I SOLID

I.1 Single Responsibility Principle (SRP)

- **Definition**: A class should have only one reason to change, meaning it should only handle one responsibility.
- **Violation**: A `User` class handles both storing user data and sending emails to new users. This means it's responsible for managing data and handling email communication, which are two distinct tasks.
- **Fix**: Create separate classes: a `UserDataService` to handle user data and an `EmailService` to handle email communication.
- **Explanation**: By splitting the responsibilities, we make each class easier to maintain and understand. If there's a change in how we send emails, we only modify the `EmailService`, leaving the `UserDataService` untouched.

I.2 Open/Closed Principle (OCP)

- **Definition**: Classes should be open for extension but closed for modification, allowing new functionality without altering existing code.
- **Violation**: A `DiscountCalculator` class has logic to apply discounts based on customer types like "Regular" and "VIP." When a new customer type, "Premium," is introduced, the class needs modification to add the new discount rule.
- **Fix**: Create individual discount strategies for each customer type that can be applied without modifying the `DiscountCalculator`. For example, use separate classes like `RegularDiscount`, `VIPDiscount`, and `PremiumDiscount` that `DiscountCalculator` can use without needing changes.
- **Explanation**: This makes it easy to add new discounts without modifying existing code, reducing the risk of bugs and keeping the system stable as it grows.

I.3 Liskov Substitution Principle (LSP)

- **Definition**: Subtypes should be substitutable for their base types. If a function uses a base class, it should work with any subclass without issue.
- **Violation**: Suppose there's a `Bird` class with a `fly` method. If we create a `Penguin` class that inherits from `Bird` but overrides the `fly` method to throw an error (because penguins can't fly), it breaks the substitution.
- **Fix**: Use a different hierarchy or interface that better represents different types of birds. For instance, create a `FlyingBird` class that includes `fly` only for birds that actually fly, and a `NonFlyingBird` class for those that don't.
- **Explanation**: This maintains predictable behavior. By ensuring subclasses behave consistently with the base class, the code is easier to understand and maintain.

I.4 Interface Segregation Principle (ISP)

- **Definition**: A class should not be forced to implement interfaces it doesn't use. Instead, create smaller, more specific interfaces.
- **Violation**: Imagine an interface called `Worker` with methods like `startWork`, `stopWork`, and `submitReport`. If some workers only need `startWork` and `stopWork` and have no use for `submitReport`, they're forced to implement an unnecessary method.
- **Fix**: Split the `Worker` interface into smaller interfaces like `Workable` (with `startWork` and `stopWork`) and `Reportable` (with `submitReport`). Classes can implement only the interfaces they need.
- **Explanation**: By dividing interfaces into focused ones, classes only depend on what they actually need. This keeps classes and interfaces lean, reducing complexity and making them easier to maintain.

I.5 Dependency Inversion Principle (DIP)

- **Definition**: High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details.
- **Violation**: A `Report` class directly creates a `PDFGenerator` to generate reports. This tightly couples the `Report` to the specific `PDFGenerator`, making it difficult to switch to another report generator without modifying `Report`.
- **Fix**: Define an interface, such as `ReportGenerator`, that `PDFGenerator` implements. Now, `Report` depends on `ReportGenerator` rather than the concrete `PDFGenerator`, allowing flexibility.
- **Explanation**: By depending on abstractions, the `Report` class can work with any `ReportGenerator` implementation. This makes the system more modular and flexible for changes, as new types of report generators can be added without altering the `Report` class.

II Recursion

- **Tail recursion** uses less memory than non-tail recursion.

II.1 Tail Recursion

II.1.1 Cost of recursion

- Each call to a function adds another frame on the stack
- Each frame contains local variables and parameters and where to return the result

II.1.2 Reducing what needs to be stored

- If we can guarantee we won't need them, we can free the memory for the local variables and parameters.
- We won't need them as long as **we do not use them after the recursive call**.
- Tail recursive functions have the **recursive call as the last thing the function does before it returns**.

