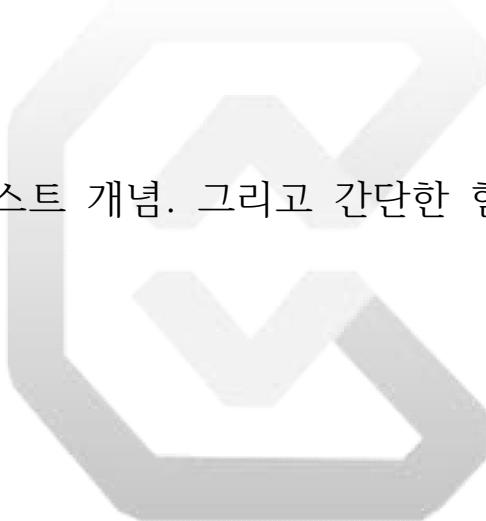


# C언어 스터디

## 5.5주차

<연결리스트 개념. 그리고 간단한 함수만.>



# CAPS

# 연결리스트

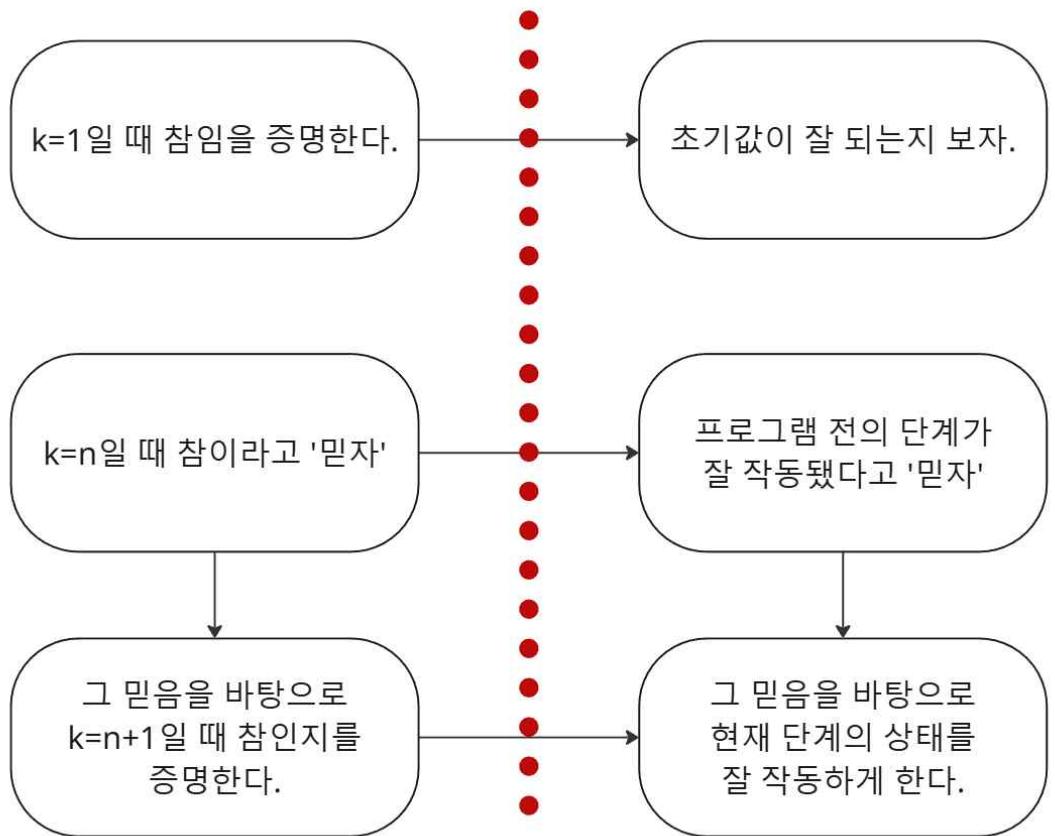
실상 ‘연결리스트’라는 자료구조를 써서 백준에 쓰는 경우는 거의 없습니다. 저희 학교에서는 ‘자료구조’ 교과목의 60%가 연결리스트의 내용입니다. 그래서 자료구조 수업에서 연결리스트를 잘 배우시길 바랍니다.

연결리스트를 잘 하시게 된다면, 향후 다룰 재귀함수 등에 큰 도움이 될 겁니다. 아니? 사실 재귀함수를 잘하시면 연결리스트를 잘하게 될 겁니다. 아니? 사실 이 모든 걸 아우르는 하나의 능력이 있습니다.

