# Data Structure And Algorithms:

## 1.Arrays:

- **An array is a data structure used to store a collection of elements, typically of the same type, in a contiguous block of memory.**
- **Code:**
  - **arr = [10, 20, 30, 40]  # A list is a flexible type of array in Python**
    **print(arr[2])  # Outputs: 30**
  - **arr=[[2,3],[3,4]] # 2D Array**
    **print(arr[0][1]) # Outputs:3**

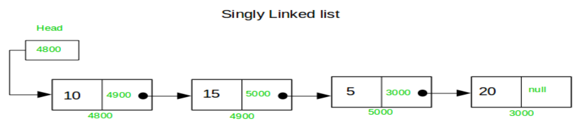
- **Types of Arrays:**
  - **One-dimensional array: A simple list of elements.**
  - **Multi-dimensional array: Like a table (2D), or a cube (3D), etc.**
- **Common Operations On Arrays:**
  - 1. Accessing Elements
    - Use an index to access elements.
    - `arr = [10, 20, 30]`
    - `print(arr[1])  # Output: 20`
  - 2. Inserting Elements
    - Add elements at the end or a specific position.
    - `arr.append(40)         # Adds to the end`
    - `arr.insert(1, 15)      # Inserts 15 at index 1`
  - 3. Deleting/Removing Elements
    - Remove by index or value.
    - `arr.pop(2)             # Removes item at index 2`
    - `arr.remove(20)         # Removes first occurrence of 20`
  - 4. Traversing
    - Loop through the array.
    - `for item in arr:`
    - `    print(item)`
  - 5. Searching
    - Find the index of an element.
    - `arr.index(15)          # Returns index of first 15`
  - 6. Sorting
    - Arrange elements in order.
    - `arr.sort()             # Ascending order`
    - `arr.sort(reverse=True) # Descending order`
  - 7. Reversing
    - Reverse the array.
    - `arr.reverse()`
  - 8. Slicing
    - Get a portion of the array.
    - `arr[1:3]               # Elements from index 1 to 2`
  - 9. Length
    - Find number of elements.
    - `len(arr)`

## 2.Strings:

- **Strings are immutable in many languages (e.g., Python, Java), meaning once created, they can't be changed directly.**
- **Strings are immutable in many languages (e.g., Python, Java), meaning once created, they can't be changed directly.**
- **Example:**
  - str = "mon"
  - str[0] = "h"
    - You will get an error, because strings are immutable in Python — meaning once a string created, you cannot change its characters directly.
- **Common String Operations:**
  - **Concatenation: Joining strings**
    `"Hello, " + "world!"  # Output: Hello, world!`
  - **Slicing/Substrings: Getting parts of a string**
    `"Hello"[1:4]  # Output: ell`
  - **Length:**
    `len("Hello")  # Output: 5`
  - **Searching:**
    `"world" in "Hello world"  # Output: True`

## 3.Linked_List:

- A linked list is a linear data structure in which elements (called nodes) are connected using pointers. Unlike arrays, elements are not stored in contiguous memory, and each node points to the next one.
- Types of Linked Lists:
  - **Singly Linked List**:
    - Each node points to the **next** node.
    - One-way traversal.

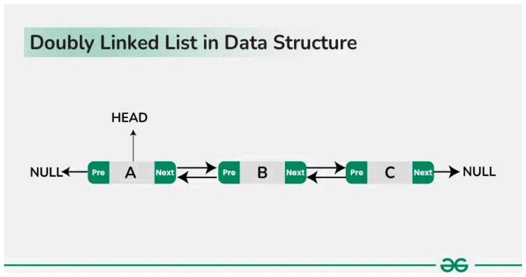


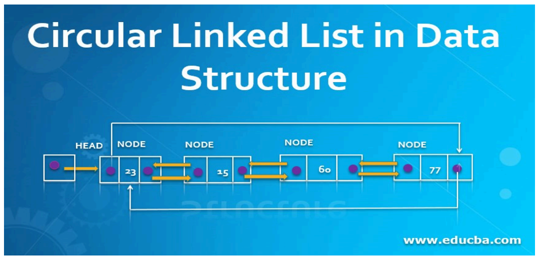
    - 
    - **Code:**

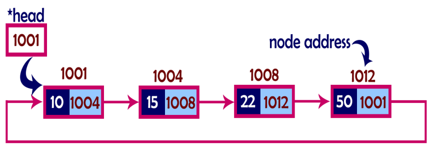
```
class Node:

    def __init__(self, data):

        self.data = data  # Value of the node

        self.next = None  # Pointer to the next node
```

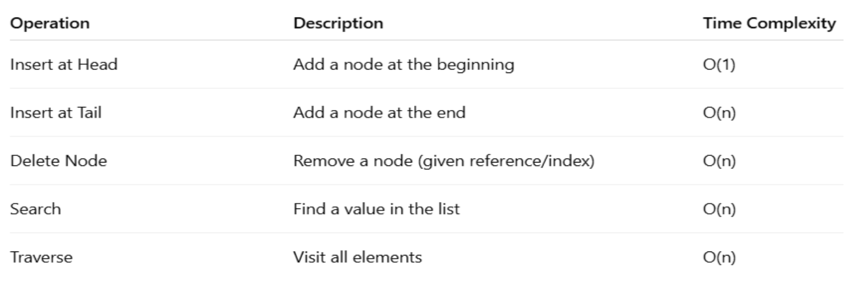
  - **Doubly Linked List**:
    - Each node points to **both** the next and previous nodes.
    - Two-way traversal.

```
class Node:

    def __init__(self, data):

        self.data = data  # Value of the node

        self.next = None  # Pointer to the next node

        self.n=pre=None  #Pointer to Previous node
```



- **Circular Linked List**:
  - The last node points back to the first node.
  - Can be singly or doubly linked.
  - 



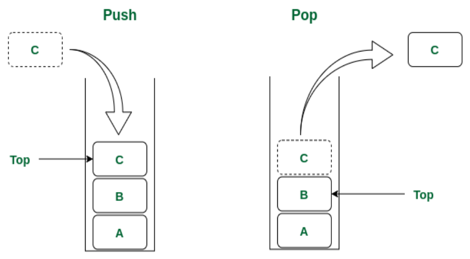
- 

| Operation | Description | Time Complexity |
|---|---|---|
| Insert at Head | Add a node at the beginning | O(1) |
| Insert at Tail | Add a node at the end | O(n) |
| Delete Node | Remove a node (given reference/index) | O(n) |
| Search | Find a value in the list | O(n) |
| Traverse | Visit all elements | O(n) |

## 4.Stacks:

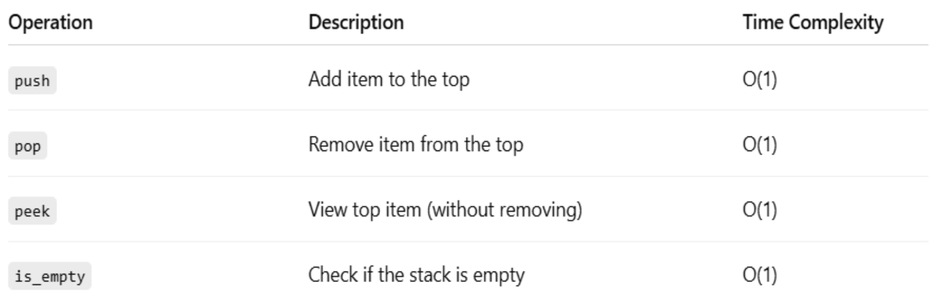
- A stack is a linear data structure that follows the LIFO principle — Last In, First Out. This means the last element added to the stack is the first one to be removed.

