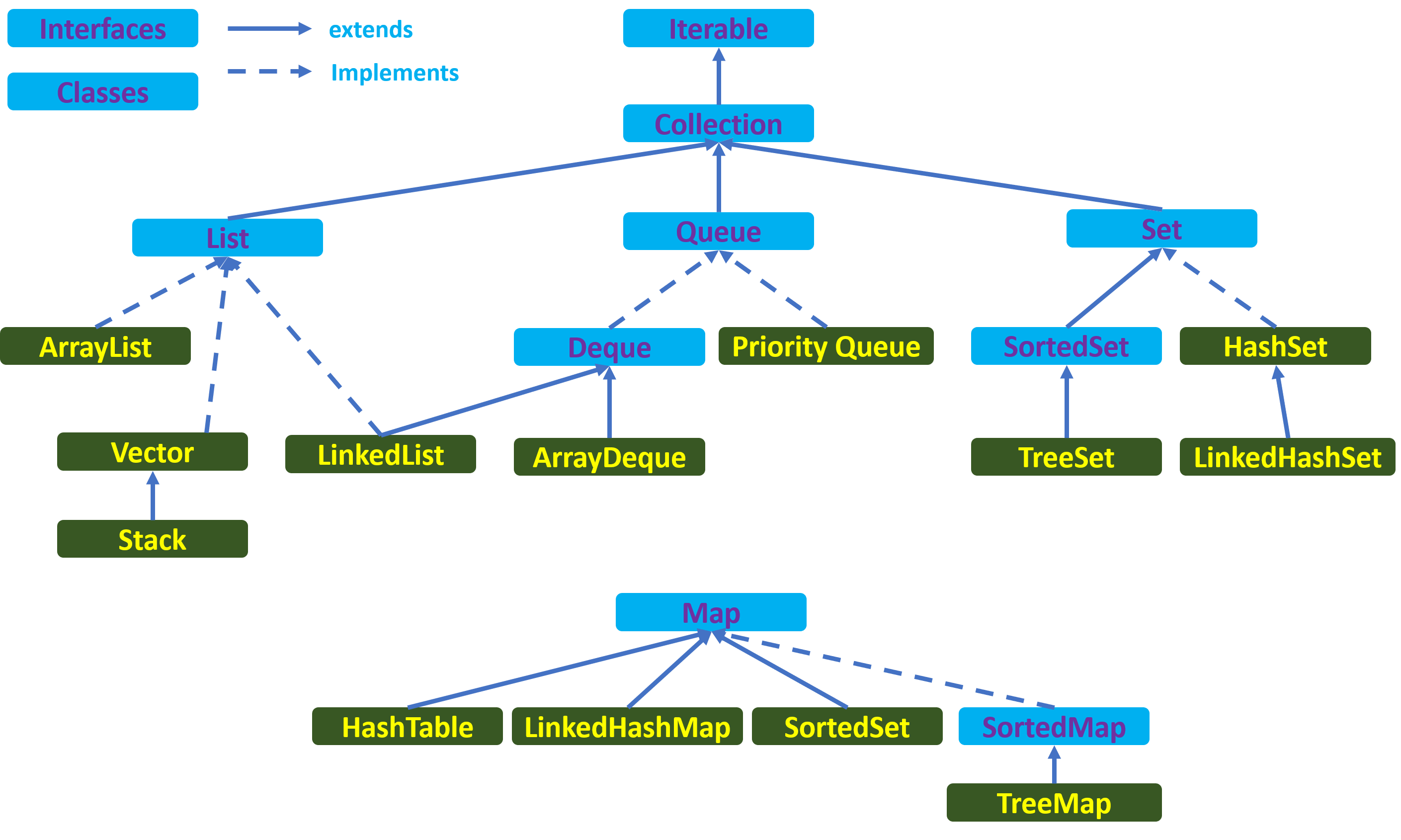
**Collection**

1. Collection và Collections
2. Collection:
   * Root Interface
   * Là super Class của List, Set, Queue, …
   * Cung cấp các quy chuẩn hành động để thao tác với các cấu trúc dữ liệu
   * Cung cấp các class được xây dựng dựa trên các cấu trúc lưu trữ phục vụ việc thao tác với và lưu trữ dữ liệu : ArrayList, LinkedList, HashSet, HashMap,…
3. Collections
   * Cung cấp các method static, các thuật toán hành động để làm việc với các cấu trúc lưu trữ của Collection
4. **Collection Interface**



