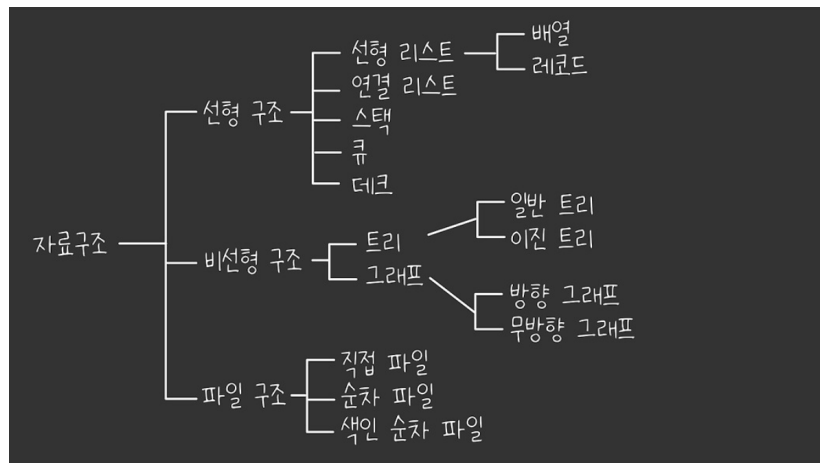
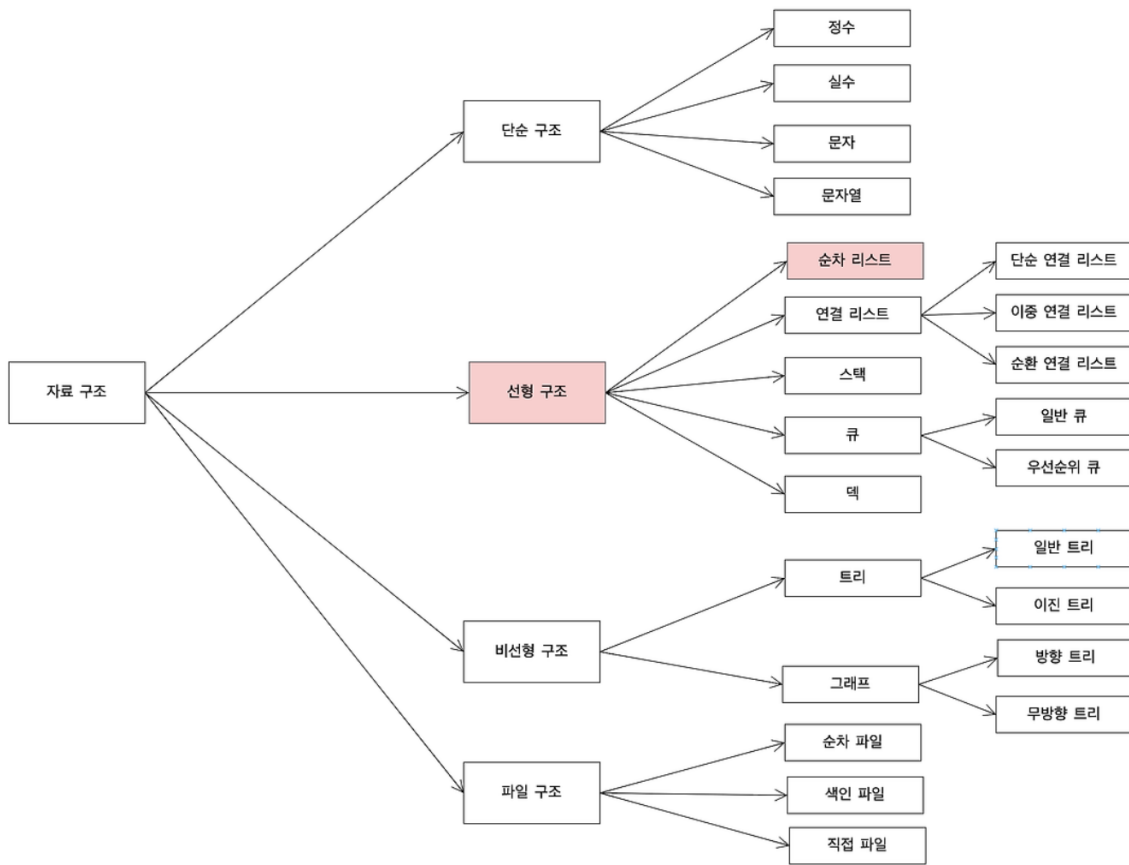


[자료구조] 선형 구조

≡ Tags	
✧ S/NS	In progress
≡ 유형	





▼ 선형 리스트(배열, 레코드)

- 데이터 요소들이 **순서대로** 배치되어 있는 선형구조
- 각 요소는 다음 요소와 연결되어 있으며 각 요소는 **인덱스**를 통해 접근 가능!

