

# 10반 알고리즘 스터디 1반

## Week 1

좌표이동

정렬

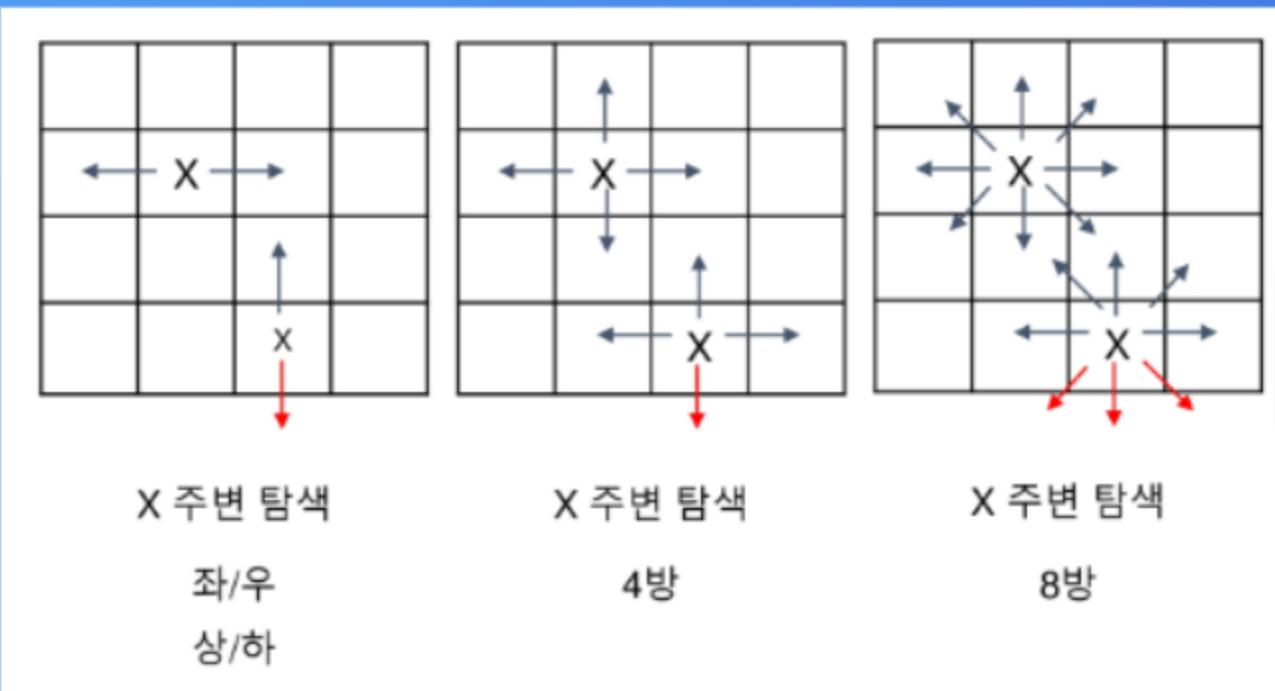
BFS  
DFS

구현  
LinkedList

etc

# dx dy 테크닉

방향에 따라 좌표를 갱신하기 위해 사용하는 기법



1. 방향  $\Rightarrow$  int[] 배열로 나타낸다.
2. 범위  $\Rightarrow$  boolean 메소드로 불력화!

```
// 상/하 만 이동하는 경우  
static int[] rx = {-1,0};  
static int[] ry = {0,1};  
  
// 8방향 (대각선) 까지 포함하는 경우  
static int[] px = {-1,-1,-1,0,1,1,1,0};  
static int[] py = {-1,0,1,1,1,0,-1,-1};
```

