2주차: 배열, 리스트, 연결 리스트

#배열이란? (Array)

- 자바스크립트에서 배열(array)은 이름과 인덱스로 참조되는 정렬된 값의 집합으로 정의됩니다.

```
var arr = new Array();
var arr = [];
```

  - 배열을 구성하는 각각의 값을 배열 요소(element)라고 하며, 배열에서의 위치를 가리키는 숫자를 인덱스(index)라고 합니다.

```
var arr = [ 1, '2', ture ];
var arr1 = [ undefined, null, false, NaN, 1000 ];
```

#배열의 특징

- 배열 요소의 타입이 고정되어 있지 않으므로, 같은 배열에 있는 배열 요소끼리의 타입이 서로 다를 수도 있습니다.
- 배열 요소의 인덱스가 연속적이지 않아도 되며, 따라서 특정 배열 요소가 비어 있을 수도 있습니다.
- 자바스크립트에서 배열은 Array 객체로 다뤄집니다.

배열의 항목들을 순환하며 처리하기

```
fruits.forEach(function (item, index, array) {
  console.log(item, index)
})
// 사과 0
// 바나나 1
```

배열 끝에 항목 추가하기

```
let newLength = fruits.push('오렌지')
// ["사과", "바나나", "오렌지"]
Copy to Clipboard
```

배열 끝에서부터 항목 제거하기

```
let last = fruits.pop() // 끝에있던 '오렌지'를 제거
// ["사과", "바나나"]
Copy to Clipboard
```

배열 앞에서부터 항목 제거하기

```
let first = fruits.shift() // 제일 앞의 '사과'를 제거
// ["바나나"]
Copy to Clipboard
```

배열 앞에 항목 추가하기

```
let newLength = fruits.unshift('딸기') // 앞에 추가
// ["딸기", "바나나"]
Copy to Clipboard
```

배열 안 항목의 인덱스 찾기

```
fruits.push('망고')
// ["딸기", "바나나", "망고"]

let pos = fruits.indexOf("바나나")
```

```
// 1
Copy to Clipboard
```

인덱스 위치에 있는 항목 제거하기

```
let removedItem = fruits.splice(pos, 1) // 항목을 제거하는 방법

// ["딸기", "망고"]
Copy to Clipboard
```

인덱스 위치에서부터 여러개의 항목 제거하기

```
let vegetables = ['양배추', '순무', '무', '당근']
console.log(vegetables)
// ["양배추", "순무", "무", "당근"]

let pos = 1
let n = 2

let removedItems = vegetables.splice(pos, n)
// 배열에서 항목을 제거하는 방법
// pos 인덱스부터 n개의 항목을 제거함

console.log(vegetables)
// ["양배추", "당근"] (원 배열 vegetables의 값이 변함)

console.log(removedItems)
// ["순무", "무"]
Copy to Clipboard
```

배열 복사하기

```
let shallowCopySpread = [...fruits]
// ["딸기", "망고"]
```

배열의 Custom 메소드

JS에서 배열은 객체이기 때문에 원하는 기능을 메소드로 추가할 수 있습니다.

```
Array.prototype.reduce = function(f, value){
    var i;
    for(i=0 ; i<this.length ; i++){
        value = f(this[i], value);
```

```
    }
    return value
};
```

reduce함수는 f 함수에 값을 전달하여 배열의 차원을 줄이는 역할을 합니다. value는 누적된 값을 저장하는 초기값입니다. 따라서 항등원 값을 초기값을 사용합니다. f 함수가 곱연산을 하면 1, 합연산을 하면 0을 초기값으로 사용합니다.

```
function add(a, b){
    return a + b;
}function multipy(a, b){
    return a * b;
}list = [1,2,3,4,5];
list.reduce(add, 0); // 15
list.reduce(multiply, 1); 120
```

배열의 크기와 차원

JS에서는 배열 초기화 함수가 없기 때문에 직접구현해야합니다. 아래는 1차원 배열을 생성하고 초기화하는 함수입니다

```
Array.dim = function(dimension, initial){
    var a = [], i;
    for( i=0 ; i<dimension, i++ ){
        a[i] = initial;
    }
    return a;
}
list = Array.dim(5, 0:
list; // [0, 0, 0, 0, 0]
```

아래는 2차원 배열을 생성 및 초기화하는 함수입니다

```
Array.dim_2d = function(m, n, initial) {
    var outer=[], inner=[], i=0, j=0;
    for (i=0 ; i<n ; i++){
        inner[i]=initial;
    }    for (j=0 ; j<m ; j++){
        outer[j]=inner;
    }
    return outer;
}list = Array.dim(5, 3, 1);
list;// [ [1,1,1], [1,1,1], [1,1,1], [1,1,1], [1,1,1]]
```

