배열(Array) & 리스트(List)&연결 리스트(Linked List)

▼ 배열(Array)

어느 프로그래밍 언어에서든 존재하는 자료구조이다.

자바스크립트에서 배열

자바스크립트의 배열은 배열이 아닌 **일반적인 배열의 동작을 흉내 낸 특화된 객체**이다.

하나의 배열이 다양한 종류의 요소를 포함할 수 있다. 요소를 위한 각각의 메모리 공간은 동일한 크기를 갖지 않아도 되며, 연속적으로 이어져 있지 않을 수도 있다.

```
const arr = [1, 2, 3];
```

length 프로퍼티

배열의 길이를 나타내는 0 이상의 정수를 값으로 갖는다.

```
const arr = [1, 2, 3];
console.log(arr.length);  // 3

arr.push(4);
console.log(arr.length);  // 4
```

배열인지 확인

자바스크립트에서는 배열은 객체로 만들어져 있기 때문에 보통 사용하는 `typeof` 로는 배열인지 알 수가 없다. 그래서 `Array.isArray()` 라는 배열인지 체크하는 메서드를 제공한다.

```
const arr = [1, 2, 3];

console.log(typeof arr);  // object
console.log(Array.isArray(arr));  // true
```

배열 요소 접근

- for문으로 요소 접근

```
let num = [1, 2, 3, 4];
let sum = 0;

for(let i = 0; i < num.length; i++){
  sum += num[i];
}
console.log(sum); //10
```

문자열로 배열 생성

- `.split()` : 문자열 분할 함수

```
let a = "Hllo, Algorithms";
let words = a.split(' ');

console.log(words); //["Hllo,","Algorithms"]
```

▼ 배열 시간 복잡도

- 알고리즘이 문제를 해결하는 데 걸리는 시간을 정량화하여 나타낼 수 있는 방법
- 일반적으로 문제에서 주어지는 최악의 경우에 대한 소요 시간을 나타내는 데 사용한다.
- **Big-O 표기법 - 최악의 경우의 시간 복잡도를 의미**

접근자 함수

배열 요소에 접근할 수 있는 다양한 함수를 제공한다.

- `indexOf()` : 제공되 값이 배열에 존재하는지 알려준다.
- `join()`,`toString()` : 배열을 문자열 형식으로 반환하는 함수.

```
var names = ['David', 'Cynthia', 'Raymond', 'Clayton'];
var namestr = names.join();
console.log(namestr); // David,Cynthia,Raymond,Clayton
var namestr = names.toString();
console.log(namestr); // David,Cynthia,Raymond,Clayton
```

- `concat()` : 두 개 이상의 배열을 합쳐 새 배열을 만든다.
- `splice()` : 배열의 기존 요소를 삭제 또는 교체하거나 새 요소를 추가하여 배열의 내용을 변경한다. 원본 배열 자체를 수정한다.

```
// concat()
var cisDept = ['David', 'Cynthia', 'Raymond', 'Clayton'];
var dmpDept = ['Mike', 'Bryan'];
var itDiv = cisDept.concat(dmpDept);
console.log(itDiv); //[ 'David', 'Cynthia', 'Raymond', 'Clayton', 'Mike', 'Bryan' ]

itDiv = dmpDept.concat(cisDept);
console.log(itDiv); // [ 'Mike', 'Bryan', 'David', 'Cynthia', 'Raymond', 'Clayton' ]

// splice()
itDiv = ['David', 'Cynthia', 'Raymond', 'Clayton','Mike', 'Bryan'];
var dmpDept = itDiv.splice(3, 3);
var cisDept = itDiv;
console.log(dmpDept); // [ 'Clayton', 'Mike', 'Bryan' ]
console.log(cisDept); // [ 'David', 'Cynthia', 'Raymond' ]
```

변형자 함수

자바스크립트는 개별적으로 요소를 건드리지 않고 배열 전체 내용을 고치는 여러 변형자 함수를 제공한다.

- `push()` : 배열의 끝에 요소를 추가한다. length 프로퍼티를 이용해 배열을 확장하는 것보다 더 직관적이다.
- `unshift()` : 한번에 여러 요소를 배열 앞으로 추가할 수 있다.
- `pop()` : 배열의 마지막 요소를 제거할 수 있다.
- `shift()` : 배열의 맨 처음 요소를 제거 할 수 있다.

```
// push()
var nums = [1,2,3,4,5];
console.log(nums); //1,2,3,4,5
nums.push(6);
console.log(nums); // 1,2,3,4,5,6

// unshift()
var nums = [4,5];
console.log(nums) // 4,5

var newnum = 1;
nums.unshift(newnum);
console.log(nums); // 1,4,5

nums = [4,5];
```

```
nums.unshift(newnum, 2);
console.log(nums); // 1,2,4,5

//pop()
var nums = [1,2,3,4,5,8];
nums.pop();
console.log(nums); // 1,2,3,4,5

// shift()
var nums = [9,1,2,3,4,5];
nums.shift();
console.log(nums); // 1,2,3,4,5

//pop(), shift()는 제거된 요소를 반환하므로 필요하다면 반환된 요소를 변수에 저장할 수 있다.
var nums = [6,1,2,3,4,5];
var first = nums.shift();
nums.push(first);
console.log(nums); // 1,2,3,4,5,6
```