- **Real-Life Analogy:**
    - Think of a stack like a pile of plates — you add a plate to the **top**, and you remove a plate from the **top**.



Stack Data Structure

-
- **Time Complexity:**



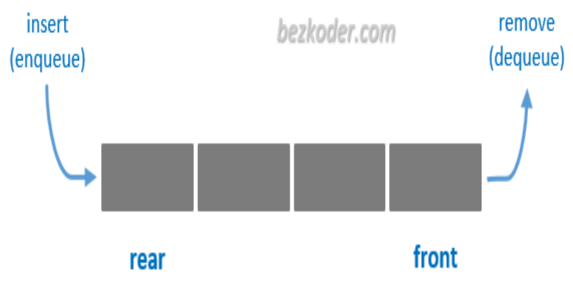
| Operation | Description | Time Complexity |
|---|---|---|
| `push` | Add item to the top | O(1) |
| `pop` | Remove item from the top | O(1) |
| `peek` | View top item (without removing) | O(1) |
| `is_empty` | Check if the stack is empty | O(1) |

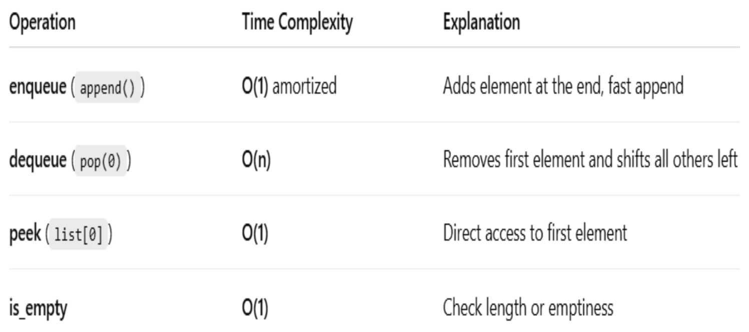
-
- **Monotonic Increasing Stack :***A Monotonic Increasing Stack is a stack that maintains its elements in increasing order from bottom to top. This means each new element pushed is greater than or equal to the element before it.*
- **Monotonic Decreasing Stack:***A **Monotonic Decreasing Stack** is a stack that maintains its elements in **decreasing order** from **bottom to top**.This means **each new element pushed is less than or equal to the one before it** (from top to bottom).*

```
class Stack:

    def __init__(self):

        self.stack = []

    def push(self, item):

        # Add item to the top of the stack

        self.stack.append(item)

    def pop(self):

        # Remove and return the top item

        if not self.is_empty():

            return self.stack.pop()

        else:

            print("Stack is empty")

            return None

    def peek(self):
        # Return the top item without removing it
        if not self.is_empty():
            return self.stack[-1]
        else:
            print("Stack is empty")
            return None
    def is_empty(self):
        # Check if stack is empty
        return len(self.stack) == 0
    def size(self):
        # Return the size of the stack
        return len(self.stack)
    def display(self):
        # Print stack elements (top on the right)
        print("Stack:", self.stack)
# Example usage
s = Stack()
s.push(10)
s.push(20)
s.push(30)
s.display()        # Stack: [10, 20, 30]
print(s.pop())     # Output: 30
s.display()        # Stack: [10, 20]
print(s.peek())    # Output: 20
```

## 5.Queues: *Note When We Use Linked List For Implementing Queue We Get O(1) For Removing And Adding Element.Where If We Use Array We Get O(1) For Adding And O(n) For Removing.*

- **Types:***Normal Queue,Circular Queue,Deque*
- *Normal Queue:*
    - *A **queue** is a **linear data structure** that follows the **FIFO** principle — **First In, First Out**.*
    - *This means the **first element** added to the queue is the **first one** to be removed.*

- 

- 

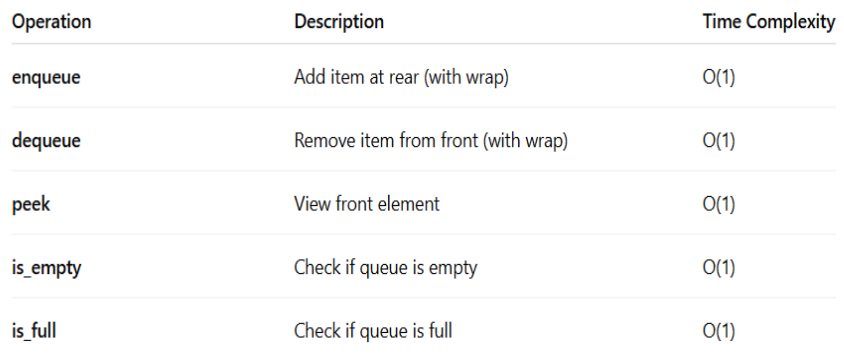
| Operation | Time Complexity | Explanation |
|---|---|---|
| enqueue ( `append()` ) | O(1) amortized | Adds element at the end, fast append |
| dequeue ( `pop(0)` ) | O(n) | Removes first element and shifts all others left |
| peek ( `list[0]` ) | O(1) | Direct access to first element |
| is_empty | O(1) | Check length or emptiness |

- *Here* dequeue is O(n) because we shift all elements to left.

```
class NormalQueue:

  def __init__(self):

    self.queue = []

  def enqueue(self, item):

    # Add item at the end (rear)

    self.queue.append(item)

  def dequeue(self):

    # Remove item from the front (inefficient in list)

    if not self.is_empty():

      return self.queue.pop(0)  # O(n) operation due to shifting

    else:

      print("Queue is empty")

      return None

  def peek(self):
        # View the front item without removing
        if not self.is_empty():
            return self.queue[0]
        else:
            print("Queue is empty")
            return None
    def is_empty(self):
        # Check if the queue is empty
        return len(self.queue) == 0

    def size(self):
        # Return the current size of the queue
        return len(self.queue)
    def display(self):
        # Print queue elements from front to rear
        print("Front ->", self.queue, "<- Rear")
# Example usage:
q = NormalQueue()
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
q.display()           # Front -> [10, 20, 30] <- Rear
print(q.dequeue())    # Output: 10
q.display()           # Front -> [20, 30] <- Rear
print(q.peek())       # Output: 20
```

