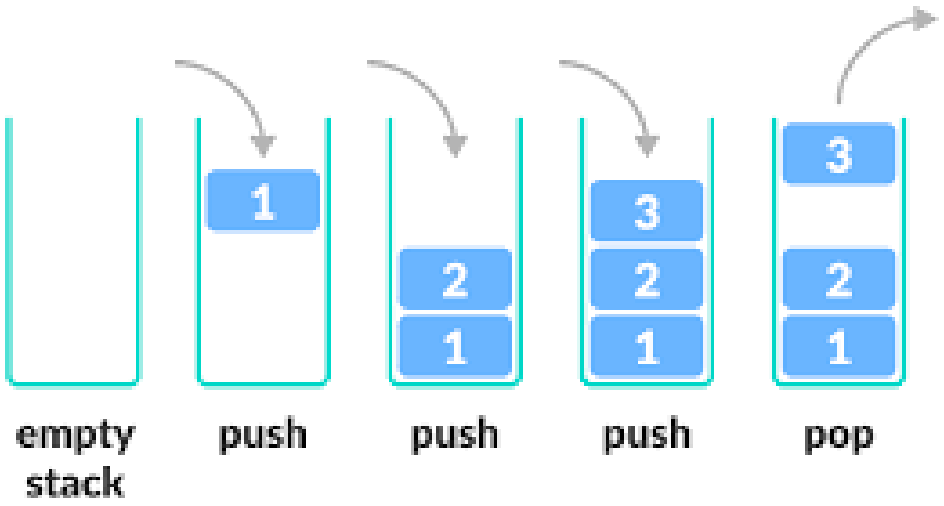
선형 자료구조 (Stack / Queue)

Stack

Stack



https://www.programiz.com/dsa/stack

특징

- LIFO (Last In First Out)

활용

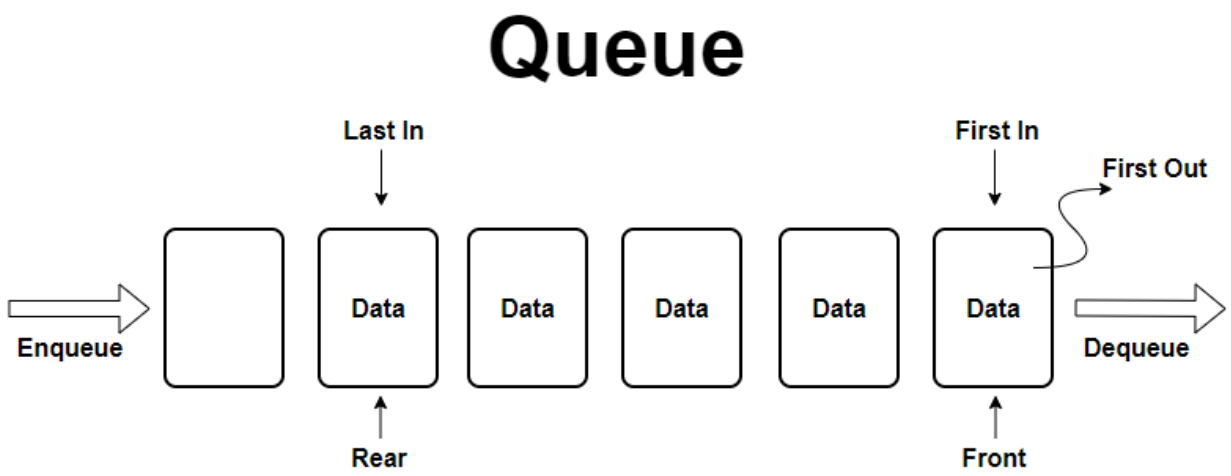
- 웹 브라우저 방문 기록 (undo)
- 괄호 유효성 검사 (현욱은 괄호왕이야!! 문제 풀면 stack 정복 가능!)
- 깊이 우선 탐색(DFS)
- 재귀 알고리즘
- 후위 표기법 연산 (예시)

시간복잡도

- 삽입(push): O(1)
- 삭제(pop): O(1)
- 탐색: O(n)