- `sort()` : 배열의 요소를 오름차순으로 바꿔준다.
- `reverse()` : 배열의 요소를 내림차순으로 바꿔준다.

```
// sort()
var names = ['David', 'Cynthia', 'Raymond', 'Clayton','Mike', 'Bryan'];
names.sort();
console.log(names); // [ 'Bryan', 'Clayton', 'Cynthia', 'David', 'Mike', 'Raymond' ]

var nums = [3, 1, 4, 100, 202];
nums.sort();
console.log(nums); //[ 1, 100, 202, 3, 4 ]
                   //숫자는 순서대로 정렬되지 않는다.

// reverse()
var nums = [6,5,4,3,2,1];
nums.reverse();
console.log(nums); //[ 1, 2, 3, 4, 5, 6 ]

var names = ['David', 'Cynthia', 'Raymond', 'Clayton','Mike', 'Bryan'];
names.reverse();
console.log(names); // [ 'Bryan', 'Mike', 'Clayton', 'Raymond', 'Cynthia', 'David' ]
                    //문자는 알파벳 순이 아닌 배열을 뒤집어 놓은 값이 나온다.


// array.sort([Compare Function])
// 위의 방법으로 숫자가 제대로 정렬되게 할 수 있다.
var nums = [3, 1, 4, 100, 202];
nums.sort((a, b) => a - b);
console.log(nums); // [ 1, 3, 4, 100, 202 ]

// sort().reverse()를 사용해서 문자열을 순서대로 정렬할 수 있다.
var names = ['David', 'Cynthia', 'Raymond', 'Clayton','Mike', 'Bryan'];
names.sort().reverse();
console.log(names); // [ 'Raymond', 'Mike', 'David', 'Cynthia', 'Clayton', 'Bryan' ]
```

반복자 함수

배열의 각 요소에 함수를 적용한 다음 그 결과 값 또는 값의 집합 또는 새로운 배열을 반환한다.

- `forEach()` : 배열의 모든 요소에 인자로 받은 함수를 호출
- `every()` : 불린 함수를 배열에 적용해 배열의 모든 요소가 참이면 true를 반환
- `some()` : 배열 요소 중에 한 요소라도 인자로 받은 요소의 기준을 만족하면 true를 반환한다.
- `reduce()` : 누적자 함수를 인자로 받은 다음 배열의 모든 요소를 누적자 함수에 적용.

```
// forEach()
function square(num){
    console.log(num, num * num);
}
var nums = [1,2,3,4,5,6,7,8,9];
nums.forEach(square); /* 1 1
                         2 4
                         3 9
                         4 16
                         5 25
                         6 36
                         7 49
                         8 64
                         9 81 */

// every()
function isEven(num){
    return num % 2 == 0;
}

var nums = [2,4,6,8,10];
var even = nums.every(isEven);
if (even){
    console.log('true');
} else{
    console.log('false');
} // true


// some()
function isEven(num){
    return num % 2 == 0;
}

var nums = [1,2,3,4,5,6,7,8,9];
var someEven = nums.some(isEven);
if (someEven){
```

```
    console.log('true');
} else{
    console.log('false');
}  // true
nums = [1,3,5,7,9];
someEven = nums.some(isEven);
if (someEven){
    console.log('true');
} else{
    console.log('false');
} // false


// reduce()
function concat(acc, item){
    return acc + item;
}

var words = ['the ', 'quick ', 'brown ', 'fox'];
var sentence = words.reduce(concat);
console.log(sentence); // the quick brown fox
```