#리스트란?(List)

- 목록, 순서가 있는 일련의 집합체(몇번째 항목)
- 리스트에 저장된 각 데이터 항목을 요소라 함, 프로그램의 가용 메모리가 리스트에 저장할 수 있는 최대 요소 수

#List ADT

- ADT란? Abstract Data Type을 말함, 리스트 자료구조가 어떤 구현부를 가져야하는가를 인터페이스 제시하는 것

1. listSize(프로퍼티): 리스트 요소 수
2. pos(프로퍼티) : 현재 위치
3. length(프로퍼티) : 리스트 요소수 반환
4. clear(함수): 모든 요소 삭제
5. toString(함수) : 리스트를 문자열로 표현해 반환
6. getElement(함수) : 현재 위치의 요소를 반환
7. insert(함수) : 기존 요소 위로 새 요소를 추가
8. append(함수) : 새 요소를 리스트 요소 끝에 추가
9. remove(함수) : 리스트의 요소 삭제
10. front(함수) : 현재 위치(탐색 위치)를 리스트 첫번째 요소로 설정
11. end(함수) : 현재 위치를 리스트 마지막 요소로 설정
12. prev(함수) : 현재 위치를 한 요소 뒤로 이동
13. next(함수) : 현재 위치를 한 요소 앞으로 이동
14. currPos(함수) : 리스트의 현재 위치 반환
15. moveTo(함수) : 현재 위치를 지정된 위치로 이동

#List 추상화

- 자바스크립트는 클래스가 없음, 함수가 객체 역할(인스턴스 변수, 메서드 구현)
- List 추상 객체 역할을 할 function 구현하기
- 자바스크립트 표준 내장객체 Array를 이용해서 List를 구현함(Array 래핑) : Array.prototype 객체 메서드 사용
- 배열로 만들지만 추상화된 List 객체를 인스턴스 생성해서 사용 : listName.메서드 혹은 프로퍼티

```
var List = function(){
    this.dataStore = [];
    this.pos = 0;
    this.listSize = 0;
}
```

1) **append** : 리스트 마지막 요소 다음 순서에 추가 : 크기 늘려주기

```
List.prototype.append = function(element){
     this.dataStore[this.listSize] = element;
     this.listSize++;
}
```

- 기본적으로 listSize를 구현해야함 : 배열의 length, size

2) **find** : 특정 값이 리스트에 포함되어있는지 있다면 position을 리턴해줌, 없으면 -1 리턴

```
List.prototype.find = function(element){
      for(var i=0; i<this.listSize; i++){
          if(this.dataStore[i] === element){
                return i;
          }
      }
      return -1;
}
```

3) **remove** : 특정 값을 가진 요소를 찾고, 요소를 삭제하고, 당기고 - 빈틈 없는 데이터 적재

```
List.prototype.remove = function(element){
    var removePos = this.find(element);
```

```
    if(removePos > -1){
      this.dataStore.splice(removePos, 1);
      this.listSize--;
      return true;
    }
    return false;
}
- 기본적으로 remove를 호출했을 때 false이고, 특정 상황에 일치할 때 true가 되도록 코드를 짜야함
```

- 기본적으로 remove를 호출했을 때 false이고, 특정 상황에 일치할 때 true가 되도록 코드를 짜야함
- 내부적으로는 array의 splice 메서드를 사용해서 삭제

4) **length** : 리스트에 저장된 요소의 개수

```
List.prototype.length = function(){
    return this.listSize;
}
```

5) **toString** : 리스트 요소 확인

```
List.prototype.toString = function(){
    return this.dataStore;
}
```

6) **insert** : 기존 리스트 뒤에 추가하기 - index 안씀

```
List.prototype.insert = function(element, after){
    var insertPos = this.find(after);

    if(insertPos > -1){
      this.dataStore.splice(insertPos+1, 0, element);
      this.listSize++;
      return true;
    }

    return false;
}
- 앞서 구현한 find를 헬퍼로 사용
```

- after : 기본 요소 값
- 앞서 구현한 find를 헬퍼로 사용

7) **clear** : 리스트 모든 요소 제거

```
List.prototype.clear = function(){
    this.dataStore = [];
    this.listSize = 0;
    this.pos = 0;
}
```