Queue

Queue



https://velog.io/@yyj8771/자료구조-큐-Queue

특징

- FIFO (First In First Out)
- 입력된 시간 순서대로 처리

활용

- 프로세스 관리

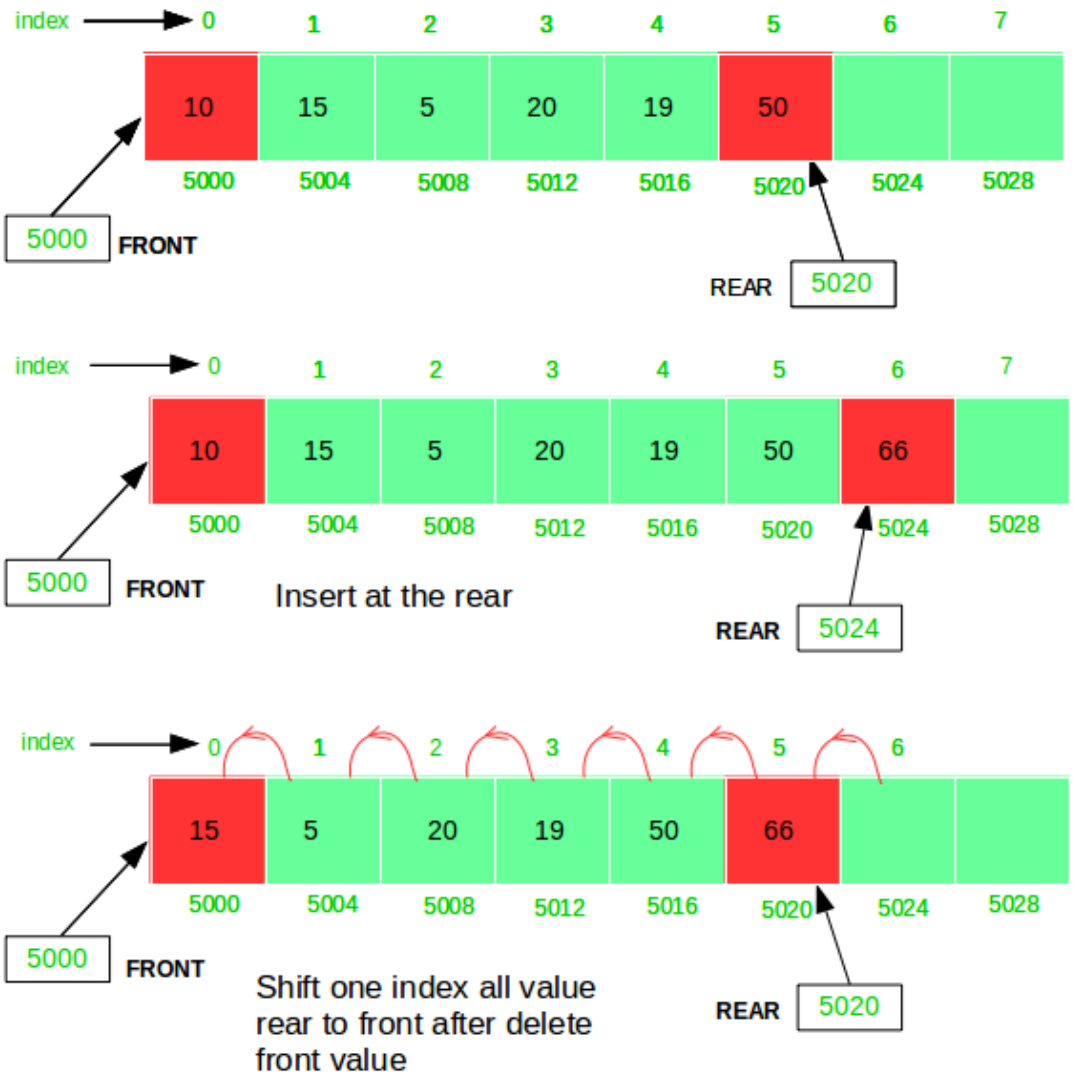
- 너비 우선 탐색(BFS)
- 캐시(Cache) 구현
  (최근 사용된 데이터 추가, 오래된 데이터 삭제)

시간복잡도

- 삽입(enqueue): O(1)
- 삭제(dequeue): O(1)
- 탐색: O(n)