- `map()` : `forEach()`처럼 배열의 각 요소에 함수를 적용.
  다만 배열 요소에 함수를 적용한 결과를 포함하는 새 배열을 반환한다는 점이 다르다.
- `filter()` : `every()` 와 유사하지만, 함수를 만족하는 요소를 포함하는 새로운 배열을 반환한다.

```
//map(), 숫자
function curve(grade){
    return grade += 5;
}

var grades = [34,56,67,83];
var newgrades = grades.map(curve);
console.log(newgrades); //[ 39, 61, 72, 88 ]

//문자열
function first(word){
    return word[1];
}

var words = ['for', 'your', 'information'];
var acronym = words.map(first);
console.log(acronym.join(',')); //o,o,n


//filter()
function isOdd(num){
    return num % 2 != 0;
}
```

```
var nums = [];
for(var i = 0; i < 20; i++){
    nums[i] = i + 1;
}

var odds = nums.filter(isOdd);
console.log('Odd numbers: ' + odds); //Odd numbers: 1,3,5,7,9,11,13,15,17,19
```

이차원 배열과 다차원 배열

- 자바스크립트는 기본적으로 일차원 배열만 지원함
- 액셀 스프레드시트를 생각하면 쉬움
- 가로 방향(row, 행) / 세로 방향(column, 열)
- row를 셀 때에는 세로 방향 갯수를 세고, column을 셀 때에는 가로 방향 갯수를 세면 됨
- 이차원 배열 : 배열에 배열을 중첩하므로 구현 가능

```
var arr = [];
var rows = 5;

for(var i=0; i<5; i++){
    arr[i] = [i, i+1];
}
console.log(arr); //[ [ 0, 1 ], [ 1, 2 ], [ 2, 3 ], [ 3, 4 ], [ 4, 5 ] ]
```

▼ 리스트(List)

리스트라는 이름의 추상적 자료형이 있다. **순서가 존재하며, 일렬로 나열된 값들이 들어있다.**

리스트가 담는 자료에 적용되는 연산

- 조회, 삽입, 삭제 등

▼ List 클래스 구현

- `append()` : 리스트의 다음 변수 값에 새 요소를 추가
- `remove()` : 리스트 클래스의 함수들 중 가장 구현하기 어려운 함수에 속함.
  우선 삭제할 요소를 찾아 삭제하고 나머지 배열 요소를 왼쪽으로 이동시켜 요소가 삭제된 자리를 메워야 한다.

- `find()` : 루프로 리스트의 요소를 검색.
  요소를 발견하면 요소의 위히를 반환. 요소를 발견하지 못하면 찾지 못했을 때 반환하는 표준 값인 -1을 반환.
- `length()` : 리스트의 요소의 수를 반환.
- `toString()` : 리스트의 요소를 확인. 문자열이 아니 배열을 객체를 반환.
- `insert()` : 리스트의 기존 요소 뒤에 새로운 요소를 삽입하게 함.
- `clear()` : delete 명령어로 리스트의 모든 요소를 삭제
- `contains()` : 어떤 값이 리스트에 포함되어 있는지 확인할 때 사용함.
- `getElement()` : 리스트 탐색 관련 기능이 있음.

리스트와 반복

반복자를 이용하면 list 클래스의 내부 저장소를 직접 참조하지 않고 리스트를 탐색할 수 있다.

- `front(), end(), prev(), next(), currPos()`

반복자의 장점

- 리스트 요소에 접근할 때 내부 데이터 저장소가 무엇인지 걱정할 필요가 없다.
- 리스투에 새 요소를 추가했을 때는 현재 인덱스가 쓸모없는 값이 되는 반면, 반복자를 이용하면 리스트가 바뀌어도 반복자를 갱신할 필요가 없다.
- List 클래스에 사용하는 데이터 저장소의 종류가 달라져도 이전과 같은 방식으로 요소에 접근할 수 있다.

▼ 연결 리스트(Linked List)

각 노드가 데이터와 포인터를 가지며, **한 줄로 연결**되어 있는 방식으로 데이터를 저장하는 자료 구조입니다.

특성

- 각각의 노드(Node)들로 구성되어 있습니다.

- `head` 는 처음 노드를 가리킵니다.
- 하나의 노드는 해당 노드의 데이터값인 `data` 와 다음 노드를 가리키는 포인터인 `next` 를 가집니다.
- 다음 노드가 존재하지 않을 경우 `next=null` 이 됩니다.
- 탐색 또는 정렬을 자주 하면 배열을, 추가/삭제가 많으면 연결 리스트를 사용하는 것이 유리하다.
- Javascript에서 연결리스트는 객체를 통해 구현할 수 있다.

```
const n1 = {
  data: 100
}

const n2 = {
  data: 200
}

n1.next = n2;

console.log(n1) // { data: 100, next: { data: 200 } }
```

노드(Node)

연결 리스트에서 각 원소는 원소 자신과 다음 원소를 가리키는 포인터가 포함된 노드(Node)로 구성된다.

'데이터를 저장할 장소'와 '다른 변수를 가리키기 위한 장소'가 구분되어 있다.