Collection Interface định nghĩa những phương thức cơ bản khi làm việc với tập hợp, đây là gốc cũng là nền móng để từ đó xây dựng lên cả bộ thư viện Java Collection Framework. Collection Interface được kế thừa từ Iterable Interface nên có thể dễ dàng duyệt qua từng phần tử thông qua việc sử dụng Iterator.

Các interface bao gồm: Set, List, Map, Stack, Queue.

Ứng với mỗi Interface trên Có các Class triển khai là:

1. List

* ArrayList
* LinkedList

1. Set

* HashSet
* TreeSet

+ SortSet

1. Map:

* HashMap
* LinkedMap
* TreeMap

4 Một số Class khác:

* Vector
* Hastable

Cung cấp các class được xây dựng dựa trên các cấu trúc lưu trữ(Set, List, Map, Queue) phục vụ việc thao tác với và lưu trữ dữ liệu : ArrayList, LinkedList, HashSet, HashMap,…

**3. Set Interface**

Set (tập hợp) là kiểu dữ liệu không chứa các giá trị trùng lặp (Bộ tú, biển số xe,số căn cước công dân) .

Set Interface cung cấp các phương thức để tương tác với set. Set Interface được kế thừa từ Collection Interface nên nó cũng có đầy đủ các phương thức của Collection Interface.

1. Một số class thực thi Set Interface thường gặp:

* **TreeSet**: là 1 class thực thi giao diện Set Interface, trong đó các phần tử trong set đã được sắp xếp.
* **HashSet**: là 1 class implement Set Interface, mà các phần tử được lưu trữ dưới dạng bảng băm (hash table) để lưu trữ
  + HashSet lưu trữ các phần tử bằng cách sử dụng một cơ chế được gọi là **hashing (băm).**
  + HashSet **không đảm bảo thứ tự** được thêm vào.
  + HashSet cho phép chứa phần tử **NULL**.

. HashSet vs TreeSet vs LinkHashSet:

* HashSet cho hiệu suất tốt hơn so với LinkHashSet TreeSet.
* TreeSet cho hiệu suất thấp hơn HashSet và LinkedHashSet vì nó phải sắp xếp các phần tử sau mỗi lần chèn và loại bỏ.
* **LinkHashSet:** Lớp **LinkedHashSet**trong Java kế thừa **HashSet** và triển khai **Set** **Interface**. Nó tạo một collection mà sử dụng một**Linked List** để lưu giữ các phần tử theo thứ tự chúng đã được chèn.
* **LinkedHashSet đảm bảo thứ tự** được thêm vào.
* LinkedHashSet sử dụng đối tượng **LinkedHashMap** nội bộ để lưu trữ và xử lý các phần tử của nó.
* LinkedHashSet cho phép chứa phần tử **NULL**.

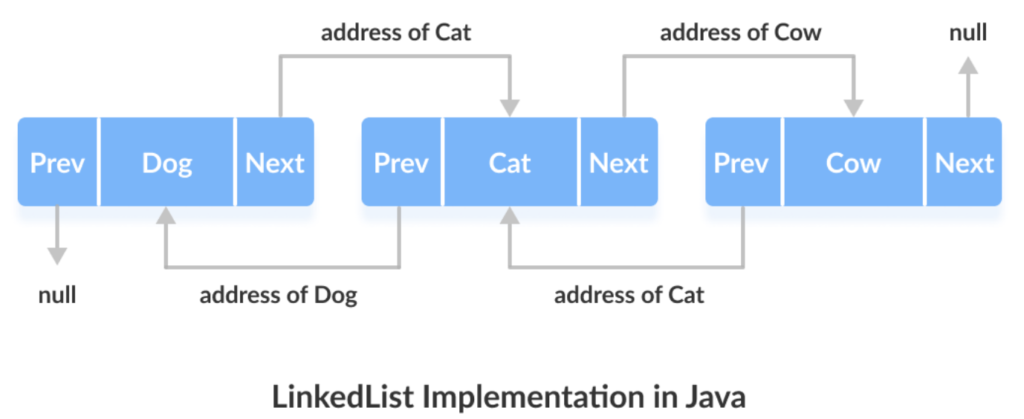
1. SortSet Interface:

* SortedSet Interface là 1 dạng riêng của Set Interface nên nó có những đặc điểm của Set đó là các phần tử trong SortedSet là duy nhất (nghĩa là giá trị của các phần tử này không được giống nhau) và SortedSet được sử dụng khi chúng ta muốn lưu trữ một danh sách các phần tử không có sự trùng lặp**. Ngoài ra, SortedSet có điểm vượt trội hơn so với Set là thứ tự các phần tử trong Set được sắp xếp tăng dần hoặc giảm dần (*mặc định là tăng dần*)**

**4. List Interface**

List (danh sách) là cấu trúc dữ liệu tuyến tính trong đó các phần tử được sắp xếp theo một thứ tự xác định, có thể chứa các phần tử giống nhau. List Interface định nghĩa các phương thức để tương tác với list cũng như các phần tử bên trong list. Tương tự như Set Interface, List Interface cũng được kế thừa và có đầy đủ các phương thức của Collection Interface.

Một số class thực thi List Interface thường sử dụng:

* **ArrayList**: là 1 class dạng list được implement dựa trên mảng có kích thước thay đổi được.
* **LinkedList**: là một class dạng list hoạt động trên cơ sở của cấu trúc dữ liệu danh sách liên kết đôi (double-linked list)
  + - chứa các phần tử trùng lặp
    - non-synchronized
    - thao tác sửa xóa nhanh hơn ArrayList
    - có thể sử dụng được như Stack hoặc Queue
    - là 1 cấu trúc phân tán
    - 

+ prev lưu trữ địa chỉ ô nhớ của node phía trước

+ data: lưu trữ dữ liệu của node

+ next: lưu trữ địa chỉ ô nhớ của node phía sau

* + - Giải quyết điểm yếu của danh sách liên kết đơn = > chỉ chứa được địa chỉ ô nhớ của node phía sau
    - Ưu điểm: có thể duyệt danh sách theo cả 2 chiều
    - Nhược điểm: tốn bộ nhớ hơn singly linkedList(chứa cả 2 ô nhớ)
    - Cách thức hoạt động khi them sửa xóa sẽ phức tạp hơn

**So sánh ArrayList và LinkedList**

- ArrayList:

+ Sử dụng để lưu trữ các phần tử

+ THAO TÁC THÊM sửa xóa chậm hơn do phải đánh lại index của các phần tử

+ chỉ có thể hoạt động như 1 list

+ Tốt hơn khi duyệt hoặc truy cập lấy ra dữ liệu do có index và các ô nhớ liền kề nhau

* + - LinkedList List:
      * Sử dụng danh sách liên kết kép để lưu trữ các phần tử
      * Thao tác them sửa xóa nhanh hơn do chỉ thay đổi dữ liệu của nơi chưa địa chỉ ô nhớ của node tiếp theo
      * Có thể hoạt động như queue, Stack, List
      * Tối ưu hơn trong việc lưu trữ vì lưu trữ phân tán
* **Vector**: là 1 class thực thi giao diện List Interface, có cách thức lưu trữ như mảng tuy nhiên có kích thước thay đổi được, khá là tương tự với ArrayList, tuy nhiên điểm khác biệt là Vector là synchronized, hay là đồng bộ, có thể hoạt động đa luồng mà không cần gọi synchronize một cách tường minh
* **Stack:** cũng là 1 class dạng list, Stack có cách hoạt động dựa trên cơ sở của cấu trúc dữ liệu ngăn xếp (stack) với kiểu vào ra LIFO (last-in-first-out hay vào sau ra trước) nổi tiếng.

