선형 자료구조 는 데이터들이 순서대로 나열되어 하나의 데이터 뒤에 하나의 데이터가 연결되는 형태의 자료구조를 의미한다.

선형 자료구조의 특징

1. 순서성 : 데이터들이 순서대로 나열되어 있기 때문에, 각 데이터의 위치를 인덱스나 순서로 쉽게 파악한다.
2. 1 : 1 관계 : 데이터는 앞 뒤 데이터와 하나씩 연결되어 있다.
3. 접근 방식 : 데이터에 접근하기 위해서 일반적으로 처음부터 순차적으로 이동한다.

선형 자료구조의 종류

1. 배열
2. 리스트
3. 스택 후입선출(LIFO) 구조
4. 큐 선입선출(FIFO) 구조
5. 양방향 큐

배열 은 동일한 자료형의 값들을 순서대로 저장하는 기본적인 선형 자료구조이다.

배열의 특징

1. 순차적 저장 : 메모리 상에서 연속된 공간에 데이터가 순차적으로 저장이 된다.
2. 인덱스 기반 : 각 요소는 인덱스를 가지며 인덱스를 통해서 요소에 접근 할 수 있다.
3. 동일 자료형 : 배열에 저장되는 모든 요소는 동일한 자료형으로 이루어져야 한다.
4. 고정된 크기 : 배열의 크기는 생성 시에 정해지며, 이후에 크기를 변경하는 것은 어렵다.
5. 중복 허용 : 동일한 값을 여러 개 저장할 수 있다.

자바에서 배열 선언시

```java
//정수형 배열 선언 및 초기화
int[] numbers = new int[4]; //크기가 5인 정수형 배열 생성

//배열 요소에 값 할당
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;

//배열 요소 출력
for (int i = 0; i < numbers.length; i++) {
    System.out.printl(numbers[i]);
}

//배열 초기화 시 값 할당
String[] names = {"대한", "민국", "만세};
```

다차원 배열 선언시

```java
int[][] matrix = new int[3][4]; //3행 4열의 2차원 배열
```

[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]

matrix 변수 초기화 형태

`int[][]` : 2차원 정수형 배열을 의미한다.
`matrix` : 배열의 이름이다.

`new int[3][4]` : 3행 4열의 2차원 배열을 생성한다.

리스트 는 데이터를 순서대로 저장하는 선형 자료구조이다. 크기가 동적으로 변할 수 있고 데이터를 추가하거나 삭제할 때 마다 크기를 조절할 수 있어 유연하게 사용할 수 있다.

리스트의 특징

1. 순차적 저장 : 데이터가 순서대로 저장된다.
2. 인덱스 기반 : 요소는 인덱스를 가지고 인덱스를 통해 요소에 접근할 수 있다.
3. 동일 자료형 : 저장되는 모든 요소는 동일한 자료형이어야 한다.
4. 가변적인 크기 : 데이터를 추가하거나 삭제해도 크기가 자동으로 조절된다.
5. 중복 허용 : 동일한 값을 여러 개 저장할 수 있다.

자바에서 리스트 선언시

자바에서는 List 인터페이스를 통해 리스트를 사용하며 ArrayList와 LinkedList가 대표적인 구현체이다.

```java
import java.util.ArrayList;
import java.util.List;

public class ListExample {
    public static void main(String[] args) {
        List<String>  listData = new ArrayList<>();
        //데이터 추가
        listData.add(1);
        listData.add(2);
        listData.add(3);
        //데이터 접근
        System.out.println(listData.get(1)); //banana 출력
        //데이터 삭제
        listData.remove(0); //인덱스 0의 데이터 삭제
        //리스트 크기 확인
        System.out.println(listData.size()); // 2출력
    }
}
```

배열 리스트 (ArrayList)

배열을 기반으로 연속된 메모리 공간에 데이터를 저장한다.