💡 배열

- 메모리의 연속 공간에 값이 채워져 있는 형태의 자료구조
- **동일한 타입**의 데이터 요소를 저장하는데 사용되며, 고정된 크기를 가지고 있어서 동일한 배열의 유형과 크기를 지정하여 생성하여 사용한다

🖋️ 특징

- 인덱스를 사용하여 요소에 빠르게 접근 가능: 키 값을 이용하여 접근하는 방식보다 인덱스로 접근하는 방식이 빠름
- 메모리에 연속적으로 저장되기 때문에 연속 접근이 가능함

- 요소들이 동일한 자료형을 가지고 있기 때문에 데이터 처리가 가능
- 크기가 고정되어 있어 추가적인 메모리 할당이 필요하지 않음
- 배열의 요소들이 순서가 있기 때문에 정렬된 데이터를 다루기에 유용하다

활용

```
// 1. 크기와 값이 둘다 없는 빈 배열
String[] strArr1 = {};

// 2. 크기는 존재하지만 값이 없는 빈 배열
String[] strArr2 = new String[3];

// 3. 크기가 자동 지정된 값이 있는 배열
String[] strArr3 = {"one", "two", "three"};
```

? 배열의 크기만 지정하고 값을 할당하지 않으면 무슨 값을 가질까?

int → 0

boolean → false

char → null

String → null

float, double → 0.0

객체자료형(Integer, Boolean, String) → null


배열의 값 할당

값이 없는 빈 배열에 값을 할당하거나 인덱스를 기반으로 값을 할당할 수 있다


```
// 1. 빈 배열에 값 넣기
String[] strArr1 = new String[3];
strArr1[0] = "one";
strArr1[1] = "two";

// 2. 빈 배열에 사이즈를 할당하고 값 넣기
```

```
String[] strArr2 = {};
strArr2 = new String[3];
for(int i=0; i<strArr2.length; i++) {
    strArr2[i] = Integer.toString(i);
}
```

 2차원 배열: 행과 열로 구성된 1차원 배열을 모아서 2차원 배열을 형성한다

```
// 생성
int[][] matrix2Array = new int[3][4];
// 접근
for(int i=0; i< matrix2Array.length; i++) {
    for(int j=0; i<matrix2Array[i].legnth; j++) {
        System.out.print(array[i][j] + " ");
    }
}
```

 Arrays를 collection 함수인 ArrayList로 전환을 하거나 ArrayList를 Array로 전환하는 방법

- Array는 정적으로 초기에 사이즈를 지정해야 배열을 구성할 수 있고, length()로 배열의 사이즈를 확인
- ArrayList는 초기에 사이즈를 지정할 필요가 없고 동적으로 사이즈를 지정하여 배열을 구성할 수 있고, size()로 배열의 사이즈를 확인할 수 있음

```
// 1. Array 초기화
String[] strArr = {"one", "two", "three"};

// 2. 배열 -> 컬렉션 함수, 배열 리스트 선언 및 초기화
List<String> strArrList = new ArrayList<>(Arrays.asList(strArr));

// 3. 컬렉션 함수 -> 배열, 사이즈 지정 및 배열 리스트 값을 전환
strArr = strArrList.toArray(new String[strArrList.size()]);
```

```
// 4. 컬렉션 함수 -> 배열, 사이즈 지정 및 배열 리스트 값을 전환
for(int j=0; j<strArrList.size(); j++) {
    strArr[j] = strArrList.get(j);
}
```

Arrays를 이용한 배열 조작

```
import java.util.Arrays 를 해야함
```

Arrays.copyOf(array, copyArrLength)

배열 전체를 복사하여서 복사할 길이만큼 지정하여 복사한 새로운 배열로 반환해주는 함수
원본의 배열의 사이즈를 넘어가면 0으로 값을 채워줌

```
int[] arr = {0,1,2,3,4};

// 동일한 배열을 복사
int[] newArray1 = Arrays.copyOf(arr, arr.length); //[0,1,2,3,4]
int[] newArray2 = Arrays.copyOf(arr, 3); // [0,1,2]
int[] newArray3 = Arrays.copyOf(arr, arr.length+3); [0,1,2,3,4,0,0,0]
```