III Complexity

III.1 Master Theorem

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$T(x) = \begin{cases} \Theta(n^d) \text{ if } a<b^d \\ \Theta(n^d \log(n)) \text{ if } a=b^d \\ \Theta\left(n^{\log_b(n)}\right) \text{ if } a>b^d \end{cases}$$

a: number of subproblems in the recursion
n/b: size of each subproblem
d: the exponent in the cost of the work done outside the recursive calls, specifically in the non-recursive part of the algorithm (like splitting or merging the problem)

e.g. **Binary Search** a=1 b=2 d=0 $\log(n)$ **Merge Sort** a=2 b=2 d=1 $n\log(n)$

III.2 Bound

III.2.1 Big O - Upper Bound

$f(n) = O(g(n))$ iff (if and only if)
$\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \text{such that } \forall n \geq n_0 : f(n) \leq cg(n)$

III.2.2 Big Ω - Lower Bound

$f(n) = \Omega(g(n))$ iff (if and only if)
$\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \text{such that } \forall n \geq n_0 : f(n) \geq cg(n)$

III.2.3 Big Θ - Tight Bound

$f(n) = \Theta(g(n))$ iff (if and only if)
$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

III.2.4 Little o upper Bound

$\exists c \in \mathbb{R}^*, \exists n_0 \in \mathbb{N}, \text{such that } \forall n \geq n_0 \text{ such that } f(n) < cg(n)\left(\left|\frac{f(n)}{g(n)}\right| < c\right)$
In other words: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

III.3 Quiz

III.3.1

- if $f(n) = \Theta(g(n))$ then $g(n) = \Theta(f(n))$
- if $f(n) = O(n)$ then $f(n) = o(n)$
- if $f(n) = o(n)$ and $g(n) = o(n)$ then $f(n) + g(n) = o(n)$
- $O(n)$ is the complexity in the worst case
- if $f(n) = \Omega(g(n))$ and $f(n) = O(h(n))$, then $g(n) \leq h(n)$ for all $n > 0$

III.3.2

$o(n^2)$ = $\sqrt{n}$ / 1 / $n\log(n)$ / n

III.3.3

$\mathbf{1 < \log(n) < \log(n^2) < (\log(n))^2 < 5n\log(\log(n)) + 100 < n < n\log(n) < n^{1.5} + 1000000 < n^2 < n^3 < 2^n < 3^n < n! < n^n}$

III.3.4

- If $f(n) = O(g(n))$, then $f(n) = o(g(n))$. The opposite is not true, because Little o assumes that $f(n)$ and $g(n)$ can't be of the same order.
- If $f(n) = o(g(n))$, then $f(n) = O(g(n))$. The opposite is not true, because Little o assumes that $f(n)$ and $g(n)$ can't be of the same order.
- $f(n) = o(g(n))$ if and only if $f(n) = O(g(n))$.
- If $f(n) = O(g(n))$ it does not mean $f(n) = o(g(n))$, and if $f(n) = o(g(n))$ it does not mean $f(n) = O(g(n))$.

III.3.5

- $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- $n = o(n^2)$ and $n = O(n^2)$
- If $f(n) = \Omega(n)$, then $f(n) \geq n$ for all n starting from some n_0
- If $f(n) = o(g(n))$ and $g(n) = O(h(n))$, then $f(n) = o(h(n))$

III.3.6

Suppose we need to find an occurrence of the array of size m inside a larger array of size n. We know that the array of size n is sorted and consists of unique elements. What will be the time complexity of the optimal algorithm?

- $\Theta(\log(n) + m)$

IV Sorting

IV.1 Selection

不断从未排序部分中选择最小（或最大）元素，并将其与第一个未排序元素交换，每次迭代后将排序边界向前移动一步 repeatedly selects the smallest (or largest) element from the unsorted portion of an array and swaps it with the first unsorted element, moving the sorted boundary one step forward with each iteration

```
for (int i = 0; i < array.size(); i++) {
  // Find min element from i to n-1
  for (int j = i + 1; i < array.size(); j++) {
  }
  // Swap elements at index i and min elements
}
```