배열 리스트 특징

1. 랜덤 접근 빠름 : 연속된 메모리 공간에 데이터를 저장하므로 인덱스를 이용하여 메모리 주소를 직접 계산할 수 있다. 데이터의 크기에 상관없이 일정한 시간에 특정 요소에 접근할 수 있어랜덤 접근시 연결 리스트 보다 빠르다. (O(1) 시간 복잡도)
2. 데이터 삽입,삭제 느림 : 연속된 메모리 공간을 사용하기 때문에 중간에 데이터를 삽입하거나

삭제하려면 메모리 블록 전체를 이동해야하고 데이터를 이동할 때 기존 데이터를 새로운 위치로 복사해야 하므로 추가적인 메모리 연산이 필요해 시간이 오래 걸린다. (O(n) 시간 복잡도)
5. 메모리 효율성 : 배열을 기반으로 연속된 메모리 공간에 데이터를 저장하기 때문에 메모리 효율성이 좋다.

데이터를 순서대로 저장하고 자주 랜덤 접근을 해야하는 경우 많이 사용한다.

연결 리스트 (LinkedList)

노드를 기반으로 객체를 연결하여 데이터를 저장한다. 각 노드는 데이터 와 다음 노드를 가리키는 포인터를 가지고 있다.

연결 리스트 특징
1. 랜덤 접근이 느림 : 특정 인덱스의 데이터를 찾으려면 처음부터 노드를 따라가야 하므로 시간이 오래 걸린다. (O(n) 시간 복잡도)
2. 데이터 삽입,삭제가 빠름 : 특정 위치에 데이터를 삽입하거나 삭제할 때 앞뒤 노드의 포인터만 변경하면 되므로 빠르다.
3. 메모리 비효율적 : 각 노드에 포인터를 저장해야 하므로 배열 리스트에 비해 메모리를 더 많이 사용한다.

데이터를 자주 삽입하거나 삭제해야 하는 경우 스택이나 큐를 구현할 때 많이 사용한다.

스택 은 가장 최근에 추가된 데이터가 가장 먼저 삭제되는 후입선출(LIFO) 방식의 선형 자료구조이다.

스택의 특징

1. 후입선출 (LIFO) : 가장 나중에 들어온 데이터가 가장 먼저 나간다.
2. 한쪽 끝에서만 접근 : 데이터의 삽입과 삭제가 스택의 한쪽 끝에서만 이루어진다.
3. 스택의 시간 복잡도 : push, pop 연산은 일반적으로 O(1)의 시간 복잡도를 가진다.

자바에서 스택 선언시

```
import java.util.Deque;
import java.util.ArrayDeque;

public class StackExample {
    public static void main(String[] args) {
        Deque<String> stack = new ArrayDeque<>();
        //스택처럼 사용
        stack.push("aaa");
        stack.push("bbb");
        stack.push("ccc");

        //스택의 맨 위 요소 확인
        System.out.println("맨 위의 요소: " + stack.peek());
        //ccc

        //스택의 맨 위 요소 제거
        while (!stack.isEmpty()) {
            System.out.println(stack.pop());
        }
        //스택에 특정 요소가 있는지 확인
        System.out.println(stack.contains("bbb"));
        //false
    }
}
```

java.util.Stack 클래스는 더 이상 권장되지 않고 ArrayDeque 클래스로 스택과 큐의 기능을 구현한다. ArrayDeque클래스는 Deque 인터페이스를 구현한 클래스이다.

주요 메소드

메소드	설명
push(e)	스택의 맨 위에 요소 추가

메소드	설명
pop()	스택의 맨 위 요소 제거 및 반환
peek()	스택의 맨 위 요소 확인
isEmpty()	스택이 비어 있는지 확인
contains(Object o)	스택에 특정 객체가 있는지 확인 (정확한 위치는 제공하지 않음)

큐 는 먼저 들어온 데이터가 먼저 나가는 특징을 가지고 있는 선형 자료구조이다.

큐의 특징

1. 선입선출 : 먼저 들어온 데이터가 먼저 나간다.
2. 삽입 : 큐의 뒷부분에 데이터를 추가한다.
3. 삭제 : 큐의 앞부분에서 데이터를 삭제한다.
4. 단방향 접근 : 일반적으로 큐의 앞부분과 뒷부분에서만 데이터에 접근할 수 있다.