Arrays.copyOfRange(array, startIndex, endIndex)

원본 배열의 시작 인덱스와 끝 인덱스를 지정하여서 복사한 새로운 배열로 반환해주는 함수

```
int[] arr = {0,1,2,3,4};
int[] copyArrIdx = Arrays.copyOfRange(arr, 0, arr.length+2);
```

분류	Arrays.CopyOf()	Arrays.copyOfRange
복사 시작 인덱스 지정 가능 여부	○	○
복사 끝 인덱스 지정 가능 여부	X	○
복사 사이즈 초과시 값 채워줄 여부	○	○

 Arrays.fill(array, value), Arrays.fill(array, startIndex, endIndex, value)

배열 내에 지정된 범위 내에 '동일한 값'으로 채워주는 함수


```
// 모든 빈 공간에 채워주는 fill
int[] fillArr1 = new int[5];
Arrays.fill(fillArr1, 369);

// 모든 공간에 같은 값 채워주는 방식
int[] fillArr2 = new int[5];
fillArr2[0] = 123;
fillArr2[1] = 456;
fillArr2[2] = 789;
Arrays.fill(fillArr2, 369) // [369, 369, 369, 369, 369]

// 특정 구간에 채워주는 방식
int[] fillArr3 = new int[5];
fillArr3[0] = 123;
fillArr3[1] = 456;
fillArr3[2] = 789;
Arrays.fill(fillArr3, 3, fillArr3.length, 369); // [123,456,789,369,369]

// fill 함수와 repeat 함수를 사용하여 순차적인 별 채워넣기
String[] fillArr4 = new String[5];
for(int i=0; i<fillArr4.length; i++) {
    Arrays.fill(fillArr4, i, i+1, "*".repeat(i, i+1)); // ""
}
}
```

```
// 배열의 길이보다 더 채워주는 경우 -> 에러 발생 Array index out of
```

 `Arrays.setAll(array, generator)`

배열을 채우는데 사용하는 메소드이며, 배열의 값은 순차적으로 메소드로 '구성한 값'으로 채워주는 함수

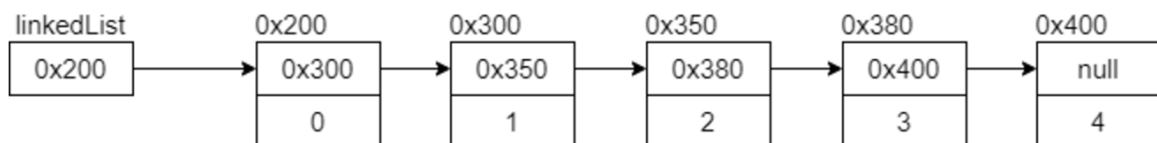
```
// setAll을 통해 배열 내에 변형된 숫자를 채워넣음
int[] arr1 = new int[5];
Arrays.setAll(arr1, i -> (int)(Math.random() * 101)); //64, 1:
```

▼ 연결 리스트

자바의 linked list도 arraylist과 같이 인덱스로 접근하여 조회, 삽입이 가능하지만 내부 구조는 다르다

arraylist는 내부적으로 배열을 이용하여 메소드로 조작이 가능하게 만든 컬렉션이면, linked list는 노드(객체)끼리의 주소 포인터를 서로 가리키며 참조(링크)함으로써 이어지는 구조이다

1. 단순 연결 리스트



```
clas node {
    node next; // 다음 노드 주소를 저장하는 필드
```

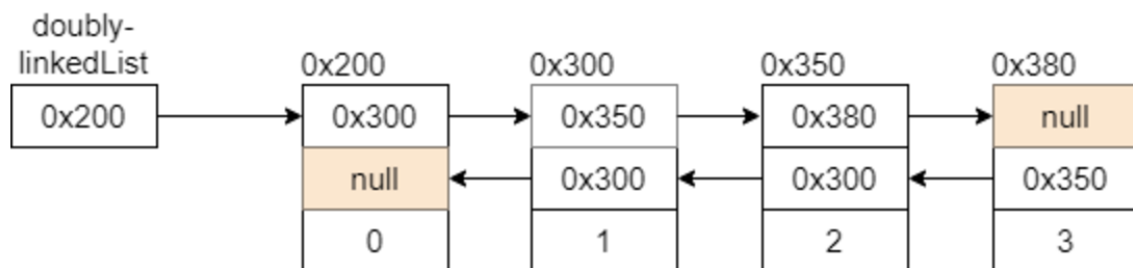
```
int data; // 데이터를 저장하는 필드
};
```

다음 노드를 가리키기 위한 포인터 필드 next만을 가지고 있는 linked list를 말함

! 현재 요소에서 이전 요소로 접근해야할 때 매우 부적합함

타고타고타고아가야함 → 해결책이 이중 연결 리스트이다

2. 이중 연결 리스트



```
class node {
    node next; // 다음 노드 주소를 저장하는 필드
    node prev; // 이전 노드 주소를 저장하는 필드
    int data; // 데이터를 저장하는 필드
}
```