- *Circular Queue:*
    - *A **Circular Queue** is a variation of a normal queue where the last position is connected back to the first position to make a **circle**. This allows efficient use of space by **reusing empty slots** left by dequeued elements.*

```
class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
    self.front = -1

    self.rear = -1

def is_empty(self):

    return self.front == -1

def is_full(self):

    return (self.rear + 1) % self.size == self.front

def enqueue(self, data):

    if self.is_full():

        print("Queue is full")

        return

    if self.is_empty():

        self.front = 0

    self.rear = (self.rear + 1) % self.size

    self.queue[self.rear] = data

def dequeue(self):

    if self.is_empty():

        print("Queue is empty")

        return None

    data = self.queue[self.front]

    if self.front == self.rear:

        # Queue has only one element, reset queue after removing it

        self.front = -1

        self.rear = -1

    else:

        self.front = (self.front + 1) % self.size

    return data

def peek(self):

    if self.is_empty():

        print("Queue is empty")

        return None

    return self.queue[self.front]

def display(self):
        if self.is_empty():
            print("Queue is empty")
            return
        i = self.front
        print("Queue elements:", end=" ")
        while True:
            print(self.queue[i], end=" ")
            if i == self.rear:
                break
            i = (i + 1) % self.size
        print()

# Example usage
cq = CircularQueue(5)
cq.enqueue(10)
cq.enqueue(20)
cq.enqueue(30)
cq.display()       # Output: Queue elements: 10 20 30
print("Dequeued:", cq.dequeue())  # Output: Dequeued: 10
cq.display()       # Output: Queue elements: 20 30
cq.enqueue(40)
cq.enqueue(50)
cq.enqueue(60)      # Queue is full now
cq.display()       # Output: Queue elements: 20 30 40 50 60
```

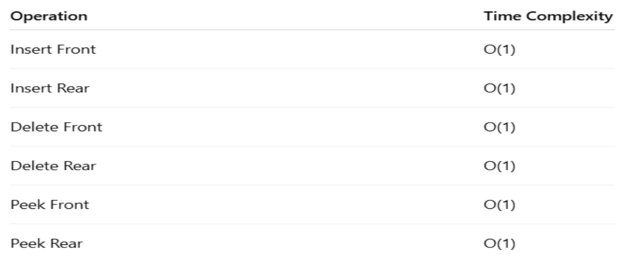


| Operation | Description | Time Complexity |
|---|---|---|
| **enqueue** | Add item at rear (with wrap) | O(1) |
| **dequeue** | Remove item from front (with wrap) | O(1) |
| **peek** | View front element | O(1) |
| **is_empty** | Check if queue is empty | O(1) |
| **is_full** | Check if queue is full | O(1) |

- 
- *Here the dequeue* is O(n) as we do not shift elements after dequeue operation.

- **Double-Ended Queue (Deque):**A **Double-Ended Queue (Deque)** is a data structure that allows insertion and removal of elements from **both the front and the rear ends**.

- **Key points about Deque:**
    - Supports operations at **both ends**:
        - Insert Front, Insert Rear
        - Delete Front, Delete Rear
    - Can be used as a queue (FIFO) or a stack (LIFO).
    - Efficient for problems needing flexible insertion/removal.



| Operation | Time Complexity |
|---|---|
| Insert Front | O(1) |
| Insert Rear | O(1) |
| Delete Front | O(1) |
| Delete Rear | O(1) |
| Peek Front | O(1) |
| Peek Rear | O(1) |

```
from collections import deque

class MyDeque:

    def __init__(self):

        self.dq = deque()

    def insert_rear(self, item):

        self.dq.append(item)

        print(f"Inserted at rear: {item}")

    def insert_front(self, item):

        self.dq.appendleft(item)

        print(f"Inserted at front: {item}")

    def delete_front(self):

        if self.is_empty():

            print("Deque is empty, cannot delete front.")

            return None

        item = self.dq.popleft()

        print(f"Deleted from front: {item}")

        return item


    def delete_rear(self):
        if self.is_empty():
            print("Deque is empty, cannot delete rear.")
            return None
        item = self.dq.pop()
        print(f"Deleted from rear: {item}")
        return item
    def peek_front(self):
        if self.is_empty():
            print("Deque is empty, no front element.")
            return None
        return self.dq[0]
    def peek_rear(self):
        if self.is_empty():
            print("Deque is empty, no rear element.")
            return None
        return self.dq[-1]
    def is_empty(self):
        return len(self.dq) == 0
    def size(self):
        return len(self.dq)
    def display(self):
        print("Deque contents:", list(self.dq))
# Example usage:
dq = MyDeque()
dq.insert_rear(10)
dq.insert_front(5)
dq.insert_rear(20)
dq.display()  # Deque contents: [5, 10, 20]
print("Front element:", dq.peek_front())  # Front element: 5
print("Rear element:", dq.peek_rear())    # Rear element: 20
```

```
dq.delete_front()  # Deleted from front: 5

dq.delete_rear()   # Deleted from rear: 20

dq.display()       # Deque contents: [10]

print("Size:", dq.size())  # Size: 1

print("Is empty?", dq.is_empty())  # Is empty? False

dq.delete_front()  # Deleted from front: 10

dq.delete_front()  # Deque is empty, cannot delete front.

print("Is empty?", dq.is_empty())  # Is empty? True
```

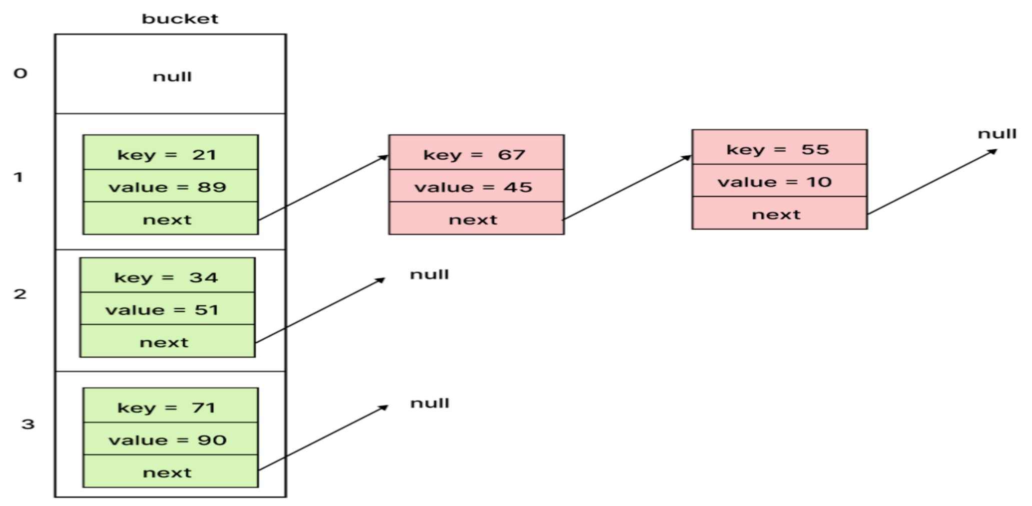
## 6.HashMap:

### →How HashMap works (Internals) :

• HashMap internally uses an array of buckets to store entries. Each bucket can hold a linked list (or a balanced tree in later Java versions for better performance with many collisions) of Entry objects.
• When you put a key-value pair , the hash code of the key is calculated , and this hash code is used to determine the bucket where the entry should be placed.

- So when we want to search key in HashMap it HashValue is Calculated which is like index by using this index we can access value in O(1) ,in Worst Case it is log(n) ,if we have more keys at one bucket(hashValue) which are stored in Balances tree form it take lon(n) to search on that key.