연결리스트에서의 중요한 건 ‘귀납적인 생각’입니다. dp, 재귀 모두 ‘귀납적인 생각’이 중요하거든요. 사실 알고리즘을 배우실 때 가장 처음 만나고 가장 어려운 벽이 귀납적인 생각일 겁니다.

## 귀납법

## 귀납적생각

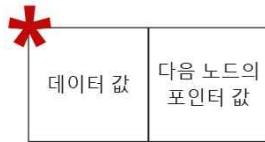


miro

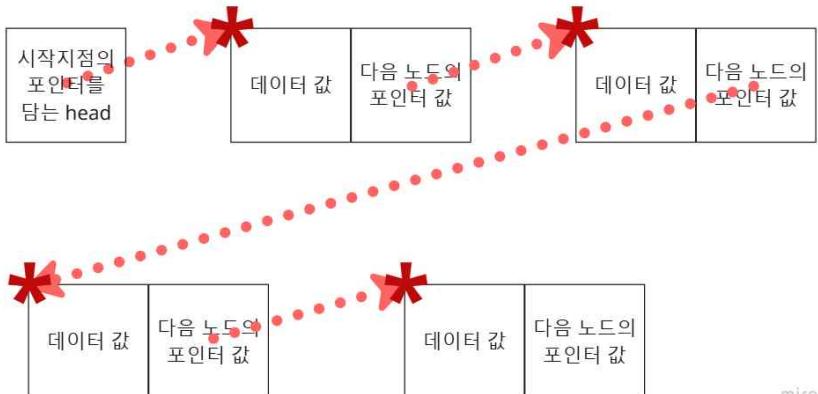
사실 이번주차 내용에 이걸 제대로 다룰 생각은 아닙니다. 하지만, 유념해 줍시다.

# 연결 리스트

**노드 :**



**연결리스트 :**



## 1. 노드

```
typedef struct Node {
    int data;           // 데이터 값
    struct Node * point; // 다음 노드의 포인터 값!
}Node;

typedef struct Node {
    int data;
    Node * point;      //오류!
}Node;
```

## 2. 문제 하나 내 볼게요

```
#include<iostream>
using namespace std;
typedef struct Node {
    int data;
    struct Node* point;
}Node;
int main(void) {
    Node tmp;
    cout<<sizeof(tmp); // 어떤 값을 출력할까요? 참고로 int의 크기는 8입니다.
}
```

```
#include<iostream>
using namespace std;
typedef struct Node {
    int data;           // int의 사이즈 : 8
    struct Node* point; // 포인터를 담는 건 결국 8.
}Node;
int main(void) {
    Node tmp;
    cout << sizeof(tmp); // 16을 출력합니다.
}
```

가장 실수하기 쉬운 데입니다.

struct Node의 크기가 x라고 합시다.

그럼 data는 8이고,, point는 x로 보이는데,,,

그럼  $x=8+x??$  으로 오해하기 딱 좋습니다.

\*는 일종의 ‘자료형’이라고 했죠?

struct Node\* point의 주인공은 struct Node가 아니라 \*입니다.

```
#include<iostream>
using namespace std;
typedef struct Node {
    int data;
    struct Node* point;
}Node;
typedef struct Big {
    int data[1000000];
}Big;
int main(void) {
    Node tmp;
    cout << sizeof(tmp) << "\n"; // 16을 출력합니다.
    cout << sizeof(int*) << "\n"; // 8
    cout << sizeof(double*) << "\n"; // 8
    cout << sizeof(Node*) << "\n"; // 8

    cout << sizeof(Big) << "\n"; // 4000000
    cout << sizeof(Big*) << "\n"; // 8
}
```

# \*는 자료형입니다. 앞의 있는 애는 들러리입니다.

# 연결리스트 구현

## 1. init

```
#include<iostream>
#include<stdlib.h>
using namespace std;
typedef struct Node {
    int data;
    struct Node* point;
}Node;
Node* init(void);
int main(void) {
    Node* head; // 1. 그냥 포인터만 담는 깡통 head를 만들어 줍시다.
    head=init(); // 2. 이 친구는 뭐하는 걸까요? 해당 함수로 갑시다.
    cout<<head->data; // 2.1참고로 a->b 는 (*a).b 와 같습니다. 해당 포인터로
뛰어 들어서 구조체의 원소를 봅니다.
}

Node* init(void) {
    Node* tmp = (Node*)malloc(sizeof(Node)); // 3. malloc을 하는 이유는 데
이터를 main문 밖에서 만들기 때문입니다.
    tmp->data = 10; // 4. 데이터를 대충 넣어주고
    tmp->point = NULL;
    return tmp; // 5. 포인터값만 haed에 넣어줍시다.
}
```

어렵죠? 괜찮습니다. init과 head의 오개념만 잡으면 그 이후는 쉬워요.

