The three main notations are:

1. **Big-O Notation (O)**:
   - **Definition**: $f(n)=O(g(n))$ if there exist constants $c>0$ and $n_0 \geq 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
   - **Meaning**: Provides an **upper bound** on the growth rate of $f(n)$. It describes the worst-case scenario for an algorithm's resource usage.
   - **Example**: If an algorithm's running time is $3n^2+5n+10$, it is $O(n^2)$, as n^2 dominates for large n.
2. **Omega Notation (Ω)**:
   - **Definition**: $f(n)=\Omega(g(n))$ if there exist constants $c>0$ and $n_0 \geq 0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.
   - **Meaning**: Provides a **lower bound** on the growth rate, describing the best-case scenario or minimum resource usage.
   - **Example**: The same algorithm with running time $3n^2+5n+10$ is $\Omega(n^2)$, as it grows at least as fast as n^2.
3. **Theta Notation (Θ)**:
   - **Definition**: $f(n)=\Theta(g(n))$ if $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$, i.e., there exist constants $c_1, c_2>0$ and $n_0 \geq 0$ such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.
   - **Meaning**: Provides a **tight bound**, indicating that the algorithm's resource usage grows exactly at the rate of $g(n)$.
   - **Example**: The algorithm with running time $3n^2+5n+10$ is $\Theta(n^2)$, as it is both $O(n^2)$ and $\Omega(n^2)$.
4. **Little-o Notation (o)**:
   - **Definition**: $f(n)=o(g(n))$ if for every constant $c>0$, there exists an $n_0 \geq 0$ such that $f(n)<c \cdot g(n)$ for all $n \geq n_0$.
   - **Meaning**: A **strict upper bound**, where $f(n)$ grows strictly slower than $g(n)$.
   - **Example**: $5n=o(n^2)$, as $5n$ grows slower than n^2.
5. **Little-omega Notation (ω)**:
   - **Definition**: $f(n)=\omega(g(n))$ if for every constant $c>0$, there exists an $n_0 \geq 0$ such that $f(n)>c \cdot g(n)$ for all $n \geq n_0$.
   - **Meaning**: A **strict lower bound**, where $f(n)$ grows strictly faster than $g(n)$.
   - **Example**: $n^2=\omega(n)$, as n^2 grows faster than n.

Hierarchy of Complexity Classes

- **Growth Rate Order**: $O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset \cdots \subset O(2^n) \subset O(n!)$.
- **Comparison**: For large n, logarithmic functions grow much slower than linear, which are slower than polynomial, and so on. For example, $\log n < n < n^2 < 2^n$.

A queue is a linear data structure that stores elements in a sequence and processes them in the First-In, First-Out (FIFO) order — meaning the first element added is the first one to be removed, just like people standing in a real-world queue.

Definition

A **Queue** is a **linear abstract data type** that stores a collection of elements in which insertion of elements is allowed **only at one end** (called the **rear**) and deletion of elements is allowed **only at the other end** (called the **front**), following the **First In, First Out (FIFO)** principle.

Formal ADT Definition

Queue ADT is described by:

- **Objects**:
  An ordered list of elements of any type.
- **Operations**:
  1. **Create(Q)** → Creates an empty queue `Q`.
  2. **IsEmpty(Q)** → Returns `true` if `Q` has no elements, otherwise `false`.
  3. **IsFull(Q)** → Returns `true` if `Q` cannot accept more elements (in a bounded queue).
  4. **Enqueue(Q, x)** → Inserts element `x` at the **rear** of `Q`.
  5. **Dequeue(Q)** → Removes and returns the element from the **front** of `Q`.
  6. **Peek(Q)** or **Front(Q)** → Returns the element at the **front** without removing it.
  7. **Size(Q)** → Returns the number of elements currently in `Q`.

Key Features

1. **FIFO Principle** – The element inserted earliest is removed first.
2. **Two Ends** –
   - **Front**: where elements are removed (dequeued).
   - **Rear (Back)**: where elements are added (enqueued).
3. **Operations are restricted** – You can't insert in the middle or remove from the middle directly; all insertions happen at the rear, and removals happen from the front.

Basic Operations

Operation	Description
Enqueue(x)	Add element `x` at the rear of the queue.
Dequeue()	Remove the element from the front.
Peek / Front	Get the element at the front without removing it.
IsEmpty()	Check if the queue has no elements.
IsFull()	(For fixed-size queues) Check if the queue is at maximum capacity.

Types of Queues

1. **Simple Queue (Linear Queue)** – Basic FIFO behavior.
2. **Circular Queue** – Rear wraps around to the front when space is free, avoiding wasted space.
3. **Deque (Double-Ended Queue)** – Insertions and deletions can happen at both ends.
4. **Priority Queue** – Elements are dequeued based on priority rather than arrival time.

Real-World Examples

- Ticket booking systems.
- Printer job scheduling.
- Call center request handling.
- CPU task scheduling.

Queue Representation

We'll store:

- **queue[]**: an array to hold elements
- **front**: index of the first element (initially 0)
- **rear**: index of the last element (initially -1)
- **maxSize**: maximum capacity of the queue

```
procedure ENQUEUE(queue, element):
   if rear == maxSize - 1:
      print "Queue Overflow"
      return
   rear ← rear + 1
```

```
    queue[rear] ← element
    print element, "inserted into queue"

procedure ENQUEUE(queue, element):
    if (front == 0 AND rear == maxSize - 1) OR (rear + 1) % maxSize == front:
        print "Queue Overflow"
        return

    if front == -1:        // First element
        front ← 0

    rear ← (rear + 1) % maxSize
    queue[rear] ← element
    print element, "inserted into queue"

procedure DEQUEUE(queue):
    if front > rear:
        print "Queue Underflow"
        return
    element ← queue[front]
    front ← front + 1
    print element, "removed from queue"
procedure DEQUEUE(queue):
    if front == -1:
        print "Queue Underflow"
        return

    element ← queue[front]

    if front == rear:      // Only one element
        front ← -1
        rear ← -1
    else:
        front ← (front + 1) % maxSize

    print element, "removed from queue"

procedure PEEK(queue):
    if front > rear:
        print "Queue is empty"
        return
    print "Front element is", queue[front]

procedure PEEK(queue):
    if front == -1:
        print "Queue is empty"
        return
    print "Front element is", queue[front]
```

```
procedure ISEMPTY():
    if front > rear:
        return TRUE
    else:
        return FALSE
```

```
procedure ISEMPTY():
    if front == -1:
        return TRUE
    else:
        return FALSE
```

Simple (Linear) Queue

- **Order:** FIFO (First In, First Out)
- **Operations:** Enqueue at rear, dequeue at front
- **Limitation:** Once `rear` reaches the end, no further insertions even if there is space at the front (unless shifted or made circular).
- **Example use:** Printing tasks queue

Circular Queue

- **Order:** FIFO
- **Structure:** Uses modulo arithmetic to "wrap around" the array when it reaches the end.
- **Advantage:** Efficient space utilization; avoids shifting elements.
- **Example use:** CPU task scheduling in round-robin

Double-Ended Queue (Deque)

- **Order:** Can insert and remove elements at **both ends**.
- **Variants:**
    - **Input-restricted deque:** Insertion at only one end, deletion at both ends
    - **Output-restricted deque:** Deletion at only one end, insertion at both ends
- **Example use:** Browser history navigation (back and forward)

Priority Queue

- **Order:** Elements dequeued according to **priority**, not arrival order.
- **Variants:**
    - **Ascending priority queue:** smallest value removed first

    - **Descending priority queue:** largest value removed first
- **Example use:** Job scheduling where urgent tasks are handled first

Double-Ended Priority Queue

- **Order:** Supports removal of both the highest and lowest priority elements efficiently.
- **Example use:** Event simulation with both earliest and latest event removal

Circular Deque

- Combination of **deque** and **circular queue** features for efficient memory usage with double-ended access.

Pointer implementation:

Structure:

```
Node:
    data
    next
```

We maintain:

- `front` → pointer to the first node
- `rear` → pointer to the last node

Create(Q):

front ← NULL

rear ← NULL

IsEmpty(Q):

if front = NULL then

return true

else

return false

Enqueue(Q, x):

newNode ← allocate memory for Node

newNode.data ← x

newNode.next ← NULL

if rear = NULL then // Queue is empty

front ← newNode

rear ← newNode

else

rear.next ← newNode

rear ← newNode

Dequeue(Q):

if IsEmpty(Q) then

print "Queue Underflow"

else

temp ← front

item ← temp.data

front ← front.next

if front = NULL then // Queue became empty

rear ← NULL

free temp

return item

Peek(Q):

if IsEmpty(Q) then

print "Queue is empty"

else

return front.data

Size(Q):

return count

Circular Queue Implementation (Linked List)

Structure:

```
Node:
    data
    next
```

We maintain:

- `front` → pointer to the first node
- `rear` → pointer to the last node
  (`rear.next` always points to `front` when the queue is non-empty.)

Create(Q):

front ← NULL

rear ← NULL

IsEmpty(Q):

if front = NULL then

return true

else

return false

Enqueue(Q, x):

newNode ← allocate memory for Node

newNode.data ← x

if front = NULL then // Queue is empty

front ← newNode

rear ← newNode

rear.next ← front // Circular link

else

newNode.next ← front // Point to first node

rear.next ← newNode // Link old rear to new node

rear ← newNode // Update rear

Dequeue(Q):

if IsEmpty(Q) then

print "Queue Underflow"

else if front = rear then // Only one element

item ← front.data

free front

front ← NULL

```
        rear ← NULL
        return item
    else
        item ← front.data
        temp ← front
        front ← front.next
        rear.next ← front            // Maintain circular link
        free temp
        return item
```

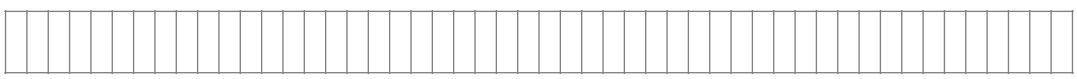
Peek(Q):

```
    if IsEmpty(Q) then
        print "Queue is empty"
    else
        return front.data
```

Optional

Traverse(Q):

```
    if IsEmpty(Q) then
        print "Queue is empty"
    else
        temp ← front
        do
            print temp.data
            temp ← temp.next
        while temp ≠ front
```



Stack ADT

Definition

A **Stack** is a **linear abstract data type** in which insertion and deletion of elements are allowed **only at one end** called the **top**. It follows the **Last In, First Out (LIFO)** principle — the most recently inserted element is the first one to be removed.

Formal ADT Definition

- **Objects**:
  An ordered list of elements of any type, with a distinguished end called the **top**.
- **Operations**:
  1. **Create(S)** → Creates an empty stack `S`.
  2. **IsEmpty(S)** → Returns `true` if `S` has no elements, otherwise `false`.
  3. **IsFull(S)** → Returns `true` if `S` cannot accept more elements (in a bounded stack).
  4. **Push(S, x)** → Inserts element `x` at the **top** of `S`.
  5. **Pop(S)** → Removes and returns the element from the **top** of `S`.
  6. **Peek(S)** or **Top(S)** → Returns the element at the **top** without removing it.
  7. **Size(S)** → Returns the number of elements currently in `S`.

Characteristics

- **Access restriction**: Only the top element can be accessed directly.
- **Order principle**: LIFO (Last In, First Out).
- **Usage**: Function call management, undo features in editors, expression evaluation, back-tracking algorithms, syntax parsing, etc.