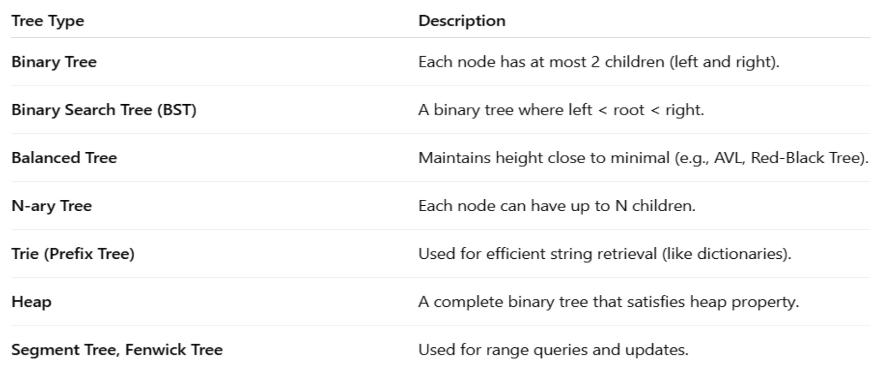
• If multiple keys have the same hash code (hash collision) , they are stored in the same bucket (as a linked list or tree).
• Operations like get, put, and remove involve calculating the hash code , finding the appropriate bucket , and then traversing the linked list/tree within that bucket to find or modify the entry.



- In Python We can Use dictionary as HashMap → w={}

## 7.Tree:a tree is a non-linear data structure that represents a hierarchical relationship between elements (called nodes).

- **Types:**

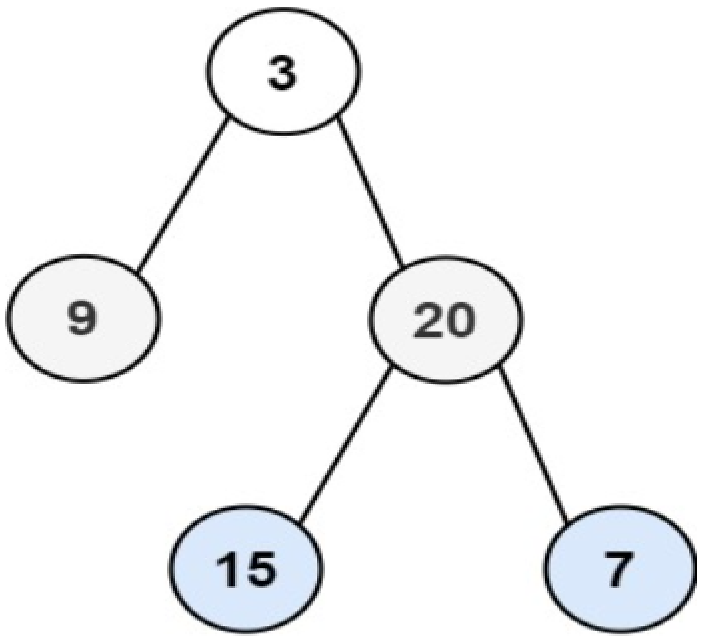


| Tree Type | Description |
|---|---|
| **Binary Tree** | Each node has at most 2 children (left and right). |
| **Binary Search Tree (BST)** | A binary tree where left < root < right. |
| **Balanced Tree** | Maintains height close to minimal (e.g., AVL, Red-Black Tree). |
| **N-ary Tree** | Each node can have up to N children. |
| **Trie (Prefix Tree)** | Used for efficient string retrieval (like dictionaries). |
| **Heap** | A complete binary tree that satisfies heap property. |
| **Segment Tree, Fenwick Tree** | Used for range queries and updates. |

- **Creation Of Tree Using BFS:**

```
from collections import deque
class Node:
    def __init__(self, val):
        self.data = val
        self.left = None
        self.right = None
def build_tree_from_list(values):
    if not values or values[0] is None:
        return None
    root = Node(values[0])
    queue = deque([root])
    i = 1
    while queue and i < len(values):
        current = queue.popleft()
        # Assign left child
        if i < len(values) and values[i] is not None:
            current.left = Node(values[i])
            queue.append(current.left)
        i += 1
        # Assign right child
        if i < len(values) and values[i] is not None:

            current.right = Node(values[i])

            queue.append(current.right)

        i += 1

    return root

# Inorder Traversal to verify

def inorder(root):

    if root:

        inorder(root.left)

        print(root.data, end=' ')

        inorder(root.right)

# Example usage

elements = [1, 2, 3, 4, 5, None, 7]

tree_root = build_tree_from_list(elements)

print("Inorder traversal of the tree:")

inorder(tree_root)  # Output: 4 2 5 1 3 7
```

- **Binary Tree Level Order Traversal:(BSF)**



```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
```

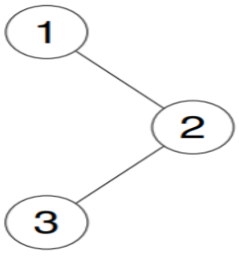
```
def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        q=[]
        w=[]
        q.append(root)
        while q:
            r=[]
            for i in range(len(q)):
                temp=q.pop(0)
                if temp.left!=None:
                    q.append(temp.left)
                if temp.right!=None:
                    q.append(temp.right)
                r.append(temp.val)
            w.append(r)
        return w
```

- **Binary Tree Inorder Traversal: (DFS)**

**Input:** root = [1,null,2,3]

**Output:** [1,3,2]

**Explanation:**



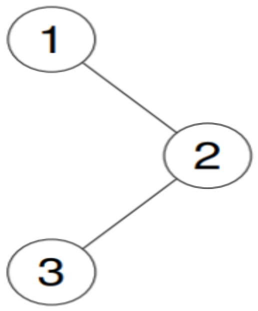
```
def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        q=[]
        def inorder(temp):
            if temp==None:
                return
            inorder(temp.left)
            q.append(temp.val)
            inorder(temp.right)

        inorder(root)

        return q
```

- **Binary Tree Preorder Traversal:(DFS)**

**Input:** root = [1,null,2,3]

**Output:** [1,2,3]

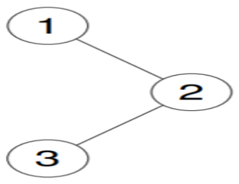
**Explanation:**



```
def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:

        q=[]

        def inorder(temp):

            if temp==None:

                return

            q.append(temp.val)

            inorder(temp.left)

            inorder(temp.right)

        inorder(root)

        return q
```

- **Binary Tree Postorder Traversal:(DFS)**

**Input:** root = [1,null,2,3]
**Output:** [3,2,1]
**Explanation:**



```
def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:

        q=[]

        def inorder(temp):

            if temp==None:

                return

            inorder(temp.left)

            inorder(temp.right)

            q.append(temp.val)

        inorder(root)

        return q
```

**7.BackTracking:***Backtracking is a general algorithmic technique that involves exploring all possible options and backtracking when a path fails to lead to a solution. It is typically used to solve problems that require searching through all combinations or permutations***.**

- ***Permutation:*** **nums=[1,2,3]**

```
def permute(self, nums: List[int]) -> List[List[int]]:

        def back(nums,temp,visit,result):

            if len(temp)==len(nums):

                result.append(temp+[])

                return

            for i in range(len(nums)):

                if visit[i]==0:

                    visit[i]=1

                    temp.append(nums[i])

                    back(nums,temp,visit,result)

                    temp.pop()

                    visit[i]=0

        temp=[]

        visit=[0]*len(nums)

        result=[]
```

```
 back(nums,temp,visit,result)

 return result
```