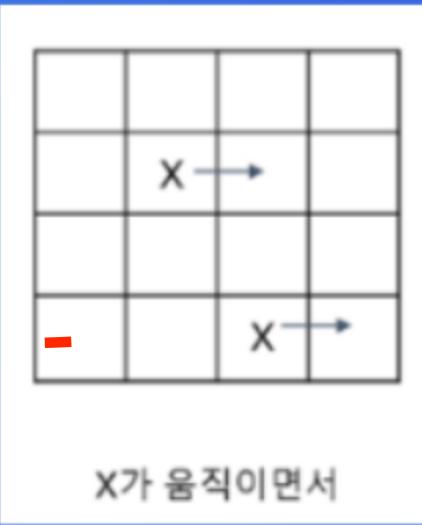
```
public class Main {  
  
    static int[][] myMap; // 2차원 격자  
    static int x, y; // 특정 객체의 좌표  
  
    // 1. 방향을 설정하는 배열  
    static int[] dx = {-1,0,1,0};  
    static int[] dy = {0,1,0,-1};  
    // 2. 격자 범위에 있는지 확인하는 함수  
    public static boolean inRange(int x, int y) {  
        return ((0 <= x && x < N) && (0 <= y && y < N));  
    }  
  
    public static void move(int x, int y) {  
        // 3. 방향의 배열 길이만큼 좌표를 이동한 값을 체크  
        for(int i = 0; i < 4; i++) {  
            int nx = x + dx[i];  
            int ny = y + dy[i];  
            if(inRange(nx, ny) && ... ) { // 이외 추가 조건 ( 대표적으로 visited, myMap 값 등 )  
                ....  
            }  
        }  
    }  
  
    public static void main(String[] args) { ... } // myMap input && 알고리즘 구현  
}
```

# dx dy 테크닉 (2)



Idea : for문을 활용,  
Row 추출 & Col 추출

```
public static void move() {  
    // Search Row  
    for(int x = 0; x < N; x++) {  
        int[] Rows = myMap[x];  
  
        doSomething();  
    }  
  
    // Search Column  
    for(int y = 0; y < N; y++) {  
        int[] Col = new int[N];  
  
        for(int x = 0; x < N; x++)  
            col[x] = myMap[x][y];  
  
        doSomething();  
    }  
  
    // 대각선 경우 => dx & dy 테크닉 활용 ...  
}
```



Idea : 범위를 벗어날 때 까  
지, 진행  
벗어난다면, 방향을 반대로  
설정  
=> while

ex) 테트리스 유형

```
// 1. 방향을 설정하는 배열  
static int[] dx = {0, -1, 1, 0};  
static int[] dy = {1, 0, 0, -1};  
// 2. 격자 범위에 있는지 확인하는 함수  
public static boolean inRange(int x, int y) {  
    return ((0 <= x && x < N) && (0 <= y && y < N));  
}  
  
public static void main(String[] args) {  
    ....  
  
    int x = a;  
    int y = b;  
    int dir = 1;  
  
    // 메인 로직  
    while( inRange(x + dx[dir], y + dy[dir]) ) {  
        // 이동한 좌표 체크  
        int nx = x + dx[dir], ny = y + dy[dir];  
  
        // 범위를 벗어나면 => 방향 반대로 전환  
        if( !inRange(nx, ny) )  
            dir = 3 - dir;  
  
        // 실제 좌표 갱신  
        x, y = x + dx[dir], ny = y + dy[dir];  
    }  
}
```

# 정렬 (기초)

⇒ `Arrays.sort( dataList, Comparator(Compare Func) )`: 임의 정렬

⇒ 원하는 기준이 존재한다면 비교자 인자 위치에 람다 처리

⇒ 단, 기준 2개 이상이면, 동적 객체 생성으로

Comparator 내에 compare 함수로 기준 구현!

```
public class Main {
    public static void main(String[] args){

        int[] arr = {1, 26, 17, 25, 99, 44, 303};
        // 오름차순 정렬
        Arrays.sort(arr);
        System.out.println("Sorted arr[] : " + Arrays.toString(arr));

        // 내림차순 정렬
        Arrays.sort(arr, Collections.reverseOrder());
        Arrays.sort(arr, (i1, i2) -> i2 - i1); // Collections 모듈 사용 불가능 시, 람다 처리
        System.out.println("Sorted arr[] : " + Arrays.toString(arr));

        // int 배열, 부분 정렬
        Arrays.sort(arr, 0, 4); // (배열, 시작idx, 끝idx)
        System.out.println("Sorted arr[] : " + Arrays.toString(arr));

        String[] arr2 = {"Apple", "Kiwi", "Orange", "Banana", "Watermelon", "Cherry"};
        // String 배열, 아스키 문자열 순 (오름차순)
        Arrays.sort(arr2);
        System.out.println("Sorted arr[] : " + Arrays.toString(arr2));

        // String 배열, 문자열 길이 순 (오름차순)
        Arrays.sort(arr, (s1, s2) -> s1.length() - s2.length());
        System.out.println("Sorted arr[] : " + Arrays.toString(arr));
    }
}
```