Create(S):

```
top ← -1
```

IsEmpty(S):

```
if top = -1 then
    return true
else
    return false
```

IsFull(S):

```
if top = MAX_SIZE - 1 then
    return true
else
    return false
```

Push(S, x):

```
    if IsFull(S) then
        print "Stack Overflow"
    else
        top ← top + 1
        S[top] ← x
```

Pop(S):

```
    if IsEmpty(S) then
        print "Stack Underflow"
    else
        item ← S[top]
        top ← top - 1
        return item
```

Peek(S):

```
    if IsEmpty(S) then
        print "Stack is empty"
    else
        return S[top]
```

Size(S):

```
    return top + 1
```

Question: Can a Stack be implemented on an array in a circular fashion?

A Binary Tree is a non-linear hierarchical data structure in which each node can have at most two children — referred to as the left child and the right child.

Formal Definition

A binary tree is either:

1. **Empty** (contains no nodes), or
2. Consists of a **root node** and two **disjoint binary trees** called the **left subtree** and the **right subtree**.

Key Properties

- The **maximum number of nodes** at level i (where root is level 0) is 2^i.
- A binary tree of height h has at most $2^{h+1} - 1$ nodes.
- A binary tree of n nodes has at least $\lceil \log_2(n+1) \rceil$ levels.

Types of Binary Trees

1. **Full Binary Tree** → Every node has either 0 or 2 children.

2. **Complete Binary Tree** → All levels are completely filled except possibly the last, which is filled from left to right.
3. **Perfect Binary Tree** → All internal nodes have 2 children and all leaf nodes are at the same level.
4. **Degenerate (or Skewed) Tree** → Each parent node has only one child (resembles a linked list).
5. **Balanced Binary Tree** → The height difference between left and right subtrees of every node is at most 1.

Structure Node:

```
data
left  // pointer to left child
right // pointer to right child
```

Function Inorder(node):

```
If node == NULL:
    return
Inorder(node.left)
Print(node.data)
Inorder(node.right)
```

Function Preorder(node):

```
If node == NULL:
    return
Print(node.data)
Preorder(node.left)
Preorder(node.right)
```

Function Postorder(node):

```
If node == NULL:
    return
Postorder(node.left)
Postorder(node.right)
Print(node.data)
```

Function LevelOrder(root):

```
If root == NULL:
    return
Create an empty Queue Q
Enqueue(Q, root)
While Q is not empty:
    node ← Dequeue(Q)
    Print(node.data)
    If node.left != NULL:
        Enqueue(Q, node.left)
    If node.right != NULL:
        Enqueue(Q, node.right)
```

A **Binary Search Tree (BST)** is a special type of **binary tree** with an additional ordering property:

Definition

A **Binary Search Tree** is a binary tree in which:

1. Each node contains a **key (or value)**.
2. For every node:
    - **Left Subtree Property:** All nodes in the left subtree have values **less than** the node's value.
    - **Right Subtree Property:** All nodes in the right subtree have values **greater than** the node's value.
3. Both the left and right subtrees are themselves **binary search trees** (recursively).

Properties of BST

- **Uniqueness:** Typically, no duplicate keys are allowed.
- **Inorder Traversal:** Produces values in **sorted ascending order**.
- **Search Efficiency:** Average case `O(log n)`, worst case `O(n)` (when tree becomes skewed).
- **Height-balanced versions** (like AVL, Red-Black Trees) improve efficiency.

ADT (Abstract Data Type) for BST

A BST supports the following standard operations:

- **Insert(x):** Insert an element `x` in the tree.
- **Delete(x):** Remove element `x` if it exists.
- **Search(x):** Find whether element `x` exists.
- **FindMin():** Return the smallest element.
- **FindMax():** Return the largest element.
- **Traverse(order):** Visit nodes in a chosen order (inorder, preorder, postorder, level-order).

Search(Node root, Key x):

```
if root == NULL:
    return "Not Found"
if x == root.key:
    return "Found"
else if x < root.key:
```

```
    return Search(root.left, x)
  else:
    return Search(root.right, x)
```

Insert(Node root, Key x):

```
  if root == NULL:
    root = new Node(x)
    return root

  if x < root.key:
    root.left = Insert(root.left, x)

  else if x > root.key:
    root.right = Insert(root.right, x)

  return root
```

Deleting a node has three cases:

1. **Leaf Node** → just remove it.
2. **One Child** → replace the node with its child.
3. **Two Children** → replace with inorder successor (smallest in right subtree).

Delete(Node root, Key x):

```
  if root == NULL:
    return root
  if x < root.key:
    root.left = Delete(root.left, x)

  else if x > root.key:
    root.right = Delete(root.right, x)

  else:
    # Case 1 & 2: Node has 0 or 1 child
    if root.left == NULL:
      return root.right
    else if root.right == NULL:
      return root.left

    # Case 3: Node has 2 children
    temp = FindMin(root.right)
    root.key = temp.key
    root.right = Delete(root.right, temp.key)
  return root
```

FindMin(Node root):

```
  while root.left != NULL:
     root = root.left
  return root
```

A **Binary Tree** can be reconstructed if you are given:

- **Inorder + Preorder traversal** OR
- **Inorder + Postorder traversal**

But since **BST** has a special property (left < root < right), you can even reconstruct it using just **Preorder** or just **Postorder**.

Inorder traversal → gives **left-root-right** sequence.

- **Preorder traversal** → gives **root-left-right** sequence.

Algorithm:

1. First element of Preorder = **root**.
2. Locate root in Inorder sequence → divides tree into **Left Inorder** & **Right Inorder**.
3. Recurse on left and right subtrees with corresponding Preorder parts.

In a normal binary tree, many pointers (left or right) are `NULL`.
A **Threaded Binary Tree (TBT)** uses these NULL pointers to store special **threads** that point to the **inorder predecessor** or **inorder successor** of a node.

This makes **inorder traversal faster** and avoids recursion or stack.

Types:

- **Single Threaded:** Each NULL pointer is replaced by either:
  - Left thread → inorder predecessor
  - OR Right thread → inorder successor
- **Double Threaded:** Both left and right NULL pointers are replaced with threads.

```
Node {
   data
   left, right   // child or thread
   lTag, rTag    // 0 = link to child, 1 = thread
}
```

- `lTag = 0` → left points to child
- `lTag = 1` → left is a thread (predecessor)
- `rTag = 0` → right points to child
- `rTag = 1` → right is a thread (successor)

INORDER-TBT(root):

```
node ← LEFTMOST(root)
while node ≠ NULL:
    print(node.data)
    if node.rTag = 1:
        node ← node.right     // follow thread
    else:
        node ← LEFTMOST(node.right)
```

LEFTMOST(node):

```
if node = NULL: return NULL
while node.lTag = 0:     // follow left children
    node ← node.left
return node
```

INSERT_RIGHT(p, newNode):

```
newNode.right ← p.right
newNode.rTag ← p.rTag
newNode.left ← p
newNode.lTag ← 1
p.right ← newNode
p.rTag ← 0
if newNode.rTag = 0:
    LEFTMOST(newNode.right).left ← newNode
```

SEARCH(root, key):

```
node ← root
while node ≠ NULL:
    if key = node.data: return node
    else if key < node.data AND node.lTag = 0:
        node ← node.left
    else if key > node.data AND node.rTag = 0:
        node ← node.right
    else:
```

```
        break
    return NULL   // not found
```

Deletion (outline only, since it's tricky)

- If node has 0 children → adjust predecessor & successor threads.
- If node has 1 child → connect child to parent, fix threads.
- If node has 2 children → replace with inorder successor, then delete.

A **Heap** is a special type of **complete binary tree** (all levels completely filled except possibly the last, filled left to right) that satisfies the **heap property**:

- **Max-Heap:** For every node `i`,

parent(i)≥child(i)

(root contains the maximum element).

- **Min-Heap:** For every node `i`,

parent(i)≤child(i)

(root contains the minimum element).

Heaps are usually implemented as **arrays** for efficiency.

- For node at index `i`:
  - **Left child** → `2*i` (or `2*i+1` in 0-based index)
  - **Right child** → `2*i+1` (or `2*i+2` in 0-based index)
  - **Parent** → `i/2` (or `(i-1)/2` in 0-based index)

Insert (Push into Heap)

Steps:

1. Insert new element at the end (last position).
2. Perform **Heapify-Up (percolate up)** to restore heap property.

```
Insert(heap, value):
    heap.size = heap.size + 1
    i = heap.size
```

```
    heap[i] = value

    while i > 1 and heap[Parent(i)] < heap[i]:
        swap(heap[i], heap[Parent(i)])
        i = Parent(i)
```

Delete Root (Extract-Max / Extract-Min)

Steps:

1. Root (max/min) is removed.
2. Replace root with last element.
3. Perform **Heapify-Down** to restore heap property.

ExtractMax(heap):

```
    if heap.size < 1:
        return ERROR

    maxValue = heap[1]
    heap[1] = heap[heap.size]
    heap.size = heap.size - 1
    Heapify(heap, 1)

    return maxValue
```

Heapify(heap, i):

```
    left = LeftChild(i)
    right = RightChild(i)
    largest = i

    if left <= heap.size and heap[left] > heap[largest]:
        largest = left
    if right <= heap.size and heap[right] > heap[largest]:
        largest = right

    if largest != i:
        swap(heap[i], heap[largest])
        Heapify(heap, largest)
```

BuildHeap(heap):

```
    for i = floor(heap.size/2) downto 1:
        Heapify(heap, i)
```

HeapSort(array):

```
BuildHeap(array)
for i = array.size downto 2:
    swap(array[1], array[i])
    array.size = array.size - 1
    Heapify(array, 1)
```

Time Complexity of Heap Operations

1. Insert

- Place new element at end, then **bubble up** (percolate up).
- Height of heap = $O(\log n)$ (since it's a complete binary tree).
  Time: $O(\log n)$

2. Extract Root (Extract-Max / Extract-Min)

- Remove root, replace with last element, then **heapify-down**.
- Again, traversal at most $O(\log n)$.

  👉 **Time:** $O(\log n)$

3. Heapify (at index i)

- Fix heap property starting at node i and moving downward.
- Worst case = go from root to leaf → $O(\log n)$.
  Time: $O(\log n)$

4. Build Heap

- Call `Heapify` from $n/2$ down to 1.
- Surprisingly, not $O(n \log n)$ but $O(n)$.
  (Because most heapify calls are on small subtrees, amortizing the cost).
  Time: $O(n)$

5. Heap Sort

- BuildHeap: $O(n)$
- Then extract n times, each $O(\log n)$ → $O(n \log n)$
  Total Time: $O(n \log n)$

6. Peek (Get Max / Get Min without deleting)

- Just return root.
  Time: $O(1)$

Space Complexity: $O(1)$ (in-place if using array).

A Height Balanced Binary Tree is a binary tree in which the balance factor of every node is:

Factor subtree) subtree)Balance Factor=Height(left subtree)−Height(right subtree)

- For AVL trees: balance factor ∈ {–1, 0, +1}
- This ensures that the tree's height = $O(\log n)$, where n = number of nodes.