***Output:[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]***

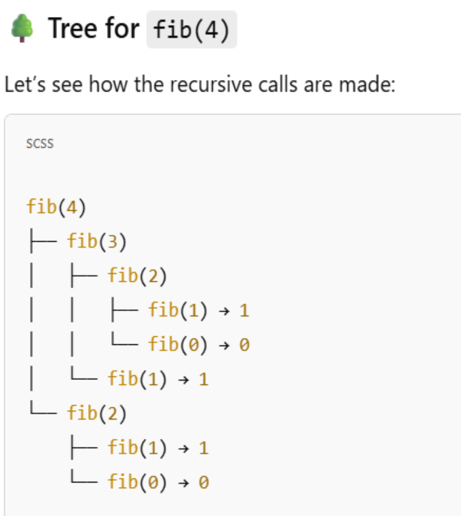
- **combinations:**
- ***Given two integers n and k, return all possible combinations of k numbers chosen from the range [1, n].***
- ***You may return the answer in any order***
- ***Input: n = 4, k = 2***
- ***Output: [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]***

```
def combine(self, n: int, k: int) -> List[List[int]]:

    visit=[0]*(n+1)

    temp=[]

    r=[]

    def back(n,visit,temp,r,k,m):

      if len(temp)==k:

          r.append(temp+[])

          return

      for i in range(m,n+1):

          if visit[i]!=1:

              visit[i]=1

              temp.append(i)

              back(n,visit,temp,r,k,i+1)

              temp.pop()

              visit[i]=0

    back(n,visit,temp,r,k,1)

    return r
```

**8.Recursion:*****Recursion is a method of solving problems where a function calls itself to solve smaller subproblems.***

***Example 2: Fibonacci:***

```
def fibonacci(n):

    if n <= 1:

        return n  # base case

    return fibonacci(n - 1) + fibonacci(n - 2)  # recursive case


print(fibonacci(6))  # Output: 8
```

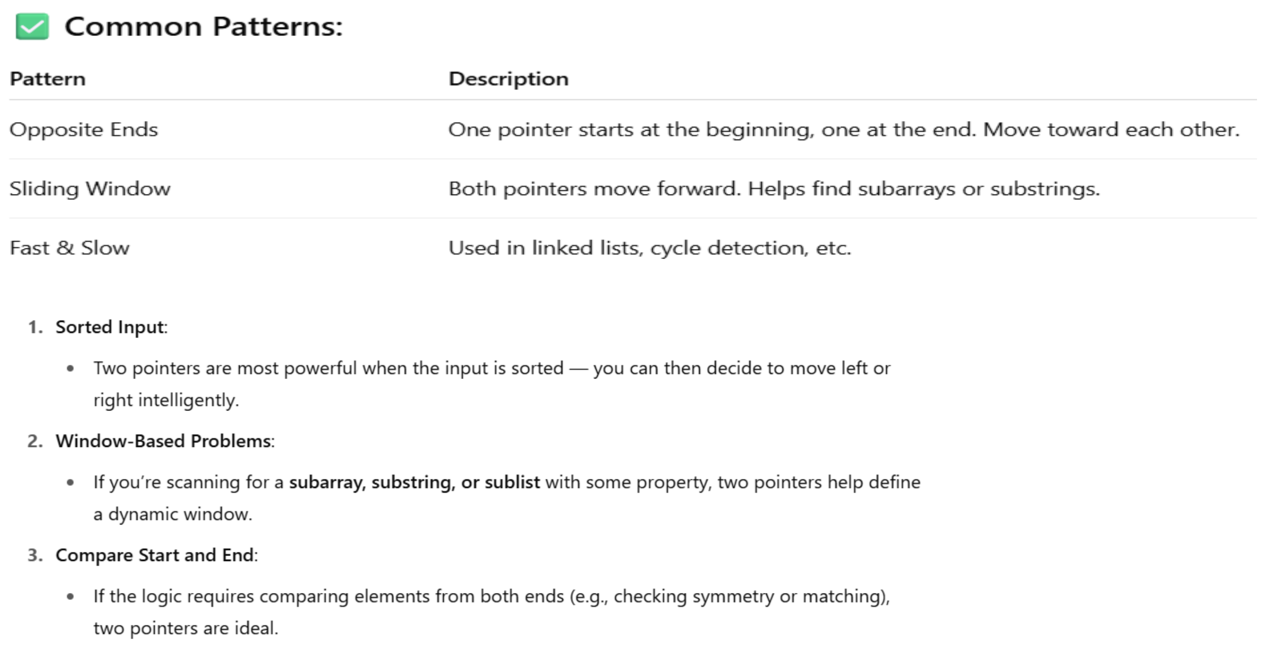


### 🌳 Tree for `fib(4)`

Let's see how the recursive calls are made:

```scss
fib(4)
├─ fib(3)
│   ├─ fib(2)
│   │   ├─ fib(1) → 1
│   │   └─ fib(0) → 0
│   └─ fib(1) → 1
└─ fib(2)
    ├─ fib(1) → 1
    └─ fib(0) → 0
```

**9.Two Pointer Technique:** *The two pointer technique is a very common and efficient approach used to solve problems that involve linear data structures like arrays, strings, or linked lists.*



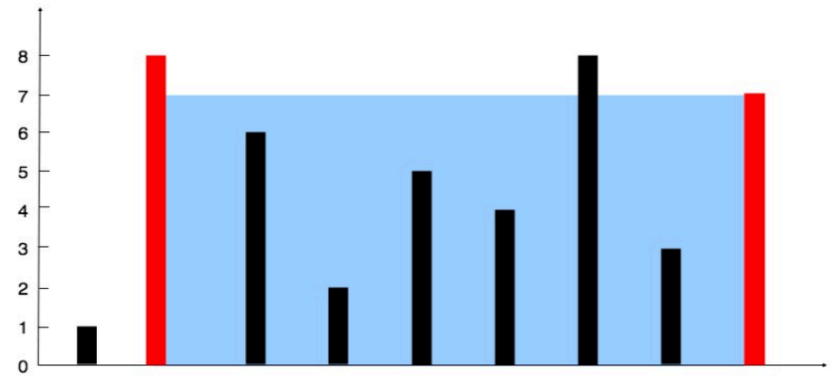
### ✅ Common Patterns:

| Pattern | Description |
|---|---|
| Opposite Ends | One pointer starts at the beginning, one at the end. Move toward each other. |
| Sliding Window | Both pointers move forward. Helps find subarrays or substrings. |
| Fast & Slow | Used in linked lists, cycle detection, etc. |

1. **Sorted Input**:
   - Two pointers are most powerful when the input is sorted — you can then decide to move left or right intelligently.
2. **Window-Based Problems**:
   - If you're scanning for a **subarray, substring, or sublist** with some property, two pointers help define a dynamic window.
3. **Compare Start and End**:
   - If the logic requires comparing elements from both ends (e.g., checking symmetry or matching), two pointers are ideal.