8) **contains** : 특정 값이 요소로 존재하는지 판단

```
List.prototype.contains = function(element){
    for(var i=0; i
```

find와 같음

9) 리스트 탐색과 관련된 메서드

```
/* 탐색 위치를 맨 앞의 요소로 */
List.prototype.front = function(){
    this.pos = 0;
}

/* 탐색 위치를 맨 뒤의 요소로 */
List.prototype.end = function(){
    this.pos = this.listSize-1;
}

/* 현재 탐색 위치보다 이전으로 이동 */
List.prototype.prev = function(){
    if(this.pos > 0){
        this.pos--;
    }
}

/* 현재 탐색 위치보다 이후로 이동 */
List.prototype.next = function(){
    if(this.pos < this.listSize-1){
        this.pos++;
    }
```

```
}

/* 현재 탐색 position 리턴 */
List.prototype.currPos = function(){
    return this.pos;
}

/* 탐색 위치 이동 */
List.prototype.moveTo = function(position){
    if(position < this.listSize){
        this.pos = position;
    }
}

/* 현재 탐색 위치에 있는 요소 리턴받기 */
List.prototype.getElement = function(){
    return this.dataStore[this.pos];
}
- 리스트 요소를 탐색하면서 요소를 리턴받음
```

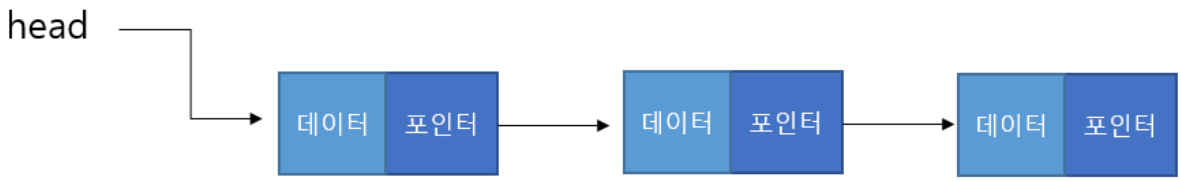
- 리스트 요소를 탐색하면서 요소를 리턴받음
- index로 요소를 찾는 것이 아니라 this.pos(탐색 위치)로 탐색위치를 옮겨다니면서 요소를 찾아서 뽑음
- => this.pos가 중요
-

[리스트 순차적으로 출력하기]

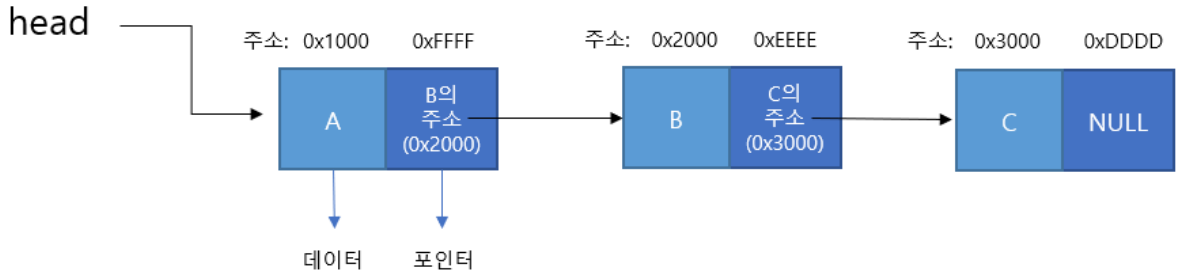
```
for(리스트명.front(); 리스트명.currPos() < 리스트명.length(); 리스트명.next()){
     console.log(리스트명.getElement());
}

for(리스트명.end(); 리스트명.currPos() >= 리스트명.front(); 리스트명.prev(){
     console.log(리스트명.getElement());
}
```

#연결리스트(Linked List)란?

- 연결리스트(Linked List)는 각 노드가 데이터와 포인터를 가지고 한 줄로 연결되어 있는 방식으로 데이터를 저장하는 자료구조이다.
- 각 노드는 다음 노드를 가리키는 포인터를 포함한다.



다음 노드를 가리키는 포인터는 **다음 노드의 주소**를 값으로 가지고 있다.



각 노드의 포인터 변수는 다음 노드의 데이터의 주소를 값으로 가진다. 또한 각 포인터 변수의 주소도 따로 존재한다.