IV.2 Insertion

逐一将未排序部分的下一个元素插入到（不断交换）已排序部分的正确位置，逐步构建出一个已排序的数组 builds the sorted array one element at a time by repeatedly taking the next unsorted element and inserting it into its correct position in the sorted part

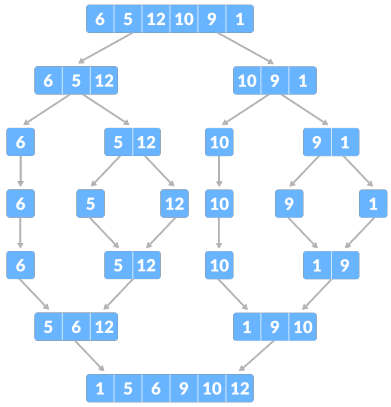
```
for (int i = 1; i < array.size(); i++) {
  for (int j = i; j > 0; j--) {
    if (array.at(j) < array.at(j - 1)) {
      swap(array.at(j - 1), array.at(j));
    } else break;
  }
}
```

IV.3 Bubble

repeatedly compares adjacent elements and swaps them if they are in the wrong order, causing larger elements to "bubble" to the end of the array. With each pass through the list, the largest (or smallest) element settles in its correct position, and this process continues until the entire array is sorted.

```
for (int i = array.size() - 1; i > 0; i--) {
    for (int j = 0; j < i; j++) {
      if (array.at(j) > array.at(j + 1)) {
        swap(array.at(j), array.at(j + 1));
      }
    }
  }
```

IV.4 Merge




```
if (array.size() == 1) return array;
// Sort left and right subarrays
int mid = array.size() / 2;
// ... Define left and right arrays
vector<int> sortedRightArray = sort(rightArray);
vector<int> sortedLeftArray = sort(leftArray);
// Merge left and right subarrays
vector<int> result;
int l = 0, r = 0;
while (l < sortedLeftArray.size() && r <
sortedRightArray.size()) {
  if (sortedLeftArray.at(l) <
sortedRightArray.at(r)) {
    result.push_back(sortedLeftArray.at(l)); l++;
  } else {
    result.push_back(sortedRightArray.at(r)); r+
+;
  }
}
// ... Add remaining elements form
sortedLeftArray or sortedRightArray
return result;
```

IV.5 Quick

```
if (start >= end) return;
// Select the last element as pivot
int pivot = array.at(end);
int pivotIndex = start;
for (int i = start; i < end; i++) {
  if (array.at(i) < pivot) {
    swap(array.at(i), array.at(pivotIndex));
    pivotIndex++;
  }
}
swap(array.at(pivotIndex), pivot);
sort(array, start, pivotIndex - 1);
sort(array, pivotIndex + 1, end);
```

V Linked List & Friend Class

```
template <class T>
class Node {
  T data;
  Node<T>* link;
  Node(T data) {
    this->data = data;
  }
public:
  template <class U>
  friend class LinkedList;
};

template <class T>
class LinkedList {
  Node<T>* head;
  bool isEmpty() {
    return this->head == NULL;
  }
public:
  LinkedList() {
    this->head = NULL;
  }
  void insert(T value) {
    Node<T>* newNode = new Node<T>(value);
    newNode->link = this->head;
    this->head = newNode;
  }
  void traverse() {
    if (isEmpty()) {
      cout << "List is empty\n";
      return;
    }
    Node<T>* tmp = this->head;
    while (tmp != NULL) {
      cout << tmp->data << " ";
      tmp = tmp->link;
    }
    cout << endl;
  }
};
```

V.1 Quiz

V.1.1

- Linked lists require at least two methods for insertion, one to cover addition at the front and another for addition at an arbitrary position.
- Linked lists have $O(1)$ access to any node.
- Linked lists are not guaranteed to have their nodes located in adjacent blocks of memory.
- Linked lists have an aggregation relationship with nodes.

V.1.2

- A singly linked list has all the nodes linking in one direction.
- A singly linked list can have a tail as well as head. When it does, insertion and deletion at both either end becomes $O(1)$
- Insertion/deletion from the front of a singly linked list is $O(1)$.
- A singly linked list cannot have a tail member variable that records where the last node is in memory, as it violates the definition of the singly linked list.