- *So we want to use 2 pointer when we see the the following pattern in question*
    - *1.elements are in increasing order*
    - *2.elements are in decreasing order*
    - *3.increasing and then decreasing*
    - *4.decreasing and then increasing*

*You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).Find two lines that together with the x-axis form a container, such that the container contains the most water.Return the maximum amount of water a container can store.*

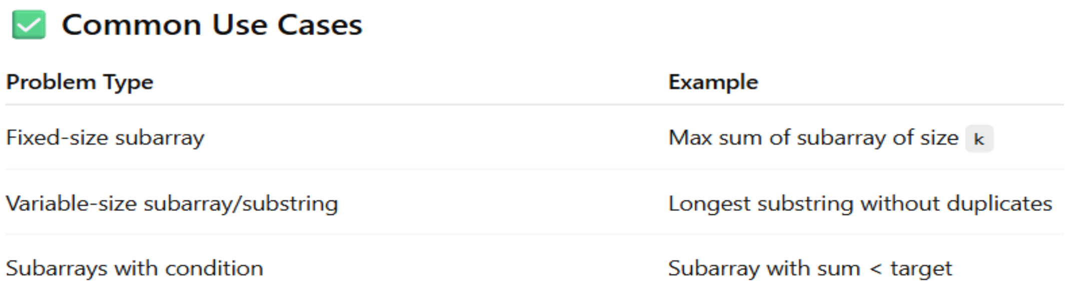


```
Input: height = [1,8,6,2,5,4,8,3,7]
Output: 49
Explanation: The above vertical lines are represented by array
[1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section)
the container can contain is 49.
```

```
def maxArea(self, height: List[int]) -> int:

        s=0

        l=len(height)-1

        a=0

        while s<l:

            w=l-s

            if height[s]>height[l]:

                a=max(a,w*height[l])

                l=l-1

            else:

                a=max(a,w*height[s])

                s=s+1

        return a
```

## *10.Sliding Window Technique:*

- *The sliding window technique is a powerful optimization for solving problems involving subarrays, substrings, or sublists within linear data structures like arrays or strings.*
- ***Basic Idea***
  - *Use two pointers (`left` and `right`) to represent a **window** that slides over the input. Expand or shrink the window to maintain some condition (like size, sum, characters, etc.).*



✅ Common Use Cases

| Problem Type | Example |
|---|---|
| Fixed-size subarray | Max sum of subarray of size `k` |
| Variable-size subarray/substring | Longest substring without duplicates |
| Subarrays with condition | Subarray with sum < target |

- 
- *Example 1: Fixed-size Window (Max Sum of Subarray of Size K).*

```
def max_sum_subarray_k(nums, k):
    max_sum = current_sum = sum(nums[:k])
    for i in range(k, len(nums)):
        current_sum += nums[i] - nums[i - k]
        max_sum = max(max_sum, current_sum)
    return max_sum
print(max_sum_subarray_k([1, 2, 3, 4, 5], 3))  # Output: 12
```

- *Example 2: Variable-size Window (Longest Substring Without Repeating Characters).*

```
def longest_unique_substring(s):
    char_index = {}
    left = max_len = 0
    for right in range(len(s)):
        if s[right] in char_index and char_index[s[right]] >= left:
            left = char_index[s[right]] + 1  # move window start
        char_index[s[right]] = right
        max_len = max(max_len, right - left + 1)
    return max_len
print(longest_unique_substring("abcabcbb"))  # Output: 3
```

- *Example 3: Minimum Size Subarray Sum.*

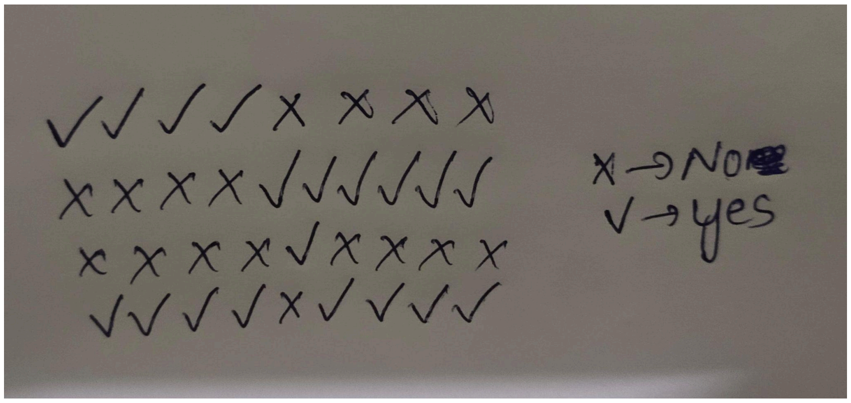
```
def min_subarray_len(target, nums):
    left = 0
    total = 0
    min_len = float('inf')
    for right in range(len(nums)):
        total += nums[right]
        while total >= target:
            min_len = min(min_len, right - left + 1)
            total -= nums[left]
            left += 1
    return 0 if min_len == float('inf') else min_len
print(min_subarray_len(7, [2,3,1,2,4,3]))  # Output: 2
```

***11.Binary Search:****Binary search is an efficient algorithm to find an element in a sorted array or list by repeatedly dividing the search interval in half.*

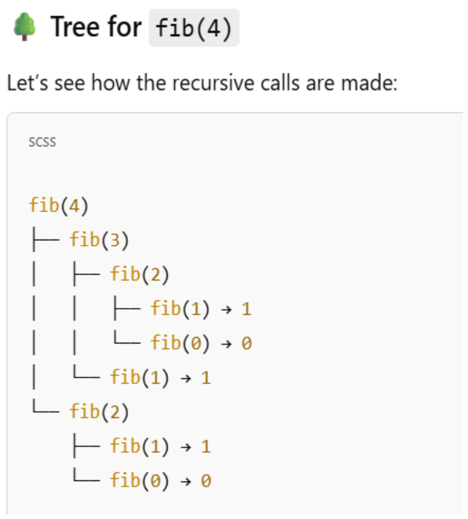
*Binary Search Template:*

```
      def binary_search(arr, target):
     left, right = 0, len(arr) - 1
 while left <= right:
         mid = (left + right) // 2
    if arr[mid] == target:
           return mid  # target found
       elif arr[mid] < target:
           left = mid + 1
       else:
           right = mid - 1
    return -1  # target not found
```

*So we want to use Binary Search when we see the following pattern in question or keywords MinMax or MaxMIn.*



**12.Dynamic Programming (DP):**Dynamic Programming is a powerful technique used to solve problems by breaking them into overlapping subproblems, storing results to avoid redundant work.

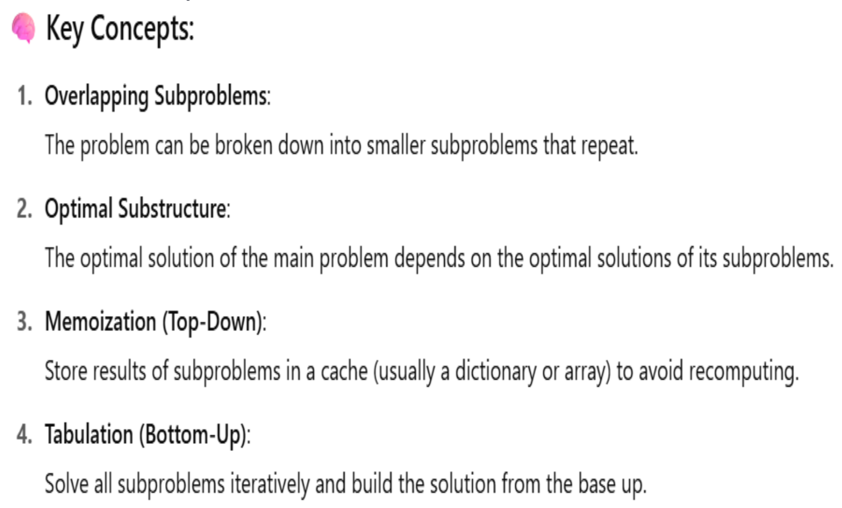


🌳 Tree for `fib(4)`

Let's see how the recursive calls are made:

```scss
fib(4)
├─ fib(3)
│   ├─ fib(2)
│   │   ├─ fib(1) → 1
│   │   └─ fib(0) → 0
│   └─ fib(1) → 1
└─ fib(2)
    ├─ fib(1) → 1
    └─ fib(0) → 0
```