(*그림에 나타난 주소는 정확하지 않다. 이해를 돕기 위해 임의로 설정했다. 실제 주소값은 저런식으로 저장되지 않는다*)

	장점	단점
배열	- 랜덤 엑세스가 빠르다 - 즉, 매우 빠르게 접근가능	- 메모리 사용 비효율적 - 배열 내의 데이터 이동 및 재구성이 어렵다
연결리스트	- 동적으로 메모리 사용가능 - 메모리 효율적 사용 - 데이터 재구성 용이 - 대용량 데이터 처리 적합	- 특정 위치 데이터 검색할때 느리다 - 메모리를 추가적으로 사용해야한다

#단순연결리스트(Singly Linked List)의 구현

가장 간단한 연결리스트인 단순 연결리스트를 구현해보자. 여러 구현 방법이 있지만, 아래 내용들을 응용하면 다양한 연결리스트를 구현할 수 있을 것이다.

노드의 구성

```
typedef struct _Node{
    int data;/* 저장할 데이터 */struct _Node* next;/* 다음 노드를 가리킬 포인터*/
}Node;
```

각 노드는 저장할 데이터와 다음 노드를 가리킬 포인터로 이루어진다.

연결리스트의 초기화(init)

가장 첫번째 노드를 가리킬 포인터 Node* head 를 전역변수로 선언하고 init() 함수를 통해 초기화한다.

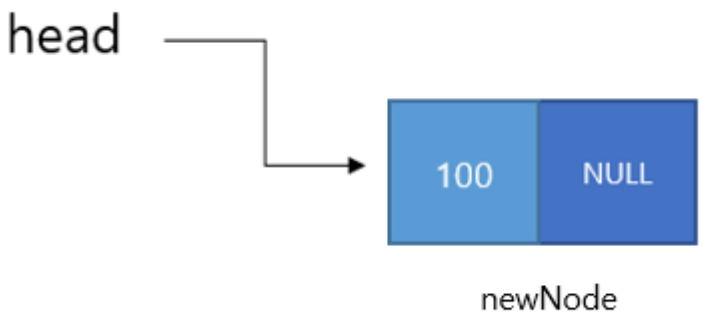
```
Node* head;

void init(){
    head = NULL;
}
```

#연결리스트의 삽입(insert)

연결리스트에 노드를 삽입하는 방법을 구현해보자.

1. 가장 앞에 노드를 삽입하는 경우

1) 연결리스트가 비어 있는 경우

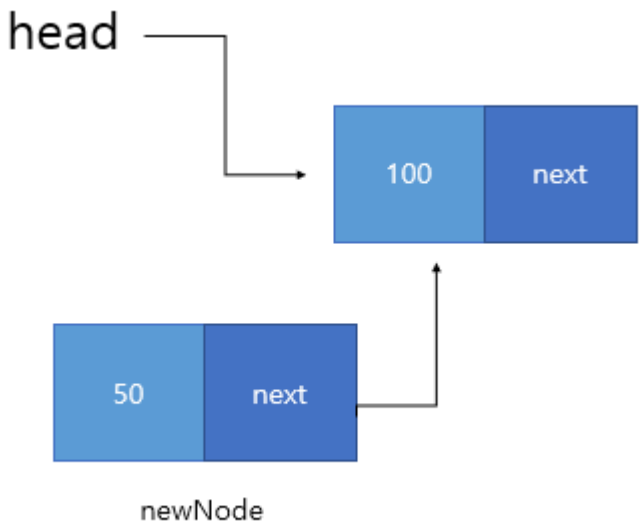


첫번째 노드 추가 - empty

첫번째 노드로 정수 100을 데이터로 갖는 노드를 추가한다고 하자. 이 경우 간단하게 newNode를 만든 후 head가 newNode를 가리키도록 하면 된다.

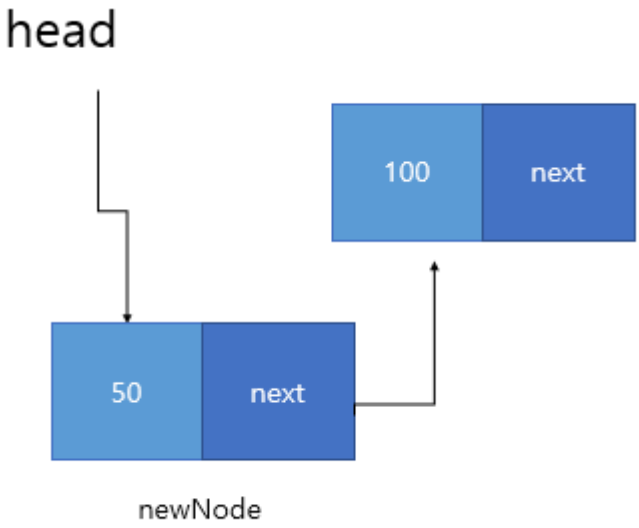
연결리스트가 비어있는 것은 어떻게 확인할 수 있을까? **head == NULL 이라면 연결리스트가 비어있는 것이다.**