# 정렬(객체)

⇒ 원하는 객체 안에 Comparable 를 구현해 비교할 수 있도록 처리

=> Collection STL : Collection.sort()

⇒ 단, 여러 기준이 존재한다면 Comparator 활용

```
public class Solution {  
  
    public static class Position implements Comparable<Position> {  
        int x, y;  
  
        public Position(int x, int y) {  
            super();  
            this.x = x;  
            this.y = y;  
        }  
  
        @Override  
        public int compareTo(@NotNull Position o) {  
            if (this.x == o.x) {  
                return this.y - o.y;  
            }  
            return this.x - o.x;  
        }  
    }  
  
    public static void main(String[] args) {  
    }  
}
```

11650번: 좌표 정렬하기 (acmicpc.net)

```
// Fruit 클래스는 Comparable<Fruit>를 구현  
// 클래스에 Comparable을 구현하여 sort()가 객체를 비교할 수 있도록 처리  
public static class Fruit implements Comparable<Fruit> {  
    private String name;  
    private int price;  
    public Fruit(String name, int price) {  
        this.name = name;  
        this.price = price;  
    }  
  
    @Override  
    public String toString() {  
        return "{name: " + name + ", price: " + price + "}";  
    }  
  
    // 이 클래스의 compareTo는 자기자신의 클래스와 인자로 전달되는 Fruit 객체의 price를 비교  
    @Override  
    public int compareTo(@NotNull Fruit fruit) {  
        return this.price - fruit.price;  
    }  
}  
  
public class Main {  
    public static void main(String[] args){  
  
        Fruit[] arr = {  
            new Fruit("Apple", 100),  
            new Fruit("Kiwi", 500),  
            new Fruit("Orange", 200),  
            new Fruit("Banana", 50),  
            new Fruit("Watermelon", 880),  
            new Fruit("Cherry", 10)  
        };  
  
        Arrays.sort(arr);  
  
        System.out.println("Sorted arr[] : " + Arrays.toString(arr));  
        // Sorted arr[] : [{name: Cherry, price: 10}, {name: Banana, price: 50},  
        // {name: Apple, price: 100}, {name: Orange, price: 200}, {name: Kiwi, price: 500},  
        // {name: Watermelon, price: 880}]  
    }  
}
```

# DFS

## 기본 로직

1. Graph & Visited

2. dx & dy & inRange

3. Recursive 종료 조건 메소드

4. 핵심 로직

5. DFS

```
public class Test {  
  
    static int N;  
    static int[][] graph;  
    static boolean[][] visited;  
  
    static int[] dx = {0,-1,1,0};  
    static int[] dy = {1,0,0,-1};  
  
    public static boolean inRange(int x, int y) {  
        return ((0 <= x && x < N) && (0 <= y && y < N));  
    }  
  
    public static boolean endCondition(){  
        return true;  
    }  
  
    public static void do_something() {  
        return;  
    }  
  
    public static void dfs(int x, int y) {  
  
        if(endCondition()){  
            do_something(); // 클래스 멤버 변수 등으로 정답값을 갱신하는 로직..  
            return;  
        }  
  
        visited[x][y] = true;  
  
        for(int i = 0; i < 4; i++) {  
            int nx = x + dx[i];  
            int ny = y + dy[i];  
  
            if(inRange(nx,ny) && !visited[nx][ny]){  
                dfs(nx, ny);  
            }  
        }  
    }  
  
    public static void main(String[] args) {  
        // STEP 1. Input && 초기화  
        N = 10;  
        graph = new int[N][N];  
        visited = new boolean[N][N];  
  
        int st_x = 5;  
        int st_y = 5;  
  
        // STEP 2. dfs 시작  
        dfs(st_x, st_y);  
    }  
}
```

# BFS

## 기본 로직

1. Graph & dx & dy & inRange

2. 좌표 객체

3. 핵심 로직 & 종료 조건

4. BFS

1. Queue & data 초기화

2. while ( !isEmpty )

