**A.GIT**

1. **Repo:**

* Repo (Repository) là một kho lưu trữ kỹ thuật số tập trung được dùng nhằm thực hiện và quản lý các thay đổi đối với mã nguồn của các ứng dụng.

(Hiểu đơn giản, repo là một kho lưu trữ những thay đổi của một dự án)

* Có 2 loại repo:
  + - Local Repository: được lưu trữ trên máy tính cá nhân.
    - Remote Repository: được lưu trữ ở một sever nào đó (Vd: Github,…)

1. **Branch:**

* Branch là một phần của repo, có thể được hiểu là một khu vực độc lập. Một repo có thể được coi là một cái cây và nhánh chính (main/ master branch) được coi như thân cây (nơi giữ mã nguồn ổn định nhất), nối liền với nó là các cành cây (nhánh con). Mỗi nhánh con hoạt động độc lập mà không ảnh hưởng đến thân cây hoặc các nhánh khác.

Khi công việc trên nhánh con hoàn thành, nó có thể được nối với thân cây (merge vào nhánh chính).

* Cách tạo branch mới:
  + $ git branch <tên branch>
  + $ git checkout -b <tên branch>

1. **Merge Request (Pull Request):**

* Merge Request là một quá trình trong Git yêu cầu hợp nhất (merge) một nhánh này vào một nhánh khác.
* Các loại Merge Request:
  + Feature Merge Request :

1. **Các cách merge branch vào branch chính:**

* Cách 1: Merge trực tiếp
  + git checkout main

git merge <tên branch>

* Cách 2: Merge bằng Pull Request
* Cách 3: Rebase trước khi merge
* git checkout <tên-branch>
* git rebase main
* git checkout main
* git merge <tên-branch>

**Sự khác nhau giữa Git Merge và Git Rebase**

Cả hai đều được dùng để tích hợp các thay đổi từ nhánh này sang nhánh khác

* **Merge:**
  + **Cách hoạt động:**
    - Git sẽ tạo một merge commit mới để kết hợp lịch sử commit của hai branch.
    - Lịch sử của các commit từ hai branch vẫn được giữ nguyên, bao gồm cả thông tin về việc hợp nhất

**Vd:**

Sau khi merge:

main: A --- B --- C ----------- F

\ /

feature: D ----- E --------

Trước khi merge:

main: A --- B --- C

\

feature: D --- E

* + - * Commit F là merge commit.
      * Lịch sử của hai nhánh main và feature vẫn được giữ nguyên
* **Rebase:**
  + **Cách hoạt động:**
    - Git sẽ di chuyển các commit từ branch hiện tại lên trên branch chính bằng cách tạo ra các lịch sử commit mới
    - Nó sẽ thay đổi thứ tự commit

**Vd:**

Sau khi rebase:

main: A --- B --- C --- D' --- E'

Trước khi rebase:

main: A --- B --- C

\

feature: D --- E

* + - * Hai commit D và E được đẩy lên sau B, tạo ra hai commit mới là D’ và E’

**B.**

**1. var, let, const:**

* **var:**
  + Phạm vi (Scope) của **var**: *Phạm vi toàn cục* (global) khi biến **var** được khai báo bên ngoài một function. *Phạm vi cục bộ/ hàm* nếu biến **var** được khai báo bên trong hàm.
  + Biến **var** có thể khai báo lại và update: Có nghĩa là chúng ta có thể khai báo lại một biến đã được khai báo trước đó với cùng một scope mà không bị lỗi.

Vd: **var** myName = “Dat”;

**var** myName = “TrinhDat”;

* + Hoisting (Hành vi ngầm định) của **var**: Khi ta khai báo một biến hoặc một function bằng **var** thì chúng sẽ được hoisted (đẩy lên đầu scope trước khi đoạn mã được thực thi) và đồng thời biến đó được khởi tạo với giá trị là *undefined*. Cho nên chúng ta có thể sử dụng biến đó trước khi khai báo mà không gặp lỗi.

Vd: console.log(myName);

**var** myName = “Dat”;

Thì thực chất đoạn code sẽ là như này:

**var** myName;

console.log(myName); // myName is undefined

myName = ”Dat”;

Ở đây biến **var** đã được đẩy lên đầu scope và khởi tạo với một giá trị là undefined (không xác định).