2) 연결리스트가 비어 있지 않은 경우



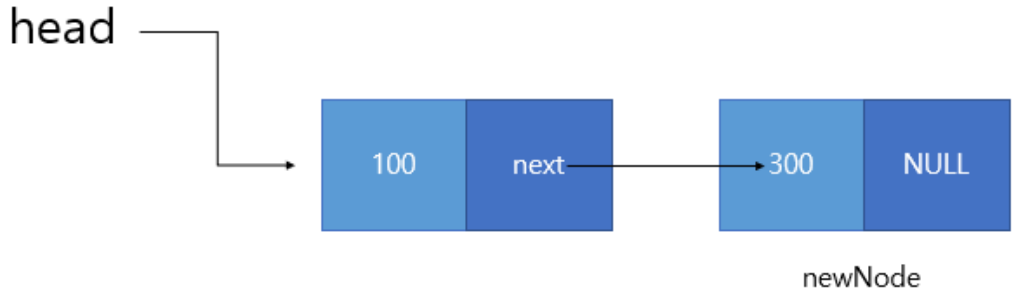
첫번째 노드 추가 - not empty

newNode를 생성 한 후 head가 가리키는 노드를 newNode의 next 포인터가 가리키게 한다.



이후 head 가 newNode를 가리키게 하면 끝이다.

2. 가장 뒤에 노드를 삽입하는 경우

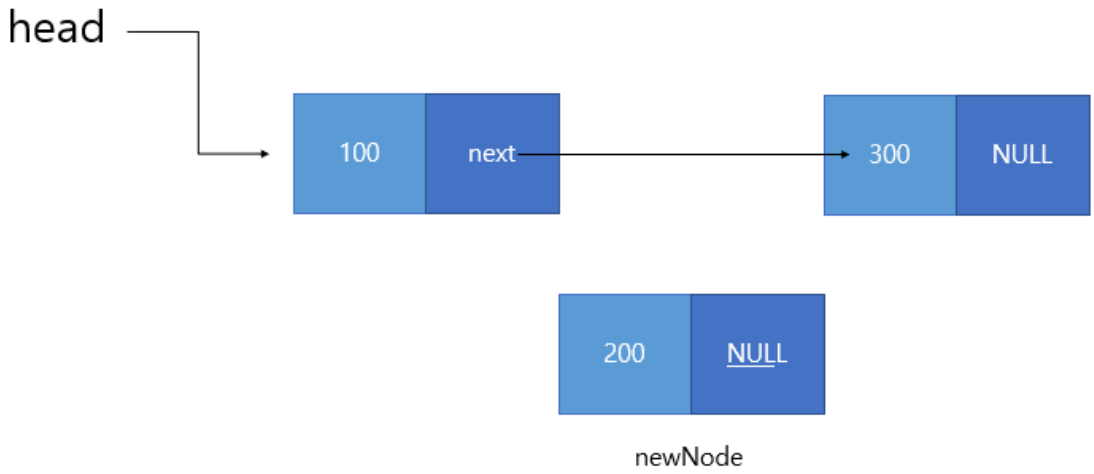


마지막 노드 추가

마지막 노드로 정수 300을 데이터로 갖는 노드를 추가한다고 하자. 이 경우 newNode를 만든 후 마지막 노드의 next 포인터가 newNode를 가리키게 하면 될 것이다.

