1 Time Complexity

Big-O Notation: describes the **upper bound** of an algorithm's runtime. It tells us how the **worst-case** execution time of an algorithm grows relative to the input size n.

$$T(n) = O(f(n)) \text{ if:}$$
$$\text{there exist constants } c > 0$$
$$\text{there exist constants } n_0 > 0$$
$$\text{such that for all } n > n_0$$
$$T(n) < c \times f(n)$$

Complexity Rules

Let $T(n) = O(f(n))$ and $S(n) = O(g(n))$:

- **Addition:** $T(n) + S(n) = O(f(n) + g(n))$
- **Multiplication:** $T(n) \times S(n) = O(f(n) \times g(n))$
- **Composition:** $f_1 \circ f_2 = O(g_1 \circ g_2)$
- **If/Else Statements:** $\text{cost} = \max(c_1, c_2) \leq c_1 + c_2$
- **Max Function:** $\max(f(n), g(n)) \leq f(n) + g(n)$

Notable

- $\sqrt{n} \log n$ is $O(n)$
- $O(2^{2n}) \neq O(2^n)$
- $O(\log(n!)) = O(n \log n)$ → **Stirling's Approximation**
- $T(n-1) + T(n-2) + ... + T(1) = 2T(n-1)$

Big-Ω-notation: The lowest bound

$$T(n) = \omega(f(n)) \text{ if:}$$
$$\text{there exist constants } c > 0$$
$$\text{there exist constants } n_0 > 0$$
$$\text{such that for all } n > n_0$$
$$T(n) \geq c \times f(n)$$

1.1 Order of size

Function	Name
5	Constant
loglog(n)	Double log
log(n)	Logarithmic
$\log^2$(n)	Polylogarithmic
n	Linear
nlog(n)	Log-linear
n^3	Polynomial
$n^3\log(n)$	Polynomial
n^4	Polynomial
2^n	Exponential
2^{2n}	Exponential
n!	Factorial

1.2 Master Theorem

The **Master Theorem** is a method used to **analyse recurrence relations** of the form:

$$T(n) = \mathrm{aT}\left(\frac{n}{b}\right) + f(n)$$
$$f(n) = O(n^k \times \log^p\{n\})$$

Where:
- $T(n)$ represents the running time of an algorithm.
- a is the **number of subproblems**. $age1$
- $\frac{n}{b}$ is the **size of each subproblem**. $b > 1$
- $f(n)$ is the **extra work** done outside the recursive calls

We need to find two values $\log_b\{a\}$ and k

1. If $\log_b(a) > K$ then $O(n^{\log_b(a)})$
2. If $\log_b(a) = K$
   1. If $P > -1$ then $O(n^k \log^{p+1} n)$
   2. If $P = -1$ then $O(n^k \log(\log n))$
   3. If $P < -1$ then $O(n^k)$
3. If $\log_b(a) < K$
   1. If $P \geq 0$ then $O(n^k \log^p n)$
   2. If $P < 0$ then $O(n^k)$

1.2.1 Recurrence Relation

$$T(n) = \mathrm{aT}(n-b) + f(n)$$

Where:
- $T(n)$ represents the running time of an algorithm.
- a is the **number of subproblems**. $a > 0$
- $n - 1b$ is the **size of each subproblem**. $b > 0$
- $f(n)$ is the **extra work** done outside the recursive calls $f(n) = O(n^k)$ where $k \geq 0$
1. if $a < 1$ then $O(n^k)$ or $O(f(n))$
2. if $a = 1$ then $O(n^{k+1})$ or $O(n \times f(n))$
3. if $a > 1$ then $O\left(n^k \times a^{\frac{n}{b}}\right)$

1.3 Useful Math Formula

1. $a^x = m \iff x = \log_{a(m)}$ → definition of logarithm
2. $\log_{a(\mathrm{mn})} = \log_{a(m)} + \log_{a(n)}$ → product property
3. $\log_{a(\frac{m}{n})} = \log_{a(m)} - \log_{a(n)}$ → quotient property
4. $\log(m^n) = n\log(m)$ → power property
5. $\log_{a(b)} = \frac{1}{\log_{b(a)}}$
6. $\log_{b(a)} = \log_{c(a)} \times \log_{b(c)} = \frac{\log_{c(a)}}{\log_{c(b)}}$ → change of base property
7. $a^{\log_{a(x)}} = a$ → number raised to log
8. $\log_{a(a)} = 1$
9. $\log_{a(1)} = 0$
10. $\log_{a(\frac{1}{b})} = -\log_{a(b)}$
11. $a^{\log_{b(x)}} = x^{\log_{b(a)}}$ (when $x, a > 0$) → can take log on both sides to verify its true
12. $a^m a^n = a^{m+n}$
13. $\frac{a^m}{a^n} = a^{m-n}$
14. $\frac{1}{a^m} = a^{-m}$
15. $(a^m)^n = a^{\mathrm{mn}}$
16. $(\mathrm{ab})^m = a^m b^m$