- ***See** above figure fib(2) and fib(1) and other are overlapping subproblems . so when we store result of this overlapping subproblems we can use that results again when that overlapping subproblems came.this is dp.*



🧠 Key Concepts:

1. **Overlapping Subproblems:**
   The problem can be broken down into smaller subproblems that repeat.
2. **Optimal Substructure:**
   The optimal solution of the main problem depends on the optimal solutions of its subproblems.
3. **Memoization (Top-Down):**
   Store results of subproblems in a cache (usually a dictionary or array) to avoid recomputing.
4. **Tabulation (Bottom-Up):**
   Solve all subproblems iteratively and build the solution from the base up.

-

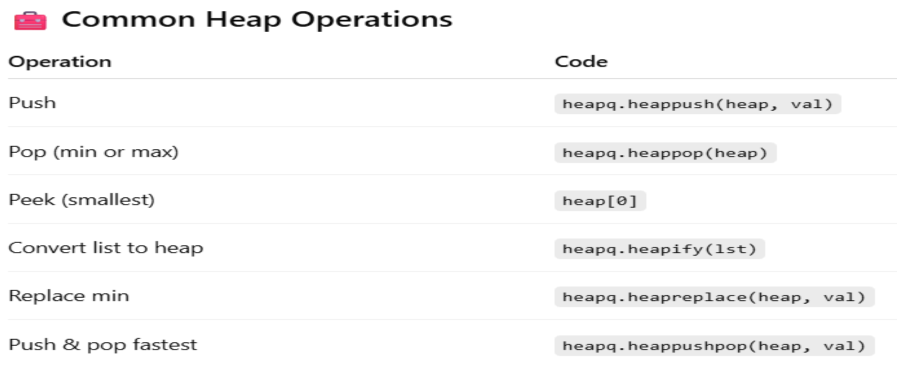
```
def fib(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]
```

- ***strictly increasing subsequence:**Given an integer array nums, return the length of the longest **strictly increasing subsequence**.*

```
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101],
therefore the length is 4.
```

```
def lengthOfLIS(self, nums: List[int]) -> int:
        result=[]
        Dp={}
        Max_Reach=len(nums)
        def LIS(pre,i):
            nonlocal Max_Reach
            if i>=Max_Reach:
                return 0
            if (pre,i) in Dp:
                return Dp[(pre,i)]
            x=y=0
            if pre<nums[i]:
                x=1+LIS(nums[i],i+1)
            y=LIS(pre,i+1)
            Dp[(pre,i)]=max(x,y)
            return Dp[(pre,i)]
        return LIS(-10**6,0)
```

***13. Min Heap & Max Heap.*** *Python provides a min heap implementation out of the box using the* `heapq` *module. For a max heap, you need to invert the values because* `heapq` *only supports min heaps directly.*



🧰 Common Heap Operations

| Operation | Code |
|---|---|
| Push | `heapq.heappush(heap, val)` |
| Pop (min or max) | `heapq.heappop(heap)` |
| Peek (smallest) | `heap[0]` |
| Convert list to heap | `heapq.heapify(lst)` |
| Replace min | `heapq.heapreplace(heap, val)` |
| Push & pop fastest | `heapq.heappushpop(heap, val)` |

```
import heapq

# ----------- MIN HEAP -----------

print("🟢 Min Heap Operations")

min_heap = []

# Push elements

heapq.heappush(min_heap, 5)

heapq.heappush(min_heap, 1)

heapq.heappush(min_heap, 3)

print("Heap after pushes:", min_heap)

# Pop the smallest element

min_val = heapq.heappop(min_heap)

print("Popped min:", min_val)

print("Heap after pop:", min_heap)

# Peek the smallest element

print("Peek min (top):", min_heap[0])

# Push and then pop

print("Pushpop (value=2):", heapq.heappushpop(min_heap, 2))

print("Heap after pushpop:", min_heap)

# Replace root (smallest)

print("Replace min with 4:", heapq.heapreplace(min_heap, 4))

print("Heap after replace:", min_heap)
```

```
# Convert list to heap

nums = [7, 9, 1, 3]

heapq.heapify(nums)

print("Heapified list:", nums)

# ----------- MAX HEAP -----------

print("🔴 Max Heap Operations")

max_heap = []

# Push elements (negated for max heap)

heapq.heappush(max_heap, -5)

heapq.heappush(max_heap, -1)

heapq.heappush(max_heap, -3)

print("Heap after pushes:", [-x for x in max_heap])

# Pop the largest element

max_val = -heapq.heappop(max_heap)

print("Popped max:", max_val)

print("Heap after pop:", [-x for x in max_heap])

# Peek the largest element

print("Peek max (top):", -max_heap[0])

# Push and then pop

print("Pushpop (value=2):", -heapq.heappushpop(max_heap, -2))

print("Heap after pushpop:", [-x for x in max_heap])

# Replace root (largest)

print("Replace max with 4:", -heapq.heapreplace(max_heap, -4))

print("Heap after replace:", [-x for x in max_heap])

# Convert list to max heap

nums = [7, 9, 1, 3]

max_heap = [-x for x in nums]

heapq.heapify(max_heap)

print("Heapified max list:", [-x for x in max_heap])
```

## 14.Dijkstra’s Algorithm – Shortest Path in a Graph

- *Dijkstra’s Algorithm finds the shortest path from a source node to all other nodes in a graph with non-negative edge weights.*

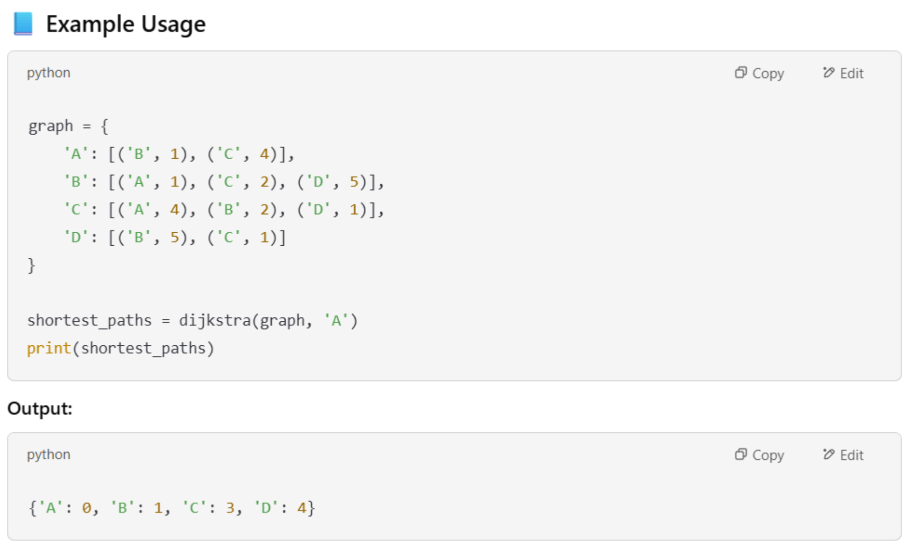
```
import heapq

from collections import defaultdict

def dijkstra(graph, start):

    # graph is a dict: { node: [(neighbor, weight), ...] }

    min_heap = [(0, start)]  # (distance, node)

    distances = {node: float('inf') for node in graph}

    distances[start] = 0

    visited = set()

    while min_heap:

        curr_dist, u = heapq.heappop(min_heap)

        if u in visited:

            continue

        visited.add(u)

        for neighbor, weight in graph[u]:

            if neighbor not in visited:

                new_dist = curr_dist + weight

                if new_dist < distances[neighbor]:

                    distances[neighbor] = new_dist

                    heapq.heappush(min_heap, (new_dist, neighbor))

    return distances
```

