C – 자료 구조 1

Data Structure

자료 구조

- **자료 구조**

  자료구조는 프로그래밍에서 데이터를 효율적으로 저장하고, 접근하고, 관리하기 위한 구조를 의미합니다.

자료 구조	특징	C 언어 구현 방법
스택(Stack)	LIFO(후입선출)	배열 또는 구조체
큐(Queue)	FIFO(선입선출)	배열, 원형 큐, 구조체
연결 리스트	동적 크기, 삽입/삭제 용이	단순, 이중, 원형 연결 리스트
트리	계층 구조(비선형 구조)	구조체, 연결리스트
해시	해시 함수로 변환	배열 및 연결리스트

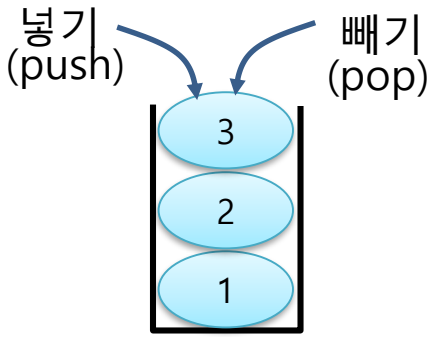
스택(Stack) 자료 구조

- **스택(Stack)**
  - 후입선출(LIFO : Last in First Out) 구조

    배열에서 나중에 들어간 자료를 먼저 꺼냄

    (응용 예: 스택 메모리, 문자열 역순 처리, 게임 무르기)

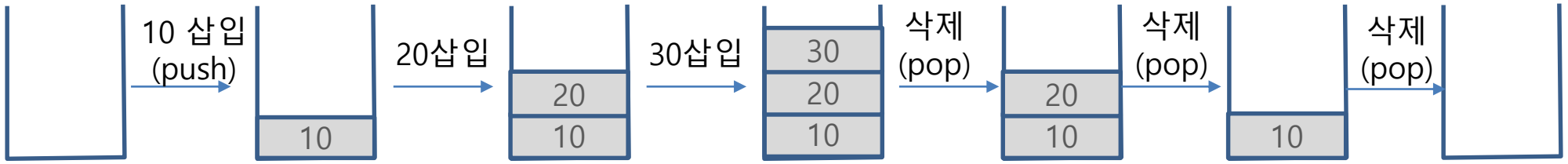
메소드명	설명
push()	원소(자료)를 스택에 넣는다.
pop()	스택의 맨 위 원소를 가져온다.

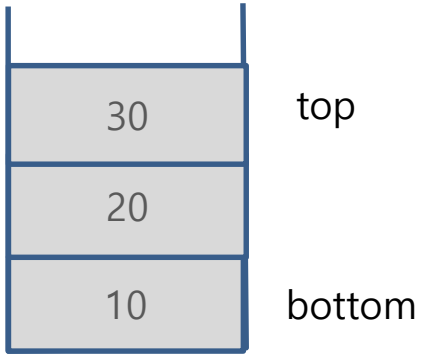


스택(Stack)

스택(Stack)

- **스택(Stack)**





스택(Stack)

- **스택(Stack) – 정수형**

```c
#define MAX_LEN 4

int stack[MAX_LEN];   // 스택 배열
int top = -1;   // 스택의 top을 나타냄

// 요소 추가 함수 (push)
void push(int value) {
    if (top >= MAX_LEN - 1)
        printf("스택이 가득 찼습니다. PUSH 실패: %d\n", value);
    else {
        stack[++top] = value; //선증가
        printf("PUSH: %d\n", value);
    }
}
```

스택(Stack)

- 스택(Stack) - 정수형

```
int pop() {  // 요소 제거 함수 (pop)
    if (top < 0) {
        printf("스택이 비어 있습니다. POP 실패\n");
        return -1;
    }
    else
        return stack[top--];
}

int main() {
    puts("=== 스택에 자료 삽입 ===");
    // 값 추가 (push) : 80 - 70 - 95 - 85
    push(80);
    push(70);
    push(95);
    push(85);
    //push(100); // 초과 입력 시 실패
```

스택(Stack)

- **스택(Stack) - 정수형**

```
puts("\n=== 스택에서 자료 삭제 ===");
// 값 제거 (pop) : 85 - 95 - 70 - 80
/*printf("%d\n", stack[top]); //현재 맨 위 요소
pop();
printf("%d\n", stack[top]);
pop();
printf("%d\n", stack[top]);
pop();
printf("%d\n", stack[top]);
pop();
printf("%d\n", stack[top]);
pop(); //빈 상태에서 pop 시도 */

while (top != -1) { //안전한 삭제
    printf("%d ", stack[top]);
    pop();
}
```

```
=== 스택에 자료 삽입 ===
PUSH: 80
PUSH: 70
PUSH: 95
PUSH: 85

=== 스택에서 자료 삭제 ===
85 95 70 80
```

스택(Stack)

- 스택(Stack) – 문자형

```
#define MAX_LEN 3

char stack[MAX_LEN];
int top = -1;

void push(char c) { //요소 삽입(저장)
    if (top >= MAX_LEN - 1) {
        printf("스택이 가득 찼습니다.\n");
        return;  //return '\0' (널문자, 비어있음)
    }
    stack[++top] = c;
    printf("%c ", stack[top]);
}
```

스택(Stack)