2 Searching

2.1 Binary Search

- **Preconditions:** Array is of size n and is sorted
- **Postcondition:** $A[\text{begin}] = \text{key}$ if element is in the array
- **Invariant:** $(\text{end} - \text{begin}) \leq \frac{n}{2^k}$ in iteration k
- **Loop Invariant:** $A[\text{begin}] \leq \text{key} \leq A[\text{end}]$

2.2 Peak Finding

- **Limitations:**
  - Cannot find global peaks efficiently
  - Cannot handle duplicate values well

- **Invariant:** If we recurse in the right half, then every peak in the right half is a peak in the array

- **Correctness:**
  - There exists a peak in the range ([text{begin}, text{end}])
  - Every peak in ([text{begin}, text{end}]) is a peak in ([0, n - 1])

- For finding **steep peaks**, the worst time complexity could be (O(n)).

3 Sorting

3.1 Bogo Sort

Pseudocode

```
while deck is not sorted:
    shuffle(deck)
```

- It repeatedly shuffles the deck in no particular pattern until the array is sorted. `Worst Case: O(n * n!)`
- No Stable

3.2 Bubble Sort

```
repeat (until no swaps):
  for j <- 1 to n-1
    if A[j] > A[j+1] then swap(A[j], A[j+1])
```

- Worst Case: $O(n^2)$
- Best Case: $O(n)$
- At the end of iteration j, the biggest j items are correctly sorted in the final j positions of the array.
- Stable

3.3 Selection Sort

```
repeat (numOfElements - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
```

- Worst Case: $O(n^2)$
- Best Case: $O(n^2)$
- At the end of iteration j: the smallest j items are correctly sorted in the first j positions of the array.
- Not Stable

3.4 Insertion Sort

```
FOR i from 1 to n-1:
    key = A[i]  # Select the current element
    j = i - 1   # Start comparing with previous elements

    WHILE j ≥ 0 AND A[j] > key:
        A[j + 1] = A[j]  # Shift elements to the right
        j = j - 1  # Move to the previous index

    A[j + 1] = key  # Place the key in the correct position
```

- Worst Case: $O(n^2)$
- Best Case: $O(n)$
- At the end of iteration j: the first j items in the array are sorted order.
- Stable

3.5 Merge Sort

1. Divide: Split the array into two halves
2. Recurse: Sort the two halves
3. Combine: Merge the two sorted halves

```
MergeSort(A, n)
  if(n==1) then return;
  else:
    X <- MergeSort(A[0...n/2], n/2)
    Y <- MergeSort(A[n/2+1, n/2], n/2)
  return Merge(X,Y, n/2)
```

- Worst Case: $O(n \log n)$
- Best Case: $O(n \log n)$
- $T(n) = 2T(\frac{n}{2}) + \mathrm{cn}$
- Each recursive call sorts a subarray correctly before merging.
- Stable

3.6 Quick Sort

```
QuickSort(A, low, high):
    if low < high:
        pivotIndex = Partition(A, low, high)
        QuickSort(A, low, pivotIndex - 1)   // Sort left part
        QuickSort(A, pivotIndex + 1, high)  // Sort right part

Partition(A, low, high):
    pivot = A[high]    // Choose pivot (last element)
    i = low - 1        // Index of smaller element

    for j = low to high - 1:
        if A[j] ≤ pivot:
            i = i + 1
            Swap(A[i], A[j])

    Swap(A[i + 1], A[high])  // Move pivot to its correct place
    return i + 1  // Return pivot index
```

1. Choose a pivot (commonly the last element, first element, or a random element).
2. Move the pivot to the start of the array (if not already chosen as the first element).
3. Set two pointers:
   - `low` starts from index 1.
   - `high` starts from the length of the array - 1.
4. Move `low` right until it finds an element greater than or equal to the pivot.
5. Move `high` left until it finds an element smaller than or equal to the pivot.
6. If `low < high`, swap `arr[low]` and `arr[high]`.
7. Repeat steps 4-6 until `low ≥ high`.
8. Swap the pivot (first element) with `arr[high]`, placing it in its correct position.
9. Return the pivot's final index for further recursive sorting.