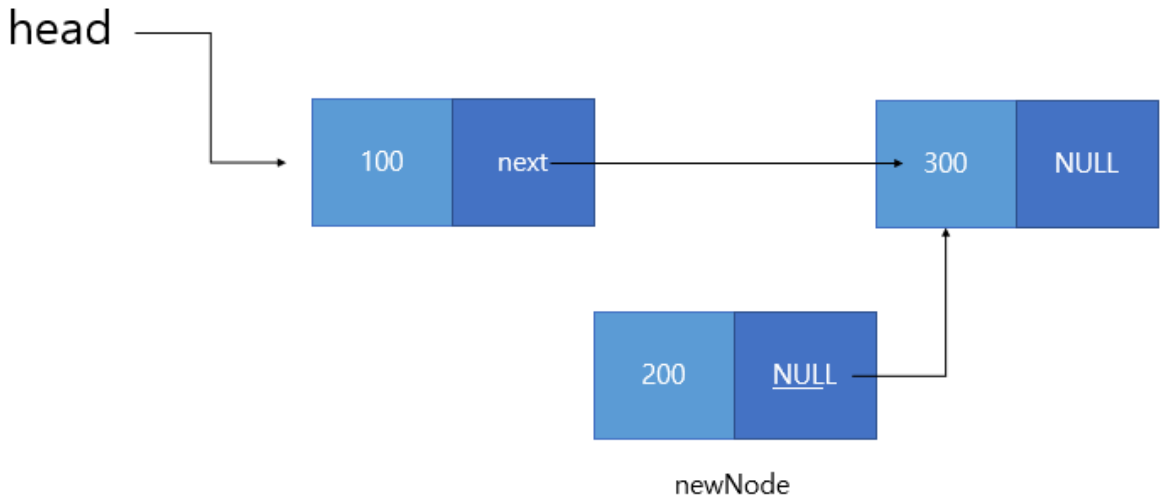
3. 중간에 노드를 삽입하는 경우

연결리스트 중간에 정수 200을 데이터로 갖는 노드를 추가한다고 하자. 이 경우 아래와 같은 과정을 거치면 된다.



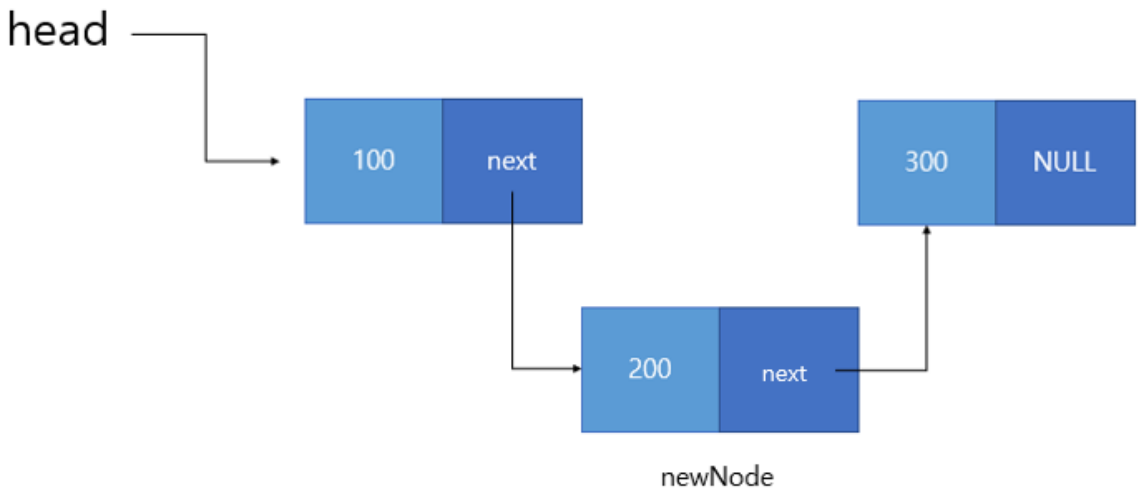
1) newNode 생성

1) 가장 먼저 newNode를 생성한다.



2) newNode의 next 포인터 설정

2) newNode의 next 포인터가 이전 노드의 next가 가리키는 노드를 가리키도록 한다.



3) 이전 노드의 next 포인터 설정

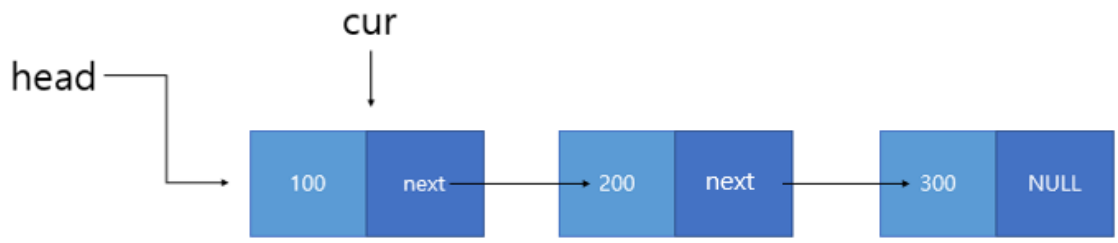
3) 마지막으로 이전 노드의 next 포인터가 newNode를 가리키도록 하면 된다.