2. Insertion

- Insert node like in a BST.
- Update heights.
- Check balance factor.
- If unbalanced, perform **rotations** (LL, RR, LR, RL).

```
INSERT(root, key):
   if root == NULL:
      return NEWNODE(key)

   if key < root.key:
      root.left = INSERT(root.left, key)
   else if key > root.key:
      root.right = INSERT(root.right, key)
   else:
      return root   // no duplicates

   UPDATE_HEIGHT(root)

   balance = BF(root)   // Balance Factor

   // Case 1: Left Left
   if balance > 1 and key < root.left.key:
      return ROTATE_RIGHT(root)

   // Case 2: Right Right
   if balance < -1 and key > root.right.key:
      return ROTATE_LEFT(root)
```

```
    // Case 3: Left Right
    if balance > 1 and key > root.left.key:
        root.left = ROTATE_LEFT(root.left)
        return ROTATE_RIGHT(root)

    // Case 4: Right Left
    if balance < -1 and key < root.right.key:
        root.right = ROTATE_RIGHT(root.right)
        return ROTATE_LEFT(root)

    return root
```

ROTATE_RIGHT(y):

```
    x = y.left
    T2 = x.right

    x.right = y
    y.left = T2

    UPDATE_HEIGHT(y)
    UPDATE_HEIGHT(x)

    return x
```

ROTATE_LEFT(x):

```
    y = x.right
    T2 = y.left

    y.left = x
    x.right = T2

    UPDATE_HEIGHT(x)
    UPDATE_HEIGHT(y)

    return y
```

3. Deletion

- Delete as in BST.
- Update heights.
- Rebalance using rotations.

```
DELETE(root, key):
    if root == NULL:
        return root

    if key < root.key:
        root.left = DELETE(root.left, key)
    else if key > root.key:
        root.right = DELETE(root.right, key)
    else:
        if root.left == NULL:
            return root.right
        else if root.right == NULL:
            return root.left
        else:
            temp = MIN_VALUE_NODE(root.right)
            root.key = temp.key
            root.right = DELETE(root.right, temp.key)

    UPDATE_HEIGHT(root)
    balance = BF(root)

    // Rebalance Cases same as in insertion
    if balance > 1 and BF(root.left) >= 0:
        return ROTATE_RIGHT(root)
    if balance > 1 and BF(root.left) < 0:
        root.left = ROTATE_LEFT(root.left)
        return ROTATE_RIGHT(root)
    if balance < -1 and BF(root.right) <= 0:
        return ROTATE_LEFT(root)
    if balance < -1 and BF(root.right) > 0:
        root.right = ROTATE_RIGHT(root.right)
        return ROTATE_LEFT(root)

    return root
```

Time Complexity: $O(\log n)$

Definition of Red-Black Tree

A **Red-Black Tree** is a type of **self-balancing binary search tree (BST)** where each node has an extra attribute: **color (red or black)**.

It ensures that the tree remains approximately balanced after insertions and deletions, keeping the height $O(\log n)$.

Properties of a Red-Black Tree

1. Every node is either **red** or **black**.
2. The **root** is always **black**.
3. All **leaves (NIL nodes)** are considered **black**.
4. If a node is **red**, both its children must be **black** (no two consecutive red nodes).
5. **Every path from a node to its descendant NIL nodes has the same number of black nodes (black-height).**

RB-SEARCH(root, key):

```
if root == NIL or key == root.key:
    return root
if key < root.key:
    return RB-SEARCH(root.left, key)
else:
    return RB-SEARCH(root.right, key)
```

Time Complexity: O(log n)

2. **Insertion(x)**

Steps:

1. Insert new node z as in BST (color = RED).
2. Fix violations of RBT properties using **rotations** and **recoloring**.

```
RB-INSERT(root, z):
  BST-INSERT(root, z)       // insert like normal BST
  z.color = RED
  while z.parent.color == RED:
    if z.parent == z.parent.parent.left:
      y = z.parent.parent.right   // uncle
      if y.color == RED:          // Case 1: uncle is red
        z.parent.color = BLACK
        y.color = BLACK
        z.parent.parent.color = RED
        z = z.parent.parent
      else:
        if z == z.parent.right: // Case 2: uncle black + z right child
          z = z.parent
```

```
            LEFT-ROTATE(root, z)
        // Case 3: uncle black + z left child
        z.parent.color = BLACK
        z.parent.parent.color = RED
        RIGHT-ROTATE(root, z.parent.parent)
    else:   // same but mirror for right child
        ...
root.color = BLACK
```
Time Complexity: O(log n)

Definition

A **Linked List** is a linear data structure where elements (called **nodes**) are stored at non-contiguous memory locations.
Each node contains:

- **Data** (value)
- **Pointer** (reference to the next node)

Unlike arrays, linked lists allow **dynamic memory allocation** and **efficient insertion/deletion**, but random access is not possible.

Abstract Data Type (ADT)

We define a **Linked List ADT** as follows:

- **Objects:** A finite sequence of nodes where each node contains `(data, next)`.
- **Functions:**
  - `CREATE()` → create an empty list.
  - `INSERT(L, pos, x)` → insert element `x` at position `pos`.
  - `DELETE(L, pos)` → delete element at position `pos`.
  - `SEARCH(L, x)` → return position of `x` in list (if exists).
  - `TRAVERSE(L)` → visit all nodes in sequence.
  - `IS_EMPTY(L)` → check if list is empty.
  - `LENGTH(L)` → return number of elements.

Structure Node:

```
data
next
```

CREATE():
head ← NIL
return head
Complexity: $O(1)$

INSERT_BEGIN(head, x):
newNode ← allocate Node
newNode.data ← x
newNode.next ← head
head ← newNode
return head
Complexity: $O(1)$

INSERT_END(head, x):
newNode ← allocate Node
newNode.data ← x
newNode.next ← NIL
if head == NIL:
head ← newNode
return head
temp ← head
while temp.next ≠ NIL:
temp ← temp.next
temp.next ← newNode
return head

Complexity: $O(n)$

InsertAfter(node, x):
if node = NULL:
print "Invalid position"
return
newNode ← Node(x)
newNode.next ← node.next
node.next ← newNode

InsertAfter: $O(1)$

DELETE_BEGIN(head):
if head == NIL:
print "Underflow"
return NIL
temp ← head

```
    head ← head.next
    free(temp)
    return head
```
Complexity: $O(1)$

```
DELETE_END(head):
    if head == NIL:
        print "Underflow"
        return NIL
    if head.next == NIL:
        free(head)
        return NIL
    temp ← head
    while temp.next.next ≠ NIL:
        temp ← temp.next
    free(temp.next)
    temp.next ← NIL
    return head
```
Complexity: $O(n)$

```
SEARCH(head, x):
    pos ← 1
    temp ← head
    while temp ≠ NIL:
        if temp.data == x:
            return pos
        temp ← temp.next
        pos ← pos + 1
    return -1   // not found
```

Complexity: $O(n)$

```
TRAVERSE(head):
    temp ← head
    while temp ≠ NIL:
        print(temp.data)
        temp ← temp.next
```
Complexity: $O(n)$
```
LENGTH(head):
    count ← 0
    temp ← head
    while temp ≠ NIL:
        count ← count + 1
        temp ← temp.next
    return count
```
Complexity: $O(n)$

Doubly Linked List

Definition

A **doubly linked list** is a linear data structure where each node has:

- **Data** – stores the element.
- **Prev** – pointer to the previous node.

- **Next** – pointer to the next node.

This allows traversal in **both forward and backward** directions.

ADT for DLL

- **Objects:** A sequence of nodes, each containing `(data, prev, next)`.
- **Operations:**
    1. `InsertAtBeginning(x)`
    2. `InsertAtEnd(x)`
    3. `InsertAfter(p, x)`
    4. `Delete(x)`
    5. `Search(x)`
    6. `TraverseForward()`
    7. `TraverseBackward()`
    8. **Node Structure**

```
Node {
    data
    prev
    next
}
```

InsertAtBeginning(head, x):

```
newNode ← Node(x)
newNode.prev ← NULL
newNode.next ← head
if head ≠ NULL:
    head.prev ← newNode
head ← newNode
return head
```

InsertAtEnd(head, x):

```
newNode ← Node(x)
newNode.next ← NULL
if head = NULL:
    newNode.prev ← NULL
    head ← newNode
    return head
temp ← head
while temp.next ≠ NULL:
    temp ← temp.next
temp.next ← newNode
newNode.prev ← temp
return head
```

InsertAfter(node, x):

```
if node = NULL:
    print "Invalid position"
    return
newNode ← Node(x)
newNode.next ← node.next
newNode.prev ← node
if node.next ≠ NULL:
    node.next.prev ← newNode
node.next ← newNode
```

Delete(head, x):

```
temp ← head
while temp ≠ NULL and temp.data ≠ x:
    temp ← temp.next
if temp = NULL:
    print "Element not found"
    return head
if temp.prev ≠ NULL:
    temp.prev.next ← temp.next
else:
    head ← temp.next   // deleting first node
if temp.next ≠ NULL:
    temp.next.prev ← temp.prev
return head
```

Search(head, x):

```
temp ← head
while temp ≠ NULL:
    if temp.data = x:
        return true
    temp ← temp.next
return false
```

TraverseForward(head):

```
temp ← head
while temp ≠ NULL:
    print temp.data
    temp ← temp.next
```

TraverseBackward(tail):

```
temp ← tail
while temp ≠ NULL:
    print temp.data
    temp ← temp.prev
```

Complexity Analysis

- **InsertAtBeginning:** $O(1)$

- **InsertAtEnd:** $O(n)$ ($O(1)$ if a tail pointer is maintained)
- **InsertAfter:** $O(1)$
- **Delete:** $O(n)$ (search needed)
- **Search:** $O(n)$
- **Traverse:** $O(n)$

A **circular singly linked list** is like a singly linked list except that the **last node's next pointer points back to the head**, forming a circle.

- Can be traversed starting from any node.
- Useful for applications like **round-robin scheduling**.

ADT for CSLL

- **Objects:** Sequence of nodes `(data, next)` with `last.next = head`.
- **Operations:**
    1. `InsertAtBeginning(x)`
    2. `InsertAtEnd(x)`
    3. `Delete(x)`
    4. `Search(x)`
    5. `Traverse()`

InsertAtBeginning(head, x):
```
newNode ← Node(x)
if head = NULL:
    newNode.next ← newNode
    head ← newNode
    return head
temp ← head
while temp.next ≠ head:
    temp ← temp.next
newNode.next ← head
temp.next ← newNode
head ← newNode
return head
```

InsertAtEnd(head, x):
```
newNode ← Node(x)
```

```
    if head = NULL:
        newNode.next ← newNode
        head ← newNode
        return head
    temp ← head
    while temp.next ≠ head:
        temp ← temp.next
    temp.next ← newNode
    newNode.next ← head
    return head
```

Delete(head, x):

```
    if head = NULL: return NULL
    curr ← head
    prev ← NULL
    do:
        if curr.data = x:
            if prev = NULL:   // deleting head
                temp ← head
                while temp.next ≠ head:
                    temp ← temp.next
                if head = head.next:   // only 1 node
                    head ← NULL
                else:
                    temp.next ← head.next
                    head ← head.next
            else:
                prev.next ← curr.next
            return head
        prev ← curr
        curr ← curr.next
    while curr ≠ head
    return head
```

Traverse(head):

```
    if head = NULL: return
    temp ← head
    do:
        print temp.data
        temp ← temp.next
    while temp ≠ head
```

Complexity

- Insert at beginning: $O(n)$ (unless tail pointer is kept, then $O(1)$)
- Insert at end: $O(n)$ ($O(1)$ with tail pointer)
- Delete/Search: $O(n)$
- Traverse: $O(n)$

Definition

A **circular doubly linked list** is a doubly linked list where:

- `head.prev = tail`
- `tail.next = head`

This allows **bidirectional circular traversal**.