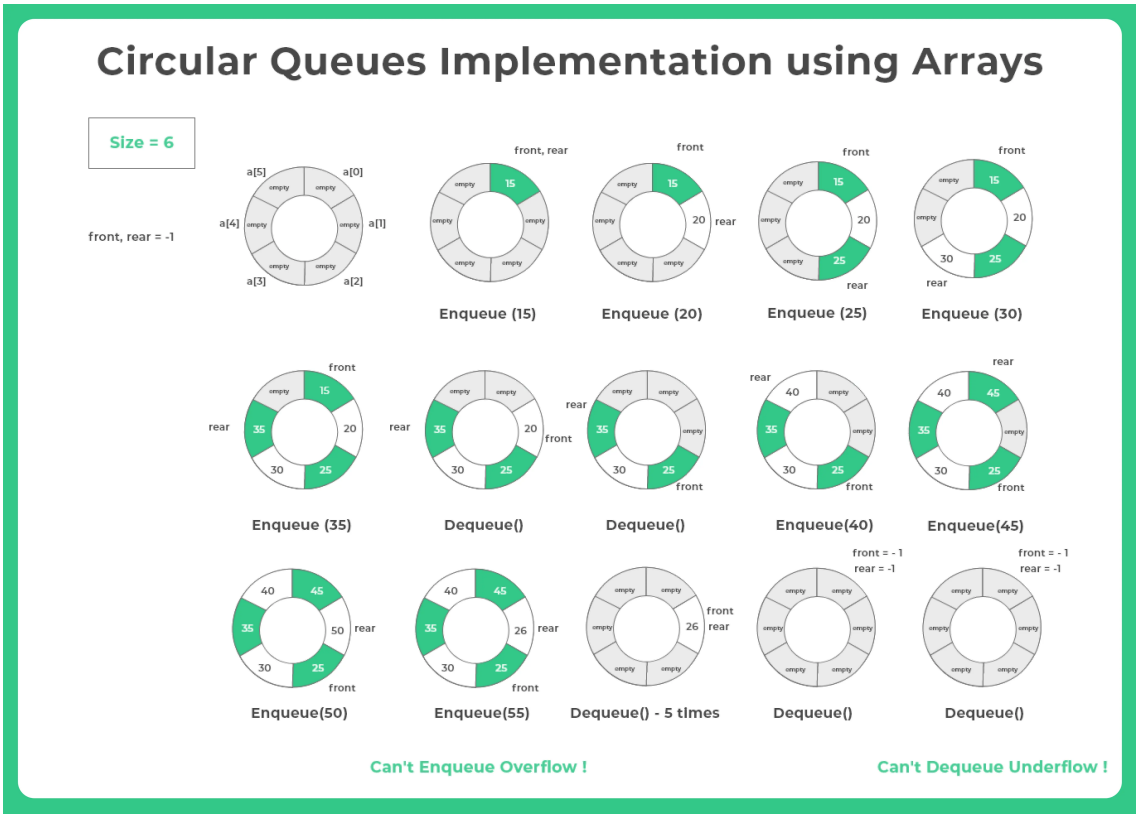
구현

- Array



https://www.geeksforgeeks.org/array-implementation-of-queue-simple/

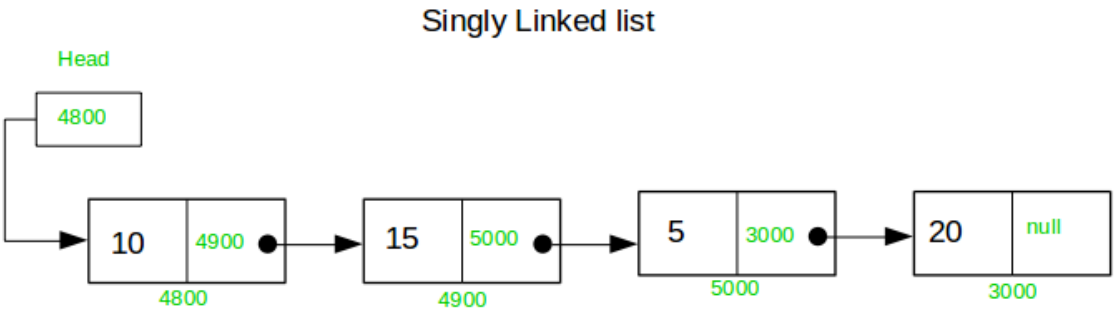
- 고정된 배열 크기 선언
- 시간 복잡도
  - Enqueue: O(1)
  - Dequeue: O(n)
  - access: O(1)
- 사이즈를 넘을 시에는 동적 배열(Dynamic array)와 같은 방식으로 배열 크기 확장
- 보통 **원형 큐(Circular Queue)**로 구현



https://prepinsta.com/data-structures-algorithms/circular-queue-using-array-in-c/

- Dequeue의 O(n)을 O(1)로 줄일 수 있음

- List



https://www.geeksforgeeks.org/difference-between-a-static-queue-and-a-singly-linked-list/

- 동적 크기
- 더 많은 메모리 사용

    - 시간 복잡도
        - Enqueue: O(1)
        - Dequeue: O(1)
        - access: O(n)
    - 보통 Single-linked list로 구현

