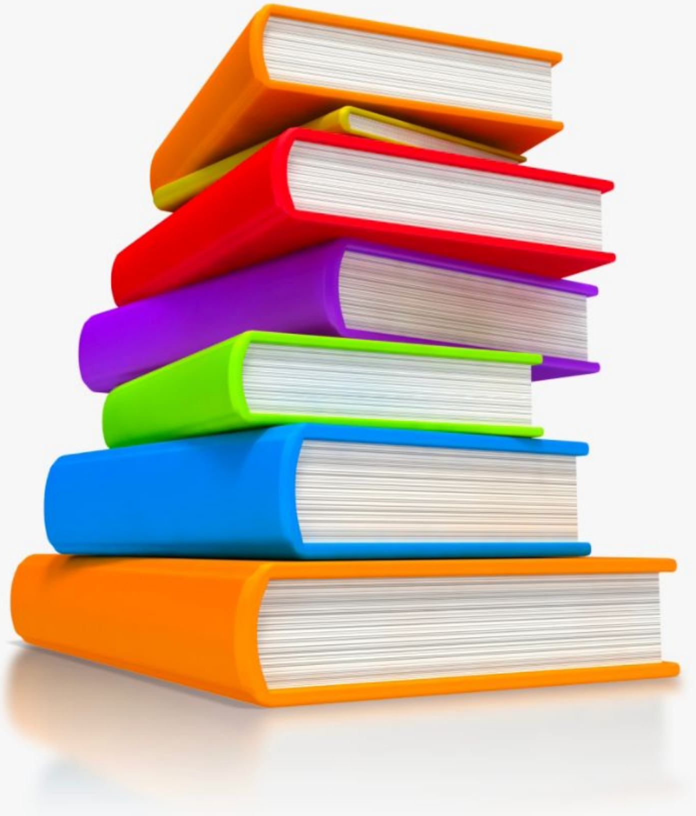
STACKS

L
I
F
O

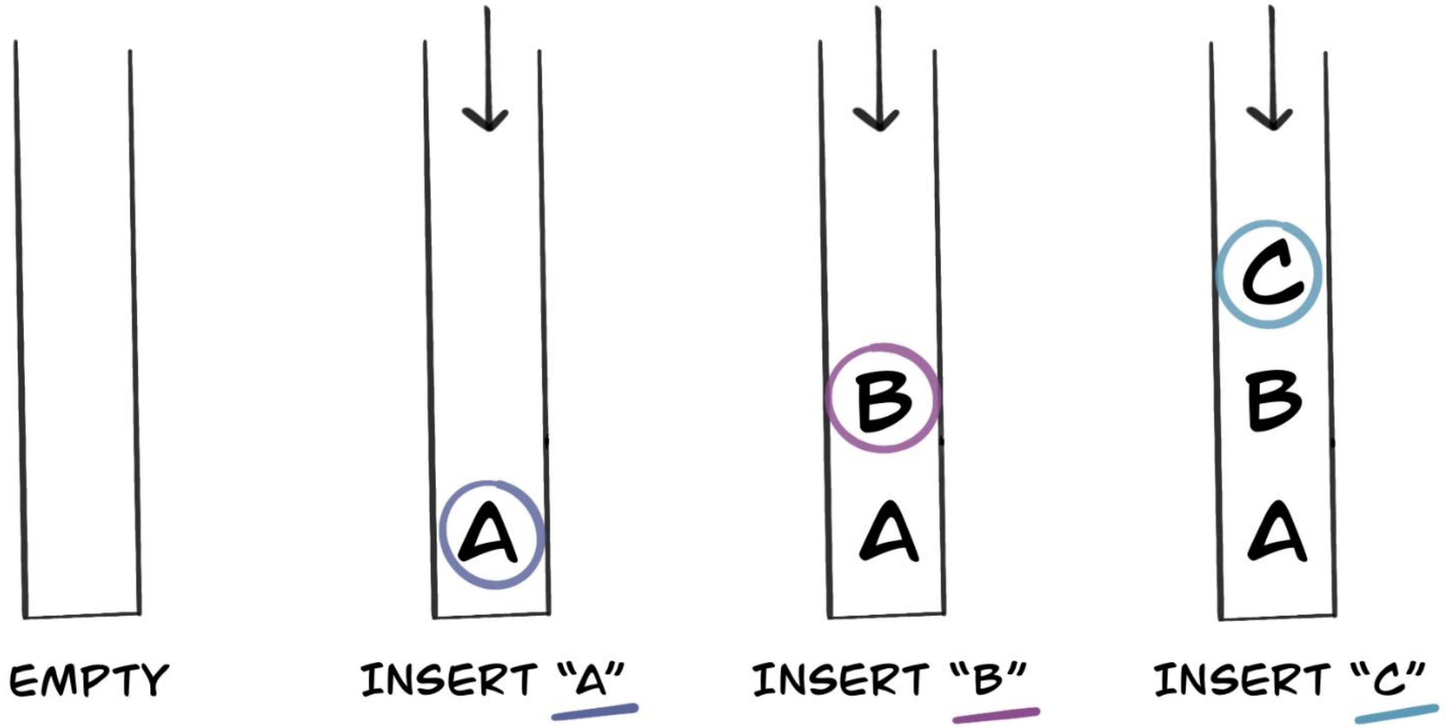
$\xrightarrow{}$

LAST-IN

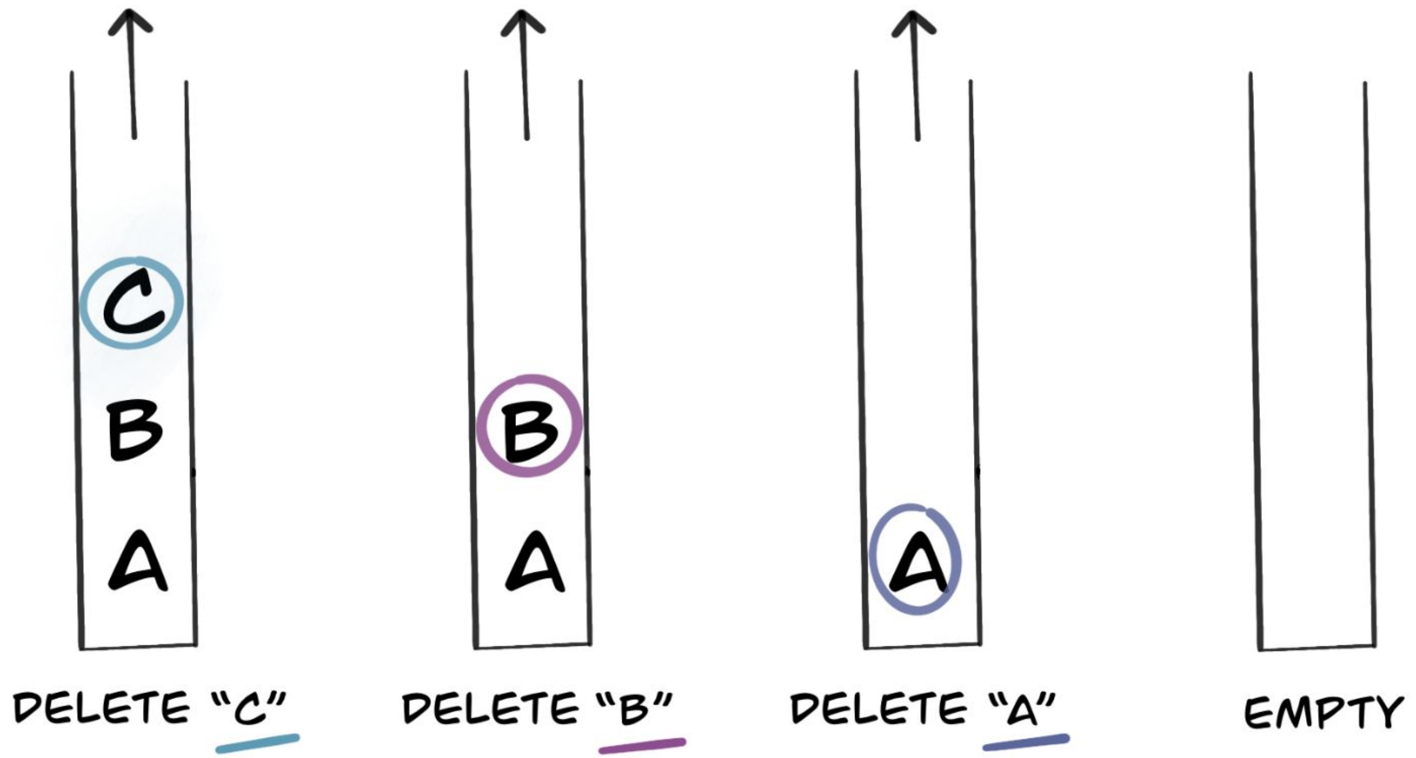
FIRST-OUT



STACKS INSERTION

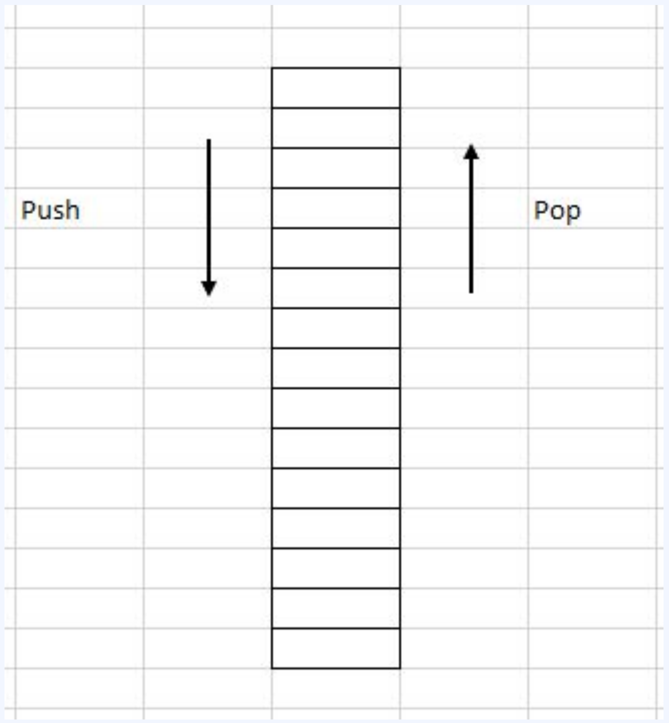


STACKS DELETE



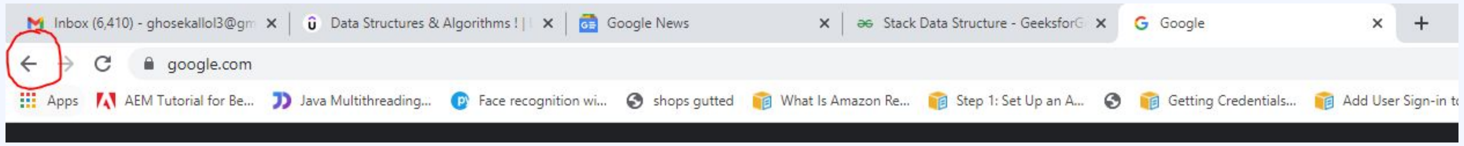
What is stack

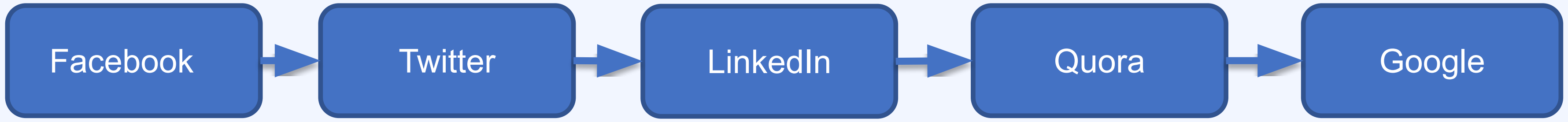
- Stack is a linear data structure which follows a particular order in which the operations are performed.
- Last in First Out (LIFO) property
- Can be implemented using
  - Arrays
  - Link List



Uses

- Back button on browser or in mobile apps





- Perform certain mathematical operation like postfix evaluation

Common Operation

- Push (Insertion)
- Pop (Deletion of a node)
- Peek (Shows the data of top element)
- IsEmpty – Checks if the stack is empty
- ~~isFull() : Only for Array Implementation~~
- Delete (Emptying the whole stack)

Push (Insertion) –

```
Push(nodeValue)
    create node with nodeValue ---------- O(1)
    node.next = top ------------------------ O(1)
    top = node ------------------------------- O(1)
```

Time Complexity : O(1)

Space Complexity : O(1)

Pop –

```
Pop()
    if stackIsEmpty() ----------------------------------- O(1)
        return error Message-------------- O(1)
    tempNode = top --------------------------- O(1)
    top = top.next ---------------------------- O(1)
    return tempNode --------------------------- O(1)
```

Time Complexity : O(1)
Space Complexity : O(1)