VI Stack & Queue

VI.1 Stack

- LIFO (Last-in/First-out)
- implemented using linked lists or dynamic arrays
- `push(item)` `pop()` `isEmpty()`
- only have access to the top of the stack

VI.2 Queue

- FIFO (First-in/First-out)
- usually implemented using linked lists
- `enqueue(item)` `dequeue()` `isEmpty()`
- You have access to both the front and back of the queue.

VI.3 Quiz

VI.3.1

- Removing an element from a queue requires the queue to search through all of the remaining elements to update their positions and ensure that the queue is properly ordered, resulting in a time complexity of O(n).
- Solving the Tower of Hanoi maps well to the queue data structure.
- Queues are FIFO/LILO.
- Since queues involve operations at both ends, they must be implemented using a doubly linked list.

VI.3.2

- A priority queue implemented using a single queue where each node has a priority value, has both $O(n)$ insertion and $O(n)$ removal.
- A priority queue with n elements that is implemented using multiple queues (with a queue for each priority) have $O(n)$ removal.
- A priority queue with n elements that is implemented using multiple queues (with a queue for each priority) have $O(1)$ insertion.
- A priority queue implemented using a single queue where each node has a priority value, has either $O(n)$ insertion or $O(n)$ removal.

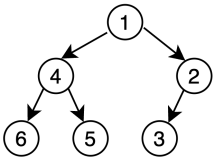
VI.3.2.1 Explanation

- Priority queues with n nodes implemented using multiple queues, have:
  - $O(1)$ insertion, as at the time of insertion you know what priority the node will have so you can simply find which queue it should belong to in $O(1)$, and then insert to the back of that queue in $O(1)$.
  - $O(m)$ removal, where m is the number of queues. This is because we need to run the isEmpty function on up to m queues.
- Priority queues implemented using one queue will either have either $O(n)$ insertion or removal. Because you have to check who the highest priority is either:
  - At insertion, to place them in the appropriate position. They "cut" in line.
  - At removal, to take out the appropriate item.

VII Tree

VII.1 Tree Traversals

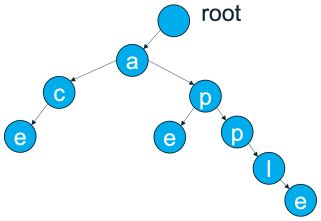
- Level Order (level by level) 1, 4, 2, 6, 5, 3
- In-order (left, root, right) 6, 4, 5, 1, 3, 2
- Pre-order (root, left, right) 1, 4, 6, 5, 2, 3
- Post-order (left, right, root) 6, 5, 4, 3, 2, 1



VII.2 BST Delete

- **0 child**: just delete
- **1 child**: set parent's pointer to point to the one child
- **2 children**: replace with in-order successor child (leftmost child of right subtree)

VII.3 Trie

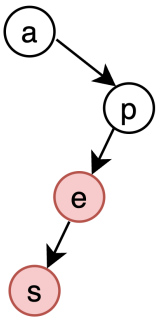


- Tries are a tree data structure that are particularly suited for searching for keys that begin with a specific prefix
- Usually, these keys are strings
- Each path from root represents one key

```
struct TrieNode {
  bool isEndOfWord;
  vector<TrieNode*> childern;
}
```

VII.3.1 Adding keys

- While the key forms a path in the trie, follow the trie while next key character matches the characters in trie
- Add remaining non-matching key characters to tree, mark node with last character as end of word
- Example: Insert "ape" and "apes" - nodes coloured red have `isEndOfWord == true`



VII.3.2 Deleting key

```
// If root is nullptr, key is not in the Trie
if (!root) return nullptr;
// If we've reached the end of the key
if (depth == key.length()) {
  // This node is no longer an end of a word
  if (root->isEndOfWord) root->isEndOfWord =
false;
  // If node has no children, delete it (free
memory) and return nullptr
  if (root->children.empty()) {
    delete root;
    return nullptr;
  }
  return root;
}
// Recursive case for deleting in child
char ch = key[depth];
root->children[ch] = deleteKey(root-
>children[ch], key, depth + 1);
// If no children and not end of word, delete
this node
if (root->children.empty() && !root->isEndOfWord)
{
  delete root;
  return nullptr;
}
return root;
```