Priority Queue

특징

- 우선순위가 높은 데이터를 먼저 처리

활용

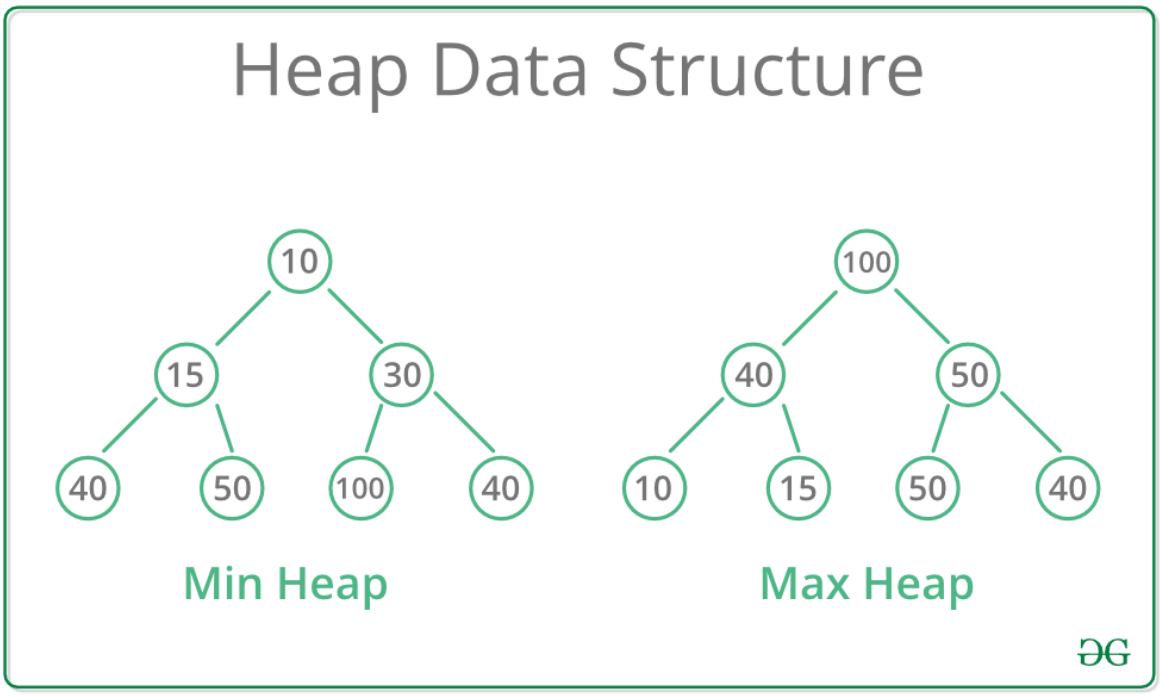
- 작업 스케줄링

시간복잡도

- 삽입(push): O(logn)
- 삭제(pop): O(logn)

구현

- Heap



https://www.geeksforgeeks.org/heap-data-structure/

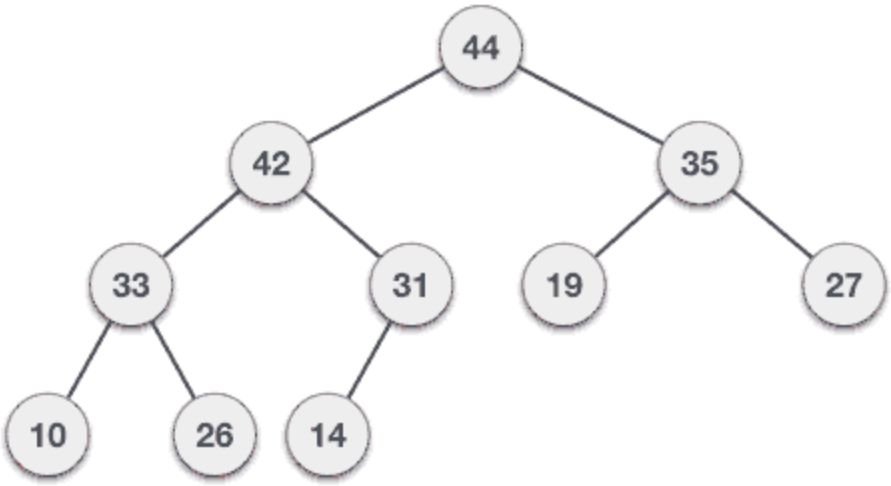
- 완전이진트리
- 최소 힙: 부모 노드의 값 ≤ 자식 노드의 값
- 최대 힙: 부모 노드의 값 ≥ 자식 노드의 값
- 보통 배열로 구현!
  - 0번째 index는 공백 (사용하지 않음)
  - 루트 노드인덱스는 1부터
    (계산 쉽게 하기 위하여)
  - n번 째 노드의 왼쪽 자식 노드 = 2n
  - n번 째 노드의 오른쪽 자식 노드 = 2n+1
  - n번 째 노드의 부모 노드 = n/2
  - h높이의 첫 번째 노드 = 2^h (h = 0, 1, …)
- 시간복잡도
  - 삽입(push): O(logn)
  - 삭제(pop): O(logn)

- 삽입

https://www.tutorialspoint.com/data_structures_algorithms/heap_data_structure.htm

1. 힙의 끝에 새로운 데이터에 대한 노드 삽입
2. 부모 노드와 새로 생긴 노드에 대한 데이터 비교하여 swap
    a. 최소 힙: 부모 노드의 값 ≤ 자식 노드의 값
    - 최대 힙: 부모 노드의 값 ≥ 자식 노드의 값
3. 힙의 특징을 만족할 때까지 2.를 반복하여 적절하게 배치

- 삭제



https://www.tutorialspoint.com/data_structures_algorithms/heap_data_structure.htm

1. 루트 노드를 지운다 (최대 / 최소)
2. 힙의 가장 마지막 노드를 루트 노드로 올린다
3. 부모 노드(현재는 루트 노드)와 새로 생긴 노드에 대한 데이터 비교하여 swap
    a. 최소 힙: 부모 노드의 값 ≤ 자식 노드의 값
    - 최대 힙: 부모 노드의 값 ≥ 자식 노드의 값
4. 힙의 특징을 만족할 때까지 3.를 반복하여 적절하게 배치