Peek

```
Peek() – Check what is the value at the top most location
    return top.value --------------------- O(1)
```

IsEmpty

```
IsEmpty()
    if top is null ---------------------------------- O(1)
        return True ------------------------- O(1)
    Else ---------------------------------------------- O(1)
        return False ------------------------ O(1)
```

Delete

Delete() – Emptying the whole stack

    top=null ----------------------------- O(1)

Queues

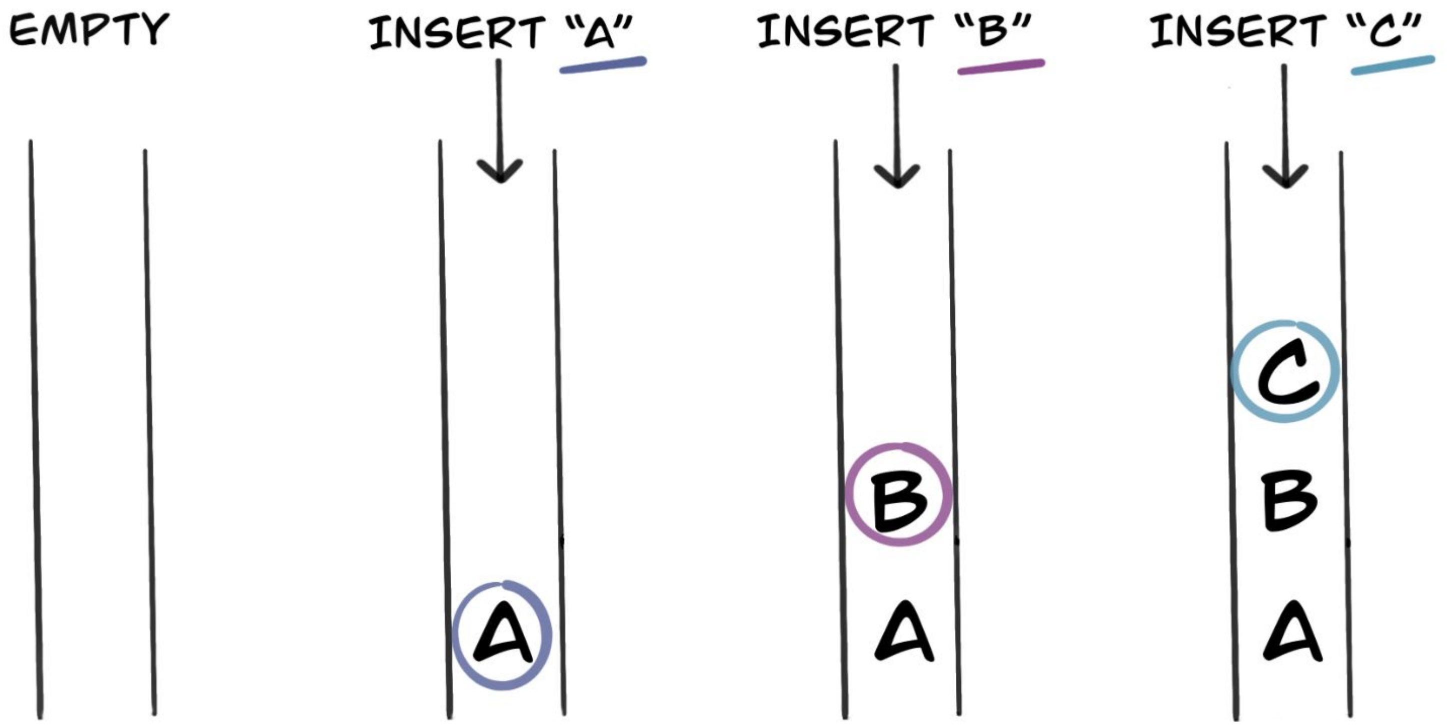
QUEUES

F
I
F
O

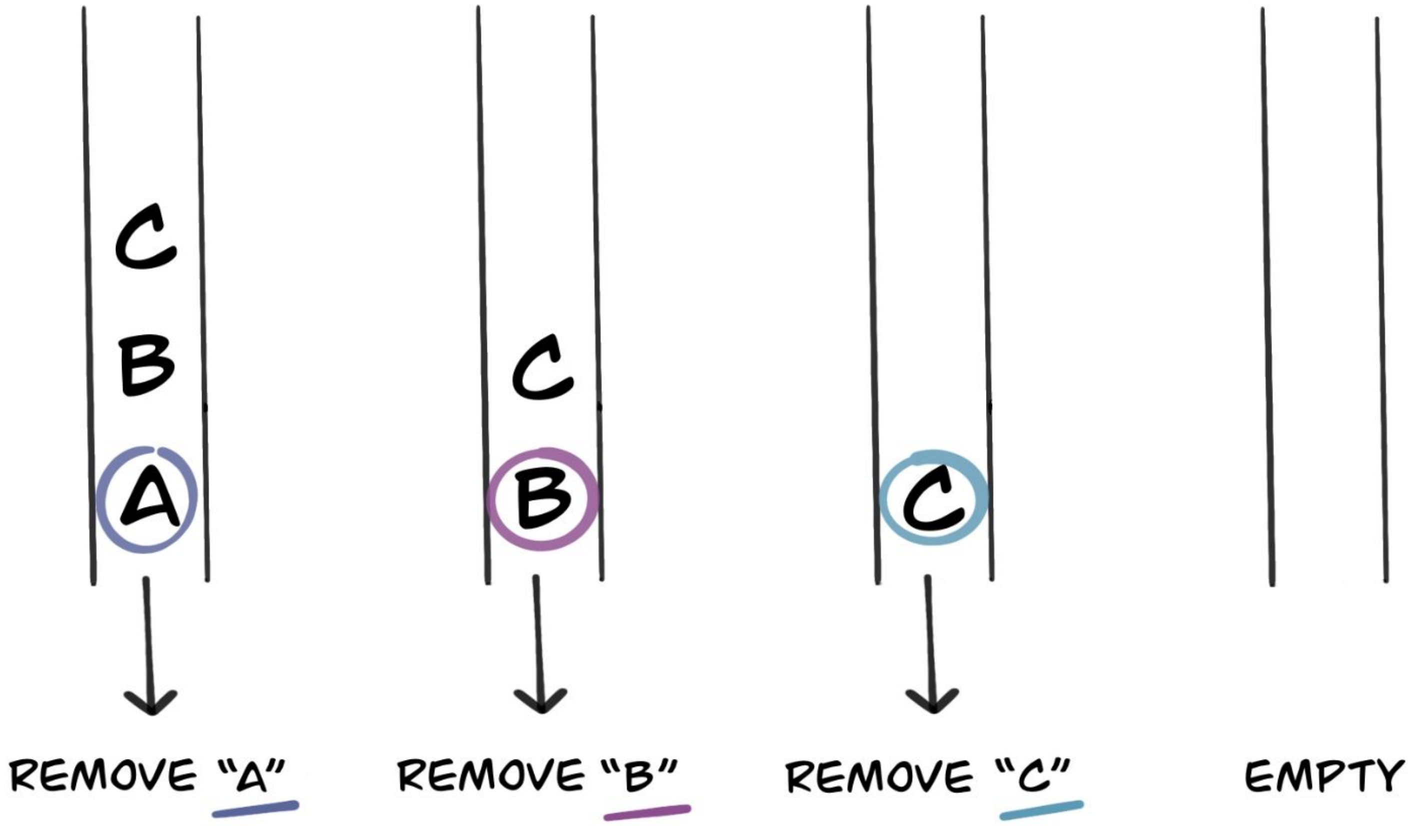
→ FIRST-IN

FIRST-OUT

QUEUE INSERTION



QUEUES REMOVE ITEM



What is queue

- Observations
  - New members addition is at the end of the queue
  - First person in the queue is the first person to get out of the queue
- First In First Out (FIFO)
- Example
  - Billing Counter
- Can be implemented using both Arrays and LL

Operations in a Queue

- createQueue()
- enQueue() – Adding to the queue (at the tail or end)
- deQueue() – Removing from the queue (at the start or head)
- peekInQueue()
- isEmpty()
- ~~isFull() : Only for Array Implementation~~
- deleteQueue()

Implementing Queue using LL

```
CreateQueue()
    head = None ---------------- O(1)
    tail = None ------------------- O(1)
```

Time : O(1)

Space : O(1)

Enqueue

```
Enqueue(nodeValue)
    create a New Node --------------------------- O(1)
    if tail is None ------------------------------ O(1)
        tail = new_node ------------------------ O(1)