1. init함수를 굳이 만드는 이유는 또 다른 연결 리스트 관련 함수에 쓸라고 그립니다. 그럼 init 함수도 main 밖의 원소이기 때문에 malloc을 잘 써 줍시다. (free는 나중에)
2. 참고로 head는 Node가 아닙니다. 단순히 연결 리스트의 시작점을 따로 저장하고 있는 변수입니다.

## 2. push

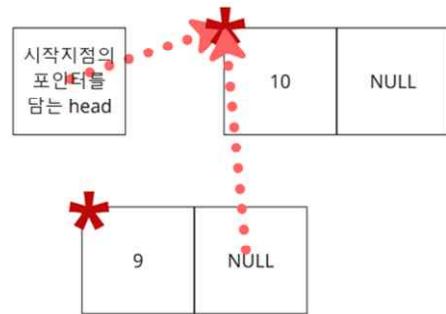
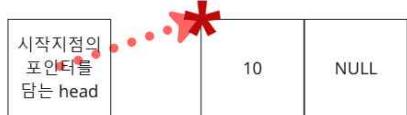
더 심화적으로 가면 push의 형태가 다양하지만, stack의 형태로 만들어 볼게요  
연결리스트의 모든 함수들 push는 pop이든,, 모든 함수들은 다음과 같은 과정이 있습니다.

1. 새로운 데이터를 만들고
2. 새로운 데이터를 수정하고
3. 기존의 데이터를 수정한다.

모든 함수들이 이렇게 돌아갑니다. 유념하세요!

### 2. 새로운 데이터를 수정하고

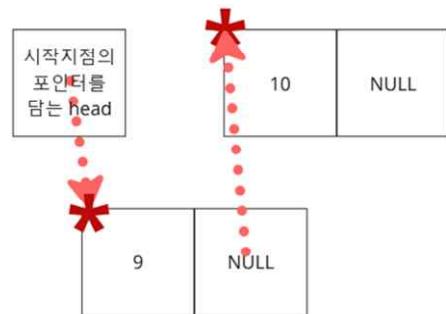
**init :**



**1. 새로운 데이터를 만들고:**



**3. 기존의 데이터를 수정한다.**



```
#include<iostream>
#include<stdlib.h>
using namespace std;
typedef struct Node {
    int data;
    struct Node* point;
}Node;
Node* init(void);
Node* push(Node* head, int x);
int main(void) {
    Node* head;
    head = init();
    cout << head->data << "\n";

    head = push(head, 9); // 3. 기존의 데이터를 수정한다.
    cout << head->data << "\n";
}

Node* push(Node* head, int x) {
    Node* tmp = (Node*)malloc(sizeof(Node)); // 1. 새로운 데이터를 만들고
    tmp->data = x;

    tmp->point = head; // 2. 새로운 데이터를 수정하고

    return tmp; // 3. 기존의 데이터를 수정한다.
}

Node* init(void) {
    Node* tmp = (Node*)malloc(sizeof(Node));
    tmp->data = 10;
    tmp->point = NULL;
    return tmp;
}
```

### 3. pop

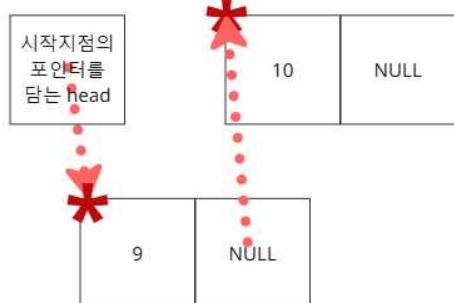
사실 pop이 가장 중요할 수 있습니다.

저희가 malloc을 하면 free를 해 주어야 한다고 했죠?