* **let:**
  + Phạm vi (Scope) của **let**: **let** có scope là block scope (Block là một đoạn mã được giới hạn bởi {} ). Một biến khi được khai báo bằng **let** bên trong một block thì chỉ có thể sử dụng trong block đó.
  + **let** có thể update nhưng không thể khai báo lại:

Vd:

* Có thể khai báo như thế này:

**let** myAge = 20;

myAge = 19;

* Nhưng không được khai báo như này:

**let** myAge = 20;

**let** myAge =19;

* + Hoisting của **let:** khi báo bằng **let** cũng được hoisted (di chuyển lên đầu scope) nhưng không được khởi tạo giá trị mặc định ban đầu. Do đó không thể sử dụng biến trước khi khai báo.
* **const:**

Biến được khai báo với const chứa giá trị hằng (giá trị không thể thay đổi trong khi thực hiện)

* + Scope của **const:** Scope của **const** là Block scope.
  + **const** không thể được cập nhật và khai báo lại: Giá trị của biến được khai báo bằng **const** không thể được thay đổi và không được khai báo trùng tên trong cùng một phạm vi tồn tại.

Vd:

**const** myMark = 10;

myMark = 9; // Error (Lỗi do thay đổi giá trị của biến myMark)

Hoặc:

**const** myMark = 10;

**const** myMark = 9; // Error (Lỗi do myMark đã được khai báo trước đó).

🡪Do đó mọi biến được khai báo với **const** phải được khởi tạo giá trị khi khai báo.

* Tuy nhiên, trong khi đối tượng thì không thể thay đổi giá trị hay cập nhật, thì thuộc tính của đối tượng được khai báo bằng **const** thì có thể.

Vd:

**const** person = {

age = 19,

mark = 10

};

person.age = 20;

**Sự khác biệt trong cơ chế của var, let và const:**

* **var** có global scope/ function scope, **let** và **const** có block scope.
* Các biến **var** có thể update và khai báo lại trong scope của nó, **let** có thể update nhưng không thể khai báo lại, **const** không thể update cũng không thể khai váo lại.
* Cả ba cách khai báo đều sở hữu cơ chế hoisting nhưng riêng **var** được khởi tạo giá trị *undefined* còn **let** và **const**  thì không được khởi tạo giá trị.
* **var** và **let** có thể khai báo mà không cần khởi tạo còn **const** thì phải được khởi tạo trong quá trình khai báo.

**2. Các thuật toán tìm kiếm:**

1. **Bubble Sort:**

* Cách thức hoạt động:
  + So sánh từng cặp phần tử trong mảng
  + Hoán đổi vị trí của hai phần tử nếu phần tử đứng trước lớn hơn phần tử đứng sau
  + Lặp lại quá trinh nhiều lần cho đến khi mảng được sắp xếp.
* Cài đặt thuật toán:

function **bubbleSort**(arr) {

let n = arr.length;

for (let i = 0; i < n - 1; i++) {

for (let j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

[arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];

}

}

}

return arr;

}

1. **Selection Sort:**

* Cách thức hoạt động:
  + Tìm phần tử nhỏ nhất để đưa lên đầu mảng.
  + Lặp lại quá trình cho đến khi mảng được sắp xếp.
* Cài đặt thuật toán:

function **selectionSort**(arr) {

let n = arr.length;

for (let i = 0; i < n - 1; i++) {

let minIndex = i;

for (let j = i+1; j < n ; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

[arr[i], arr[minIndex]] = [arr[minIndex], arr[i]];

}

return arr;

}

1. **Insertion Sort:**

* Cách thức hoạt động:
  + Bắt đầu xét từ phần tử thứ hai của mảng, chọn một phần tử làm key (số làm chuẩn).
  + So sánh key với các phần tử trước đó và chèn nó vào đúng vị trí.
  + Lặp lại với các phần tử tiếp theo cho đến hết.
* Cài đặt thuật toán:

function **insertionSort**(arr) {

let n = arr.length;

for (let i = 1; i < n ; i++) {

let key = arr[i];

let j = i – 1;

while (j>=0 && arr[j] > key){

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

return arr;

}

1. **Quick Sort:**

* Cách thức hoạt động:
  + Chọn một phần tử làm điểm đánh dấu (pivot)
  + Chia mảng làm hai mảng con bằng cách so sánh với điểm đánh dấu. Mảng bên trái gồm các phần tử nhỏ hơn điểm đánh dấu và mảng bên phải gồm các phần tử lớn hơn điểm đánh dấu.
  + Đệ quy để chọn lại pivot cho hai mảng trái và phải.
  + Gộp các phần tử lại với nhau để tạo thành mảng đã được sắp xếp.
* Cài đặt thuật toán:

function **quickSort**(arr) {

if(arr.length <=1) return arr;

const pivot = arr[Math.floor(arr.length / 2)]; // chọn phần tử giữa mảng làm pivot

//Phân chia mảng theo pivot

const left = arr.filter( number => number < pivot);

const right = arr.filter( number => number > pivot);

return(...) //đệ quy hai mảng

}