- Worst Case: $O(n^2)$ if always choose the worst pivot the first and last element
- Best Case: $O(n \log n)$
- $T(n) = 2T(\frac{n}{2}) + \mathrm{cn}$
- Not Stable

Invariant
- All elements below index `i` are smaller than or equal to the pivot.
- All elements above index `j` are greater than or equal to the pivot.
- The pivot is in its correct position after partitioning.
- Unprocessed elements exist between `i` and `j`, which will be checked next.

4 Tree

4.1 Critical Components of a Tree Data Structure

- Nodes:
  - A **node** is the fundamental unit of a tree.
  - It contains information stored in the node
  - Might have a reference
- Root Node:
  - The **topmost node** in a tree.
  - It has no parent
- Edges:
  - An edge is the connection between two nodes
  - Each edge represents a parent-child relationship
- Leaf Nodes
  - Nodes with no children
- Depth:
  - The number of edges from the root to that node
- Height:
  - The number of edges in the **longest path** from that node to a leaf.
- Degree of node
  - The number of children it has
- Degree of tree
  - The maximum degree of any node in the tree
- Siblings
  - If the same parent

4.2 Types of Tree

- **Binary Tree** – Each node has at most **two** children.
- Binary Search Tree (BST) – A binary tree where left < root < right.
- **Balanced Trees (AVL, Red-Black Tree, etc.)** – Trees that self-balance to maintain efficiency.
  - AVL Trees (Adelson-Velskii & Landis 1962)
- **Heap Tree** – A complete binary tree with specific ordering properties.

4.3 Binary Search Tree

1. All the keys in the left sub-tree are lesser than the key
2. All the keys in the right sub-tree are greater than the key

4.3.1 Basic Operations

4.3.1.1 Height

1. If leaf nodes:
   - Height = 0
2. Internal Nodes:
   - The max height of the left and right child + 1
3. Root Node:
   - The root node height is the overall height of the tree

4.3.1.2 Search Max and Min keys

Look for the max key it is the rightmost element of the BST the reason is based on the property of BST we know that the right sub-tree has a greater value than the root

Look for the min key it is the leftmost element of the BST the reason is based on the property of BST we know that the left sub-tree has a lesser value than the root

4.3.1.3 Search for a key

1. **Start at the Root Node**
   - Compare the target key with the root.
2. **Compare the Key**
   - If the key is **equal** to the current node -> return the value of the current node
   - If the key is **smaller** → Move to the **left subtree**.
   - If the key is **greater** → Move to the **right subtree**.
3. **Repeat Until Found or NULL**
   - If you reach a **NULL (empty node)**, the key **does not exist** in the BST.

Time Complexity Analysis of Searching in a BST
- Best Case: $O(1)$
- Average-Case: $O(\log(n))$
- Worst-Case: $O(n)$

4.3.1.4 Insert a key

1. **Start at the Root Node**
   - If the tree is **empty**, insert the key as the **root**.
2. **Compare the Key with the Current Node**
   - If the key is **equal** to the current node → ignore
   - If the key is **smaller** → Move to the **left subtree**.
   - If the key is **greater** → Move to the **right subtree**.
3. **Repeat Until NULL**
   - If you reach a **NULL (empty node)**, insert the key there =linebreak()

Time Complexity: O(h), where h is the height of the BST.

Auxiliary Space: O(1), this is because we don't have to store the stack

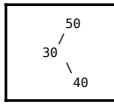
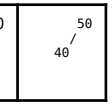
4.3.1.5 Delete a Key

4.3.1.5.1 Case 1: Node is a leaf (No children)

This is pretty straightforward forward if the node has no children just remove it pretty straightforward

4.3.1.5.2 Case 2: Node has One child

When there is only one child we just have to remove the node and replace its child.

	After deleting 30	

4.3.1.5.3 Case 3: Node Has Two Children

This is a bit complicated. When there are two children, we can't simply delete it so we have to find the [[=Inorder Successor]] to replace the node.

4.3.1.6 Inorder Successor

When we want to find the next n value ,we use the successor function. We know that BST is a key property all the elements on the left are smaller and all the elements are greater. Using the BST property we can easily find the successor.

1. Search for n in the BST
   - Move **left if n is smaller** than the current node.
   - Move **right if n is larger** than the current node.
2. If n has a right subtree
   - The successor is **the smallest value** in the **right subtree** (leftmost node).
3. If n does NOT have a right subtree
   - Start moving **upward**.
   - The successor is the **first ancestor where n is in its left subtree**.
   - If no such ancestor exists, n has **no successor**.