🔷 Comparison of Linked Lists

Feature / Type	Singly Linked List (SLL)	Circular Singly Linked List (CSLL)	Doubly Linked List (DLL)	Circular Doubly Linked List (CDLL)
Structure	Each node has **data + next**	Same as SLL, but last node points back to head	Each node has **data + next + prev**	Same as DLL, but last node connects to head and head's prev points to tail
Direction of Traversal	Forward only	Forward only, but circular (no NULL)	Forward & backward	Forward & backward, circular
First Node Pointer	`head`	`head`	`head`	`head`
Last Node Pointer	`NULL`	Points to `head`	`NULL`	Points to `head`
Insertion at Front	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Insertion at End	$O(n)$ (unless tail pointer kept)	$O(1)$ with tail pointer	$O(n)$ (unless tail pointer kept)	$O(1)$ with tail pointer
Deletion at Front	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Deletion at End	$O(n)$	$O(n)$	$O(n)$ ($O(1)$ if tail available)	$O(1)$
Traversal	Stops at NULL	Infinite unless stopped manually	Stops at NULL	Infinite unless stopped manually
Memory per Node	Data + 1 pointer	Data + 1 pointer	Data + 2 pointers	Data + 2 pointers
Ease of Implementation	Easiest	Slightly harder than SLL	More complex than SLL	Most complex
Typical Use Cases	Simple lists, stacks, queues	Round-robin scheduling, buffers	Deques, navigation (forward/back)	Advanced structures, memory management, editors

---

✅ Summary:

- **SLL** → Simple, one-way navigation.
- **CSLL** → Useful when you need circular traversal (e.g., round robin).
- **DLL** → Bi-directional navigation (backtracking possible).
- **CDLL** → Most flexible (both circular and bi-directional).

A **Splay Tree** is a self-adjusting **binary search tree (BST)** where recently accessed nodes are moved to the root using a process called **splaying**.

- Guarantees **amortized O(log n)** time for standard operations (search, insert, delete).
- Frequently accessed elements move closer to the root, improving access time for non-uniform access patterns.

Core Operations

1. Splaying (basic operation)

Moves a given node `x` to the root using tree rotations. There are three cases:

- **Zig**: `x` is child of root → single rotation
- **Zig-Zig**: `x` and parent are both left or both right children → double rotation
- **Zig-Zag**: `x` is left child and parent is right child (or vice versa) → two rotations

Pseudocode (Splay)

```
Splay(T, x):
  while x is not root:
    if parent(x) is root:
      if x is left child: RightRotate(parent(x))
      else: LeftRotate(parent(x))
    else if x and parent(x) are both left children:
      RightRotate(grandparent(x))
      RightRotate(parent(x))
    else if x and parent(x) are both right children:
```

```
        LeftRotate(grandparent(x))
        LeftRotate(parent(x))
    else if x is left child and parent is right child:
        RightRotate(parent(x))
        LeftRotate(grandparent(x))
    else:
        LeftRotate(parent(x))
        RightRotate(grandparent(x))
```

Search(T, key):

```
x = BST_Search(T, key)
if x ≠ NULL:
    Splay(T, x)
return x
```

Insert(T, key):

```
x = BST_Insert(T, key)
Splay(T, x)
```

Delete(T, key)

- Search the node → splay it to root
- Remove the root
- Merge the left and right subtrees by splaying the maximum node in the left subtree, then attaching the right subtree

Delete(T, key):

```
x = Search(T, key)
if x == NULL:
    return
Splay(T, x)
if x.left == NULL:
    T = x.right
else:
    y = x.left
    maxNode = FindMax(y)
    Splay(y, maxNode)   // bring max of left subtree to root
    root = y
    root.right = x.right
```

🔷 **The Access Lemma (Sleator & Tarjan, 1985)**

For a Splay Tree with n nodes, accessing (searching/inserting/deleting) a node x costs:

$O(\log (n/s(x)))$

where $s(x)$ is the size of the subtree rooted at x.

This is the **Access Lemma**, and it forms the basis of amortized analysis.

Amortized Complexity Results

1. **Search**
    - Worst case: $O(n)$ (when the node is deep).
    - Amortized: $O(\log n)$ per search.
    - After access, the node is splayed to root → improves future accesses.
2. **Insert**
    - Worst case: $O(n)$ (like search).
    - Amortized: $O(\log n)$.
    - New node is added, then splayed to the root.
3. **Delete**
    - Worst case: $O(n)$.
    - Amortized: $O(\log n)$.
    - Involves splaying predecessor/successor to restructure the tree.
4. **Other operations** (join, split, min, max):
    - All $O(\log n)$ **amortized**.

🔷 Hashing

Hashing is a technique to map data (keys) into a fixed-size table (hash table) using a **hash function**.

- The hash function `h(key)` returns an index in the table.
- Ideally, different keys should map to different indices (no collisions).
- In practice, collisions happen, so collision resolution techniques are used.

🔷 Abstract Data Type (ADT) for Hash Table

A Hash Table supports:

- **Insert(key, value)** – Store a key–value pair.
- **Search(key)** – Retrieve value associated with key.
- **Delete(key)** – Remove key–value pair.

🔷 Hash Functions

A good hash function should:

- Be fast to compute.
- Spread keys uniformly across slots.
- Minimize collisions.

Common methods:

1. **Division method**: $h(k) = k \bmod m$
2. **Multiplication method**: $h(k) = \text{floor}(m * (k * A \bmod 1))$ where $0<A<1$
3. **Universal hashing**: Choose hash function randomly from a family.

🔷 Collision Resolution Techniques

Since multiple keys may hash to the same slot:

1. Chaining

- Each slot stores a linked list of keys that hash to it.
- Operations involve searching inside the chain.

2. Open Addressing

- All keys are stored directly in the table.
- On collision, probe for next available slot.
  - **Linear probing**: $h(k, i) = (h(k) + i) \bmod m$
  - **Quadratic probing**: $h(k, i) = (h(k) + c_1 * i + c_2 * i^2) \bmod m$
  - **Double hashing**: $h(k, i) = (h_1(k) + i * h_2(k)) \bmod m$

🔷 Pseudocode for Hash Table Operations (Chaining)

```
INSERT(T, key, value):
    index = h(key)
    if key not in T[index]:
        add (key, value) to T[index]

SEARCH(T, key):
    index = h(key)
    for each (k, v) in T[index]:
        if k == key:
            return v
    return NOT_FOUND

DELETE(T, key):
    index = h(key)
    remove (key, value) from T[index] if exists
```

🔷 Time Complexity

Operation	Average Case	Worst Case
Insert	$O(1)$	$O(n)$ (if all collide)
Search	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

-

- Average case assumes **good hash function** + **load factor α = n/m is small**.
- With chaining: Expected search cost = $O(1 + \alpha)$
- With open addressing: Expected search cost ≈ $O(1 / (1 - \alpha))$

🔷 Load Factor (α)

$\alpha = n/m$

- n = number of stored elements
- m = number of slots in table
- Controls efficiency: keep α small (≤ 1 for open addressing, can be >1 for chaining).

🔷 Linear Probing: Concept

- In **open addressing**, all keys are stored directly in the table (no external chains).
- On collision, **Linear Probing** looks for the next free slot by checking sequentially.

$h(k,i)=(h(k) + i) \bmod m$

- $h(k)$ = base hash function
- i = probe number ($0,1,2,...,m-1$)
- m = table size

So if $h(k)$ is occupied, try $(h(k)+1) \bmod m$, then $(h(k)+2) \bmod m$, and so on.

🔷 Data Structure

We store elements in an **array of size *m***.
Each slot can be in one of three states:

- **EMPTY** (never used)
- **OCCUPIED** (contains a key-value pair)
- **DELETED** (used earlier but deleted → helps in search continuation)

🔷 Pseudocode

```
INSERT(T, key, value):
    for i = 0 to m-1:
        j = (h(key) + i) mod m
        if T[j] is EMPTY or T[j] is DELETED:
            T[j] = (key, value)
            return SUCCESS
    return FAILURE   # Table full
```

```
SEARCH(T, key):
    for i = 0 to m-1:
        j = (h(key) + i) mod m
        if T[j] is EMPTY:      # Stop if an empty slot is found
            return NOT_FOUND
        if T[j].key == key:
            return T[j].value
    return NOT_FOUND
```

```
DELETE(T, key):
    for i = 0 to m-1:
        j = (h(key) + i) mod m
        if T[j] is EMPTY:
            return FAILURE    # Key not found
        if T[j].key == key:
            mark T[j] as DELETED
            return SUCCESS
    return FAILURE
```

🔷 Example

Suppose $m = 7$, hash function $h(k) = k \bmod 7$.

Insert keys: 10, 20, 15, 7

- Insert 10 → h(10)=3 → slot 3 empty → place at 3
- Insert 20 → h(20)=6 → slot 6 empty → place at 6
- Insert 15 → h(15)=1 → slot 1 empty → place at 1
- Insert 7 → $h(7)=0$ → slot 0 empty → place at 0

If we now insert 17 → $h(17)=3$ → slot 3 occupied → try 4 → empty → place at 4

🔷 Time Complexity

- **Best case**: $O(1)$ (no collision)
- **Average case**: $O(1 / (1 - \alpha))$ where $\alpha = n/m$ (load factor)
- **Worst case**: $O(m)$ (table almost full → long probe sequence)

🔷 Problem: Primary Clustering

- In linear probing, **consecutive occupied slots form clusters**.
- Future insertions into the cluster make it grow further.

- This increases average probe length.

  👉 Alternative: **Quadratic Probing** or **Double Hashing** to reduce clustering.

🔷 Quadratic Probing: Concept

- Like **linear probing**, we handle collisions by trying alternative slots.
- But instead of checking sequentially (`+1, +2, +3, …`), we probe using **quadratic increments** to reduce **primary clustering**.

General probe function:

$h(k,i)=(h(k) + c_1 i + c_2 i^2) \bmod m$

- $h(k)$ = base hash function
- i = probe number (`0, 1, 2, …`)
- c_1, c_2 = constants (usually small, e.g., 1/2)
- m = table size

So the probing sequence goes like:
$h(k),\ h(k)+1,\ h(k)+4,\ h(k)+9,\ h(k)+16,\ \ldots \pmod m$

🔷 Advantages

- **Avoids primary clustering** (consecutive filled slots don't create a giant cluster).
- Each new probe "jumps away" in quadratic steps.

🔷 Disadvantages

- **Secondary clustering** still exists: two keys with the same starting slot follow the exact same quadratic sequence.
- Requires careful choice of m, c_1, c_2 to guarantee that a free slot will eventually be found. (Often m is chosen as a prime number, and load factor $\alpha \leq 0.5$).