        head = new_node --------------------- O(1)
    else -------------------------------------------------- O(1)
        tail.next = new_node ------------------ O(1)
        tail = new_node ------------------------- O(1)
```

Time Complexity – O(1)
Space Complexity – O(1)

Is Empty

```
isEmpty()
    if head is None
        return true
    else
        return false
```

Time Complexity – O(1)

Space Complexity – O(1)

DeQueue

```
deQueue()
    if isEmpty ---------------------------------------------- O(1)
        return error Message ------------------------- O(1)
    tmpNode = head -------------------------------------- O(1)
    head = head.next ------------------------------------- O(1)
    return tmpNode.data ---------------------------------- O(1)
```

Time Complexity – O(1)
Space Complexity – O(1)

Peek

```
Peek()
    if isEmpty()
        return error
    else
        return head.data
```

Time Complexity - O(1)

Space Complexity – O(1)

Delete

```
DeleteQueue()
    head = None
    tail = None
```

Time Complexity – O(1)

Space Complexity – O(1)

Postfix Evaluation

```
Read each character in the expression
loop unit expression is empty {
    When character read is an operand
        - Push to stack
    WHen character read is an operator (+, -, /)
        - Pop 2 elements from stack
        - Perform the mathmetical operation on the 2 operands using the operator
        - Push the result from step 2 to stack
}

Peek/pop stack -> This should be the answer of the postfix expression
```