4.4 Tree Shape

Number Unique Binary Search Tree We can use the Catalan Numbers to find the number of unique Binary Search Trees that can be formed for n numbers elements

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

Number of different ways n elements can be inserted

$$n!$$

4.5 Traversal

Inorder	Preorder	Postorder	Level-Order
Algorithm: • Traverse the **left subtree**. • Visit the **root node**. • Traverse the **right subtree**.	Algorithm: • Visit the **root node**. • Traverse the **left subtree**. • Traverse the **right subtree**.	Algorithm: • Traverse the **left subtree**. • Traverse the **right subtree**. • Visit the **root node**.	Algorithm: • Visit the **root node**. • Visit all nodes at **level 1**. • Visit all nodes at **level 2**, and so on.

5 AVL

The time complexities of all operations (search, insert and delete, max, min, floor and ceiling) become O(Log n). This happens because the height of an AVL tree is bounded by O(Log n). In case of a normal BST, the height can go up to O(n).

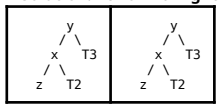
5.1 Define the Invariant

1. A **node** v is **height-balanced** if the absolute difference between the heights of its left and right subtrees is at most **1**: $|\text{v.left.height} - \text{v.right.height}|$
2. A **binary search tree (BST)** is considered **height-balanced** if **every** node in the tree satisfies the height-balanced condition.

5.2 Rebalancing

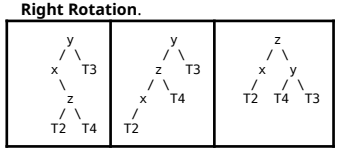
5.2.1 Case 1: Left Heavy

- Occurs when a **node becomes unbalanced due to excessive height in the left subtree**.
- **Solution:** Perform a **Right Rotation** (Single Rotation).



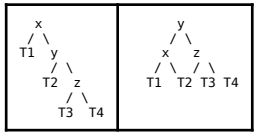
5.2.2 Case 2: Left Right Heavy

- Occurs when the **left subtree of a node is right-heavy**.
- **Solution:** Perform a **Left Rotation on the left child**, followed by a **Right Rotation**.



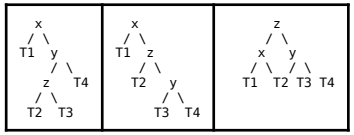
5.2.3 Case 3: Right Heavy

- Occurs when a **node becomes unbalanced due to excessive height in the right subtree**.
- **Solution:** Perform a **Left Rotation** (Single Rotation).



5.2.4 Case 4: Right Left Heavy

- Occurs when the **right subtree of a node is left-heavy**.
- **Solution:** Perform a **Right Rotation on the right child**, followed by a **Left Rotation**.



5.3 Inserting and Deleting Key

Insert

To insert into an AVL is the same as inserting into any BST. Once you have inserted you have to update the height of the nodes. If the nodes are unbalanced we just have to rebalance them based on whether it is greater than 1 or –1. 1 means is left heavy and –1 means it is right heavy. Once we figured that out we need to know where the key was inserted in the respective subtree.

We only have to balance the lowest unbalanced node in the root-to-leaf path. Once the lowest unbalanced node is **rotated** (single or double rotation), all its ancestors above remain **balanced**. This is because AVL trees **always maintain the height balance property** at every level.

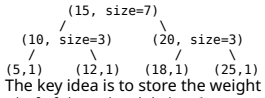
Deleting

The logic for deleting a key starts with the same logic as deleting a key from BST. So after deleting the key we go upward and find the first unbalanced tree and perform rotation

6 Dynamic Order Statistics

6.1 Select

To find the n element in ascending order.



The key idea is to store the weight in each node

`w(v.left) + w(v.right) + 1`

Run-time: `O(log n)`

Algorithm

- Start at the root of the BST.
- Compute `left_size`:
  - left_size = size of the left subtree (number of nodes in left subtree).
- Compare `k` with `left_size + 1` (1-based index):
  - If `k == left_size + 1`, return the current node's value (this is the k-th smallest element).
  - If `k < left_size + 1`, search in the left subtree.
  - If `k > left_size + 1`, search in the right subtree, adjusting `k` to `k - (left_size + 1)` because we are skipping `left_size + 1` nodes.

6.2 Rank

The purpose of this method is to find the rank of `x` node.

Algorithm

- The **rank of** `x` is the number of nodes **less than or equal to** `x`.
- If `x == v.data`, rank = `left.size + 1`.
- If `x < v.data`, search in the **left subtree**.
- If `x > v.data`, search in the **right subtree** and **add left.size + 1** to the rank.

Pseudocode

```
rank(node)
  rank = node.left.weight + 1;
  while (node != null) do
    if node is left child then
      do nothing
    else if node is right child then
      rank += node.parent.left.weight + 1;
    node = node.parent;
  return rank;
```