역순으로도 검색이 가능함, 각 요소에 대한 접근과 이동이 쉽기 때문에 많이 사용

!! 자바의 컬렉션 프레임워크에 구현된 LinkedList 클래스는 이중 연결 리스트이다

3. 양방향 원형 연결 리스트

첫번째 노드와 마지막 노드를 각각 연결시켜, 마치 원형 리스트처럼 만들었다

LinkedList vs ArrayList 비교

	ArrayList	LinkedList
컬렉션 구성	배열을 이용	노드를 연결
데이터 접근 시간	모든 데이터 상수 시간 접근	위치에 따라 이동시간 발생

삽입/삭제 시간	삽입/삭제 시 데이터 이동이 필요한 경우 추가시간 발생	삽입/삭제 위치에 따라 그 위치까지 이동하는 시간 발생
리사이징 필요	공간이 부족할 경우 새로운 배열에 복사하는 추가 시간 발생	
데이터 검색	최악의 경우 리스트에 있는 item수만큼 확인	최악의 경우 리스트에 있는 item수만큼 확인
CPU Cache	캐시 이점을 활용 ?????	

LinkedList 클래스 역시 List 인터페이스를 구현하므로, ArrayList 클래스와 사용할 수 있는 메소드는 거의 비슷함

LinkedList 객체 생성

- LinkedList(): LinkedList 객체를 생성
- LinkedList(Collection c): 주어진 컬렉션을 포함하는 LinkedList 객체를 생성

```
// 라이브러리 가져와야함
import java.util.LinkedList;

// 타입 설정 int 타입만 적재 가능
LinkedList<Integer> list = new LinkedList<>();

// 생성시 초기값 설정
LinkedList<Integer> list2 = new LinkedList<Integer>(Arrays.asList(1, 2, 3, 4, 5));
```

LinkedList 요소 추가/삽입

메소드	설명
void addFirst(Object obj)	맨 앞에 객체를 추가
void addLast(Object obj)	맨 뒤에 객체를 추가
boolean add(Object obj)	마지막에 객체를 추가하고 성공하면 true반환

<code>void add(int index, Object element)</code>	지정된 위치에 객체를 저장
<code>void addAll(Collection c)</code>	주어진 컬렉션의 모든 객체를 저장
<code>void addAll(int index, Collection c)</code>	지정된 위치부터 주어진 컬렉션의 데이터를 저장

추가되거나 삭제될 노드의 위치의 바로 앞뒤쪽에 있는 노드의 참조를 변경하기만 하면 된다
`addFirst(Object obj)`, `addLast(Object obj)`는 요소를 첫번째, 마지막에 추가하는 것이기 때문에 $O(1)$ 의 시간이 걸리는데, 중간 삽입일 경우 $O(N)$ 의 시간이 걸린다

LinkedList 요소 삭제

메소드	설명
<code>Object removeFirst()</code>	첫 번째 노드 제거
<code>Object removeLast()</code>	마지막 노드를 제거
<code>Object remove()</code>	LinkedList의 첫 번째 요소를 제거
<code>Object remove(int index)</code>	지정된 위치에 있는 객체를 제거
<code>boolean remove(Object obj)</code>	지정된 객체를 제거
<code>boolean removeAll(Collection c)</code>	지정된 컬렉션에 저장된 것과 동일한 노드들을 LinkedList에서 제거
<code>boolean retainAll(Collection c)</code>	LinkedList에 저장된 객체 중에서 주어진 컬렉션과 공통된 것들만 남기고 제거
<code>void clear()</code>	LinkedList를 비움

