1. Các đặc điểm **Set Interface**

* Không cho phép lưu trũ giá trị trùng lặp tức là mỗi phần tử lưu trong set là duy nhất .
* Không lưu trữ theo thứ tự thêm vào của phần tử và cũng không có cơ chế đánh index .
* Hiệu suất cao cho cá theo tác tìm kiếm hay thêm xoá phần tử trong set.
* Set cũng được cấp phát đọng bộ nhớ vầ có thể lưu giá trị null nhưng chỉ luuw duy nhất 1 giá trị null.

1. Các class triển khai từ Set Interface và trường hợp sử dụng :

2.1 HashSet:

* Được sử dụng để lưu trữ các phần tử duy nhất . Nó dựa trên HashMap để quản lý dữ liệu và hash code để lưu trữ các phần tử.
* Ưu điểm :
  + Tốc độ tìm kiếm phần tử trong mảng nhanh do lưu trữ bằng hashcode .
  + Giảm bộ nhớ do không cần lưu trữ thứ tụe hoặc liên kết các phần tử từ đó giúp việc thêm xoá nhanh hơn.
* Nhược điểm
  + Có thể xung đột hashcode khi lưu trữ .
  + không an toàn trong môi trường đa lường
* Cơ chế hoạt động :
  + Thêm phần tử :
    - **Tính mã băm**: HashSet sẽ tính toán giá trị mã băm (hashCode()) của phần tử cần thêm.
    - **Xác định vị trí bucket**: hashCode được dùng để xác định bucket mà phần tử sẽ được lưu trữ.
    - **Kiểm tra xung đột**: Nếu bucket đã có phần tử khác , HashSet sẽ sử dụng phương thức equals() để kiểm tra xem phần tử mới có trùng với phần tử đã tồn tại không:
      * Nếu phần tử mới và phần tử cũ bằng nhau (theo equals()), HashSet sẽ không thêm phần tử mới.
      * Nếu không, phần tử mới sẽ được thêm vào cùng bucket đó (tạo ra một danh sách liên kết trong bucket nếu cần).
  + Xoá phần tử :
    - Tính mã băm: HashSet tính toán mã băm của phần tử cần xóa.
    - Xác định bucket: Sử dụng mã băm để tìm vị trí bucket chứa phần tử.
    - Kiểm tra và xóa: HashSet duyệt qua tất cả các phần tử trong bucket đó, sử dụng phương thức equals() để tìm phần tử cần xóa.
    - Nếu tìm thấy phần tử cần xóa, HashSet sẽ xóa nó khỏi bucket.
  + Tìm kiếm phần tử :
    - Tính mã băm: Tính toán mã băm của phần tử cần tìm kiếm.
    - Xác định bucket: Sử dụng mã băm để xác định bucket mà phần tử có thể được lưu trữ.
    - Kiểm tra bằng equals(): HashSet kiểm tra từng phần tử trong bucket đó bằng cách sử dụng phương thức equals():
      * Nếu tìm thấy phần tử tương đương, trả về true.
      * Nếu không tìm thấy, trả về false.
  + Duyệt phần tử : sử dụng foreach hoăc Iteẻator , tuy nhiên set không trả về đúng thứ tự khi đã lưu.
* Trường hợp sử dụng : **HashSet** rất phù hợp cho các trường hợp cần lưu trữ các phần tử duy nhất và yêu cầu hiệu suất cao khi thêm/tìm kiếm .

2.2 LinkedHashSet:

* Tương tự như hashSet tuy nhiên có cơ chế duy trì thứ tự của phần tử khhi thêm vào trong mảng .
* Ưu điểm :
  + Ưu điểm lớn nhất của LinkedHashSet là nó duy trì thứ tự các phần tử theo thứ tự chúng được thêm vào. Điều này khác với HashSet, vốn không bảo đảm thứ tự phần tử.
* Nhược điểm
  + Tăng kích thước bộ nhớ .
  + Chậm hơn hashSet do duy trì thứ tụ.
* Cơ chế hoạt động :
  + Thêm phần tử :
    - Kiểm tra trùng lặp: Trước khi thêm, nó sử dụng bảng băm (HashTable) để tính toán hash code của phần tử và kiểm tra xem phần tử này đã tồn tại hay chưa. Nếu phần tử đã tồn tại, nó sẽ không thêm vào.
    - Duy trì thứ tự chèn: Bên cạnh việc lưu trữ phần tử trong bảng băm, nó cũng sử dụng một doubly linked list (danh sách liên kết kép) để theo dõi thứ tự các phần tử được thêm vào
  + Xoá phần tử :
    - Tìm kiếm phần tử trong bảng băm.
    - Nếu tìm thấy, xóa phần tử đó khỏi bảng băm và danh sách liên kết kép để duy trì thứ. tự
  + Tìm kiếm phần tử :
    - Tính mã băm: Tính toán mã băm của phần tử cần tìm kiếm.
    - Xác định bucket: Sử dụng mã băm để xác định bucket mà phần tử có thể được lưu trữ.
    - Kiểm tra bằng equals(): LinkedHashSet kiểm tra từng phần tử trong bucket đó bằng cách sử dụng phương thức equals():
      * Nếu tìm thấy phần tử tương đương, trả về true.
      * Nếu không tìm thấy, trả về false.
  + Duyệt phần tử : sử dụng foreach hoăc Iteẻator , Iterator sẽ truy cập qua danh sách liên kết kép, giữ thứ tự các phần tử như chúng được thêm vào.
* Trường hợp sử dụng : Khi cần một collection không chứa các phần tử trùng lặp và bạn cần duy trì thứ tự các phần tử được thêm vào.