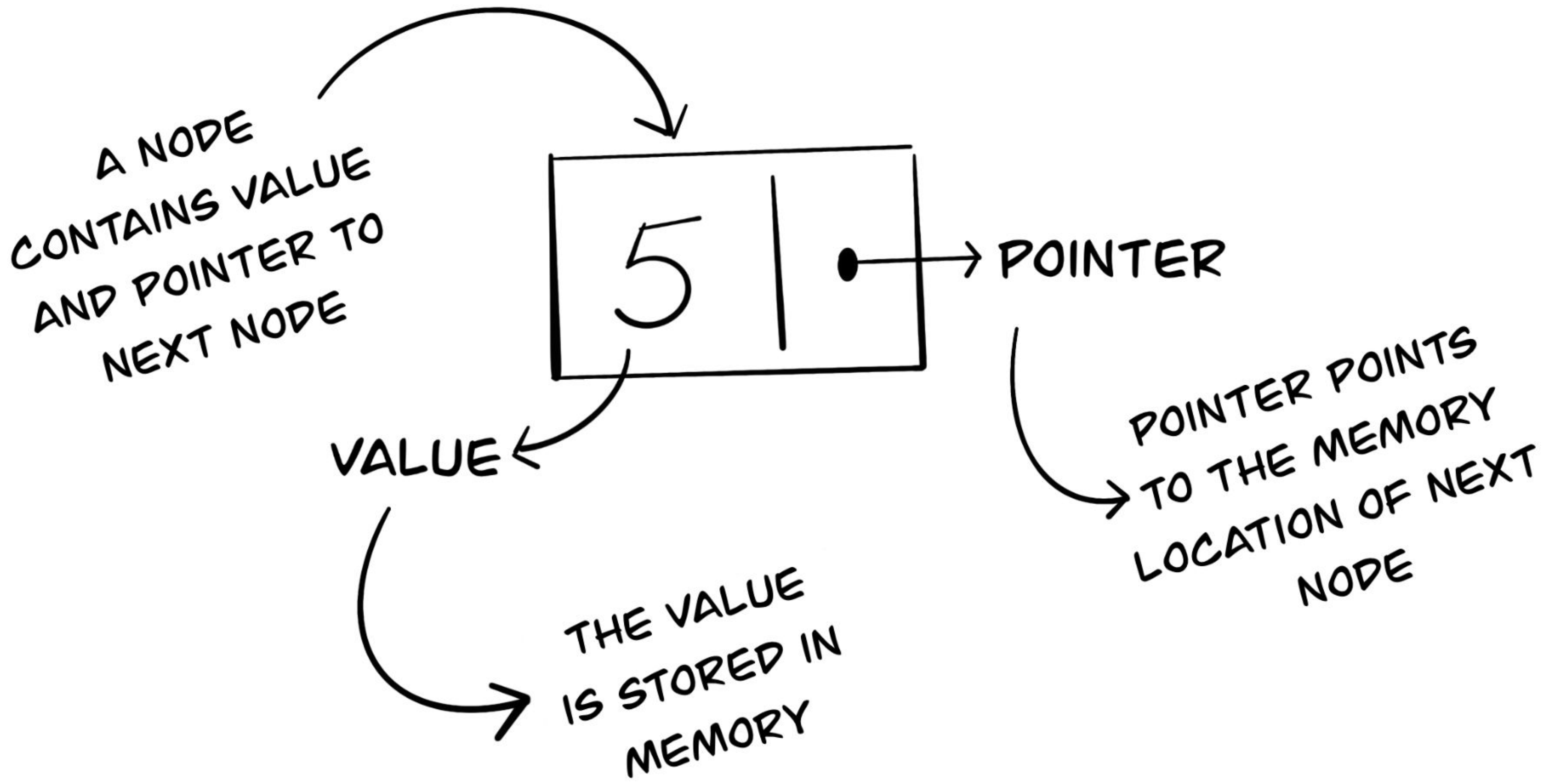
EDYODA

Linked List

Linked List

- A linked list is a sequence of data elements, which are connected together via links. Each data element contains a connection to another data element in form of a pointer.
- Each element (Node) consists of 2 components data and pointer.
- It is variable in size
    - Since variable in size much more flexible than array
    - Array size is static
    - Linked List is variable (increase/decrease in runtime[execution of the program])

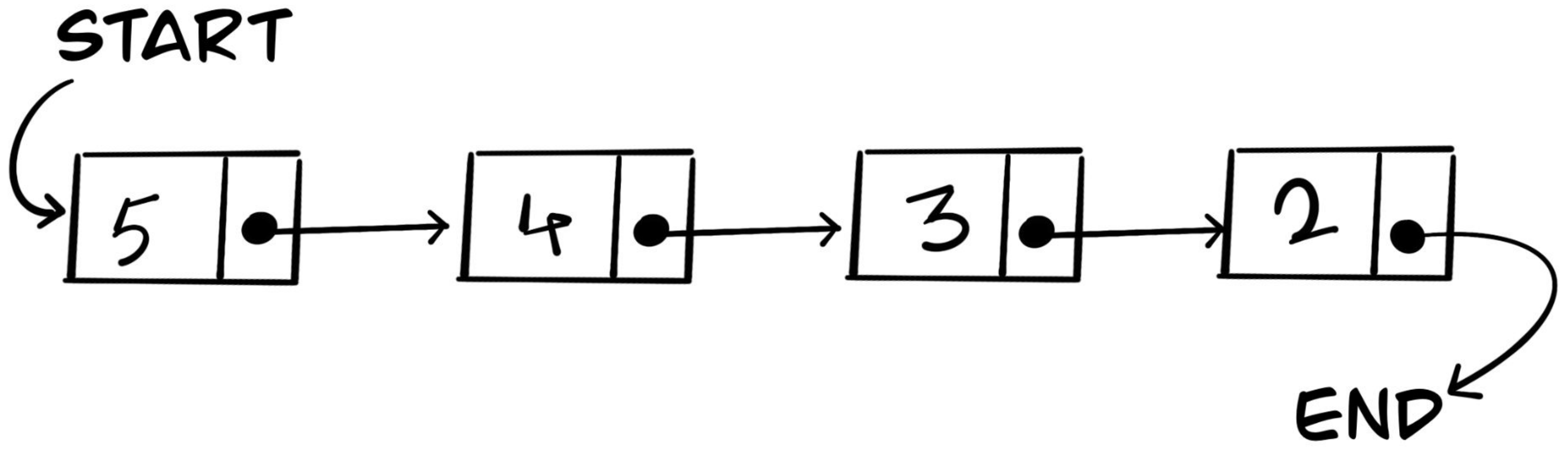
LINKED LIST


A NODE CONTAINS VALUE AND POINTER TO NEXT NODE
5
POINTER
VALUE
POINTER POINTS TO THE MEMORY LOCATION OF NEXT NODE
THE VALUE IS STORED IN MEMORY