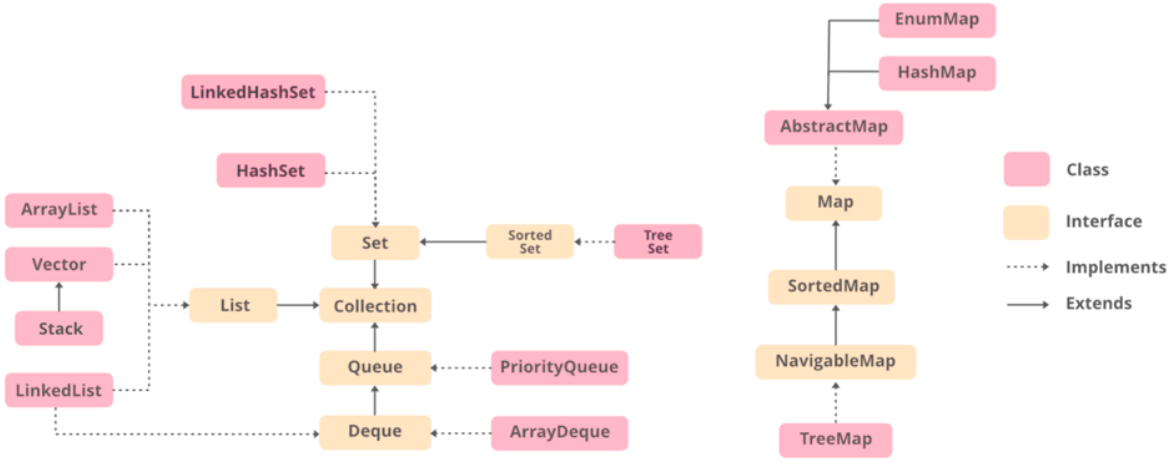
Queue 시간 복잡도 비교

Data Structure	Complexity	Time Complexity			Space Complexity
	Operations	enqueue	dequeue	peek	
Queue	Array	O(1)	O(n)	O(1)	O(n)
	Linked list	O(1)	O(1)	O(1)	O(n)
Circular queue		O(1)	O(1)	O(1)	O(n)
Deque		O(1)	O(1)	O(1)	O(n)
Priority queue		O(log n)	O(log n)	O(1)	O(n)