2.3 TreeSet:

* Là cấu trúc dựa trên Binary Tree và mọi thao tác đến dữ liệu liên quan đến cấu trúc này .
* Ưu điểm:
  + Các phần tử trong TreeSet luôn được sắp xếp theo thứ tự hoặc theo một comparator tùy chỉnh được cung cấp khi khởi tạo.
  + TreeSet không cho phép các phần tử trùng lặp. Điều này đảm bảo rằng mỗi phần tử trong TreeSet là duy nhất.
  + TreeSet dựa trên cấu trúc cân bằng nhị phân. Điều này làm cho các thao tác như tìm kiếm, thêm và xóa phần tử khá hiệu quả cho các tập dữ liệu lớn.
  + TreeSet hỗ trợ các phương thức đặc biệt như first(), last(), higher(), lower() để lấy phần tử lớn hơn, nhỏ hơn, phần tử đầu tiên, phần tử cuối cùng, giúp dễ dàng điều hướng trong tập hợp.
  + TreeSet cho phép lấy các tập hợp con từ một tập hợp lớn thông qua các phương thức như subSet(), headSet(), và tailSet().

### Nhược điểm:

* + Mặc dù TreeSet có thời gian truy xuất tốt hơn nhiều so với các cấu trúc dữ liệu khác như LinkedList hay ArrayList, nhưng nó vẫn chậm hơn HashSet..
  + TreeSet không cho phép phần tử null.
  + Do phải duy trì cấu trúc cây nhị phân cân bằng, TreeSet yêu cầu nhiều bộ nhớ hơn so với HashSet hoặc LinkedHashSet, đặc biệt là khi làm việc với các tập dữ liệu lớn.
  + Để có thể sắp xếp phần tử, các đối tượng trong TreeSet phải triển khai giao diện Comparable hoặc phải cung cấp một Comparator khi khởi tạo TreeSet.
* Cơ chế triển khai
  + Thêm phần tử
    - Bắt đầu từ root node, TreeSet sẽ so sánh phần tử mới với các phần tử hiện có trong cây (dựa trên Comparable hoặc Comparator nếu được cung cấp).
    - Phần tử được thêm vào vị trí đúng trong cây theo thứ tự sắp xếp.
    - Sau khi thêm, nếu tree mất cân bằng (chiều cao của các nhánh không đồng đều), thuật toán cân bằng của tree sẽ được áp dụng.
  + Xoá phần tử
    - Khi xóa một phần tử, TreeSet sẽ tìm phần tử đó trong cây và loại bỏ nó.
    - Sau khi xoá, nếu tree mất cân bằng (chiều cao của các nhánh không đồng đều), thuật toán cân bằng của tree sẽ được áp dụng.
  + Duyệt phần tử
    - Bắt đầu từ node nhỏ nhất (tìm bằng cách đi đến node trái nhất của cây), sau đó duyệt các nút theo thứ tự đã được sắp xếp. Mỗi lần di chuyển tới nút tiếp theo, thuật toán duyệt sẽ sử dụng lại cấu trúc binary tree.
  + Tìm kiếm phần tử
    - Bắt đầu từ root node, nếu phần tử tìm kiếm nhỏ hơn phần tử gốc, tìm trong nhánh con trái; nếu lớn hơn, tìm trong nhánh con phải. Cứ thế, duyệt qua các nút cho đến khi tìm được phần tử hoặc không còn nhánh để tiếp tục tìm kiếm.
    - Phép có thể so sánh: Dựa trên thứ tự tự nhiên (hoặc Comparator nếu có).
* Trường hợp sử dụng :
  + Khi cần lưu trữ các phần tử không trùng lặp và duy trì thứ tự tự nhiên hoặc tùy chỉnh.

2.4 EnumSet

* Được sử dụng để quản lý các tập hợp (set) của các kiểu enum.
* Ưu điểm của EnumSet:
  + EnumSet được triển khai rất tối ưu về bộ nhớ và thời gian, đặc biệt là so với các tập hợp khác như HashSet. Vì EnumSet sử dụng các bit để đại diện cho các phần tử, các phép toán như thêm, xóa, và kiểm tra phần tử rất nhanh.
  + Tiết kiệm bộ nhớ do lưu trữ dựa vào bit.
  + EnumSet lưu trữ các phần tử theo thứ tự xuất hiện của chúng trong kiểu enum, giúp việc duyệt qua các phần tử của tập hợp theo thứ tự tự nhiên dễ dàng.
  + EnumSet cung cấp nhiều phương thức hữu ích để tạo và thao tác với tập hợp như allOf(), noneOf(), range(), và complementOf(). Những phương thức này giúp dễ dàng làm việc với tập hợp các hằng số enum.

### Nhược điểm của EnumSet:

* + Điểm yếu lớn nhất của EnumSet là chỉ làm việc với các kiểu enum. Điều này có nghĩa là bạn không thể sử dụng EnumSet cho các kiểu dữ liệu khác.
  + EnumSet không phải là thread-safe, vì vậy nếu bạn cần sử dụng nó trong môi trường đa luồng.
  + EnumSet không cho phép chứa giá trị null.
  + Kích thước của EnumSet bị giới hạn bởi số lượng các hằng số của kiểu enum mà nó biểu diễn.
* Trường hợp sử dụng :Khi cần làm việc với các tập hợp của enum vì hiệu suất cao và tính tối ưu về bộ nhớ.