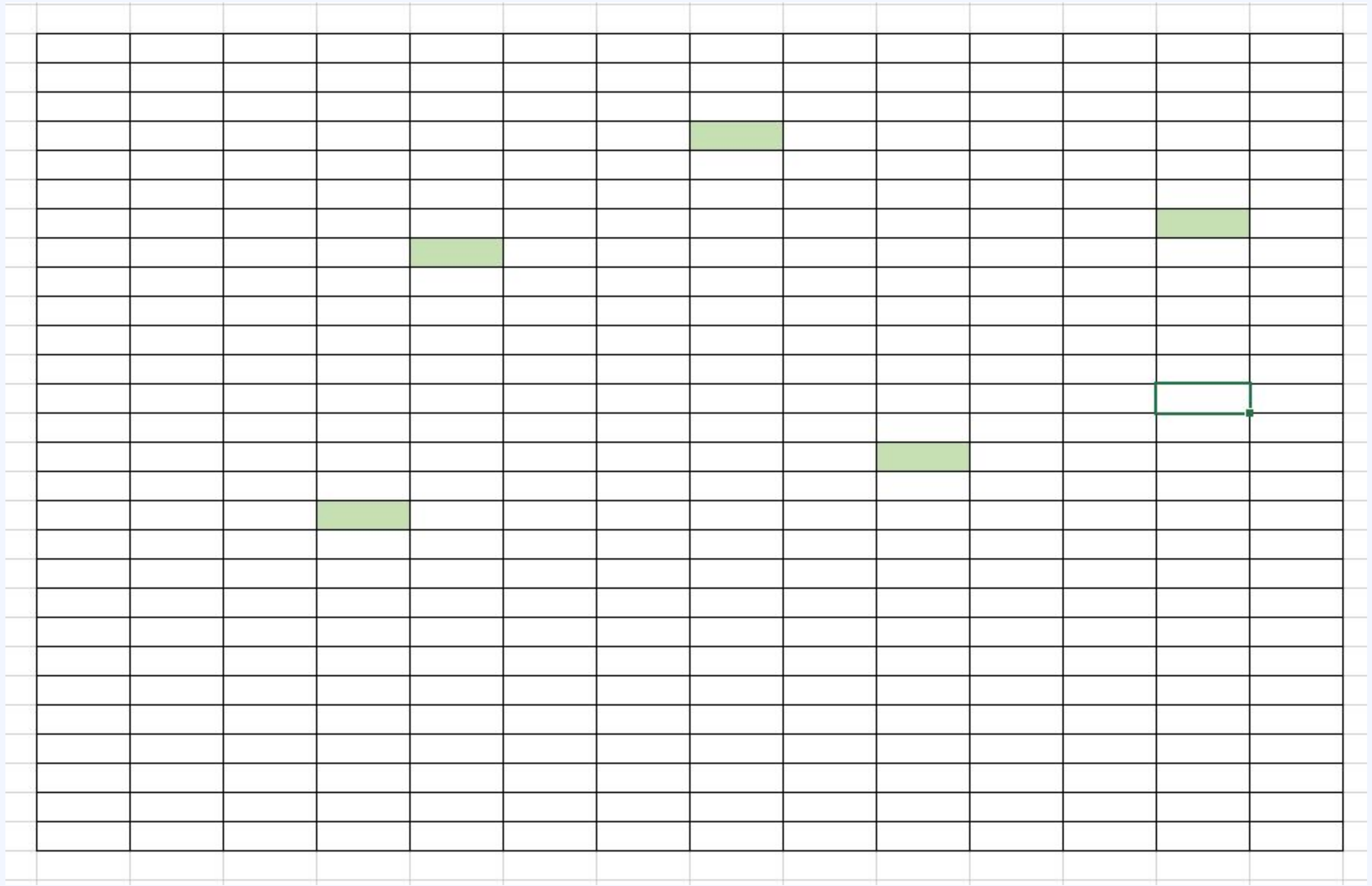
Components in Linked List

- Head – Point to 1st Node
- Tail – Point to Last Node
- Node
  - Data
  - Reference

LINKED LIST



Link List Storage in Memory



Types of link list

- Single Link List
  - In a single link list each node store the data and reference to the next node
- Circular Single Link List
  - The last node of the link list refers to the first node
- Double Link List
  - Each node refers to previous and next node
- Circular Double Link List
  - Double Link List which is circular

Common Operations On Single LL

- Creation of LL
- Insertion of Data
- Traversal
- Searching a given Value
- Deletion of Node
- Deletion of Linked List

Creation

```
CreateSingleLL(nodeValue)
    create head, tail initial value as Null ---- O(1)
    create a blank node ------------------------- O(1)
    node.data = nodeValue -------------------- O(1)
    node.next = None -------------------------- O(1)
    head = Node --------------------------------- O(1)
    tail = Node ------------------------------------ O(1)
```

Time Complexity – O(1)
Space Complexity – O(1)

Insertion

1. New Node at start of the Link List (before head)
2. New Node at end of the Link List (after tail)
3. New Node at specific location

Insert

```
InsertLinkList(head, nodeValue, Location)
    create a blank Node
    node.Value = nodeValue
    if not existsLL()
        return error
    else if location is 0 //First Node
        newNode.next = head
        head = newNode
    else if location equals last
        newNode.next = null
        tail.next = newNode
        tail = newNode
    else
        loop tempNode = 0 to location – 1
        newNode.next = tempNode.next
        tempNode.next = newNode
```

Time Complexity : O(n)
Space Complexity : O(1)

Traversal

```
Traversal(head):
    if !exsitsLL()
        return
    loop:head to tail
        print currentNode.value
```

Time Complexity – O(n)

Space Complexity – O(1)

Search For a given value

```
Search(head, nodevalue)
    loop: head to tail
        if currentNode.value == nodeValue
            print currentNode.value
            return
    return value not found
```

Time Complexity : O(n)

Space Complexity : O(1)

Delete a node

- Delete first Node
    - When more than one node
    - When only one node
- Delete last Node
    - When more than one node
    - When only one node
- Delete Middle Node

Delete…

Time Complexity = O(n)
Space Complexity = O(1)

```
DeletionOfNode(head, location)
    if !exisitLL()
        return error
    if location == 0
        head = head.next
        if this is the only element then update tail = null
    else if location >= last
        if  only element then head=tail = null return
        loop tmpNode till 2nd last element
        tail = tmpNode
        tmpNode.next = Null
    else
        loop tmpNode: 0 to location -1
        tempNode.next = tempNode.next.next
```

Delete entire LL

```
deleteEntireLL(head)
    head = null
    tail = null
```

Time Complexity – O(1)

Space Complexity – O(1)

Circular LL

Common Operation

- Creation
- Insertion
- Traversal
- Searching
- Deletion of a Node
- Delete entire LL

Insert

```
InsertLinkList(head, nodeValue, Location)
    create a blank Node
    node.Value = nodeValue
    if !existsLL()
        return error
    else if location is 0
        node.next = head
        head = node
        tail.next = head
    else if location equals last
        node.next = head
        tail.next = Node
        tail = node
    else
        loop tempNode = 0 to location – 1
        node.next = tempNode.next
        tempNode.next = node
```

Traversal

```
Traversal(head):
    if head == NULL
        return
    loop:head to tail
        print currentNode.value
```

Time Complexity – O(n)

Space Complexity – O(1)

Search

```
Search(head, nodevalue)
    loop: tempNode = head to tail
        if tempNode.value == nodeValue
            print tempNode.value
            return
    return value not found
```

Time Complexity : O(n)

Space Complexity : O(1)

Delete a node (1st , Last, Middle)

```
DeletionOfNode(head, location)
    if !exisitLL()
        return error
    if location == 0
        head = head.next
        tail.next = head
        if this is the only element then update head = tail = node.next = null
    else if location >= last
        if  only element then head=tail = node.next null return
        loop tmpNode till 2nd last element
        tail = tmpNode
        tmpNode.next = head
    else
        loop tmpNode: start to location -1
        tempNode.next = tempNode.next.next
```

Usage

- Implement Logical DS
- **Python
    - It is a substitute for the array when the size is not known.
- It is used in many programming languages to implements other language specific collections
    - In Java, HashMap -> the internal working of HashMap used linked list
- In DB linked list is used to add relations/storage
    - B+ Tree along with linked list

Recursion

What is Recursion

- Example from Non – Computing World:

Recursion Properties

- Same operation being performed with different input
- In every step we try to make the problem smaller
- We mandatorily need to give a base condition
  - A condition which stops the recursion
    - Either found the answer
    - Answer does not exists
- It is managed using stacks

Usage

- Recursion simplifies implementation in many cases
- Greatly used in below scenarios
  - Trees
  - Graphs
  - Dynamic Algorithm
  - Sorting Algorithms

Format

- **Recursive Case** : Case where function recur with different (smaller) parameters
- ****Base Case** : Case where the function does not recur (recursion stops)

```
SampleRecursion(parameter)
    if base case
        return some base case value
    else
        SampleRecursion(modifiedParameter)
```

Recursion Find Factorial

- Definition
  - Factorial of non – negative integer n
  - Denoted by n!
  - Is the product of all positive integers from 1 to n
- Example
  - 5! = 1*2*3*4*5 = 120
  - 6! = 1*2*3*4*5*6 = 720
  - 0! = 1