🔷 Pseudo-code

```
INSERT(T, key, value):
    for i = 0 to m-1:
        j = (h(key) + c1*i + c2*i*i) mod m
        if T[j] is EMPTY or T[j] is DELETED:
            T[j] = (key, value)
```

```
            return SUCCESS
    return FAILURE   # Table full
```

```
SEARCH(T, key):
    for i = 0 to m-1:
        j = (h(key) + c1*i + c2*i*i) mod m
        if T[j] is EMPTY:
            return NOT_FOUND
        if T[j].key == key:
            return T[j].value
    return NOT_FOUND
```

```
DELETE(T, key):
    for i = 0 to m-1:
        j = (h(key) + c1*i + c2*i*i) mod m
        if T[j] is EMPTY:
            return FAILURE    # Key not found
        if T[j].key == key:
            mark T[j] as DELETED
            return SUCCESS
    return FAILURE
```

🔷 Example

Suppose `m = 7`, $h(k) = k \bmod 7$, and probing rule:

$h(k,i)=(h(k) + i^2) \bmod 7$

Insert keys: **10, 17, 24**

- Insert 10 → h(10)=3 → slot 3 empty → place at 3
- Insert 17 → h(17)=3 → slot 3 occupied → try $(3+1^2)=4$ → empty → place at 4
- Insert 24 → h(24)=3 → slot 3 occupied → try $(3+1^2)=4$ (occupied) → try $(3+2^2)=7\equiv0$ → empty → place at 0

Final table: slot[0] = 24, slot[3] = 10, slot[4] = 17

🔷 Time Complexity

- **Best case**: $O(1)$
- **Average case**: $O(1 / (1 - \alpha))$, where $\alpha = n/m$
- **Worst case**: $O(m)$ (if table nearly full and unlucky collisions)

✅ Comparison with Linear Probing

- Linear probing suffers from **primary clustering**.
- Quadratic probing avoids it but still has **secondary clustering**.
- To fully avoid clustering, **Double Hashing** is usually preferred.

Heap: Definition

A **Heap** is a specialized **binary tree-based** data structure that satisfies the **heap property**:

- **Max-Heap Property**: Every parent node is **greater than or equal to** its children.
  (Root contains the maximum element.)
- **Min-Heap Property**: Every parent node is **less than or equal to** its children.
  (Root contains the minimum element.)

✅ Properties

1. **Shape property**: Heap is a **complete binary tree** (all levels filled except possibly the last, which is filled left to right).
2. **Heap property**: Maintains ordering between parent and child nodes (as above).

✅ Types

- **Binary Heap**: Each node has at most 2 children.
- ***d*-ary Heap**: Each node has at most *d* children (generalization).
- **Min-Heap / Max-Heap**: Based on the property used.

📌 Heap Operations

Insert (Heapify Up)

- Place the new element at the next available position (to keep the tree complete).
- Compare with its parent; swap if the heap property is violated.
- Repeat until property is restored.

```
Insert(heap, value):
    heap.size ← heap.size + 1
    i ← heap.size
    heap[i] ← value
    while i > 1 and heap[parent(i)] < heap[i]:
        swap(heap[i], heap[parent(i)])
        i ← parent(i)
```

- **Time Complexity:** $O(\log n)$

---

2. Delete Root (Extract-Max / Extract-Min)

- Remove the root (max or min).
- Replace root with the last element.
- **HeapifyDown**: Compare with children and swap with the larger (max-heap) or smaller (min-heap) child until heap property is satisfied.

```
ExtractMax(heap):
    if heap.size == 0:
        return error
    max ← heap[1]
    heap[1] ← heap[heap.size]
    heap.size ← heap.size - 1
    HeapifyDown(heap, 1)
    return max

HeapifyDown(heap, i):
    left ← 2*i
    right ← 2*i + 1
    largest ← i

    if left ≤ heap.size and heap[left] > heap[largest]:
        largest ← left
    if right ≤ heap.size and heap[right] > heap[largest]:
        largest ← right
    if largest ≠ i:
        swap(heap[i], heap[largest])
        HeapifyDown(heap, largest)
```

- **Time Complexity:** O(log n)

---

3. Heapify (Build-Heap)

- Convert an arbitrary array into a heap.
- Start from the lowest non-leaf nodes and apply `HeapifyDown`.

```
BuildHeap(array, n):
    for i ← ⌊n/2⌋ downto 1:
        HeapifyDown(array, i)
```

- **Time Complexity:** $O(n)$

---

4. Heap Sort

- Build a max-heap.
- Repeatedly extract max and place it at the end of the array.

Time Complexity: $O(n \log n)$

📌 Complexities (Binary Heap)

Operation	Time Complexity
Insert	$O(\log n)$
Extract Max/ Min	$O(\log n)$
Find Max/Min	$O(1)$
Build Heap	$O(n)$
Heap Sort	$O(n \log n)$

SELECTION_SORT(A[0..n-1]):

```
for i = 0 to n-2 do
    minIndex ← i
    for j = i+1 to n-1 do
        if A[j] < A[minIndex] then
            minIndex ← j
    swap A[i] with A[minIndex]
```

Input: `A = [29, 10, 14, 37, 14]`

- **Pass 1 (i = 0):**
  - Find min in [29,10,14,37,14] → min = 10 at index 1
  - Swap(29,10) → [10,29,14,37,14]
- **Pass 2 (i = 1):**
  - Find min in [29,14,37,14] → min = 14 at index 2
  - Swap(29,14) → [10,14,29,37,14]
- **Pass 3 (i = 2):**

- Find min in [29,37,14] → min = 14 at index 4
- Swap(29,14) → [10,14,14,37,29]

- **Pass 4 (i = 3):**
  - Find min in [37,29] → min = 29 at index 4
  - Swap(37,29) → [10,14,14,29,37]

Sorted array = [10,14,14,29,37]

🔷 Time Complexity

- **Best Case:** $O(n^2)$
  (Even if already sorted, still scans all elements to find min).
- **Worst Case:** $O(n^2)$
  (Always scans remaining unsorted elements).
- **Average Case:** $O(n^2)$.

🔷 Space Complexity

- **Auxiliary Space:** $O(1)$ (in-place sort).

🔷 Stability

- **Not stable** in the standard form, since swapping can change relative order of equal elements.

(But with small modification—placing min at correct position by shifting elements instead of swapping—it can be made stable.)

Bubble Sort

1. Idea

- Repeatedly compare adjacent elements and **swap if they are in the wrong order**.
- After the i-th pass, the **largest element bubbles up** to its correct position at the end.
- Process is repeated until the array is sorted.
-

BUBBLE_SORT(A[0..n-1]):

```
for i = 0 to n-2 do
    swapped ← false
    for j = 0 to n-2-i do
        if A[j] > A[j+1] then
            swap A[j] with A[j+1]
            swapped ← true
```

```
    if swapped = false then
        break   // optimization: stop if already sorted
```

4. Analysis

Time Complexity

- **Best Case:** $O(n)$
  (if array already sorted, with optimization check).
- **Worst Case:** $O(n^2)$
  (reverse sorted, needs max swaps).
- **Average Case:** $O(n^2)$.

Space Complexity

- **Auxiliary Space:** $O(1)$ (in-place).

Stability

- **Stable** — equal elements keep their relative order (since swap only occurs when strictly greater).

Tournament Sort

- Think of sorting like a **knockout tournament** (like tennis or football).
- Each element is a "player."
- Elements are compared pairwise until the "winner" (minimum or maximum element) is found.
- Remove the winner and repeat the tournament to find the next one.

So, Tournament Sort repeatedly selects the smallest (or largest) element using a **tournament tree**.

Tournament Tree

- A **complete binary tree** where:
    - Leaves represent the input elements.

    - Each internal node stores the **winner** (min or max) among its children.
- The **root** contains the overall winner.

3. Steps

1. Build a tournament tree with all elements.
2. The root gives the smallest element (for ascending sort).
3. Replace that leaf with +∞ (or remove it).
4. Recompute winners along the path to the root.
5. Repeat until all elements are extracted.

```
TOURNAMENT_SORT(A[0..n-1]):
    # Step 1: Build tournament tree
    T ← buildTournamentTree(A)

    sortedList ← []

    # Step 2: Extract winners
    for i = 0 to n-1 do
        winner ← T.root.value
        append winner to sortedList

        # Step 3: Replace winner with +∞
        leaf ← leaf corresponding to winner
        leaf.value ← ∞

        # Step 4: Recompute winners up to root
        updateTournamentTree(leaf, T)

    return sortedList


buildTournamentTree(A):
    # Build leaves
    leaves ← A
    while number of nodes > 1 do
        pair nodes, compute winner, make parent
    return root


updateTournamentTree(leaf, T):
    while leaf has parent do
        parent.value ← min(leaf, sibling)
        leaf ← parent
```

Dry Run Example

Input: `A = [4, 7, 2, 9]`

- Build tree:
    - Compare (4,7) → 4 wins.
    - Compare (2,9) → 2 wins.
    - Compare (4,2) → 2 wins → Root = 2.

- Extract **2**, replace with ∞, update tree: next winner = 4.
- Extract **4**, replace with ∞, update tree: next winner = 7.
- Extract **7**, then **9**.

Sorted = [2,4,7,9]

Complexity Analysis

- **Building tree:** $O(n)$ (first round has $n/2$ comparisons, next $n/4$, … total = n-1 comparisons).
- **Each extraction:** $O(\log n)$ (only update along the path to root).
- ***n* extractions:** $O(n \log n)$.

Time Complexity

- Best Case: $O(n \log n)$
- Worst Case: $O(n \log n)$
- Average Case: $O(n \log n)$

Space Complexity

- $O(n)$ (for storing tournament tree).

7. Properties

- **Not stable** (relative order of equal elements not preserved).
- Mainly useful for **external sorting** (when data is on disk, e.g., merge runs).
- Guarantees $O(n \log n)$ regardless of input order.

Merge Sort

- It's a **Divide and Conquer** algorithm.
- Divide the array into two halves.
- Recursively sort each half.
- Merge the two sorted halves into one.

Algorithm Steps

1. If the array has **0 or 1 element**, it is already sorted → return.
2. Otherwise, Split the array into two halves.
    - Recursively apply **Merge Sort** to each half.

- Merge the two sorted halves.

Pseudocode

```
MERGE_SORT(A, left, right):
    if left < right:
        mid = (left + right) // 2
        MERGE_SORT(A, left, mid)        # Sort left half
        MERGE_SORT(A, mid+1, right)     # Sort right half
        MERGE(A, left, mid, right)      # Merge both halves


MERGE(A, left, mid, right):
    # Create temp arrays
    n1 = mid - left + 1
    n2 = right - mid
    L[0..n1-1] ← A[left..mid]
    R[0..n2-1] ← A[mid+1..right]

    i = 0, j = 0, k = left

    # Merge two sorted arrays
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            A[k] = L[i]
            i = i + 1
        else:
            A[k] = R[j]
            j = j + 1
        k = k + 1

    # Copy remaining elements
    while i < n1:
        A[k] = L[i]
        i = i + 1
        k = k + 1

    while j < n2:
        A[k] = R[j]
        j = j + 1
        k = k + 1
```

Dry Run Example

Input: `A = [38, 27, 43, 3, 9, 82, 10]`

- Split: `[38,27,43,3]` and `[9,82,10]`
- Split again: `[38,27]`, `[43,3]`, `[9,82]`, `[10]`
- Split further: `[38]`, `[27]`, `[43]`, `[3]`, `[9]`, `[82]`

Now start merging:

- Merge `[38]` and `[27]` → `[27,38]`
- Merge `[43]` and `[3]` → `[3,43]`
- Merge `[9]` and `[82]` → `[9,82]`

Then:

- Merge `[27,38]` and `[3,43]` → `[3,27,38,43]`
- Merge `[9,82]` and `[10]` → `[9,10,82]`

Finally:

- Merge `[3,27,38,43]` and `[9,10,82]` → `[3,9,10,27,38,43,82]`

Sorted result.