`clear()`는 참조 체인을 따라가면서 일일이 null로 설정해주기때문에 $O(N)$ 의 시간이 걸림

LinkedList 요소 검색

요소 자체가 리스트에 있는지 검사하는 `contains()`와 인덱스 위치도 반환해주는 `indexOf()`가 존재

메소드	설명
<code>int size()</code>	LinkedList에 저장된 객체의 개수를 반환
<code>boolean isEmpty()</code>	비어있는지 확인

boolean contains(Object obj)	지정된 객체가 LinkedList에 포함되어 있는지 확인
boolean containsAll(Collection c)	지정된 컬렉션의 모든 요소가 포함되었는지 알려줌
int indexOf(Object obj)	지정된 객체가 저장된 위치를 찾아 반환
int lastIndexOf(Object obj)	지정된 객체가 저장된 위치를 뒤에서부터 역방향으로 찾아 반환함

```
LinkedList<String> list = new LinkedList<>(Arrays.asList("a",
// 해당 값을 가지고 있는 요소 위치를 반환함, 찾고자 하는 값이 없으면 -1
list1.lastIndexOf("D"));
```

LinkedList 요소 얻기, 변경

메소드	설명
Object get(int index)	지정된 위치에 저장된 객체를 반환
List subList(int fromIndex, int toIndex)	fromIndex부터 toIndex사이에 저장된 객체를 List로 반환함
Object set(int index, Object obj)	지정된 위치의 객체를 주어진 객체로 바꿈

LinkedList 배열 변환

메소드	설명
Object[] toArray()	LinkedList에 저장된 모든 객체들을 객체 배열로 반환
Object[] toArray(Object[] objArr)	LinkedList에 저장된 모든 객체들을 객체배열에 담아 반환

```
LinkedList<Number> list1 = new LinkedList<>();
list1.add(1);
list1.add(2);
```

```
Number[] arr = (Number[])list1.toArray();
System.out.println(Arrays.toString(arr)); [1,2]
```

```
LinkedList<Number> list1 = new LinkedList<>();
list1.add(1);
list1.add(3);
list1.add(4);

Object[] tmp = {0,1,2,3,4,5};

Number[] arr = (Number[])list1.toArray(tmp);
System.out.println(Arrays.toString(arr)); // [1, 2, null, 3,
// 1,3,4,null,4,5
```

toArray(Object[] objArr) 메소드의 결과값 배열 출력에서 null이 삽입되어 있는 이유는 자바 메서드 스펙이다, javadoc에 따르면 삽입된 리스트의 길이를 알리기 위해서 null을 넣는다

LinkedList 순회

```
LinkedList<String> list = new LinkedList<>();
// for문을 이용한 순회
for(String data : list) {
    System.out.println(data);
}
```

메서드	설명
Iterator iterator()	LinkedList의 Iterator 객체를 반환
ListIterator listIterator()	LinkedList의 ListIterator 객체를 반환
ListIterator listIterator(int index)	LinkedList의 지정된 위치부터 시작하는 ListIterator를 반환

Collection 인터페이스에서는 Iterator 인터페이스를 구현한 클래스의 인스턴스를 반환하는 iterator() 메소드를 정의하여 각 요소에 접근하도록 정의하고 있다. 따라서 Collection 인터페이스를 상속받는 List, Set 인터페이스에서도 iterator() 메소드를 사용할 수 있음! map 사용 불가

```
Iterator it = list.iterator();

while(it.hasNext()) {
    Object obj = it.next();
    System.out.println(obj);
}
```

LinkedList 스택 큐

메소드	설명
Object element()	LinkedList에 첫 번째 노드를 반환
boolean offer(Object obj)	지정된 객체를 LinkedList의 끝에 추가, 성공하면 true
Object peek()	LinkedList의 첫 번째 요소를 반환
Object poll()	LinkedList의 첫 번째 요소를 반환, LinkedList에서는 제거
void push(Object obj)	맨 앞의 객체를 추가
Iterator descendingIterator()	역순으로 조회하기 위한 DescendingIterator를 반환
Object getFirst()	LinkedList의 첫 번째 노드를 반환
Object getLast()	LinkedList의 마지막 노드를 반환
boolean offerFirst(Object obj)	지정된 객체를 LinkedList의 맨 앞에 추가, 성공하면 true
boolean offerLast(Object obj)	지정된 객체를 LinkedList의 맨 뒤에 추가, 성공하면 true
Object peakFirst()	첫번째 노드 반환
Object peakLast()	마지막 노드를 반환
Object pollFirst()	첫번째 노드를 반환하면서 제거