자바에서 큐 (List) 선언시

```java
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        //데이터 추가
        queue.offer(11);
        queue.offer(22);
        queue.offer(33);
        //데이터 확인 및 삭제
        while (!queue.isEmpty()) {
            System.out.println(queue.peek()); //맨 앞 데이터 확인
            int data = queue.poll(); //맨 앞 데이터 삭제 및 반환
            System.out.println(data);
        }
    }
}
```

offer : 큐의 뒷부분에 데이터를 추가
poll : 큐의 앞부분에서 데이터를 삭제하고 반환 만약 큐가 비어있으면 null을 반환
peek : 큐의 앞부분에 있는 데이터를 확인 큐를 변경하지 않고 데이터만 확인
isEmpty : 큐가 비어있는지 확인

양방향 큐 는 큐의 양쪽 끝에서 데이터를 삽입하거나 삭제할 수 있는 선형 자료구조이다.
스택과 큐의 기능을 모두 갖고 있어서 다양한 알고리즘 문제 해결에 유용하게 활용된다.

양방향 큐의 특징

1. 양방향 연산 : 앞 또는 뒤에서 데이터를 추가하거나 삭제할 수 있다.
2. 스택과 큐의 결합 : 스택처럼 후입선출 방식으로 데이터를 처리하거나 큐처럼 선입선출 방식으로 처리가 할 수 있다.
3. 유연성 : 다양한 알고리즘 문제에 적용이 가능하다.

자바에서 Deque 사용

자바에서는 **java.util.Deque** 인터페이스를 통해 **Deque**를 사용할 수 있으며 이 인터페이스를 구현한 클래스로는 **ArrayDeque**와 **LinkedList** 를 사용한다.

ArrayDeque: 배열을 기반으로 구현되어 있어 랜덤 접근이 빠르지만 크기 조정 시 오버헤드(어떤 작업을 수행하는 데 있어 실제로 필요한 작업 외에 추가적으로 소모되는 자원이나 시간을 의미)가 발생할 수 있다.
LinkedList: 연결 리스트를 기반으로 구현되어 있어 삽입/삭제 연산이 빠르지만 랜덤 접근이 느리다.

결론: 대부분의 연산에서 ArrayDeque가 LinkedList보다 빠르지만 배열의 크기 조정이 자주 발생하는 경우에는 LinkedList가 더 효율적일 수 있다.

자바에서 양방향 큐 선언시

```
import java.util.ArrayDeque;
import java.util.Deque;

public class DequeExample {
    public static void main(String[] args) {
        Deque<Integer> deque = new ArrayDeque<>();
        // 데이터 추가
        deque.addFirst(10);
        deque.addLast(20);
        deque.addFirst(5);

        // 데이터 출력
        while (!deque.isEmpty()) {
            System.out.print(deque.removeFirst() + " ");
        }
    }
}
```

메소드:

addFirst(e) : 큐의 앞에 요소 e를 추가
addLast(e) : 큐의 뒤에 요소 e를 추가
removeFirst() : 큐의 앞에서 요소를 제거하고 반환
removeLast() : 큐의 뒤에서 요소를 제거하고 반환
peekFirst() : 큐의 앞에 있는 요소를 반환하지만 제거하지는 않음
peekLast() : 큐의 뒤에 있는 요소를 반환하지만 제거하지는 않음

List 인터페이스

데이터를 입력한 순서대로 유지하는 선형 자료구조이다. 데이터는 고유한 번호(인덱스)를 가지고 순서대로 나열된다.

주요 특징

1. 순서 유지 : 데이터가 입력된 순서대로 저장된다.
2. 중복 허용 : 같은 값을 가진 데이터를 여러개 저장할 수 있다.
3. 인덱스 기반 접근 : 각 데이터에 부여된 인덱스를 이용하여 특정 데이터에 접근한다.