Algo For Factorial

```
factorial(n)
    if n equals 0
        return 1
    return n * factorial(n-1)
```

Time Complexity

Recursions

Calculate Time Complexity (O)

- Iterative Algorithm
- Recursive Algorithm

Iterative

Find the largest number in an array

```
largestNumber(arr):
    largestNumber = Integer.Min------------------- O(1)
    for i in range(0, len(arr)) ----------------------- O(n)
        if arr[i] > largestNumber -------------- O(1)
            largestNumber = arr[i] -------- O(1)
    return largestNumber -------------------------- O(1)
```

Time Complexity = $O(1) + O(n) + O(1) = O(n)$

Recursive #1

```
Set global highest = Integer.Min (-99999999)
biggestNumber(A, index):
    if index == -1
        return highest
    else
        if A[index] > highest
            highest = A[index]
        return biggestNumber(A, index-1)
```

Recursive #1

```
biggestNumber(A, index): ------------------------------ (Say T(n))
    if index == -1---------------------------------------- O(1)
        return highest ------------------------------ O(1)
    else ------------------------------------------------- O(1)
        if A[index] > highest ---------------------- O(1)
            highest = A[index]--------------- O(1)
        return biggestNumber(A, index-1)---- T(n-1)
```

Back Substitution

Back Substitution O(1) as 1

T(n) = 1 + T (n-1) ------------ Eq#1

T(-1) = 1 ----------------------- Base Condition

T(n-1) = 1 + T((n-1)-1) = 1 + T(n-2)----- Eq #2

T(n-2) = 1 + T((n-2)-1) = 1 + T(n-3) ----- Eq #3

T(n) = 1 + T(n-1)

= 1 + (1+T(n-2)) = 2 + T(n-2)

= 2 + (1 + T(n-3)) = 3 + T(n-3)

= k + T(n-k)

= n + 1 + T(n- (n+1)) [select a value of k such that T is removed]

= n + 1 + T(-1)

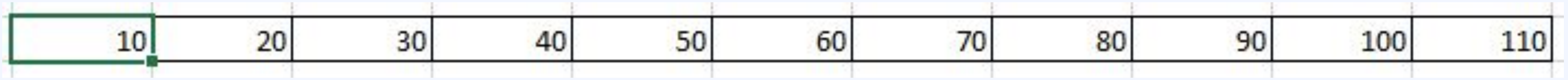
= n + 1 + 1

= n + 1

= O(n)

Example 2 (Binary Search)

- Find a number in a sorted array
- Binary Search



10	20	30	40	50	60	70	80	90	100	110

Algo

```
binarySearch(arr, findNumber, start, end)
    if start == end
        if arr[start] == findNumber
            return start
        else
            return error not present
    mid = findmid(arr, start, end) # Return the index
    if findnumber < arr[mid]
        binarySearch(arr, findNumber, start, mid)
    else if findNumber > arr[mid]
        binarySearch(arr, findNumber, mid+1, end)
         else if arr[mid] == findNumber
        return mid
```

Binary Search Time Complexity

```
binarySearch(arr, findNumber, start, end) ----------------------------- T(n)
    if starts = end ------------------------------------------------------ O(1)
        if arr[start] == findNumber ---------------------------- O(1)
            return start --------------------------------------- O(1)
        else ------------------------------------------------------ O(1)
            return error not present --------------------- O(1)
    mid = findmid(arr, start, end) --------------------------------- O(1)
    if arr[mid] > findnumber ---------------------------------------- O(1)
        binarySearch(arr, findNumber, start, mid) --------- T(n/2)
    else if arr[mid] < findNumber --------------------------------- O(1)
        binarySearch(arr, findNumber, mid+1, end) ------- T(n/2)
          else if arr[mid] == findNumber -------------------------------- O(1)
        return mid ------------------------------------------------- O(1)
```

Equations

T(n) = T(n/2) + 1 ------------ Eq(#1)
T(1) = 1 ---------------------- Base Equation
T(n/2) = T(n/4) + 1 ----------- (#2)
T(n/4) = T(n/8) + 1 ----------- (#3)

T(n) = T(n/2) + 1
= (T(n/4) + 1) + 1 = T(n/4) + 2
= (T(n/8) + 1) + 2 = T(n/8) + 3
= T(n/2^k) + k
= T(1) + log(n) = 1 + log(n) = log(n)

Element T function
n/2^k = 1
n= 2^k
k = log(n)

Advanced DS

Searching & Sorting

Linear Search

Binary Search

Bubble Sort

Insertion Sort

Selection Sort

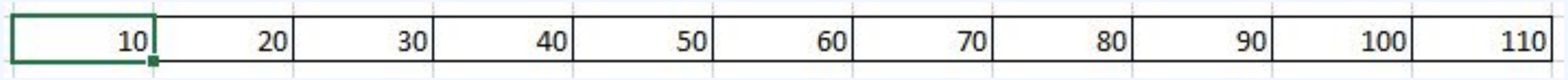
Linear Search

```
Linear_search(arr, n, search_value)
    for i = 0 to n-1
        if arr[i] == search_value
            print(“Value found”)
            return
    print(“Value not found”)
```

Time Complexity = O(n)
Space Complexity = O(1)

Binary Search

- Find a number in a **Sorted** array
- Binary Search



10	20	30	40	50	60	70	80	90	100	110

Algo(Recursion)

```
binarySearch(arr, findNumber, start, end)
    if start == end
        if arr[start] == findNumber
            return start
        else
            print(“Data is not present)”
            return
    mid = findmid(arr, start, end) # Returning the middle index
    if findnumber < arr[mid]
        binarySearch(arr, findNumber, start, mid)
    else if findNumber > arr[mid]
        binarySearch(arr, findNumber, mid+1, end)
       else if arr[mid] == findNumber
        return mid
```

Complexity

- Time Complexity = O(logn)
- Space Complexity = O(1)

Binary Search Time Complexity

```
binarySearch(arr, findNumber, start, end) ----------------------------- T(n)
    if starts = end ------------------------------------------------------ O(1)
        if arr[start] == findNumber ----------------------------- O(1)
            return start ------------------------------------------ O(1)
        else ---------------------------------------------------------------- O(1)
            return error not present --------------------- O(1)
    mid = findmid(arr, start, end) ---------------------------------- O(1)
    if arr[mid] > findnumber ---------------------------------------- O(1)
        binarySearch(arr, findNumber, start, mid) --------- T(n/2)
    else if arr[mid] < findNumber --------------------------------- O(1)
        binarySearch(arr, findNumber, mid+1, end) ------- T(n/2)
          else if arr[mid] == findNumber --------------------------------- O(1)
        return mid ---------------------------------------------------- O(1)
```