https://devopedia.org/queue-data-structure

Java에서의 구현

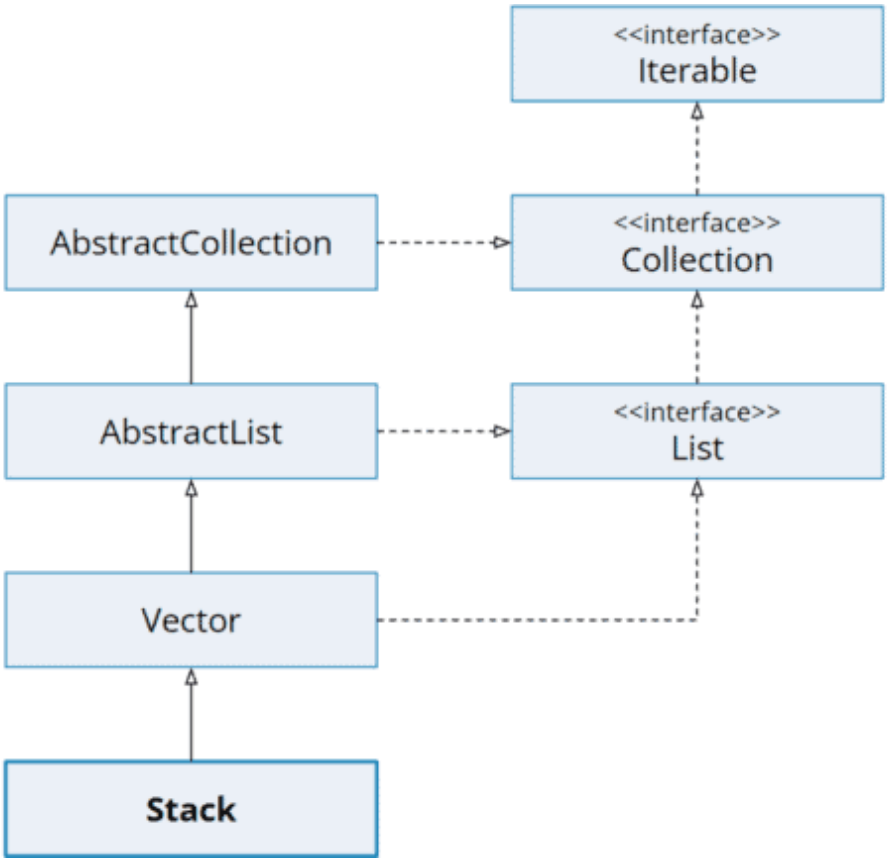
전체 다이어그램



https://www.geeksforgeeks.org/arraylist-vs-linkedlist-java/

Stack 구현

- Vector 상속

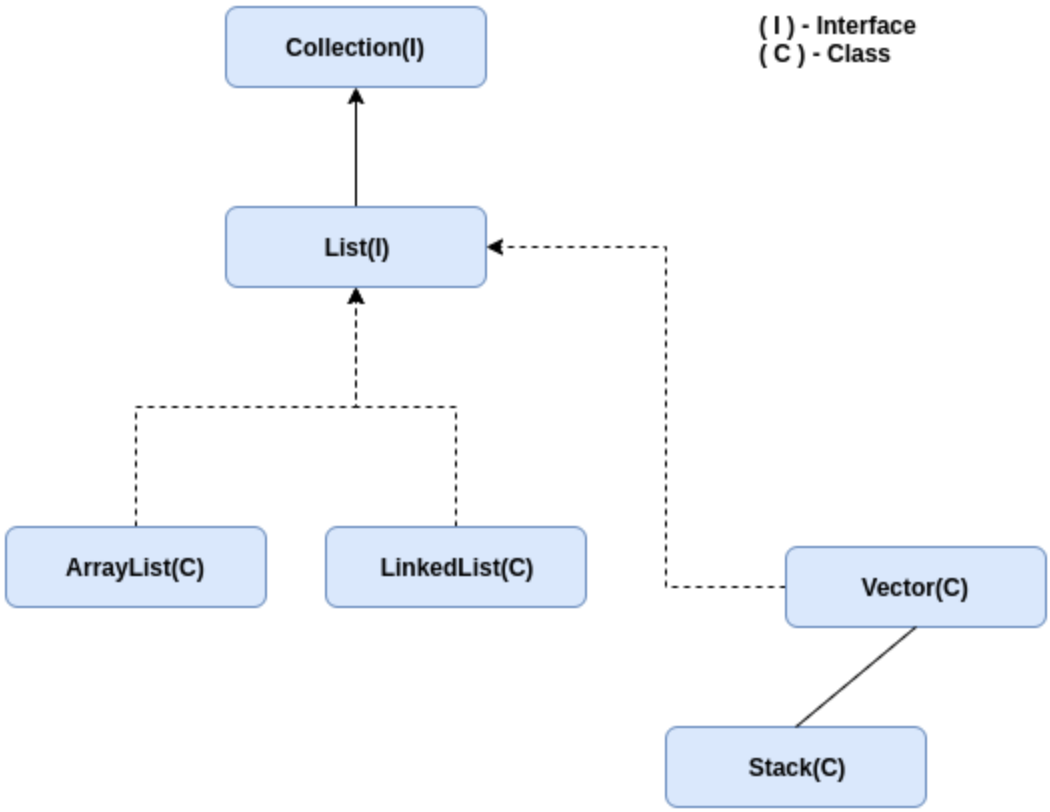


https://www.happycoders.eu/algorithms/java-stack-class/

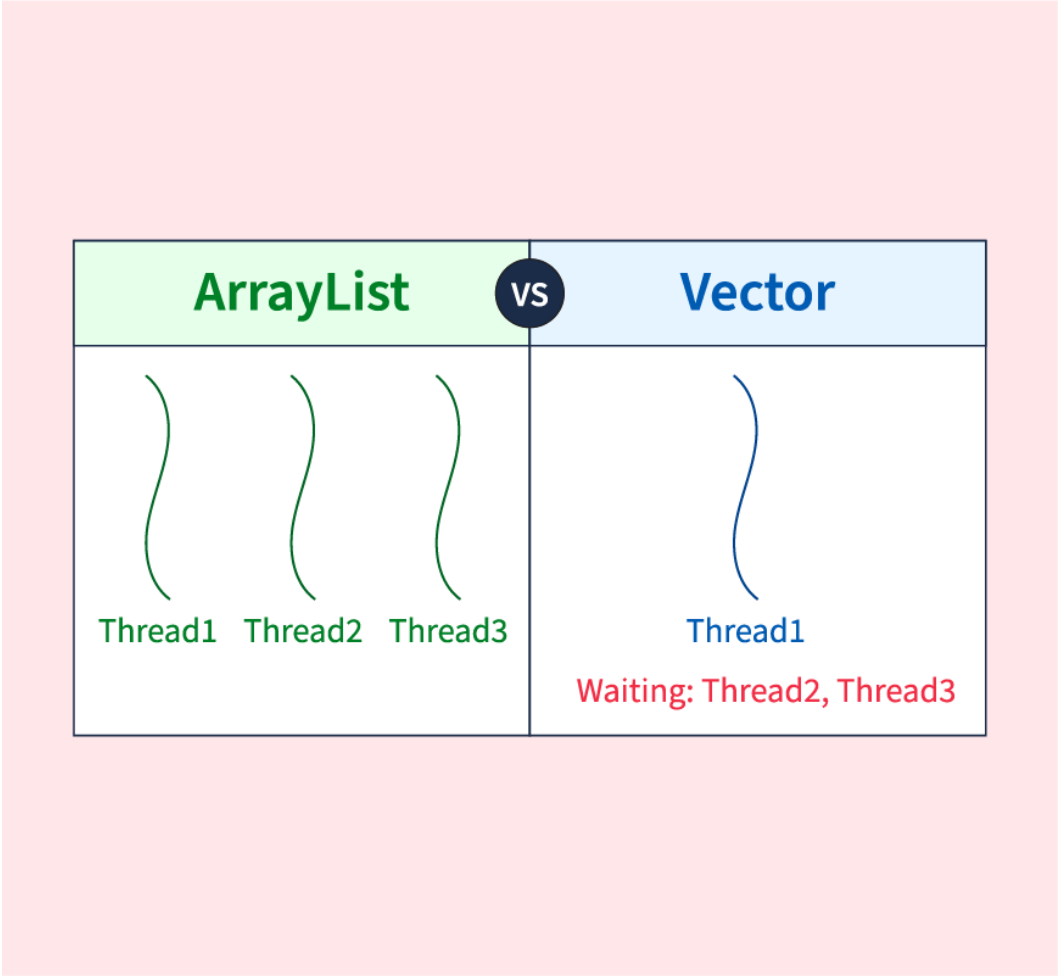
- Vector란?
  - 객체들의 가변 크기 배열 구현.
    동기화(Synchronization)된 동적 배열(dynamic array) 기반 컬렉션 클래스
  - Vector vs ArrayList