Complexity Analysis

- **Time Complexity**:
  - Splitting takes $O(\log n)$ levels.
  - Merging each level takes $O(n)$.
  - Total = $O(n \log n)$.
- **Space Complexity**:
  - $O(n)$ (extra space for temporary arrays).
- **Best / Worst / Average Case**:
  - All are $O(n \log n)$.

Properties

- **Stable** (preserves order of equal elements).
- Works well with **linked lists** (no extra space needed if merging by pointers).
- Not in-place (requires extra memory).
- Excellent for **large data sets** and external sorting (disk files).

Quicksort

Quicksort is a **divide-and-conquer** sorting algorithm.
It works by selecting a **pivot element**, partitioning the array into two subarrays (elements smaller than pivot and elements greater than pivot), and then recursively sorting the subarrays.

Algorithm Steps

1. If the array has 0 or 1 elements → it is already sorted.
2. Choose a **pivot** element (commonly first, last, middle, or random element).
3. Partition the array into:
   - Elements **less than or equal to** pivot.

- Elements **greater than** pivot.

4. Recursively apply Quicksort to left and right subarrays.
5. Combine the results.

```
QUICKSORT(A, low, high):
    if low < high:
        pivotIndex ← PARTITION(A, low, high)
        QUICKSORT(A, low, pivotIndex - 1)
        QUICKSORT(A, pivotIndex + 1, high)


PARTITION(A, low, high):
    pivot ← A[high]                 // choose last element as pivot
    i ← low - 1                     // place for swapping
    for j ← low to high - 1:
        if A[j] ≤ pivot:
            i ← i + 1
            swap A[i], A[j]
    swap A[i + 1], A[high]
    return i + 1
```

Example

Array: `[10, 80, 30, 90, 40, 50, 70]`
Pivot = 70 → Partition gives `[10, 30, 40, 50] 70 [80, 90].`
Recursive calls continue until fully sorted.

Complexity Analysis

- **Best Case** (balanced partitions):
  $T(n) = 2T(n/2) + O(n) = O(n \log n)$
- **Average Case**:
  $O(n \log n)$
- **Worst Case** (highly unbalanced partitions, e.g., sorted input with poor pivot choice): $O(n^2)$
- **Space Complexity**:
  $O(\log n)$ (for recursion stack, best/average case),
  $O(n)$ in worst case.

Randomized Quicksort

The main issue with classical Quicksort is **pivot choice**.
If the pivot is always poorly chosen (e.g., smallest/largest element), the algorithm degrades to $\boldsymbol{O(n^2)}$.

To reduce the chance of this worst case, **Randomized Quicksort** selects the pivot **uniformly at random** from the subarray.
This ensures that *every partition is equally likely*, giving an **expected runtime of $\boldsymbol{O(n \log n)}$**.

Algorithm Steps

1. If the array has 0 or 1 elements → already sorted.
2. Randomly pick a pivot index between `low` and `high`.
3. Swap that pivot with the last element (so that partitioning logic works as usual).
4. Partition the array as in standard Quicksort.
5. Recursively sort left and right parts.

```
RANDOMIZED-QUICKSORT(A, low, high):
    if low < high:
        pivotIndex ← RANDOMIZED-PARTITION(A, low, high)
        RANDOMIZED-QUICKSORT(A, low, pivotIndex - 1)
        RANDOMIZED-QUICKSORT(A, pivotIndex + 1, high)


RANDOMIZED-PARTITION(A, low, high):
    i ← RANDOM(low, high)           // choose random index
    swap A[i], A[high]              // put pivot at end
    return PARTITION(A, low, high)  // same as standard
```

Complexity Analysis

- **Expected Time Complexity**:
  $O(n \log n)$ (due to uniform random pivot choice).
- **Worst Case Complexity**:
  Still $O(n^2)$, but probability is very low.
- **Space Complexity**:
  Same as Quicksort → $O(\log n)$ expected recursion depth.

Master Theorem

The **Master Theorem** provides a direct way to solve recurrence relations of the form:

$T(n)=aT(n/b)+f(n)$

where:

- $a \geq 1$ = number of subproblems in recursion
- $b > 1$ = factor by which subproblem size shrinks
- $f(n)$ = cost of work done outside the recursive calls (divide + combine)

Intuition

- At each level of recursion, you split the problem into a subproblems.
- Each subproblem is of size n/b.
- You do some extra work $f(n)$ at each level.
- The total work = sum across recursion tree.

Three Cases

We compare $f(n)$ with n log a/b (called the **critical exponent**).

Case 1: $f(n)$ is smaller

If

some $f(n) = O(n \text{ ^}(\log_b a - \epsilon))$ for some $\epsilon>0$

then

$T(n)=\Theta(n \text{ ^}(\log a/b))$

Work is dominated by the leaves (recursive calls).

Case 2: $f(n)$ matches

If

$f(n)=\Theta(n \text{ ^}(\log_b a) \cdot \log^k n), k \geq 0$

then

$T(n)=\Theta(n \log a/b \cdot \log k +1n)$

👉 Work is evenly balanced between recursive calls and outside work.

Case 3: f(n) is larger

If some $f(n)=\Omega(n \log ba+\epsilon)$for some $\epsilon>0$

and if **regularity condition** holds:

some $a \cdot f(bn) \leq c \cdot f(n)$,for some $c<1$

then

$T(n)=\Theta(f(n))$

👉 Work is dominated by the root (outside work).

Examples

1. **Merge Sort**

$T(n)=2T(n/2)+O(n)$

- $a=2, b=2, f(n)=n$
- $\log_b a=\log_2 2=1$
- $f(n)=\Theta(n^1) \rightarrow$ Case 2
- $T(n)=\Theta(n\log n)$.

---

2. **Binary Search**

$T(n)=T(n/2)+O(1)$

- $a=1, b=2, f(n)=1$
- $\log_b a=\log_2 1=0$
- $f(n)=O(n^{0-\epsilon}) \rightarrow$ Case 1
- $T(n)=\Theta(\log n)$.

AVL Tree Deletion

Steps

1. **BST Deletion**
    - If the node has no child → delete directly.
    - If the node has one child → replace with child.
    - If the node has two children → replace with inorder successor (or predecessor) and recursively delete that.
2. **Update Heights**

- After deletion, update the height of current node.

3. **Check Balance Factor (BF = height(left) – height(right))**
    - If BF ∈ {–1,0,1} → balanced.
    - Otherwise → rotations required.
4. **Rebalance with Rotations**
    - **LL Case** → Right Rotation
    - **RR Case** → Left Rotation
    - **LR Case** → Left Rotation + Right Rotation
    - **RL Case** → Right Rotation + Left Rotation

Pseudocode

```
DELETE_AVL(root, key):
    if root = NULL:
        return root

    # --- Step 1: Perform standard BST deletion ---
    if key < root.key:
        root.left = DELETE_AVL(root.left, key)
    else if key > root.key:
        root.right = DELETE_AVL(root.right, key)
    else:
        # Node found
        if root.left = NULL or root.right = NULL:
            temp = root.left ? root.left : root.right
            if temp = NULL:
                root = NULL
            else:
                root = temp
        else:
            # Node with two children
            temp = MIN_VALUE_NODE(root.right)   # inorder successor
            root.key = temp.key
            root.right = DELETE_AVL(root.right, temp.key)

    if root = NULL:
        return root

    # --- Step 2: Update height ---
    root.height = 1 + max(HEIGHT(root.left), HEIGHT(root.right))

    # --- Step 3: Get Balance Factor ---
    balance = HEIGHT(root.left) - HEIGHT(root.right)

    # --- Step 4: Balance the tree with rotations ---

    # LL Case
    if balance > 1 and GET_BALANCE(root.left) >= 0:
        return RIGHT_ROTATE(root)

    # LR Case
    if balance > 1 and GET_BALANCE(root.left) < 0:
        root.left = LEFT_ROTATE(root.left)
        return RIGHT_ROTATE(root)
```

```
    # RR Case
    if balance < -1 and GET_BALANCE(root.right) <= 0:
        return LEFT_ROTATE(root)

    # RL Case
    if balance < -1 and GET_BALANCE(root.right) > 0:
        root.right = RIGHT_ROTATE(root.right)
        return LEFT_ROTATE(root)

    return root


# --- Utility Functions ---
HEIGHT(node):
    if node = NULL: return 0
    return node.height

GET_BALANCE(node):
    if node = NULL: return 0
    return HEIGHT(node.left) - HEIGHT(node.right)

MIN_VALUE_NODE(node):
    current = node
    while current.left != NULL:
        current = current.left
    return current
```

Complexity

- **Deletion:** $O(\log n)$ (BST delete + rebalancing).
- **Rotations:** $O(1)$.
- **Total:** $O(\log n)$.

Median of Medians algorithm (Blum–Floyd–Pratt–Rivest–Tarjan, 1973).

Linear-Time Median Finding (Median of Medians)

1. Problem

Given an unsorted array of n elements, find the **median** (or generally, the k-th smallest element).

A naive way is to sort → O(nlogn).
But with **selection algorithms**, we can do it in **linear time**.

---

2. Idea

This is based on the **Deterministic Selection Algorithm**:

1. Divide array into groups of 5 elements.
2. Sort each group (constant time per group, since size ≤ 5).
3. Pick the **median of each group**.
4. Recursively find the **median of these medians** → pivot.
5. Partition the array around this pivot.
6. Recurse into the part containing the k-th element.

This guarantees a **good pivot** (not too skewed), ensuring linear time.

---

3. Pseudocode

```
SELECT(A, k):   # returns k-th smallest element in A
    if length(A) ≤ 5:
        sort(A)
        return A[k]

    # Step 1: Divide into groups of 5
    groups = split A into groups of 5

    # Step 2: Find medians of groups
    medians = []
    for each group in groups:
        sort(group)
        append median(group) to medians

    # Step 3: Recursively find pivot (median of medians)
    pivot = SELECT(medians, length(medians) / 2)

    # Step 4: Partition A around pivot
    L = [x in A | x < pivot]
    E = [x in A | x = pivot]
    G = [x in A | x > pivot]

    # Step 5: Recurse into correct part
    if k ≤ |L|:
        return SELECT(L, k)
    else if k ≤ |L| + |E|:
        return pivot
    else:
        return SELECT(G, k - |L| - |E|)
```

---

4. Example

Array: `[12, 3, 5, 7, 4, 19, 26]`, Find median.

- Groups: `[12,3,5,7,4]`, `[19,26]`
- Medians: `5` and `26` → median of medians = `5` (pivot).
- Partition: L = `[3,4]`, E = `[5]`, G = `[12,7,19,26]`.
- Median index = 4. Since it lies in G, recurse into G.

- Eventually, find answer = `7`.

✅ Median = 7.

---

5. Complexity

- Grouping + sorting small groups = $O(n)$.
- Recursive call on ~ $n/5$ elements = $O(n/5)$.
- Partitioning = $O(n)$.
- Recursion tree is balanced → **Total = $O(n)$.**

Applications

- Finding **median** or ***k*-th order statistic**.
- Used in **Introselect** (C++ `nth_element`).
- Forms basis of **deterministic quickselect**.