3. 현재 data poll & 로직

4. 4방향 탐색 & queue push

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

public class Test {

    static int N;
    static int[][] graph;
    static boolean[][] visited;

    public static class Position {
        int x, y;

        public Position(int x, int y){
            this.x = x;
            this.y = y;
        }

        static int[] dx = {0,-1,1,0};
        static int[] dy = {1,0,0,-1};

        public static boolean inRange(int x, int y) {
            return ((0 <= x && x < N) && (0 <= y && y < N));
        }

        public static boolean endCondition(){
            return true;
        }

        public static void do_something() {
            return;
        }
    }
}
```

```
public static void bfs(int x, int y) {
    Queue<Position> queue = new LinkedList<Position>();

    queue.offer(new Position(x, y));
    visited[x][y] = true;

    while(!queue.isEmpty()) {
        Position curPos = queue.poll();

        //do_something();

        for(int i = 0; i < 4; i++) {
            int nx = x + dx[i];
            int ny = y + dy[i];

            if(inRange(nx, ny) && !visited[nx][ny]) {
                //do_something();

                queue.offer(new Position(nx, ny));
                visited[nx][ny] = true;
            }
        }
    }
}

public static void main(String[] args) {
    N = 10;
    graph = new int[N][N];
    visited = new boolean[N][N];

    int st_x = 5;
    int st_y = 5;

    bfs(st_x, st_y);
}
```

# DFS 응용 (1)

## DFS를 활용한 조합 구현

(전체 5개 중 3개의 조합을 구할 때 )

조합 sudo code

1. (종료조건) 만약 data 집합이 M개라면

a. 핵심 로직 & return

2. 탐색이 끝에 다다르면, return

3. 집합에 현재 원소를 넣은 경우, 다음 원소 탐색

4. 집합에 현재 원소를 뺀 경우, 다음 원소 탐색

```
public class Main {  
  
    int M = 3;  
    static int[] arr;  
  
    static void doSomething(ArrayList<Integer> a) {  
        return;  
    }  
  
    static void dfs(int depth, ArrayList<Integer> combi) {  
        if(combi.size() == M) {  
            doSomething(combi);  
            return;  
        }  
        else if (depth == arr.length)  
            return;  
  
        combi.add(arr(depth));  
        dfs(depth+1, combi);  
  
        combi.remove(combi.size()-1); // LinkedList => pop ≡  
        dfs(depth+1, combi);  
    }  
  
    public static void main(String[] args) {  
        arr = {1,5,4,3,2};  
  
        dfs(0, new ArrayList<Integer>());  
    }  
}
```

# DFS 응용 (2)

## DFS를 활용한 순열 구현

( 전체 5개 중 3개의 순열을 구할 때 )

순열 sudo code

순서 구분! => visited 선언!

1. (종료조건) 만약 data 집합이 M개라면

a. 핵심 로직 & return

2. 각 원소에 대해

a. 방문했다면 넘어간다.

b. 현재 원소 기준, 집합에 넣고 방문 처리한다.

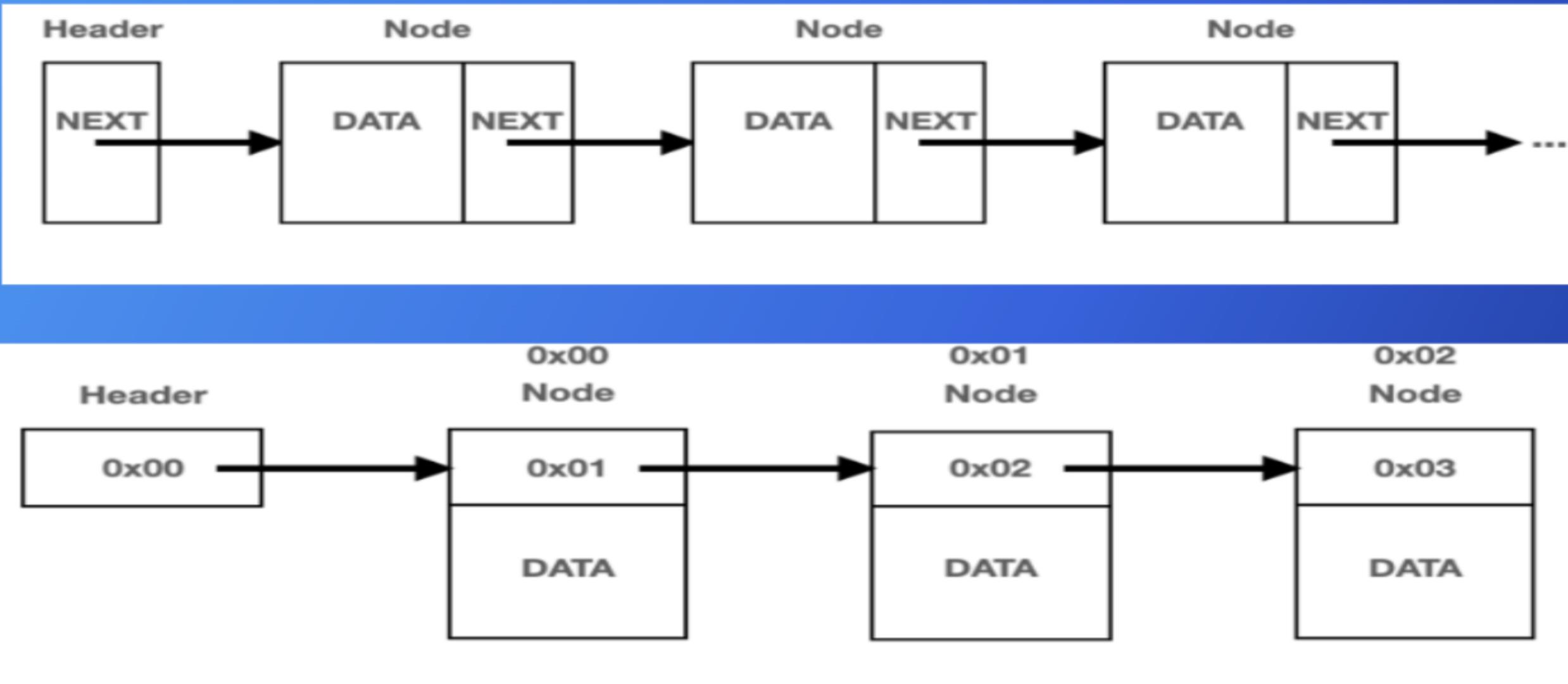
c. 다음 원소 탐색으로 넘어간다.

d. 현재 원소 기준, 집합에서 빼고 방문을 품다.

( visited => 앞 뒤 순서 구분 가능 )