Equations

T(n) = T(n/2) + 1 ------------ Eq(#1)
T(1) = 1 ---------------------- Base Equation
T(n/2) = T(n/4) + 1 ----------- (#2)
T(n/4) = T(n/8) + 1 ----------- (#3)

T(n) = T(n/2) + 1
= (T(n/4) + 1) + 1 = T(n/4) + 2
= (T(n/8) + 1) + 2 = T(n/8) + 3
= T(n/2^k) + k
= T(1) + log(n) = 1 + log(n) = log(n)

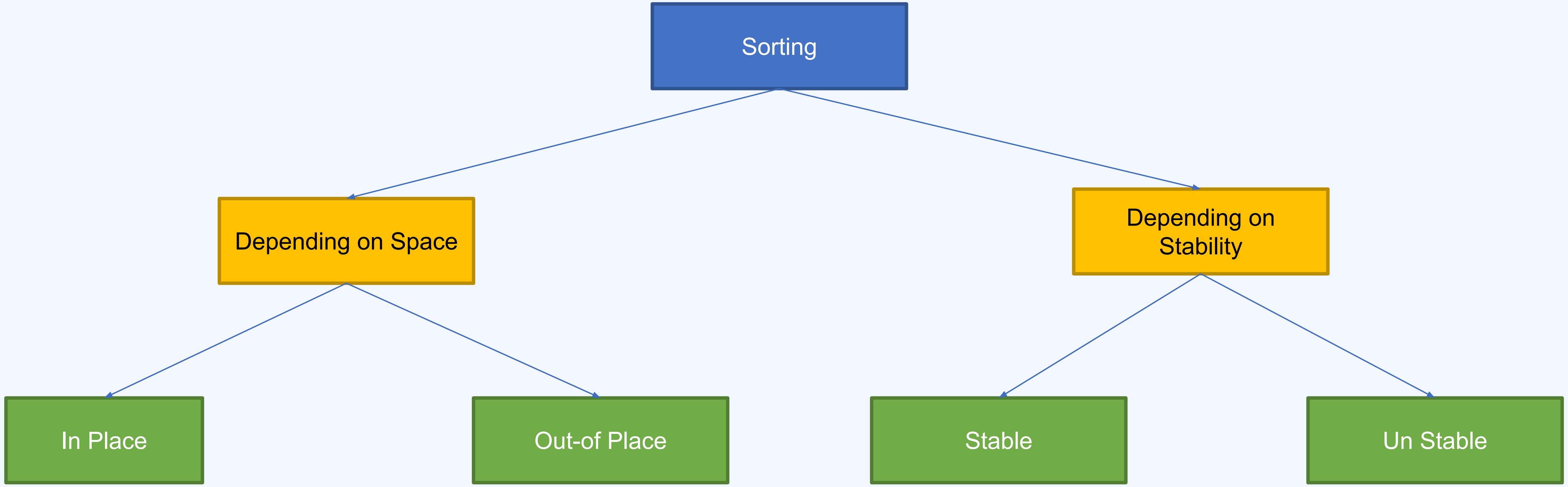
Element T function
n/2^k = 1
n= 2^k
k = log(n)

Sorting

What is sorting

- Arrange the given data in a particular format
  - Ascending order
  - Descending order
- Where do we find sorting
  - Excel
  - Online Shopping
    - Sort by Price low to high
    - Sort by Price high to low

Types of Sorting



Types of Sorting

- In Place Sorting
  - Does not require any additional space for sorting
  - Example : Bubble Sort
- Out Place
  - Does require additional space for sorting
  - Example : Merge Sort

Type of Sorting

- Stable Sort
  - If a sorting algorithm after sorting the contents **does not** change the sequence of similar content in which they appear is called Stable Sorting
  - Example : Insertion Sort
- Un Stable Sort
  - If a sorting algorithm after sorting the contents **does** changes the sequence of similar content in which they appear is called Un stable Sorting
  - Example : Quick Sort

Bubble Sort

- Also know as Sinking Sort
- Repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in wrong order.

Algorithm

```
bubbleSort(arr):
    n = arr.length
    for i in 0 to n-1
        for j in 0 to n-i-1
            if arr[j] > arr[j+1]
                swap(arr[j], arr[j+1])
```

Complexity

```
bubbleSort(arr):
    n = arr.length ------------------------------------------- O(1)
    for i in 0 to n-1 ------------------------------------------ O(n)
        for j in 0 to n-i-1 ---------------------------------- O(n)
            if arr[j] > arr[j+1] ------------------------ O(1)
                swap(arr[j], arr[j+1]) ----------- O(1)
```

Time Complexity – O(n^2)

Space Complexity – O(1)

When to Use/Not Use

- Use
  - Space is concern
  - Easy to implement
- Not Use
  - Average Time Complexity is poor

Selection Sort

- Is based on idea of finding the minimum or maximum element in the an unsorted array and then putting it in the correct position in a sorted array
- Breaks into sorted and un-sorted array

Selection Sort

```
selectionSort(arr):
    loop: j = 0 to n-1:
        iMin = j
        loop: i = j+1 to n-1
            if arr[i] < arr[iMin]
                iMin = i
        if iMin != j
            swap(arr[j], arr[iMin])
```

Complexity

```
selectionSort(arr):
    loop: j = 0 to n-1: -------------------------------------- O(n)
        int iMin = j ------------------------------------------ O(1)
        loop: i = j+1 to n-1 ------------------------------ O(n)
            if arr[i] < arr[iMin] ---------------------- O(1)
                iMin = I ---------------------------- O(1)
        if iMin != j ------------------------------------------- O(1)
            swap(arr[j], arr[iMin]) ------------------ O(1)
```

Time Complexity : O(n^2)
Space Complexity : O(1)

When to use/not use

- Use
  - No additional memory
  - Easy to implement
- Not use
  - When time complexity is a concern

Insertion Sort

- Divide the array in two part sorted and unsorted
- Then from unsorted we pick the first element and find its correct position in sorted array
- Repeat until unsorted array is empty

Algorithm

```
insertionSort(arr):
    loop: i = 1 to n
        currentNumber = arr[i]
        j = i
        while arr[j-1] > currentNumber && j > 0
            arr[j] = arr[j-1]
            j- -
        arr[j] = currentNumber
```

When to use/not use

- Use
  - No additional Memory
  - Simple Implementation
- Not Use
  - When time complexity is to be considered