Heapsort Algorithm

1. Idea

- A **heap** is a complete binary tree where each node satisfies the **heap property**.
  - **Max-Heap**: parent ≥ children.
  - **Min-Heap**: parent ≤ children.
- **Heapsort** uses a **max-heap** to sort in ascending order:
  - **Build a max heap** from the input array.
  - The maximum element is at the root → swap it with the last element.
  - Reduce heap size by 1 and **heapify** the root.
  - Repeat until the heap size = 1.

Pseudocode

Helper functions

```
MAX-HEAPIFY(A, i, n):
    # A is array, i is index, n is heap size
    left  = 2 * i
    right = 2 * i + 1
    largest = i

    if left ≤ n and A[left] > A[largest]:
        largest = left

    if right ≤ n and A[right] > A[largest]:
        largest = right

    if largest ≠ i:
        swap A[i], A[largest]
        MAX-HEAPIFY(A, largest, n)
```

Build a Heap

```
BUILD-MAX-HEAP(A, n):
    for i = ⌊n/2⌋ downto 1:
        MAX-HEAPIFY(A, i, n)
```

Heapsort

```
HEAPSORT(A):
    n = length(A)
    BUILD-MAX-HEAP(A, n)

    for i = n downto 2:
        swap A[1], A[i]          # Move max to the end
        n = n - 1                # Reduce heap size
        MAX-HEAPIFY(A, 1, n)     # Restore heap property
```

(Here array indices start from 1 for simplicity; in 0-based arrays adjust formulas.)

---

3. Example

Input: `A = [4, 10, 3, 5, 1]`

1. Build max-heap: `[10, 5, 3, 4, 1]`
2. Swap 10 with last → `[1, 5, 3, 4, 10]` → heapify → `[5, 4, 3, 1, 10]`
3. Swap 5 with 1 → `[1, 4, 3, 5, 10]` → heapify → `[4, 1, 3, 5, 10]`
4. Swap 4 with 3 → `[3, 1, 4, 5, 10]` → heapify → `[3, 1, 4, 5, 10]`
5. Continue → Final sorted array: `[1, 3, 4, 5, 10]`

---

4. Complexity Analysis

- **Build-Max-Heap**: $O(n)$

- **Heapify**: $O(\log n)$
- **Sorting loop**: $n-1$ calls to heapify → $O(n \log n)$

➡️ **Overall Time Complexity**:

- Best Case = $O(n \log n)$
- Worst Case = $O(n \log n)$
- Average Case = $O(n \log n)$

➡️ **Space Complexity**: $O(1)$ (in-place sorting)

---

5. Properties

- Not stable (relative order of equal elements may change).
- In-place, unlike mergesort.
- Performance guarantee (always O(nlogn)), unlike quicksort.

✅ **Heapsort = Build Max-Heap + Extract Max repeatedly**

Lower Bound for Sorting

1. Context

We are considering **comparison-based sorting algorithms**:

- Algorithms that only gain information about the input by comparing elements (like Quicksort, Mergesort, Heapsort, Insertion sort, etc.).

👉 The question: *What is the best possible asymptotic runtime any comparison-based sorting algorithm can achieve?*

---

2. Decision Tree Model

- Sorting `n` elements can be seen as a **decision problem**.
- Each comparison (e.g., `A[i] < A[j]?`) splits possibilities into two branches.

- The whole sorting process is represented as a **binary decision tree**.

Key facts:

- There are `n!` possible input permutations.
- A correct sorting algorithm must be able to distinguish **all** `n!` **permutations**.
- Hence, the decision tree must have at least `n!` leaves.

---

3. Height of the Decision Tree

- If the decision tree has height `h`, then it can have at most `2^h` leaves.
- To sort `n` elements, we need:

$2h \geq n!$

- Taking logs:

$h \geq \log_2(n!)$

---

4. Asymptotic Bound

Using **Stirling's approximation**:

$n! \approx 2\pi n \left(\frac{e}{n}\right)^n$

So,

$\log_2(n!) = \Theta(n \log n)$

Thus:

$h = \Omega(n \log n)$

---

5. Result

- **Lower Bound**: Any comparison-based sorting algorithm must take

$\Omega(n \log n)$

comparisons in the **worst case**.

- **Implication**: No comparison sort can beat `O(n log n)` in general.
- **Optimal algorithms** (achieve the bound): Mergesort, Heapsort, Randomized Quicksort (expected).

---

6. Exceptions

- If extra assumptions are made (e.g., numbers are small integers), then **non-comparison sorts** like **Counting Sort, Radix Sort, Bucket Sort** can achieve **O(n)**.
- But under the **pure comparison model**, `Ω(n log n)` is the best possible.

✅ **Summary:**

The **decision-tree argument** shows that comparison-based sorting has a worst-case lower bound of **Ω(n log n)**.

Radix Sort

1. Definition

Radix Sort is a **non-comparative integer sorting algorithm** that sorts numbers by processing individual digits.
It works by sorting the numbers digit by digit, starting either from:

- **Least Significant Digit (LSD)** → common in practice.
- **Most Significant Digit (MSD)** → less common, more complex.

To sort digits, Radix Sort typically uses a **stable sorting algorithm** such as **Counting Sort** as a subroutine.

---

2. Idea

1. Find the maximum number to know the number of digits `d`.
2. Starting from the **least significant digit (LSD)**, use **Counting Sort** (or another stable sort) to sort numbers according to that digit.
3. Move to the next significant digit and repeat.

4. After `d` passes, the array is sorted.

---

3. Pseudocode

```
RADIX-SORT(A, n):
    maxVal = maximum element in A
    exp = 1   # digit position (1 = LSD, 10 = tens, 100 = hundreds, ...)

    while maxVal / exp > 0:
        COUNTING-SORT-BY-DIGIT(A, n, exp)
        exp = exp * 10


COUNTING-SORT-BY-DIGIT(A, n, exp):
    output[1..n]
    count[0..9] = {0}

    # Count occurrences of digits
    for i = 1 to n:
        digit = (A[i] / exp) % 10
        count[digit] = count[digit] + 1

    # Accumulate counts
    for i = 1 to 9:
        count[i] = count[i] + count[i-1]

    # Build output (stable)
    for i = n downto 1:
        digit = (A[i] / exp) % 10
        output[count[digit]] = A[i]
        count[digit] = count[digit] - 1

    # Copy back to A
    for i = 1 to n:
        A[i] = output[i]
```

---

4. Example

Input: `[170, 45, 75, 90, 802, 24, 2, 66]`

- Sort by 1s place → `[170, 90, 802, 2, 24, 45, 75, 66]`
- Sort by 10s place → `[802, 2, 24, 45, 66, 170, 75, 90]`
- Sort by 100s place → `[2, 24, 45, 66, 75, 90, 170, 802]`

Final sorted array.

---

5. Complexity Analysis

- Let `n` = number of elements, `d` = number of digits, `k` = range of each digit (0–9 for decimal).
- Each digit sorting takes $O(n + k)$ using Counting Sort.
- Total time: $O(d \times (n + k))$.

For fixed-digit integers (like 32-bit ints), `d` is constant → ***O(n)***.
For large numbers with many digits, performance depends on `d`.

- **Space Complexity**: $O(n + k)$.
- **Stable**: Yes (because of stable subroutine like Counting Sort).

✅ **Key Uses**: Sorting integers, strings, and other keys with a fixed length.

✅ **Advantage**: Can beat `O(n log n)` comparison-based sorting for bounded `d`.

Stable Sort

1. Definition

A sorting algorithm is said to be **stable** if **two equal elements preserve their relative order** after sorting.

👉 Example:

Input:
`[(4, A), (3, B), (4, C), (2, D)]` (numbers are keys, letters are labels)

Sorted by number (ascending):

- **Stable Sort** → `[(2, D), (3, B), (4, A), (4, C)]`
- **Unstable Sort** → `[(2, D), (3, B), (4, C), (4, A)]`

In the stable version, `(4, A)` stays before `(4, C)`, just like in the input.

---

2. Why Stability Matters

- Stability is important when **sorting by multiple keys**.

    - Example: Students sorted first by **name**, then by **age**.
    - If age sort is stable, students with the same age remain in the name order.
- Essential in **radix sort** (where sorting is done digit by digit).

---

3. Stable vs Unstable Algorithms

Algorithm	Stable?
Bubble Sort	✅ Yes
Insertion Sort	✅ Yes
Merge Sort	✅ Yes
Counting Sort	✅ Yes (if implemented properly)
Radix Sort	✅ Yes (relies on stability of sub-sort)
Bucket Sort	Depends (if stable sort is used inside buckets)
Selection Sort	❌ No
Heap Sort	❌ No
Quick Sort	❌ No (can be made stable with modifications but not naturally)

---

4. Complexity

- Stability does not change the **time complexity** — it's a property of the algorithm's behavior.
- But often, stable algorithms use **extra memory** (like Merge Sort).

---

5. Summary

- **Stable Sort** = equal elements keep original relative order.
- Useful for **multi-key sorting**.
- Examples: Merge Sort, Insertion Sort, Bubble Sort, Counting Sort, Radix Sort.

The **Union-Find** (or **Disjoint Set Union (DSU)**) data structure is used to maintain a collection of **disjoint sets** under two operations:

1. **Find(*x*):** Returns the **representative** (root) of the set containing element x.
2. **Union(*x*, *y*):** Merges the sets containing x and y.

It is extremely useful in algorithms like Kruskal's algorithm for finding minimum spanning trees, dynamic connectivity, etc.

Basic Representation

Each set is represented as a **rooted tree**:

- Each node has a parent pointer.
- The **root node** of each tree is the **representative** of the set.
- Initially, each element is its own parent (a singleton set).

Two Key Optimizations

Without optimizations, `Find` can take $O(n)$ in the worst case (a tall tree). Two powerful heuristics reduce the cost drastically:

1. Union by Rank / Size

- Always attach the **smaller tree** under the root of the **larger tree**.
- Maintains shallower trees.

Formally:

- Maintain a `rank` (approximate depth) or `size` (number of nodes) for each root.
- When `Union(x, y)` is called, make the root with **smaller rank** a child of the root with **larger rank**.
- If ranks are equal, pick one arbitrarily and increase its rank by 1.

---

2. Path Compression

- During `Find(x)`, make every node on the path from `x` to root **point directly to the root**.

Example:

```
int find(int x) {
    if (parent[x] != x)
        parent[x] = find(parent[x]);  // path compression
    return parent[x];
}
```

This flattens the tree almost completely over time.

Amortized Analysis

We want to analyze the **total cost of m operations (unions + finds) on *n* elements.**

Step 1: Bound Height with Union by Rank Alone

- With **union by rank (without path compression)**, each node's rank increases only when it becomes the root of a union of **two equal-rank trees.**
- So if a node's rank is r, its subtree size is at least 2^r.

Lemma: Maximum rank $\leq \log_2 n$.
Proof:

- Base case: rank 0 nodes are singletons.

- Inductive step: When rank increases to r+1, it must merge two rank r trees, each of size $\geq 2^r$.
- So size $\geq 2^{r+1}$.
- Since there are at most n nodes, $2^r \leq n \Rightarrow r \leq \log_2 n$.

So trees are at most $O(\log n)$ tall if we use **union by rank only**.