**5. Queue Interface**

Queue (hàng đợi) là kiểu dữ liệu nổi tiếng với kiểu vào ra FIFO (first-in-first-out hay vào trước ra trước), tuy nhiên với Queue Interface thì queue không chỉ còn dừng lại ở mức đơn giản như vậy mà nó cũng cấp các phương thức để xây dựng các queue phức tạp hơn nhiều như priority queue (queue có ưu tiên), deque (queue 2 chiều), … Và cũng giống như 2 interface trước, Queue Interface cũng kế thừa và mang đầy đủ phương thức từ Collection Interface. Một số class về Queue thường sử dụng:

* **LinkedList**: tương tự ở phần List
* **PriorityQueue**: là 1 dạng queue mà trong đó các phần tử trong queue sẽ được sắp xếp.
* **ArrayDeque**: là 1 dạng deque (queue 2 chiều) được implement dựa trên mảng

**6. Map Interface**

Map (đồ thị/ánh xạ) là kiểu dữ liệu cho phép ta quản lý dữ liệu theo dạng cặp key-value, trong đó key là duy nhất và tương ứng với 1 key là một giá trị value. Map Interface cung cấp cho ta các phương thức để tương tác với kiểu dữ liệu như vậy. Không giống như các interface ở trên**, Map Interface không kế thừa từ Collection Interface** mà đây là 1 interface độc lập với các phương thức của riêng mình. Dưới đây là một số class về Map cần chú ý:

* **TreeMap**: là class thực thi giao diện Map Interface với dạng cây đỏ đen (Red-Black tree) trong đó các key đã được sắp xếp. Class này cho phép thời gian thêm, sửa, xóa và tìm kiếm 1 phần tử trong Map là tương đương nhau và đều là O(log(n))
* **HashMap**: là class thực thi giao diện Map Interface với các key được lưu trữ dưới dạng bảng băm, cho phép tìm kiếm nhanh O(1).

HashMap vs Treemap:

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| 1) HashMap có thể chứa một key là null. | TreeMap KHÔNG thể chứa bất kỳ key nào là null. |
| 2) HashMap duy trì các phần tử không theo thứ tự. | TreeMap duy trì các phần tử theo thứ tự key tăng dần. |

## Định nghĩa về Stack.

**Stack** là một loại cấu trúc dữ liệu tuyến tính không nguyên thủy. Nó là một danh sách được sắp xếp trong đó các phần tử được thêm vào hoặc xóa đi chỉ ở một đầu của danh sách, được gọi là đỉnh của ngăn xếp (TOS). Vì tất cả việc xóa và chèn trong ngăn xếp được thực hiện từ đầu ngăn xếp, phần tử cuối cùng được thêm vào sẽ là phần tử đầu tiên được xóa khỏi ngăn xếp. Đó là lý do tại sao ngăn xếp được gọi là loại danh sách Last-in-First-out (LIFO).

Ví dụ: Bạn có 1 hộp bánh và chỉ có 1 đầu để lấy bánh ra, bạn chỉ có thể lấy từng cái ra bằng đầu này ( cái này gọi là popping). Tương tự, bạn muốn cất số bánh đã lấy thừa, bạn chỉ có thể cất từng cái bánh vào hộp thông qua đầu này (gọi là pushing).

## Định nghĩa về Queue.

**Queue** là một loại cấu trúc dữ liệu tuyến tính thuộc loại không nguyên thủy. Nó là một tập hợp các loại yếu tố tương tự. Việc bổ sung các yếu tố mới diễn ra ở một đầu được gọi là Rear-end. Tương tự như vậy, việc xóa các phần tử hiện có diễn ra ở đầu kia được gọi là Front-end.

Ví dụ: Nó giống như việc sếp hàng để mua vé xem phim, người đến trước sẽ được phục vụ trước người đến sau phải đứng vào cuối hàng đợi chờ để được phục vụ.