```
public class Main {  
  
    int M = 3;  
    static boolean[] visited;  
    static int[] arr;  
  
    static void doSomething(ArrayList<Integer> a) {  
        return;  
    }  
  
    static void dfs( ArrayList<Integer> perm ) {  
        if( perm.size() == M ) {  
            doSomething(perm);  
            return;  
        }  
  
        for(int i = 0; i < arr.length; i++) {  
            if(visited[i]) continue;  
  
            perm.add(arr[i]);  
            visited[i] = true;  
            dfs( perm );  
            perm.remove(combi.size()-1);  
            visited[i] = false;  
        }  
    }  
  
    public static void main(String[] args) {  
        arr = {1,5,4,3,2};  
        visited = new boolean[5];  
  
        dfs(new ArrayList<Integer>());  
    }  
}
```

# 연결 리스트 (1)

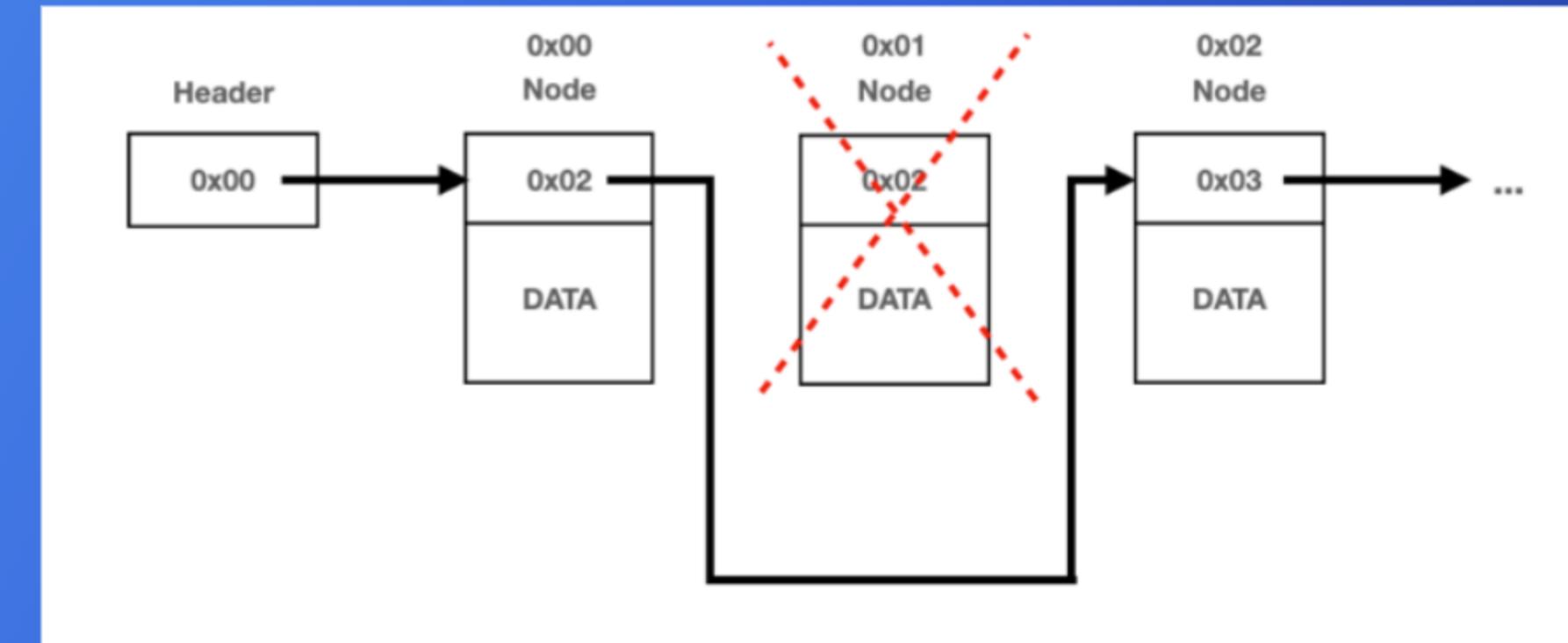


연결 리스트(LinkedList)는 각 노드(Node)가 데이터와 포인터를 가지고 한 줄로 연결되어 있는 방식으로 데이터를 저장하는 자료구조입니다.

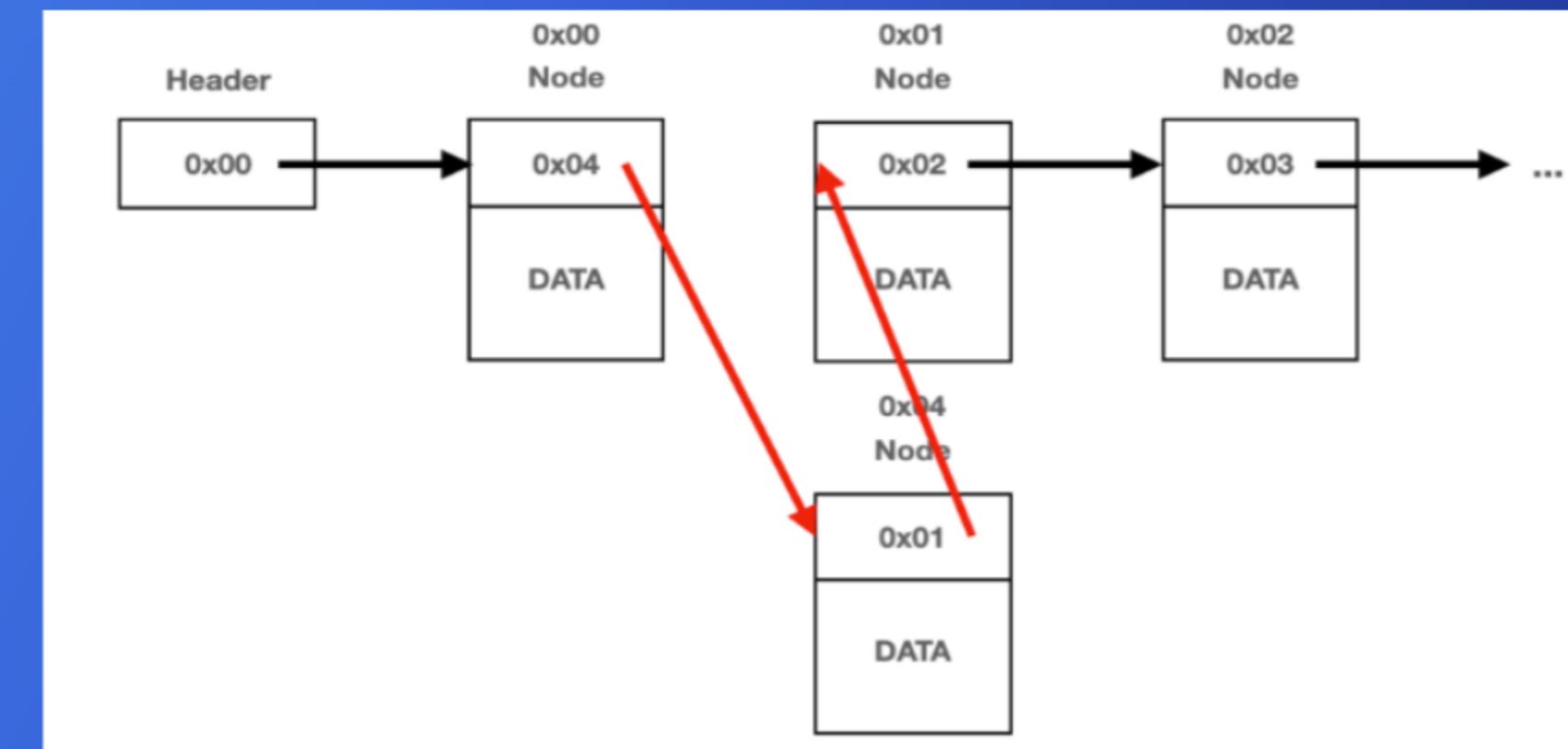
# 연결 리스트 (2)

## - 주요 기능 -

데이터 삭제



데이터 추가



# 연결 리스트 (3)

## - 메소드 -

1. `size()` -> 리스트 요소 개수 반환
2. `isEmpty()` -> 비어있는지 확인
3. `get( index )` -> index 위치에 해당하는 Node get
4. `contains( node )` -> 리스트에 특정 노드가 있는지?
5. `toArray()` -> 리스트를 배열로 반환

1. `addFirst(E data)` -> 리스트 앞에 노드 추가
2. `addLast(E data)` -> 리스트 마지막에 노드 추가
3. `add(E data, index )` -> index 위치에 Node 추가
4. `removeFirst()` -> 리스트 앞에 노드 제거
5. `removeLast()` -> 리스트 마지막에 노드 제거
6. `remove( index )` -> index 위치에 Node 제거