List 인터페이스로 구현되는 클래스 ArrayList, LinkedList

List 인터페이스의 주요 메소드

`boolean add(e)` : 리스트 끝에 요소 e 추가 성공 시 true 반환
`void add(i,e)` : 지정된 인덱스 위치에 요소 삽입
`boolean addAll(c)` : 다른 컬렉션 c의 모든 요소를 리스트 끝에 추가 성공 시 true 반환
`boolean addAll(i,c)` : 다른 컬렉션 c의 모든 요소를 지정된 인덱스 위치부터 추가
`void clear()` : 리스트의 모든 요소 삭제
`boolean contains(o):` : 리스트에 지정된 객체 o 포함 여부 확인
`boolean containsAll(c):` : 리스트에 다른 컬렉션 c의 모든 요소 포함 여부 확인
`E get(i):` : 지정된 인덱스 위치의 요소 반환
`int indexOf(o):` : 지정된 객체 o가 처음으로 나타나는 인덱스 반환 없으면 -1 반환
`int lastIndexOf(o):` : 지정된 객체 o가 마지막으로 나타나는 인덱스 반환 없으면 -1 반환
`ListIterator<E> listIterator():` : 리스트의 요소를 순서대로 탐색할 수 있는 ListIterator 객체 반환
`ListIterator<E> listIterator(i):` : 지정된 인덱스부터 시작하여 리스트의 요소를 순서대로 탐색할 수 있는 ListIterator 객체 반환
`E remove(int index):` : 지정된 인덱스 위치의 요소 삭제,삭제된 요소 반환
`boolean remove(o):` : 리스트에서 처음으로 나타나는 지정된 객체 o 삭제 성공 시 true 반환
`boolean removeAll(c):` : 리스트에서 다른 컬렉션 c의 모든 요소 삭제 변경 여부 반환
`boolean retainAll(c):` : 리스트에 다른 컬렉션 c의 요소만 남기고 나머지 삭제 변경 여부 반환
`E set(i,e):` : 다른 컬렉션 c의 모든 요소를 지정된 인덱스 위치부터 추가
`int size():` : 리스트의 요소 개수 반환
`List<E> subList(int fromIndex, int toIndex):` : 지정된 범위의 요소를 포함하는 새로운

리스트 반환

`Object[] toArray()` : 리스트의 모든 요소를 Object 배열로 변환하여 반환

`<T> T[] toArray(T[] a)` : 리스트의 모든 요소를 지정된 타입의 배열로 변환하여 반환

Queue 인터페이스

먼저 들어온 데이터가 먼저 나가는 선입선출(FIFO) 방식으로 데이터를 관리하는 선형 자료구조이다.

주요 특징

1. 순서 유지 : 데이터가 입력된 순서대로 처리된다.
2. 한쪽 끝에서 추가, 다른 쪽 끝에서 삭제 : 새로운 데이터는 뒤에 추가되고 가장 오래된 데이터는 앞에서 삭제된다.

Queue 인터페이스로 구현되는 클래스 LinkedList, ArrayDeque

Queue 인터페이스의 주요 메소드

`boolean add(e)` : 큐의 끝에 요소를 추가 큐가 가득 차 있으면 IllegalStateException을 발생

`boolean offer(e)` : 큐의 끝에 요소를 추가 큐가 가득 차 있으면 false를 반환하고 성공하면 true를 반환

`E remove()` : 큐의 맨 앞 요소를 제거하고 반환 큐가 비어 있으면 NoSuchElementException을 발생

`E poll()` : 큐의 맨 앞 요소를 제거하고 반환 큐가 비어 있으면 null을 반환

`E element()` : 큐의 맨 앞 요소를 반환 큐가 비어 있으면 NoSuchElementException을 발생

`E peek()` : 큐의 맨 앞 요소를 반환 큐가 비어 있으면 null을 반환

Deque 인터페이스

양쪽 끝에서 데이터를 추가하거나 삭제할 수 있는 선형 자료구조를 나타내는 인터페이스이다. 스택과 큐 기능이 모두 제공된다.

주요 특징

1. 양방향 접근 : 데이터를 앞쪽 또는 뒤쪽에서 삽입하고 삭제할 수 있다.

2. 유연성 : 다양한 자료구조를 구현하는 데 사용될 수 있다. 스택, 큐, 데크 자체로 사용된다.
3. 효율성 : 일반적으로 배열이나 연결 리스트를 기반으로 구현되며 삽입 및 삭제 연산의 시간 복잡도는 O(1)이다.