Object pollLast()	마지막 노드를 반환하면서 제거
Object pop()	첫번째 노드를 제거

연결 리스트 직접 구현하기

데이터의 저장 순서가 유지되고 중복을 허용함

인덱스로 요소를 접근하지만, 배열이 아니기 때문에 별도의 탐색시간이 걸려 임의의 요소에 대한 접근 성능은 좋지 않다

대신 데이터의 중간 삽입, 삭제가 빈번할 경우 빠른 성능을 보장한다

```
public class 이중연결리스트 {
    private Node<E> head; // 노드의 첫 부분을 가리키는 필드
    private Node<E> tail; // 노드의 끝 부분을 가리키는 필드

    private int size; // 리스트 요소 갯수

    // 생성자
    public 이중연결리스트() {
        this.head = null;
        this.tail = null;
        this.size = 0;
    }
    // 노드
    private static class Node<E> {
        private E item; // Node에 담을 데이터
        private Node<E> next; // 다음 Node 객체를 가리키는 필드
        private Node<E> prev; // 이전 Node 객체를 가리키는 필드
        Node(Node<E> prev, E item, Node<E> next) {
            this.item = item;
            this.next = next;
            this.prev = prev;
        }
    }
}
```

```

    }
}

```

- 왜 클래스 안에 클래스를 선언했지? → 다른 클래스에서는 필요 없어서
- private으로 했지? → 보안상의 문제
- 왜 static을 붙였는가? → memory leak 이슈 때문에(내부 클래스는 static으로 선언!)

검색 구현

추가, 삭제를 구현하기 앞서 검색을 구현할건데

- 만일, 인덱스가 0에 가깝다면 순차 방향 탐색
- 만일, 인덱스가 size(마지막)에 가깝다면 역 방향 탐색

```

private Node<E> search(int index) {
    Node<E> n; // 반환할 노드
    // 1. 만일 인덱스가 시작에 가까우면, 순차 탐색
    if((size/2) > index) {
        n = head;
        for(int i=0; i<index; i++) {
            n = n.next;
        }
    }
    // 2. 만일 인덱스가 끝에 가까우면, 역순 탐색
    } else {
        n = tail;
        for(int i=size-1; i>index; i--) {
            n = n.prev;
        }
    }
    return n;
}

```

추가 구현

1. addFirst 구현

```
public void addFirst(E value) {
    Node<E> first = head;
    // 1. 새 노드 추가: 첫 번째 노드니까 prev는 null이고, next는 head
    Node<E> new_node = new Node<>(null, value, first);
    // 2. 노드 추가했으니 리스트 크기 증가
    size++;
    // 3. 첫번째 기준이 변경되었으니 head를 삽입된 새 노드로 참조하도록
    head = new_node;
    // 4. 만약, 빈 리스트에서 최초의 요소 추가하였을 경우, tail도 첫째
    if(first == null) {
        tail = new_node;
        // 5. 빈 리스트가 아닐경우, 추가되기 이전 첫번째노드였던 노드에.
    } else {
        first.prev = new_node;
    }
}
```

2. addlast 구현

3. add 중간 삽입 구현

4. removefirst 구현

5. remove 구현

6. removelast 구현

7. 인덱스로 remove 구현

8. 값으로 remove 구현

9. get/set 구현

10. indexOf 구현

11. size

12. isEmpty

13. clear

14. contains

15. toString

▼ 스택

- 후입선출 구조, 직전의 데이터를 빠르게 가져올 수 있음

구현

```
public class 스택<E> {  
    private static final int DEFAULT_CAPACITY = 6; // 최소 용량  
    private Object[] arr; // 요소를 담을 내부 배열  
    private int top; // 가장 마지막 요소를 가리키는 포인터  
  
    public 스택() {  
        this.arr = new Object[DEFAULT_CAPACITY];  
        this.top = -1;  
    }  
}
```

isFull/isEmpty 구현

```
public boolean isFull() {  
    return top == arr.length - 1;  
}  
  
public boolean isEmpty() {  
    return top == -1;  
}
```

resize 구현

스택에 요소가 추가, 삭제 등의 동작이 될때 호출된다

```
private void resize() {
    int arr_capacity = arr.length-1;
    // 용량이 꽉찬 경우
    if(top == arr_capacity){
        // 넉넉하게 공간 지정
        int new_capacity = arr.length * 2;
        // 복사할 배열을 new_capacity 용량 만큼 설정하고 arr 원소들을
        arr = Arrays.copyOf(arr, new_capacity);
        return;
    }

    // 용량에 비해 데이터 양이 적은 경우
    if(top < (arr_capacity/2)) {
        // 공간 작게 지정
        int half_capacity = arr.length/2;
        arr = Arrays.copyOf(arr, Math.max(half_capacity, DEFAULT_CAPACITY));
        return;
    }
}
```