## Thực hiện Stack

Stack có thể thực hiện theo 2 cách:

1. **Static implementation** (triển khai tĩnh) sử dụng mảng để tạo stack. Triển khai tĩnh mặc dù là một kỹ thuật dễ dàng nhưng không phải là cách tạo linh hoạt, vì việc khai báo kích thước của ngăn xếp phải được thực hiện trong quá trình thiết kế chương trình, sau đó kích thước không thể thay đổi. Ngoài ra, triển khai tĩnh không hiệu quả lắm đối với việc sử dụng bộ nhớ. Vì một mảng (để thực hiện ngăn xếp) được khai báo trước khi bắt đầu hoạt động (tại thời điểm thiết kế chương trình). Bây giờ nếu số lượng phần tử được sắp xếp rất ít trong ngăn xếp, bộ nhớ được phân bổ tĩnh sẽ bị lãng phí. Mặt khác, nếu có nhiều số phần tử được lưu trữ trong ngăn xếp thì chúng ta không thể thay đổi kích thước của mảng để tăng dung lượng của nó, để nó có thể chứa các phần tử mới.
2. **Dynamic implementation**(triển khai động) cũng được gọi là biểu diễn danh sách được liên kết và sử dụng các con trỏ để thực hiện kiểu ngăn xếp của cấu trúc dữ liệu.

## Thực hiện Queue

1. **Static implementation** (triển khai tĩnh): Nếu hàng đợi được triển khai bằng mảng, số lượng phần tử chính xác mà chúng tôi muốn lưu trữ trong hàng đợi phải được đảm bảo trước, bởi vì kích thước của mảng phải được khai báo tại thời điểm thiết kế hoặc trước khi quá trình xử lý bắt đầu. Trong trường hợp này, phần đầu của mảng sẽ đứng trước của hàng đợi và vị trí cuối cùng của mảng sẽ đóng vai trò là phía sau của hàng đợi. Mối quan hệ sau đây cung cấp cho toàn bộ các phần tử tồn tại trong hàng đợi khi được triển khai bằng cách sử dụng các mảng.
2. **Dynamic implementation**(triển khai động): thực hiện các hàng đợi bằng cách sử dụng các con trỏ, nhược điểm chính là một nút trong biểu diễn được liên kết sẽ tiêu tốn nhiều không gian bộ nhớ hơn một phần tử tương ứng trong biểu diễn mảng. Vì có ít nhất hai trường trong mỗi nút một cho trường dữ liệu và trường khác để lưu địa chỉ của nút tiếp theo trong khi trong biểu diễn chỉ có trường dữ liệu là có. Công dụng của việc sử dụng biểu diễn được liên kết trở nên rõ ràng khi được yêu cầu chèn hoặc xóa một phần tử ở giữa một nhóm các phần tử khác.

## Hoạt động trên Stack

Các hoạt động cơ bản có thể được vận hành trên ngăn xếp như sau:

**PUSH** : khi một phần tử mới được thêm vào đầu ngăn xếp được gọi là hoạt động PUSH. Đẩy một phần tử trong ngăn xếp gọi thêm phần tử, vì phần tử mới sẽ được chèn ở trên cùng. Sau mỗi hoạt động đẩy, đỉnh được tăng thêm một. Nếu mảng đầy, và không có phần tử mới nào có thể được thêm vào, nó được gọi là điều kiện STACK-FULL hoặc STACK OVERFLOW

**POP** : Khi một phần tử bị xóa khỏi đầu ngăn xếp, nó được gọi là hoạt động POP. Ngăn xếp được giảm đi một, sau mỗi hoạt động pop. Nếu không còn phần tử nào trên ngăn xếp và pop được thực hiện, thì điều này sẽ dẫn đến điều kiện STACK UNDERFLOW, điều đó có nghĩa là ngăn xếp của bạn trống.

## Kết luận

Như vậy **Stack** và **Queue** là các cấu trúc dữ liệu tuyến tính khác nhau theo các cách nhất định như cơ chế làm việc, cấu trúc, cách thực hiện, các biến thể nhưng cả hai đều được sử dụng để lưu trữ các phần tử trong danh sách và thực hiện các thao tác trong danh sách như thêm và xóa các phần tử.