Step 2: Add Path Compression

Path compression **further speeds up** `Find`. The analysis of the combination is more subtle.

Step 3: Ackermann Function Bound

This is the famous result:

> **Theorem (Robert Tarjan, 1975):**
> Using both **union by rank** and **path compression**, any sequence of m `Union` and `Find` operations on n elements runs in
>
> $O(m\ \alpha(n))$
>
> time,
> where $\alpha(n)$ is the **inverse Ackermann function** — an *extremely slowly growing* function (≤ 5 for all practical n).

This means that **each operation runs in almost constant time amortized**.

Step 4: Sketch of Proof

Let's sketch how this amortized bound is derived (high level but rigorous):

Definitions

- Let `rank(x)` denote the rank of node `x`.
- Ranks are non-decreasing along parent pointers.
- Maximum rank $\leq \log n$.

Charging argument (Potential method)

- Every `Find` walks from a node to the root, then compresses.
- We split the cost of traversing each edge into two types:
  - **Link to a much higher rank:** These happen few times.
  - **Link to a similar rank:** These can happen only $O(\log n)$ times.
- More formally, Tarjan showed:

Lemma (Rank Grouping):
Define level groups of ranks using iterated logarithms (or Ackermann hierarchy). A node can move from a lower level group to a higher group only a bounded number of times.

- The amortized number of times any node's parent pointer is updated is $O(\alpha(n))$.

- Summing over n nodes and m operations gives O(m α(n)) total cost.

Step 5: Why α(n) is "Almost Constant"

- α(n) = inverse Ackermann function.
- It grows slower than $\log^* n$.
- For any conceivable input size (even the number of atoms in the universe), α(n)≤5.

So **practically**, each `Find`/`Union` runs in **amortized *O*(1)** time.

Summary Table

Operation	Worst-case (naive)	With union by rank only	With union by rank + path compression
`Find`	$O(n)$	$O(\log n)$	$O(\alpha(n))$ (amortized)
`Union`	$O(n)$	$O(\log n)$	$O(\alpha(n))$ (amortized)

Final Notes

- The correctness of Union-Find is trivial (set partition maintained).
- The efficiency comes from clever tree flattening and balancing.
- The α(n)-bound is one of the most celebrated results in data structure analysis.

Counting Sort is a **non-comparison-based, integer sorting algorithm**.
It works by **counting the number of occurrences of each distinct element**, and then **using that information to place each element directly into its correct position** in the output array.

- It is **stable** (preserves order of equal elements) if implemented carefully.
- It works best when:
  - the range of input elements (k) is not much larger than the number of elements (`n`)
  - elements are **integers (or can be mapped to integers)**

Steps of Counting Sort

Suppose we have an input array `A[1..`n`]` where each element is an integer in the range `[0..k]`.

1. Count occurrences

- Create an auxiliary count array `C[0..k]`.
- For each `A[`i`]`, do `C[A[`i`]]++`.

2. Compute prefix sums

- Update `C` to store the **cumulative counts**:

```
for i = 1 to k:
    C[i] = C[i] + C[i-1]
```

- Now, `C[x]` contains the **number of elements ≤ *x***.

3. Place elements into output array

- Create an output array `B[1..n]`.
- Iterate through `A` **from right to left** (to make it stable):

```
for i = n downto 1:
    B[C[A[i]]] = A[i]
    C[A[i]] = C[A[i]] - 1
```

- Now `B` is the sorted array.

Example

Let `A = [4, 2, 2, 8, 3, 3, 1]`

Here `k = 8`.

Step 1 – Count

`C` after counting:

`[0,1,0,2,1,0,0,0,1]`

Step 2 – Prefix sum

`C` after cumulative:

`[0,1,1,3,4,4,4,4,5]`

Step 3 – Place elements

Iterate `A` from right to left and fill `B`.

Result: `B = [1,2,2,3,3,4,8]`

Time and Space Complexity

Operation	Time
Counting elements	$O(n)$
Prefix sums	$O(k)$
Output placement	$O(n)$

Total: O(n+k) time and O(n+k) space.

- If $k = O(n)$, then Counting Sort runs in **linear time**.

- This is **better than comparison sorts** ($O(n \log n)$) when k is small.

Properties

Property	Value
Stable	Yes (if done right-to-left in final pass)
In-place	No (uses extra arrays)
Comparison-based	No
Best use-case	When k (range of values) is small compared to n

When to Use Counting Sort

- Sorting small integers (like exam scores, ages, IDs).
- As a subroutine in **Radix Sort** (which uses counting sort as a stable sort on each digit).
- When you need **stability and linear time** with bounded keys.

Intuition

Counting Sort doesn't compare elements — it **uses the values as indices** to count their frequencies. This is why it avoids the Ω(nlogn) lower bound that applies to comparison-based sorting algorithms.

Summary

- **Input**: n elements in range $0..k$
- **Time complexity**: $O(n+k)$
- **Space complexity**: $O(n+k)$
- **Stable**: Yes
- **In-place**: No
- **Best when**: $k = O(n)$

Radix Sort is a **non-comparison-based, stable sorting algorithm** used for sorting **integers (or strings) by processing individual digits or characters** from **least significant to most significant** (LSD radix sort), or vice versa (MSD radix sort).

It **leverages a stable subroutine** like **Counting Sort** to sort elements based on each digit/character.

- Typical version: **LSD Radix Sort**
- Works when the **number of digits is fixed or small** compared to the number of items.

How Radix Sort Works (LSD version)

Assume:

- You have n numbers
- Each number has d digits (in base b, e.g. base 10 for decimal)

Steps:

1. For $i = 1$ to d (from least significant digit to most significant digit):
    - Use a **stable sort** (like Counting Sort) to sort all numbers **based on the *i*-th digit**.

Because the sorting is stable, sorting by lower digits first will not disturb the order of numbers that share the same lower digits.

Example

Sort: `[329, 457, 657, 839, 436, 720, 355]`

Pass 1 (ones place):
→ `[720, 355, 436, 457, 657, 329, 839]`

Pass 2 (tens place):
→ `[720, 329, 436, 839, 355, 457, 657]`

Pass 3 (hundreds place):
→ `[329, 355, 436, 457, 657, 720, 839]` (sorted)

Pseudocode

```
RADIX_SORT(A, d):
    # A: array of n integers
    # d: number of digits (max digit length)
    for i = 1 to d:
        STABLE_COUNTING_SORT(A, digit=i)
```

Stable Counting Sort by digit:

```
STABLE_COUNTING_SORT(A, digit):
    n = length(A)
    B = array of size n
    C = array[0..9] initialized to 0  # for base 10
```

```
# Count occurrences of each digit
for j = 1 to n:
    digit_value = get_digit(A[j], digit)
    C[digit_value] += 1

# Compute prefix sums
for k = 1 to 9:
    C[k] += C[k-1]

# Build output (right-to-left for stability)
for j = n downto 1:
    digit_value = get_digit(A[j], digit)
    B[C[digit_value]] = A[j]
    C[digit_value] -= 1

# Copy back to A
for j = 1 to n:
    A[j] = B[j]
```

`get_digit(x, i)` returns the i-th least significant digit of number `x`.

Complexity Analysis

- n = number of elements
- d = number of digits in the largest number
- k = range of each digit (for decimal, $k = 10$)

Each counting sort pass: $O(n+k)$
Number of passes: d

Total time:

$O(d \times (n+k))$

- If k is constant (like 10) and $d = O(\log_{k}(\text{max value}))$, then time is $O(n \log max)$
- If d is small, this is essentially **linear time** $O(\boldsymbol{n})$.

Space complexity:
$O(n+k)$ (for the count and output arrays).

Properties

Property	Value
Stable	Yes (if using stable sort inside)
In-place	No (needs auxiliary arrays)
Comparison-based	No
Best use-case	Large number of items with bounded digit length

When to Use Radix Sort

- Sorting large datasets of integers (like roll numbers, phone numbers, account numbers)
- When keys have **fixed length and small alphabet/radix**
- As a **faster alternative to comparison sorts** for such data

Key Idea Recap

- Sort by each digit from **least to most significant** using a **stable sort**.
- Stability ensures previously sorted lower digits remain in correct order as you move to higher digits.
- Avoids $\Omega(n \log n)$ lower bound because it's **not comparison-based**.

3.5.2 Divide and Conquer Relations

In a divide-and-conquer algorithm, the problem is divided into smaller subproblems, each subproblem is solved recursively, and a *combine* algorithm is used to solve the original problem. Assume that there are a subproblems, each of size $1/b$ of the original problem, and that the algorithm used to combine the solutions of the subproblems runs in time cn^k, for some constants a, b, c, and k. The running time $T(n)$ of the algorithm thus satisfies

$$T(n) = aT(n/b) + cn^k. \tag{3.14}$$

We assume, for simplicity, that $n = b^m$, so that n/b is always an integer (b is an integer greater than 1). We first try to expand (3.14) a couple of times to get the feel of it:

$$T(n) = a(aT(n/b^2) + c(n/b)^k) + cn^k = a(a(aT(n/b^3) + c(n/b^2)^k) + c(n/b)^k) + cn^k.$$

In general, if we expand all the way to $n/b^m = 1$, we get

$$T(n) = a(a(\cdots T(n/b^m) + c(n/b^{m-1})^k) + \cdots) + cn^k.$$

Let's assume that $T(1) = c$ (a different value would change the end result by only a constant). Then,

$$T(n) = ca^m + ca^{m-1}b^k + ca^{m-2}b^{2k} + \cdots + cb^{mk},$$

which implies that

$$T(n) = c\sum_{i=0}^{m} a^{m-i}b^{ik} = ca^m \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i.$$

But, this is a simple geometric series. There are three cases, depending on whether (b^k/a) is less than, greater than, or equal to 1.

Case 1: $a > b^k$

In this case, the factor of the geometric series is less than 1, so the series converges to a constant even if m goes to infinity. Therefore, $T(n) = O(a^m)$. Since $m = \log_b n$, we get $a^m = a^{\log_b n} = n^{\log_b a}$ (the last equality can be easily proven by taking logarithm of base b of both sides). Thus,

$$T(n) = O(n^{\log_b a}).$$

Case 2: $a = b^k$

In this case, the factor of the geometric series is 1, and thus $T(n) = O(a^m m)$. Notice that $a = b^k$ implies that $\log_b a = k$ and $m = O(\log n)$. Thus,

$$T(n) = O(n^k \log n).$$

Case 3: $a < b^k$

In this case, the factor of the geometric series is greater than 1. We use the standard expression for summing a geometric series. Denote b^k/a by F (F is a constant). Since the first element of the series is a^m, we obtain

$$T(n) = a^m \frac{F^{m+1} - 1}{F - 1} = O(a^m F^m) = O((b^k)^m) = O((b^m)^k) = O(n^k).$$

These three cases are summarized in the following theorem.

□ Theorem 3.4

The solution of the recurrence relation $T(n) = aT(n/b) + cn^k$, *where* a *and* b *are integer constants,* $a \geq 1$, $b \geq 2$, *and* c *and* k *are positive constants, is*

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a > b^k \\ O(n^k \log n) & \text{if } a = b^k \\ O(n^k) & \text{if } a < b^k \end{cases}.$$

□

The result of Theorem 3.4 applies to many divide-and-conquer algorithms. It should be memorized. This result is also very helpful in the design stage, since it can be used to predict the running time. Generalizations of this formula are given in the exercises.