Deque 인터페이스로 구현되는 클래스 LinkedList, ArrayDeque

Deque 인터페이스의 주요 메소드
`void addFirst(e)` : 덱의 맨 앞에 요소 e 추가 덱이 가득 차면 IllegalStateException
`void addLast(e)` : 덱의 맨 뒤에 요소 e 추가 덱이 가득 차면 IllegalStateException
`boolean offerFirst(e)` : 덱의 맨 앞에 요소 e 추가 (성공 여부 확인)
`boolean offerLast(e)` : 덱의 맨 뒤에 요소 e 추가 (성공 여부 확인)
`E removeFirst()` : 덱의 맨 앞 요소 제거 및 반환 덱이 비어 있으면 NoSuchElementException
`E removeLast()` : 덱의 맨 뒤 요소 제거 및 반환 덱이 비어 있으면 NoSuchElementException
`E pollFirst()` : 덱의 맨 앞 요소 제거 및 반환 (비어있으면 null)
`E pollLast()` : 덱의 맨 뒤 요소 제거 및 반환 (비어있으면 null)
`E getFirst()` : 덱의 맨 앞 요소 반환 덱이 비어 있으면 NoSuchElementException
`E getLast()` : 덱의 맨 뒤 요소 반환 덱이 비어 있으면 NoSuchElementException
`E peekFirst()` : 덱의 맨 앞 요소 반환 (비어있으면 null)
`E peekLast()` : 덱의 맨 뒤 요소 반환 (비어있으면 null)

ArrayList 클래스
배열을 기반으로 구현된 동적 배열이다. 랜덤 접근이 빠르지만 중간에 데이터를 삽입,삭제할 때는 배열의 크기를 조절해야 하므로 성능이 저하될 수 있다. 연속적인 메모리 공간을 사용하여 캐시 효율성이 좋다(ArrayList를 사용할 때 CPU가 데이터에 더 빠르게 접근할 수 있다는 것을 의미). ArrayList는 데이터의 크기가 미리 예측 가능한 경우 순차적인 접근이 주된 경우 사용하는 것이 좋다.

주요 메소드

메소드	설명	시간 복잡도	예외
boolean add(e)	리스트의 끝에 요소 e를 추가	O(1) (평균), O(n) (최악)	
add(i,e)	지정된 위치 index에 요소 element를	O(n)	IndexOutOfBoundsException (index가 유효하지 않은 경우)

메소드	설명	시간 복잡도	예외
	삽입		
boolean addAll(c)	지정된 컬렉션 c의 모든 요소를 리스트의 끝에 추가	O(n)	
boolean addAll(i,c)	지정된 위치 index부터 컬렉션 c의 모든 요소를 삽입	O(n)	IndexOutOfBoundsException (index가 유효하지 않은 경우)
E get(i)	지정된 위치 index의 요소를 제거하고 반환	O(1)	IndexOutOfBoundsException (index가 유효하지 않은 경우)
E remove(i)	지정된 위치 index의 요소를 제거하고 반환	O(1)	IndexOutOfBoundsException (index가 유효하지 않은 경우)
boolean remove(o)	리스트에서 처음 발견되는 요소 o를 제거하고 성공 여부를 반환	O(n)	
E (기존 요소) set(i,e)	지정된 위치 index의 요소를 element로 변경	O(1)	IndexOutOfBoundsException (index가 유효하지 않은 경우)
int size()	리스트의 크기(요소 개수)를 반환	O(1)	
boolean isEmpty()	리스트가 비어 있는지 여부를 반환	O(1)	
boolean contains(o)	리스트에 요소 o가 포함되어 있는지 여부를 반환	O(n)	
int indexOf(Object o)	리스트에서 처음으로 요소 o가 나타나는 인덱스를 반환 없으면 -1을 반환	O(n)	
int lastIndexOf(o)	o가 나타나는 인덱스를 반환 없으면 -1을 반환	O(n)	
void clear()	리스트의 모든 요소를 제거	O(n)	
Object[] toArray()	리스트의 모든 요소를 담은 Object 배열	O(n)	