#연결리스트의 삭제(delete)

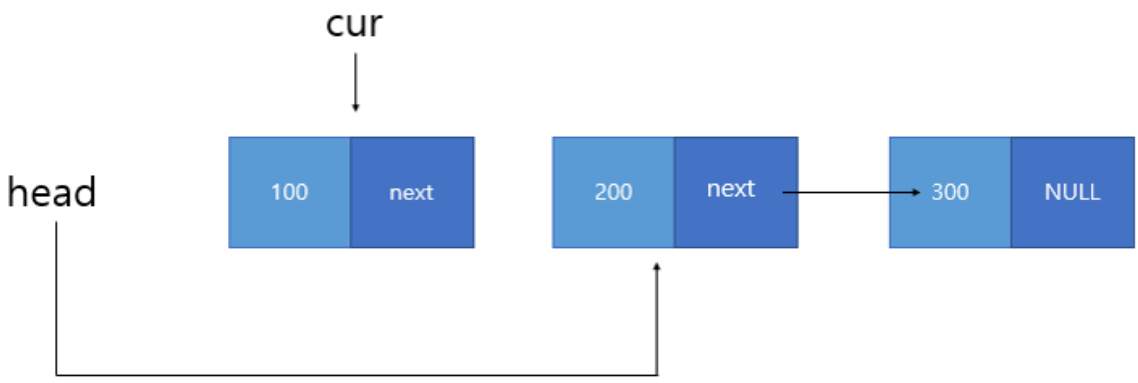
사용자가 data를 입력하면 해당 data를 갖는 노드를 연결리스트에서 삭제한다고 하자.

1. 가장 앞의 노드를 삭제하는 경우

예를 들어, 100 을 데이터로 갖는 노드를 삭제한다면, 다음과 같은 과정을 거치게 된다.

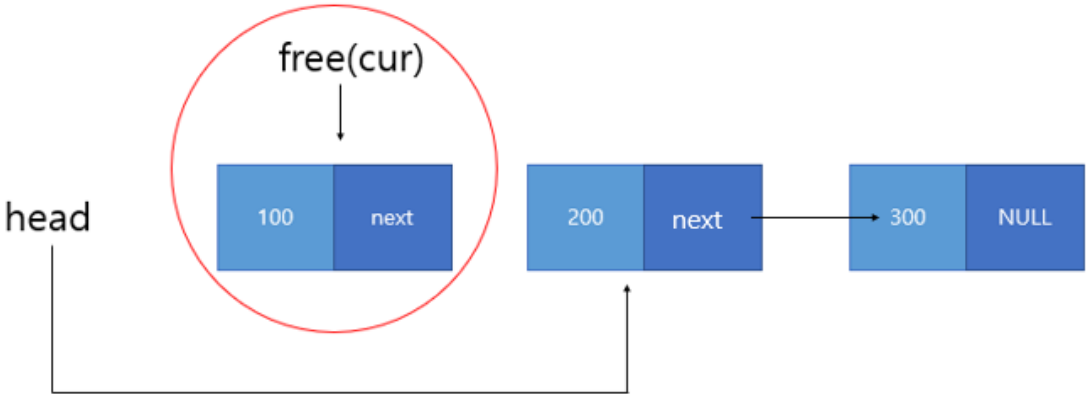


가장 앞의 노드 삭제



가장 앞의 노드 삭제

head 가 cur->next를 가리키게 하고, cur->next를 NULL 로 설정한다.