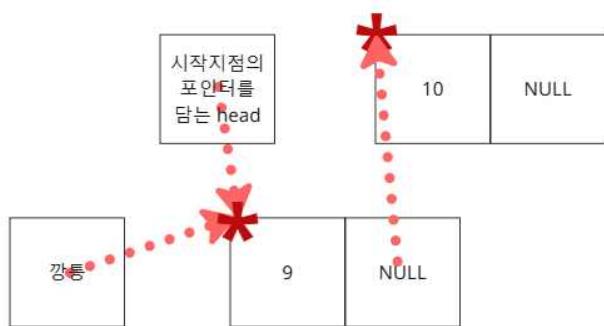
free해 줄 주솟값을 담을 깡통 변수를 만들어 줍시다.

그리고 head가 가르키는 곳을 data10을 담고 있는 노드로 향하게 한 뒤,  
깡통 변수에 담겨있는 주소지를 free해주면 마침내 해당 노드는 사라집니다.

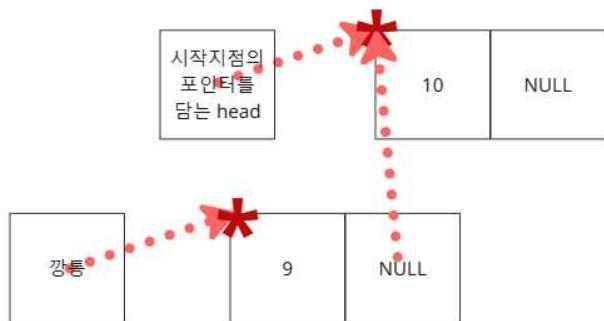
**init :**



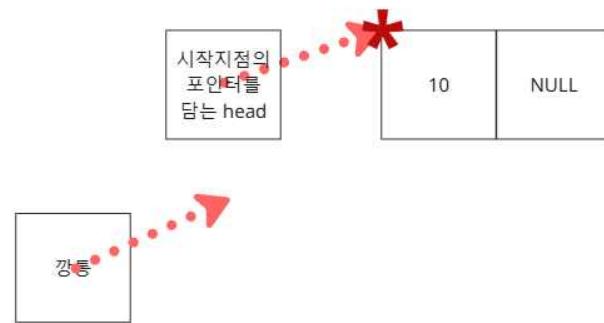
**1. free를 하기 위한 깡통을 세우고**



**2. head의 방향을 틀고**



**3. 깡통보고, 네가 보고 있는 놈을 죽이라 한다.**



miro

```

#include<iostream>
#include<stdlib.h>
using namespace std;
typedef struct Node {
    int data;
    struct Node* point;
}Node;
Node* init(void);
Node* push(Node* head, int x);
Node* pop(Node* );
int main(void) {
    Node* head;
    head = init();
    cout << head->data << "\n"; // 10

    head = push(head, 9);
    cout << head->data << "\n"; // 9

    head = pop(head);
    cout << head->data << "\n"; // 10
}

Node* pop(Node* head) {
    Node* tmp = head; // 새로운 깡통 포인터를 만들고 수정한 뒤
    head = head->point; // 기존의 데이터를 수정한다.
    free(tmp); // 이때 tmp를 free하면, 버리고 싶은 데이터가 드디어 사
라집니다.
    return head; // 다음 노드의 주소지를 반환합니다.
}

Node* push(Node* head, int x) {
    Node* tmp = (Node*)malloc(sizeof(Node));
    tmp->data = x;
    tmp->point = head;
    return tmp;
}

Node* init(void) {
    Node* tmp = (Node*)malloc(sizeof(Node));
    tmp->data = 10;
    tmp->point = NULL;
    return tmp;
}

```

# 배열로 암매 연결리스트 구현

```
#include<iostream>
using namespace std;
void push(int x);
void pop(void);
int arr[10000][2]; // value, point
int ind = -1;
int main(void) {
    push(1);
    push(10);
    push(5);
    cout << arr[ind][0] << "\n";
    pop();
    cout << arr[ind][0] << "\n";
    pop();
    cout << arr[ind][0] << "\n";
}
void push(int x) {
    arr[ind + 1][0] = x;
    arr[ind + 1][1] = ind;
    ind++;
}
void pop(void) {
    ind = arr[ind][1];
}
```

뭐 전에 했던 스택 구현과 차이가 없겠다 싶겠지만,  
연결 리스트의 구조가 아주 복잡해지면 세세하게 많이 바뀔 겁니다.  
이걸 굳이 하는 이유는 구현이 쉽기 때문입니다.  
정석은 위의 구조체와 포인터를 쓰는 것이지만, 배열을 통한 암매 구현은 향후 코딩  
테스트나 백준의 환경에서 손쉽게 구현할 수 있습니다.