메소드	설명	시간 복잡도	예외
	을 반환		
T[] toArray(T[] a)	리스트의 모든 요소를 지정된 배열 a에 복사하고 필요하면 새로운 배열을 생성하여 반환	O(n)	
List subList(int fromIndex, int toIndex)	지정된 범위의 요소를 포함하는 새로운 리스트를 반환	O(1)	IndexOutOfBoundsException (index가 유효하지 않은 경우), IllegalArgumentException (fromIndex > toIndex)
ListIterator iterator()	리스트를 순회하기 위한 Iterator를 반환	O(1)	
ListIterator listIterator()	리스트를 양방향으로 순회하기 위한 ListIterator를 반환	O(1)	
ListIterator listIterator(int index)	지정된 위치부터 시작하는 ListIterator를 반환	O(n)	IndexOutOfBoundsException (index가 유효하지 않은 경우)

LinkedList 클래스

노드를 기반으로 객체를 연결하여 데이터를 저장한다. 중간에 데이터를 삽입,삭제하는 것이 빠르지만 특정 인덱스에 접근하려면 처음부터 순차적으로 찾아가야 하므로 랜덤 접근 성능이 떨어진다. 메모리 공간이 비연속적으로 분리되어 있어 캐시 효율성도 좋지 않다. LinkedList는 데이터를 자주 삽입,삭제해야 하는 경우 데이터의 크기가 자주 변하는 경우 스택이나 큐 처럼 양쪽 끝에서 데이터를 추가하거나 삭제해야하는 경우 사용하는 것이 좋다.

주요 메소드

메소드	설명	시간 복잡도	예외
boolean add(e)	리스트의 끝에 요소 e를 추가	O(1)	
void add(i,e)	지정된 인덱스에 요소를 삽입	O(n)	IndexOutOfBoundsException (index가 유효하지 않을 때)

메소드	설명	시간 복잡도	예외
boolean addAll(c)	리스트의 끝에 다른 컬렉션 c의 모든 요소를 추가	O(n)	NullPointerException (c가 null일 때)
boolean addAll(i,c)	지정된 인덱스부터 다른 컬렉션 c의 모든 요소를 삽입	O(n)	IndexOutOfBoundsException, NullPointerException
void addFirst(E e)	리스트의 처음에 요소 e를 추가	O(1)	
void addLast(E e)	리스트의 끝에 요소 e를 추가 (add(e)와 동일)	O(1)	
void clear()	리스트의 모든 요소를 제거	O(n)	
boolean contains(o)	리스트에 지정된 객체가 포함되어 있는지 확인	O(n)	
E get(i)	지정된 인덱스의 요소를 반환	O(n)	IndexOutOfBoundsException
int indexOf(o)	지정된 객체가 처음으로 나타나는 인덱스를 반환 (없으면 -1)	O(n)	
boolean isEmpty()	리스트가 비어있는지 확인	O(1)	
int lastIndexOf(o)	지정된 객체가 마지막으로 나타나는 인덱스를 반환 (없으면 -1)	O(n)	
boolean offer(e)	리스트의 끝에 요소 e를 추가 (add(e)와 동일)	O(1)	
boolean offerFirst(e)	리스트의 처음에 요소 e를 추가	O(1)	
boolean offerLast(e)	리스트의 끝에 요소 e를 추가 (add(e)와 동일)	O(1)	
E peek()	리스트의 첫 번째 요소를 반환 (리스트가 비어있으면 null)	O(1)	
E peekFirst()	리스트의 첫 번째 요소를 반환 (peek()와 동일)	O(1)	
E peekLast()	리스트의 마지막 요소를 반환	O(1)	

메소드	설명	시간 복잡도	예외
E pop()	리스트의 첫 번째 요소를 제거하고 반환 (poll()과 동일)	O(1)	NoSuchElementException (리스트가 비어있을 때)
void push(e)	리스트의 처음에 요소 e를 추가 (addFirst(e)와 동일)	O(1)	
E remove()	리스트의 첫 번째 요소를 제거하고 반환 (poll()과 동일)	O(1)	NoSuchElementException
E remove(i)	지정된 인덱스의 요소를 제거	O(n)	IndexOutOfBoundsException
boolean remove(o)	리스트에서 처음으로 나타나는 지정된 객체를 제거	O(n)	
E removeFirst()	리스트의 첫 번째 요소를 제거 (poll()과 동일)	O(1)	NoSuchElementException
E removeLast()	리스트의 마지막 요소를 제거	O(1)	NoSuchElementException
E set(i,e)	지정된 인덱스의 요소를 변경	O(n)	IndexOutOfBoundsException
int size()	리스트의 크기를 반환	O(1)	