특성	Vector	ArrayList
동기화(Synchronization)	모든 메서드에 동기화 지원	비동기화 (non-synchronized)
성능	단일 스레드에서 성능 하락 가능	단일 스레드에서 더 효율적
활용	멀티스레드 환경에서 필요 시	단일 스레드 환경에서 선호
크기 조절	크기 동적 조절 가능	크기 동적 조절 가능

내부 구현	배열 기반 (동적 배열)	배열 기반 (동적 배열)



https://manish11414.medium.com/overview-of-collections-in-java-5d279d425ddb



https://www.scaler.com/topics/arraylist-vs-vector/

▼ [동기화 지원 여부 함수 틈새 비교]

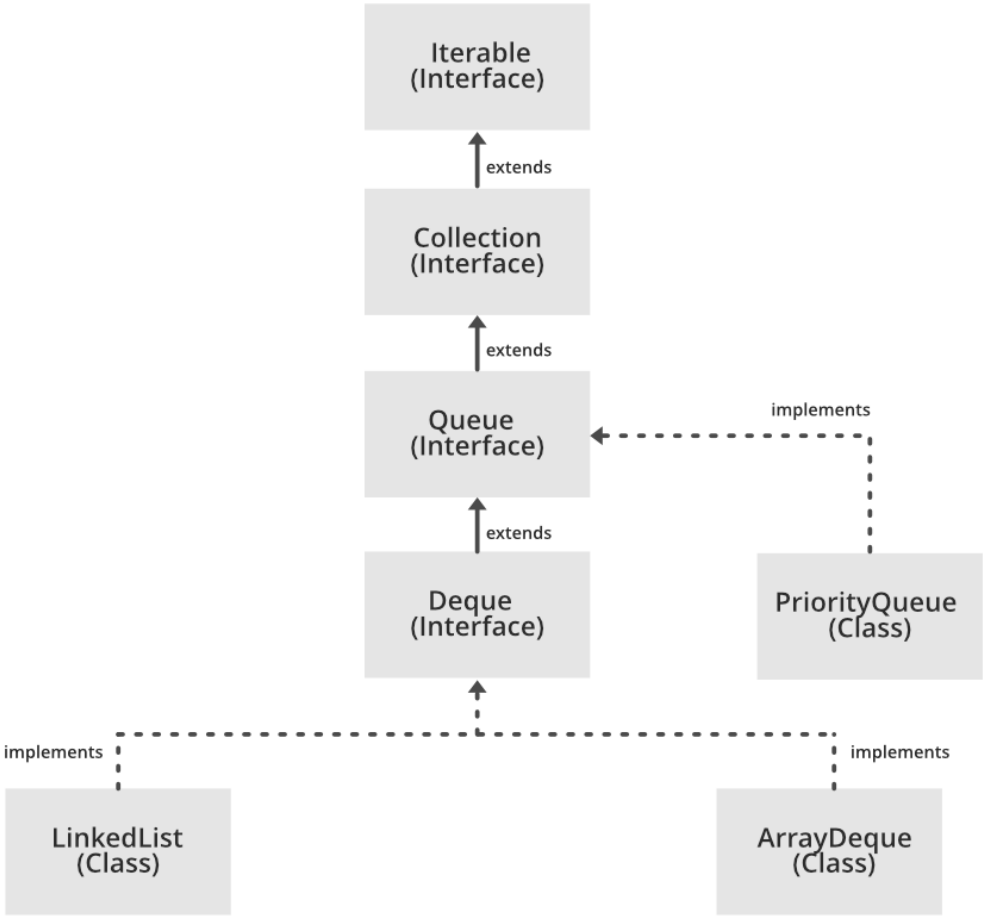
- 동기화 지원. 멀티 쓰레드 환경에서 안전
  - StringBuffer / Vector
- 동기화 지원하지 않음. 단일 쓰레드 환경에 적합
  - StringBuilder / ArrayList, LinkedList

- 성능을 위해 Stack 보다는 ArrayDeque활용을 권장

```java
package java.util;

/**
 * The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector
 * with five operations that allow a vector to be treated as a stack. The usual push and pop
 * operations are provided, as well as a method to peek at the top item on the stack, a method to
 * test for whether the stack is empty, and a method to search the stack for an item and discover
 * how far it is from the top.
 *
 * When a stack is first created, it contains no items.
 *
 * A more complete and consistent set of LIFO stack operations is provided by the Deque
 * interface and its implementations, which should be used in preference to this class. For
 * example:
 *
 *  Deque<Integer> stack = new ArrayDeque<Integer>();
 *
 * Since:  1.0
 * Author: Jonathan Payne
 */
public class Stack<E> extends Vector<E> {
    /**
     * Creates an empty Stack.
     */
    public Stack() {
    }
```