Push 구현

```
public E push(E value) {
    // 1. 배열이 꽉 차있으면
    if(isFull) {
        resize(); // 배열 리사이징 실행
        top++; // 2. 원소를 추가할 예정이니 마지막 요소 위치인 top + 1
        arr[top] = value; // 3. 해당 배열 위치에 요소를 추가
        return value; // 4. 넣은 요소 값 반환
    }
}
```

Pop 구현

```

public E pop() {
    // 1. 배열이 비어있으면 예외 발생
    if(isEmpty()) throw new EmptyStackException();
    // 2. 먼저 arr[top] 원소 백업
    E value = arr[top];
    // 3. 해당 위치의 요소를 삭제
    arr[top] = null;
    // 4. top을 -1 감소시킴
    top--;
    // 5. 들어있는 요소에 비해 빈공간이 많으면 최적화를 위해 리사이징
    resize();
    // 6. 백업한 요소 반환
    return value;
}

```

Peek 구현

```

public E peek() {
    // 1. 배열이 비어있으면 예외 발생
    if(isEmpty()) throw new EmptyStackException();
    // 2. 스택의 마지막 원소 값만 반환
    return (E) arr[top];
}

```

search 구현

```

public int search(E value) {
    // 스택의 맨 위부터 아래로 순회하여 찾고자 하는 값의 위치를 구한다
    for(int i=top; i>=0; i--) {
        if(arr[i].equals(value)) {
            return top-i+1;
        }
    }
}

```

```
    return -1;
}
```

toString 구현

```
public String toString() {
    return Arrays.toString(arr);
}
```

▼ 큐

FIFO 구조

- enqueue: 큐 맨 뒤에 데이터 추가
 - dequeue: 큐 맨 앞쪽의 데이터를 삭제
1. 큐의 한쪽 끝은 front로 정하여 삭제연산만 수행
 2. 다른 한쪽 끝은 rear로 정하여 삽입연산만 수행
 3. 그래프 넓이 우선 탐색(BFS)에서 사용
 4. 컴퓨터 버퍼에서 주로 사용

queue 클래스를 이용해 구현

자바에서 제공하는 queue 클래스를 이용해 구현해보자

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

public class Main {
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<Integer>();
        queue.offer(1);
    }
}
```

```

        queue.offer(2);

        while(!queue.isEmpty()) {
            System.out.println(queue.poll());
        }
    }
}

```

배열을 이용해서 구현

```

public class ArrayQueue {
    int MAX = 1000;
    int front; // pop할때 참조하는 인덱스
    int rear; // push할때 참조하는 인덱스
    int[] queue;
    public ArrayQueue(){
        front = rear = 0;
        queue = new int[MAX]; // 배열 생성
    }
    // 차있는지 확인하는 함수
    public boolean queueisFull() {
        if(rear == MAX-1) {return true;}
        else { return false;}
    }
    // 큐에 들어가있는 데이터의 개수
    public void size() {
        return front-rear;
    }
    // 넣는 함수
    public void enqueue(int value) {
        if(queueisFull()) {
            return;
        }
        queue[rear++] = value; // rear가 위치한 곳에 넣어주고 rear
    }
}

```

```

// 빼는 함수
public int dequeue() {
    if(queueisEmpty()) {
        return -1;
    }
    int popValue = queue[front++];
    return popValue;
}

public int peek() {
    if(queueisEmpty()) {
        return -1;
    }
    int popValue = queue[front];
    return popValue;
}
}

```

배열로 구현한 큐는 push, pop이 간단하지만 배열의 크기가 유한해 큐의 크기가 다 차버리면 사용할 수 없다.

LinkedList을 이용해서 queue 구현

똑같이 node를 이용해서 큐를 구현한다, 값이 들어갈 node와 queue를 관리할 nodeManager 클래스를 만들면 된다

```

public class QueueNode {
    int value;
    QueueNode queueNode; // 다음 노드를 가리킴
    public QueueNode(int value) {
        this.value=value;
        queueNode = null;
    }
    public int getValue() {
        return value;
    }
}

```