```
public class LinkedList<E> {  
  
    Node<E> first;  
    Node<E> last;  
    private int size = 0;  
  
    public LinkedList() { }  
    private static class Node<E> {  
        E item;  
        Node<E> next;  
        Node<E> prev;  
  
        public Node(E item, Node<E> next, Node<E> prev) {  
            this.item = item;  
            this.next = next;  
            this.prev = prev;  
        }  
    }  
}
```

```
// ***** Basic Func *****

// 리스트 요소 개수를 반환
public int size() {
    return -1;
}

// 리스트가 비어있는지 확인
public boolean isEmpty() {
    return false;
}

// index 위치의 노드를 반환하는 메소드
public E get(int index) {
    return null;
}

// 리스트에 특정 요소가 있는지 확인하는 메소드
public boolean contains(Object obj) {
    return false;
}

// 리스트를 배열로 반환하는 메소드
public Object[] toArray() {
    return null;
}
```

```
// ***** Add Func *****
// 리스트의 맨 앞에 노드를 추가하는 메소드
public void addFirst(E data){
}

// 리스트의 맨 뒤에 노드를 추가하는 메소드
public void addLast(E data) {
}

// 원하는 index 위치에 노드를 추가하는 메소드
public void add(int index, E data) {
}

// ***** Delete Func *****
// 리스트 맨 앞 노드를 제거하는 메소드
public void removeFirst() {
}

// 리스트 맨 뒤 노드를 제거하는 메소드
public void removeLast() {
}

// 리스트 index 위치의 노드를 제거하는 메소드
public void remove(int index) {
}
```

# ※ 최적화 ?

Ex) Get 메소드, 아마 찾으려는 index 위치가 head에 가깝냐, 아니면 tail에 가깝냐에 따라 시작 위치를 바꿀 수 있지 않을까?

대표 문제  
( 원형 리스트 )

백준 1021번 : 회전하는  
큐

백준 1406번 : 에디터

# 숙제

1. 삼성 기출 1 문제 풀기  
( BOJ 16236 : 아기 상어 )
2. LinkedList 구현 & 대표 문제 풀기  
( STL vs 직접구현 비교 )

이글이글