Linked List의 이점

- 새로운 elements를 삽입, 삭제 시 용이
- restructuring이 덜 복잡함

Linked List의 단점

- array보다 많은 메모리 사용
- 특정 element를 검색시 비효율적임

연결 리스트의 종류

1. Doubly Linked List (이중 연결 리스트)

연결 리스트에서 탐색을 할 때, 뒤쪽의 원소를 검색하면 loop를 돌아야 하기 때문에 탐색 속도가 많이 느려지게 된다. 이 문제를 해결하려면 뒤쪽 원소들도 빠르게 검색할 수 있는 이중 연결 리스트 구조를 활용하면 된다.

2. Circular Linked List (원형 연결 리스트)

원형 리스트를 사용하면 굳이 Head가 어디인지, Tail이 어디인지 신경 쓸 필요가 없다.

- **장점 :** 마지막 작업하던 위치에서 이어서 탐색, 삽입, 삭제 등의 연산 등을 곧바로 수행할 수 있다.
- **단점 :** 데이터 순서의 개념이 모호해지고, 때문에 인덱스를 이용한 탐색이 어렵다.

배열 vs 연결 리스트 차이

▼ 배열

▼ 장점

- 메모리를 연속적으로 저장
- n번째 원소를 접근할 때 바로 접근할 수 있음.

▼ 단점

- 메모리 사용이 비효율적
- 배열 내의 데이터 추가 및 삭제의 시간 복잡도가 O(n)

▼ 연결 리스트

▼ 장점

- 메모리를 효율적으로 사용
- 삽입, 삭제에 대해 효율적으로 할 수 있음.

▼ 단점

- 캐싱에 적합하지 않은 구조
- 복잡한 연산에 따른 오버헤드 발생
- 주소 저장으로 인한 공간 낭비

양방향 Linked List 구현

```
class Node {
  constructor(element) {
    this.element = element;
    this.next = null;
    this.prev = null;
  }
}

class LinkedList {
  constructor() {
    this.head = new Node("head");
  }

  find(item) {
    let currNode = this.head;
    while (currNode.element !== item) {
      currNode = currNode.next;
    }
    return currNode;
  }

  insert(newElement, item) {
    let newNode = new Node(newElement);
    let current = this.find(item);
    if (current.next == null) {
      newNode.next = null;
      newNode.prev = current;
      current.next = newNode;
    } else {
      newNode.next = current.next;
      newNode.prev = current;
      current.next.prev = newNode;
      current.next = newNode;
    }
  }

  remove(item) {
    let currNode = this.find(item);
    if (currNode.next !== null) {
      currNode.prev.next = currNode.next;
      currNode.next.prev = currNode.prev;
      currNode.next = null;
      currNode.prev = null;
    }
  }
}

const linkedList = new LinkedList();
linkedList.insert("Seoul", "head"); //head->Seoul
linkedList.insert("Busan", "Seoul"); //head->Seoul->Busan
linkedList.insert("Daegu", "Seoul"); //head->Seoul->Daegu->Busan
linkedList.insert("Incheon", "Busan"); //head->Seoul->Daegu->Busan->Incheon
linkedList.remove("Busan"); //head->Seoul->Daegu->Incheon
```

원형 Linked List 구현

```
class Node {
  constructor(element) {
    this.element = element;
    this.next = null;
    this.prev = null;
  }
}

class LinkedList {
  constructor() {
    this.head = new Node("head");
  }

  find(item) {
    let currNode = this.head;
    while (currNode.element !== item) {
      currNode = currNode.next;
    }
    return currNode;
  }

  insert(newElement, item) {
    let newNode = new Node(newElement);
    let current = this.find(item);
    if (current.next == null) {
      newNode.next = null;
      newNode.prev = current;
      current.next = newNode;
    } else {
      newNode.next = current.next;
      newNode.prev = current;
      current.next.prev = newNode;
      current.next = newNode;
    }
  }

  remove(item) {
    let currNode = this.find(item);
    if (currNode.next !== null) {
      currNode.prev.next = currNode.next;
      currNode.next.prev = currNode.prev;
      currNode.next = null;
      currNode.prev = null;
    }
  }
}

const linkedList = new LinkedList();
linkedList.insert("Seoul", "head"); //head->Seoul
linkedList.insert("Busan", "Seoul"); //head->Seoul->Busan
```

```
linkedList.insert("Daegu", "Seoul"); //head->Seoul->Daegu->Busan
linkedList.insert("Incheon", "Busan"); //head->Seoul->Daegu->Busan->Incheon
linkedList.remove("Busan"); //head->Seoul->Daegu->Incheon
```