Queue 구현

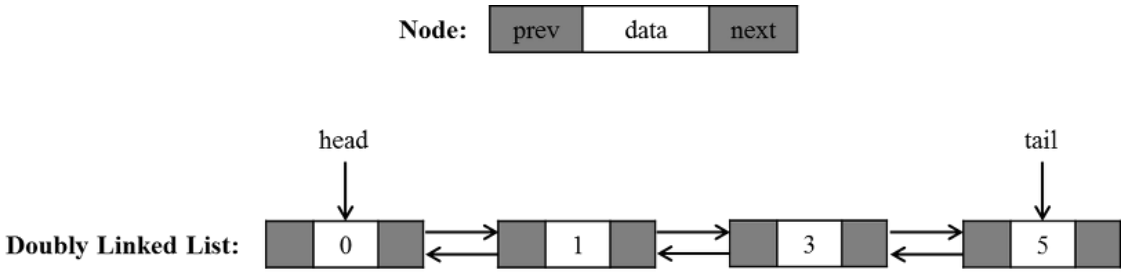


- Deque란?
  - Double Ended Queue
  - Queue, Stack의 양쪽 끝에서 데이터 추가 및 삭제 허용 (양방향 연산)
- ArrayDeque vs LinkedList

특성	**ArrayDeque**	**LinkedList**
내부 구현	배열 기반	이중 연결 리스트 (Doubly Linked List)
메모리 할당	동적으로 크기가 조절되는 배열	각 요소마다 메모리 할당
접근 시간 복잡도	O(1)	O(n)
중간에서의 삽입 및 삭제	O(n)	O(1)

활용	양쪽 끝에서 빠른 삽입 및 삭제가 빈번한 경우(큐, 스택) / 빠른 인덱스 엑세스가 필요한 경우 / 메모리 사용량이 중요한 경우	중간에서의 삽입 및 삭제가 빈번한 경우 / 데이터 크기가 유동적인 경우

- 이중 연결 리스트 (Doubly Linked List) 구조



https://untitledtblog.tistory.com/84

- Priority Queue 구현
  - 균형 이진 힙(balanced binary heap)으로 구현
  - heap은 배열로 구현

```
public class PriorityQueue<E> extends AbstractQueue<E>
    implements java.io.Serializable {

    @java.io.Serial
    private static final long serialVersionUID = -7720805057305804111L;

    private static final int DEFAULT_INITIAL_CAPACITY = 11;

    Priority queue represented as a balanced binary heap: the two children of queue[n] are queue[2*n+1]
    and queue[2*(n+1)]. The priority queue is ordered by comparator, or by the elements' natural
    ordering, if comparator is null: For each node n in the heap and each descendant d of n, n <= d. The
    element with the lowest value is in queue[0], assuming the queue is nonempty.
    transient Object[] queue; // non-private to simplify nested class access

    The number of elements in the priority queue.
    int size;
```

추가 질문

인터페이스와 구현체

- 차이점이 무엇인지?
  - 인터페이스 (interface)
    - 해당 추상적인 선언만하고 구현은 없음 (구현체가 구현해야 하는 기능들을 강제)
    - ex) 에어컨에는 운전, 정지, 온도조절 등의 기능이 있어야한다.
      에어컨에 들어가야하는 필수 기능 리스트를 담은 메뉴얼 같은 느낌
    - 다중 상속 가능
  - 구현체 (implementation)
    - 인터페이스에서 선언된 메소드 구현
    - ex) 각 회사에서 내부적으로 구현한 에어컨: S사 에어컨, L사 에어컨.
      위의 운전, 정지, 온도조절은 각자의 방식대로 구현되어있다.
    - 단일 상속만 가능
- 왜 구분해서 사용하는지?
  - 협업 시 표준화와 실수 방지
    - 일관적으로 구현해야하는 메서드 정의 가능
    - 필수적으로 구현되어야하는 추상 메서드 누락 시 컴파일 에러 발생

```
public interface MyInterface {
        /*
            - 필수적으로 구현되어야 하는 추상 메서드
            - 앞에 abstract 생략되어 있음
        */
    void myMethod();



        // --- java 8 이후에는 구현도 포함할 수 있게 해놓음 ---
    /*
```

```
            - interface에서 메소드 구현 가능
            - 구현체에서 @Override로 재정의 가능
            - 참조 변수로 함수를 호출 가능
        */
    default void myDefaultMethod() {
  }

    /*
            - interface에서 메소드 구현 가능
            - 구현체에서 @Override로 재정의 불가능
            - 반드시 클래스 명으로 메소드 호출
        */
    static void myStaticMethod() {
    }
}
```

- 사용자는 어떻게 구현되어 있는지 신경 쓸필요없이 메서드를 가져다 쓰면 된다 (다형성, 캡슐화)

```
AirConditioner ac1 = new SCompanyAC();
AirConditioner ac2 = new LCompanyAC();

ac1.on();
ac1.off();

ac2.on();
ac2.off();
```