### ⏱ Time Complexity

- Using min-heap: **O((V + E) log V)** where:
    - V = number of vertices
    - E = number of edges
    -

## 🧠 When to Use Dijkstra's Algorithm

- Graph has **non-negative weights**
- You want **shortest paths** from a single source
- Works on both **undirected** and **directed** graphs

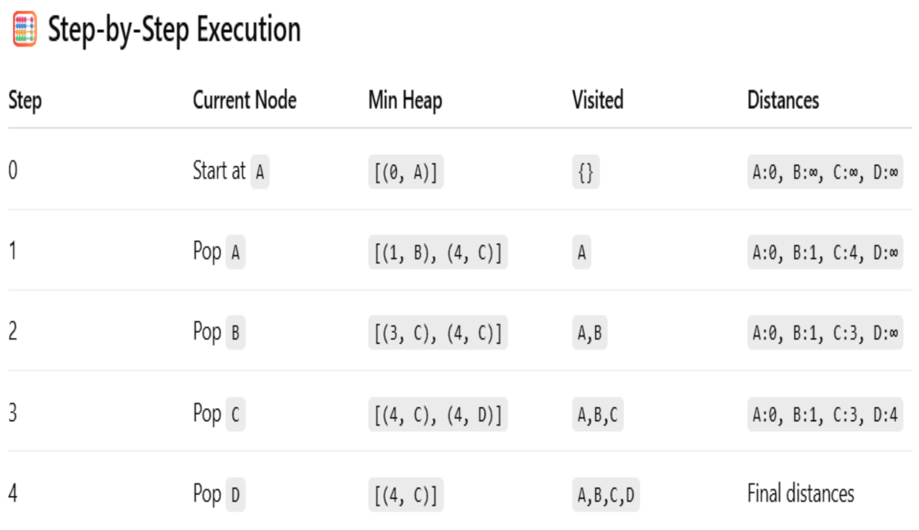


### 📘 Example Usage

```python
graph = {
    'A': [('B', 1), ('C', 4)],
    'B': [('A', 1), ('C', 2), ('D', 5)],
    'C': [('A', 4), ('B', 2), ('D', 1)],
    'D': [('B', 5), ('C', 1)]
}

shortest_paths = dijkstra(graph, 'A')
print(shortest_paths)
```

**Output:**

```python
{'A': 0, 'B': 1, 'C': 3, 'D': 4}
```



### 🧮 Step-by-Step Execution

| Step | Current Node | Min Heap | Visited | Distances |
|---|---|---|---|---|
| 0 | Start at `A` | `[(0, A)]` | `{}` | `A:0, B:∞, C:∞, D:∞` |
| 1 | Pop `A` | `[(1, B), (4, C)]` | `A` | `A:0, B:1, C:4, D:∞` |
| 2 | Pop `B` | `[(3, C), (4, C)]` | `A,B` | `A:0, B:1, C:3, D:∞` |
| 3 | Pop `C` | `[(4, C), (4, D)]` | `A,B,C` | `A:0, B:1, C:3, D:4` |
| 4 | Pop `D` | `[(4, C)]` | `A,B,C,D` | Final distances |

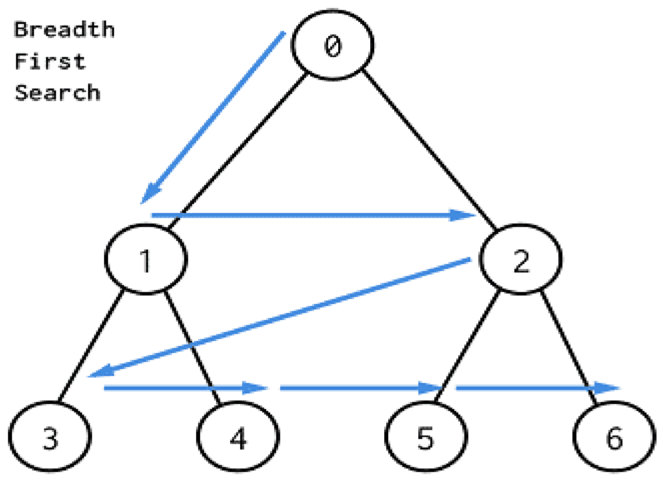
→ So It Helps to Find Shortest Path from One Node to Every Other Node.

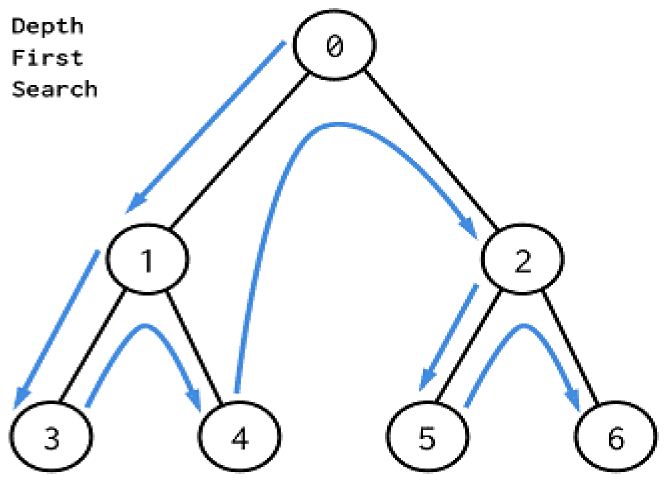
## 15.Graph(BSF):

```
from collections import deque

def bfs_shortest_path_length(graph, start, goal):
    visited = set()
    queue = deque([start])
    path_length = 0
    visited.add(start)
    while queue:
        path_length += 1
        for _ in range(len(queue)):
            node = queue.popleft()
            for neighbor in graph[node]:
                if neighbor not in visited:
                    if neighbor == goal:
                        return path_length
                    visited.add(neighbor)
                    queue.append(neighbor)
    return -1  # If no path is found
# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
start = 'A'
goal = 'F'
shortest_path_length = bfs_shortest_path_length(graph, start, goal)
print("Shortest path length:", shortest_path_length)
```

## *16.Graph(DFS):*

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end='  ')
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
# Run DFS starting from node 'A'
dfs(graph, 'A')
```


Breadth
First
Search
0
1
2
3
4
5
6


Depth
First
Search
0
1
2
3
4
5
6