```

    public QueueNode getNextNode() {
        return queueNode;
    }
    public void setNextNode(QueueNode queueNode) {
        this.queueNode = queueNode;
    }
}

public class QueueNodeManager {
    QueueNode front, rear;
    public QueueNodeManager() {
        front = rear = null;
    }
    // 비어있는지 확인하는 메소드
    public boolean queueisEmpty() {
        if(front == null && rear == null) {
            return true;
        } else {
            return false;
        }
    }
    // 넣기
    public void push(int value) {
        QueueNode queueNode = new QueueNode(value);
        // 큐 안에 데이터가 없으면 첫번째 노드에 front, rear
        if(queueisEmpty()) {
            front = rear = queueNode;
            // 큐 안에 데이터가 있으면 front를 다음 노드에 연결
        } else {
            front.setNextNode(queueNode);
            front.queueNode;
        }
    }
    // 빼기
    public QueueNode pop() {
        if(queueisEmpty()) {

```

```

        return null;
    } else {
        QueueNode popNode = rear;
        rear = rear.getNextNode();
        return popNode;
    }
}
// peek()
public QueueNode peek() {
    if(queueisEmpty()) {
        return null;
    } else {
        return rear;
    }
}
// 크기 구하는 메소드
public int size() {
    QueueNode front2 = front;
    QueueNode rear2 = rear;
    int count = 0;
    while(front2 != rear2 && rear2 != null) {
        count++;
        rear2 = rear2.getNextNode();
    }
    return count;
}
}

```

연결리스트로 큐를 만들게 되면 구현은 복잡하지만 큐의 크기도 무한하고 큐안에 원하는 기능을 넣을 수 있다

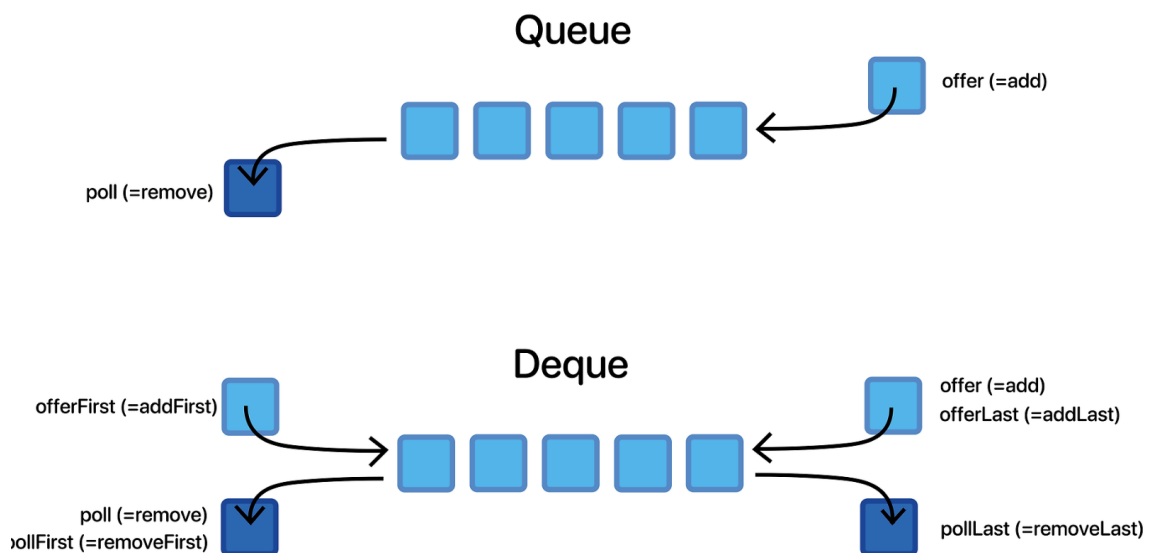
우선순위 큐

우선순위 큐는 힙 자료구조를 기반으로 구현된다

힙은 노드를 활용하여 연결리스트처럼 구현하는 방식이 있고, 배열을 활용하여 구현하는 방법이 있음

▼ 데크

양방향 큐



삽입, 삭제가 총 12개가 있다

offer, poll은 삽입과 삭제 과정에서 저장 공간이 넘치거나 삭제할 원소가 없을 때 특정 값 (null, false)를 반환하지만, add계열과 remove계열의 경우 예외를 던진다

```

public class ArrayDeque<E> implements Queue<E> {
    private static final int DEFAULT_CAPACITY = 64;
    private Object[] array;
    private int size;
    private int front;
    private int rear;
    // 초기 할당을 안할 경우
    public ArrayDeque() {
        this.array = new Object[DEFAULT_CAPACITY];
        this.size = 0;
        this.front = 0;
        this.rear = 0;
    }
    // 초기 할당을 할 경우
    public ArrayDeque(int capacity) {
        this.array = new Object[capacity];
        this.size = 0;
        this.front = 0;
        this.rear = 0;
    }
}

```

▼ vector