ArrayDeque 클래스

양쪽 끝에서 삽입,삭제가 매우 빠르며 스택과 큐로 사용하기에 적합하다. LinkedList에 비해 캐시 효율성이 좋다. ArrayDeque는 양쪽 끝에서 데이터를 자주 추가하거나 삭제해야하는 경우 사용하는 것이 좋다.

주요 메소드

메소드	설명	시간 복잡도	예외
boolean add(e)	컬렉션의 마지막에 요소를 추가	O(1)	IllegalStateException (컬렉션이 가득 찼을 때)

메소드	설명	시간 복잡도	예외
void addFirst(e)	컬렉션의 처음에 요소를 추가	O(1)	IllegalStateException
void addLast(e)	컬렉션의 마지막에 요소를 추가 (add(e)와 동일)	O(1)	IllegalStateException
boolean offer(e)	컬렉션의 마지막에 요소를 추가 (add(e)와 동일)	O(1)	
boolean offerFirst(e)	컬렉션의 처음에 요소를 추가	O(1)	
boolean offerLast(e)	컬렉션의 마지막에 요소를 추가 (add(e)와 동일)	O(1)	
E peek()	컬렉션의 첫 번째 요소를 반환 (컬렉션이 비어있으면 null)	O(1)	
E peekFirst()	컬렉션의 첫 번째 요소를 반환 (peek()와 동일)	O(1)	
E peekLast()	컬렉션의 마지막 요소를 반환	O(1)	
E poll()	컬렉션의 첫 번째 요소를 제거하고 반환 (컬렉션이 비어있으면 null)	O(1)	
E pollFirst()	컬렉션의 첫 번째 요소를 제거하고 반환 (poll()과 동일)	O(1)	
E pollLast()	컬렉션의 마지막 요소를 제거하고 반환	O(1)	
E pop()	컬렉션의 첫 번째 요소를 제거하고 반환 (poll()과 동일)	O(1)	NoSuchElementException (컬렉션이 비어있을 때)
void push(e)	컬렉션의 처음에 요소를 추가 (addFirst(E e)와 동일)	O(1)	
E remove()	컬렉션의 첫 번째 요소를 제거하고 반환 (poll()과 동일)	O(1)	NoSuchElementException
E removeFirst()	컬렉션의 첫 번째 요소를 제거 (poll()과 동일)	O(1)	NoSuchElementException
E removeLast()	컬렉션의 마지막 요소를 제거	O(1)	NoSuchElementException

정리표

자료구조	구현 방식	장점	단점	주요 사용 시나리오
ArrayList	배열	랜덤 접근 빠름, 캐시 효율성 좋음	중간 삽입/삭제 느림	데이터 자주 읽고 수정, 크기 미리 예측 가능
LinkedList	연결 리스트	중간 삽입/삭제 빠름	랜덤 접근 느림, 캐시 효율성 낮음	데이터 자주 삽입/삭제, 크기 자주 변함, 스택/큐
ArrayDeque	배열	양쪽 끝 삽입/삭제 빠름, 캐시 효율성 좋음		스택/큐, 양쪽 끝에서 데이터 자주 추가/삭제

시간 복잡도 설명

O(1) : 수행 시간이 입력 데이터의 크기에 상관없이 일정한다.
O(n) : 수행 시간이 입력 데이터의 크기에 비례하여 증가한다.

선택 가이드

주요 연산 : 랜덤 접근, 중간 삽입삭제, 양쪽 끝 삽입/삭제 등
데이터 크기 : 미리 예측 가능한지, 자주 변하는지
메모리 사용 : 연속적인 메모리 공간이 필요한지
캐시 효율성 : 중요한 요소인지