VII.4 Quiz

VII.4.1
The height of a binary tree can not have a tight bound

VII.4.2
Running time of searching a Binary Search Tree with n nodes: $\Theta(\text{height})$

VII.4.3
The minimum number of nodes that could be in a balanced binary tree of height h: 2^h

VII.4.4
In a red black, we keep the same number of black nodes between a given node and any other node. The reason for this is: to ensure search is $\log(n)$

VII.4.5
For which of the following would a Trie be a good choice of Tree data structure
- Storing key-value pairs with string keys
- Storing a collection of unsorted strings
- Storing key-value pairs with integer keys
- Storing a collection of unsorted integers

VIII Heap

VIII.1 Binary Tree in array
- If a node is at index i
- Its left child is at $2 * i$
- Its right child is at $2 * i + 1$
- Its parent is at $\lfloor \frac{i}{2} \rfloor$

VIII.2 Operations
Swap the node with the largest(smallest) child until the heap property is satisfied
- **Insert**: bottom to top
- **Delete**: top to bottom
- **Heapify**: bottom to top (back to front)

VIII.3 Quiz

VIII.3.1
- A min-heap is a binary tree
- In a min-heap node after deletion, the maximum levels that a parent will sift down during heapify is O(log(n))
- A heap can become an unbalanced tree
- When traversing by level a Heap implemented as a tree, the values will be in order from smallest to largest.

VIII.3.2
When deleting the root node, which of the following should take place to restore the Heap
- sift up each of the leaf nodes
- copy the value of the last element to the root and heapify the root element
- copy the value of the last element to the root and delete the last element, heapify the root element.
- copy the value to the root and heapify each parent node starting from the bottom right parent working across tree to the left, level by level until the root node is heapified

VIII.3.3
maximum depth of a Heap implemented as a Tree: $O(\log(n))$

VIII.3.4
- Implementing a Heap as a vector will allow faster insertion deletion of elements compared to implementing as a Tree
- Implementing a Heap as a vector uses less memory compared to implementing as a Tree
- Implementing a Heap as a vector better matches the Heap abstraction compared to implementing as a Tree
- Implementing a Heap as a vector results in a lower run-time complexity for all operations compared to implementing as a Tree

	Best	Average	Worst	Space	Stable
Selection	$O(n^2)$			$O(1)$	
Insertion	$O(n)$	$O(n^2)$		$O(1)$	✓
Bubble	$O(n^2)$			$O(1)$	✓
Merge	$O(n\log(n))$			$O(n)$	✓
Quick	$O(n\log(n))$		$O(n^2)$	$O(\log(n))$	
Bucket	$O(n)$	$O(n+k)$ (k: numbers of buckets)	$O(n^2)$	$O(n+k)$	✓
Counting	$O(n+k)$ (k: range of the input values, max-min)			$O(n+k)$	✓
Heap	$O(n\log(n))$			$O(1)$	

	Worst			Average			Space
	Insert	**Delete**	**Search**	**Insert**	**Delete**	**Search**	
Vector Ordered	$O(n)$		$O(\log(n))$	$O(n)$		$O(\log(n))$	$O(n)$
Vector Unordered	$O(1)$*	$O(n)$		$O(1)$*	$O(n)$		
Linked List	$O(1)$			$O(1)$			
Binary Search Tree	$O(n)$			$O(\log(n))$			
Balanced BST (RBT)	$O(\log(n))$						
Priority Queues	**Insert**	**RMH****	**Peek**	**Insert**	**RMH****	**Peek**	
Linked List	$O(n)$	$O(1)$		$O(n)$	$O(1)$		
Heap	$O(\log(n))$		$O(1)$	$O(1)$	$O(\log(n))$	$O(1)$	

*: Amortised - ie over a sequence of this operation. Resizing vector $O(n)$
**: Remove highest priority