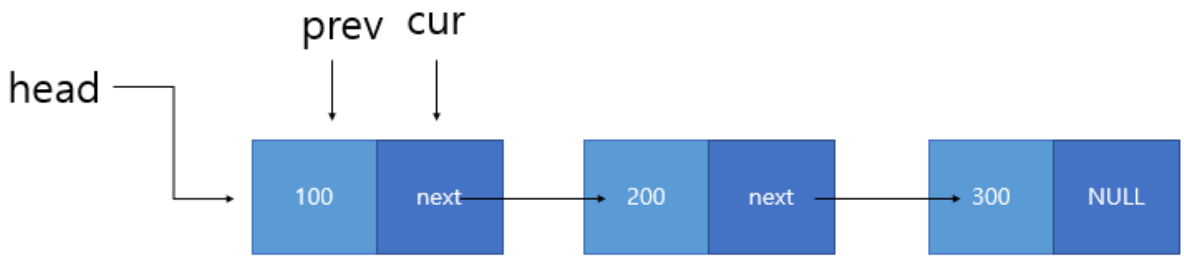


가장 앞의 노드 삭제

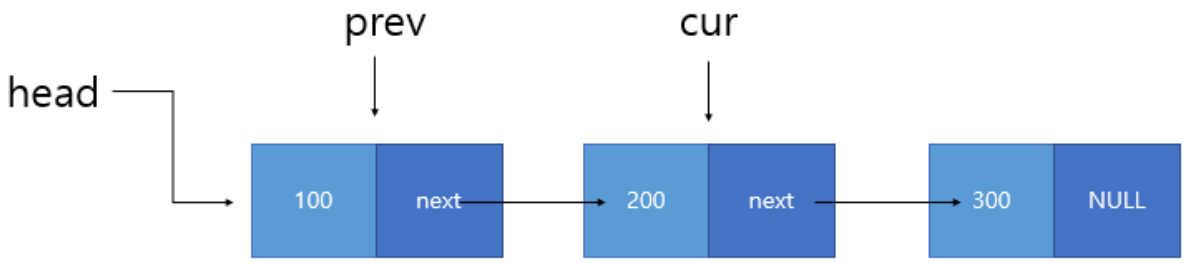
이후 cur 을 free 시키면 완료다.

2. 가장 뒤의 노드를 삭제하는 경우 & 중간 노드를 삭제하는 경우

이 경우에는, prev 라는 포인터를 이용해 삭제할 노드의 이전 노드를 가리켜야 한다.

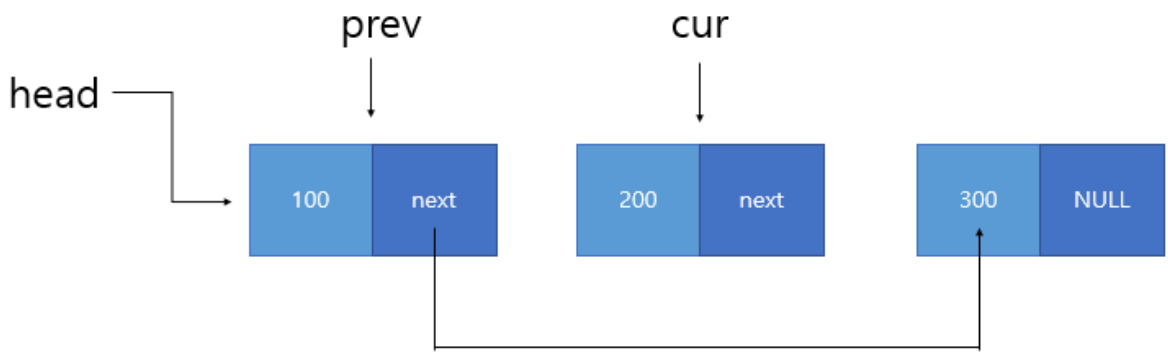


초기에 prev 포인터와 cur 포인터는 모두 head가 가리키는 첫번째 노드를 가리킨다.



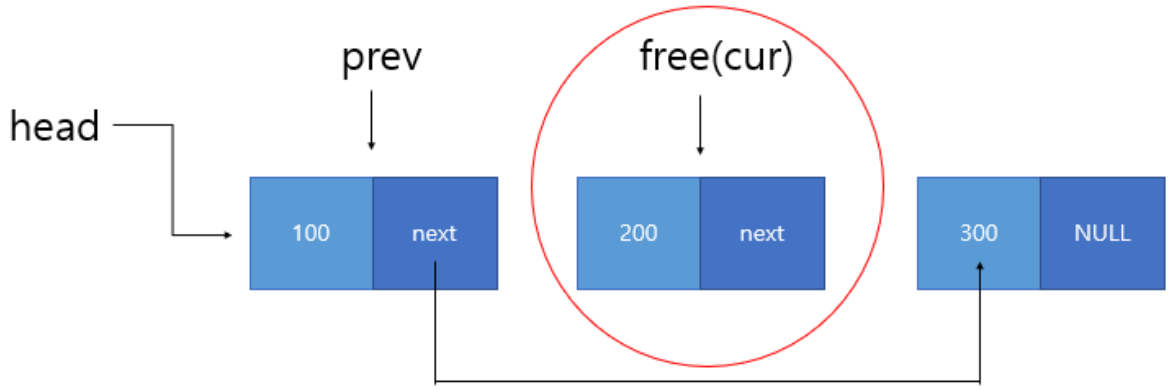
중간 노드 삭제

이후 cur 포인터가 다음 노드를 가리키고, 이때 사용자가 입력한 데이터 200과 cur->data 가 일치하므로 삭제 과정을 진행한다.



중간 노드 삭제

prev->next 가 cur->next를 가리키게 하고, 이후 cur->next 는 NULL을 가리키게 하면 된다.



중간 노드 삭제

이후 cur 을 free 해주면 삭제가 완료된다.