- **스택(Stack) – 문자형**

```
char pop() { //요소 삭제(빼기)
    if (top < 0) {
        printf("스택이 비었습니다!!\n");
        return '\0';
    }
    return stack[top--];
}

int main()
{
    //a - b - c
    printf("스택에서 자료 저장\n");
    push('a');
    push('b');
    push('c');
    push("d"); //초과했을 때 처리 확인
```

스택(Stack)

- **스택(Stack) – 문자형**

```
//c - b - a
printf("스택에서 자료 삭제\n");
/*printf("%c\n", pop());
printf("%c\n", pop());
printf("%c\n", pop());
printf("%c\n", pop()); //비었을 때 처리 확인*/

while (top != -1) { //안전한 삭제
    printf("%c ", stack[top]);
    pop();
}
```

```
스택에서 자료 저장
a b c 스택이 가득 찼습니다.
스택에서 자료 삭제
c b a
```

단어 순서 뒤집기

- **스택 구조체 활용**

```c
#define MAX_LEN 4

// 스택 구조체 정의
typedef struct {
    int data[MAX_LEN]; // 스택 요소 저장
    int top;           // top 인덱스
} Stack;

// 스택 초기화 함수
void initStack(Stack* s) {
    s->top = -1;
}
```

단어 순서 뒤집기

- 스택 구조체 활용

```
void push(Stack* s, int value) {
    if (s->top < MAX_LEN - 1) {
        s->data[++(s->top)] = value;
        printf("PUSH: %d\n", value);
    }
    else {
        printf("스택이 가득 찼습니다. PUSH 실패: %d\n", value);
    }
}

int pop(Stack* s) {
    if (s->top > -1) {
        return s->data[(s->top)--];
    }
    else {
        printf("스택이 비어 있습니다. POP 실패\n");
        return -1;
    }
}
```

단어 순서 뒤집기

- 스택 구조체 활용

```
void printStack(Stack* s) {
    printf("\n현재 스택 상태:\n");
    if (s->top == -1) {
        printf("(비어 있음)\n");
    }
    else {
        printf("남은 요소 수: %d\n", s->top + 1);
        for (int i = 0; i <= s->top; i++) {
            printf("%d\n", s->data[i]);
        }
    }
}

int main() {
    Stack stack;   // 스택 변수 생성
    initStack(&stack); // 초기화
```

단어 순서 뒤집기

- **스택 구조체 활용**

```
puts("=== 스택에 자료 삽입 ===");
// 값 추가 (push) : 80 - 70 - 95 - 85
push(&stack, 80);
push(&stack, 70);
push(&stack, 95);
push(&stack, 85);
//push(&stack, 100); // 초과 입력 시 실패

puts("\n=== 스택에서 자료 삭제 ===");
// 값 제거 (pop) : 85 - 95 - 70 - 80
/*pop(&stack);
pop(&stack);
pop(&stack);*/
//pop(&stack);
//pop(&stack); // 빈 상태에서 pop 시도

while (stack.top != -1) {
    printf("%d\n", pop(&stack));
}
printStack(&stack); // 최종 상태 출력
```

```
=== 스택에 자료 삽입 ===
PUSH: 80
PUSH: 70
PUSH: 95
PUSH: 85

=== 스택에서 자료 삭제 ===
85
95
70
80

현재 스택 상태:
(비어 있음)
```

단어 순서 뒤집기

- **문자열 뒤집기**

```c
#define MAX_LEN 128  // 문자열 최대 길이

// 스택 구조체 정의
typedef struct {
    char data[MAX_LEN];
    int top;
} Stack;

// 스택 초기화
void initStack(Stack* s) {
    s->top = -1;
}
```

단어 순서 뒤집기

- **문자열 뒤집기**

```
void push(Stack* s, char ch) {
    if (s->top >= MAX_LEN - 1) {
        printf("스택이 가득 찼습니다.\n");
        return;
    }
    else
        s->data[++(s->top)] = ch; //문자 저장
}

int pop(Stack* s) {
    if (s->top < 0) {
        printf("스택이 비어 있습니다.\n");
        return -1;
    }
    else
        return s->data[(s->top)--]; //문자 반환
}
```

단어 순서 뒤집기

- **문자열 뒤집기**

```
Stack stack;  //스택 구조체 변수 생성
char str[MAX_LEN]; //문자열 배열

initStack(&stack); //초기화 함수 호출

//사용자 입력
printf("문자열 입력: ");
fgets(str, MAX_LEN, stdin);  //공백 포함 문자 입력

// 문자열을 한 글자씩 push
for (int i = 0; str[i] != '\0'; i++) {
    push(&stack, str[i]);
}

// pop 하면서 뒤집어 출력
printf("뒤집은 문자열: ");
while (stack.top != -1) {
    printf("%c", pop(&stack));
}
```

```
문자열 입력: hello world
뒤집은 문자열:
dlrow olleh
```

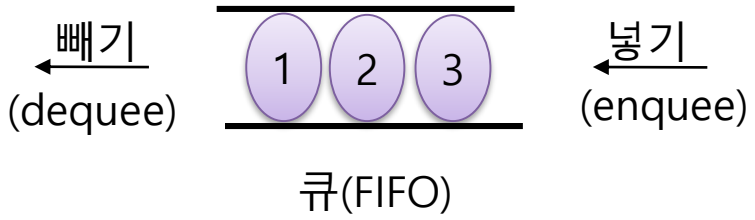
큐(Queue) 자료 구조

- **큐(Queue)**
  - 선입선출(FIFO : First in First Out) 구조

    배열에서 먼저 들어간 자료를 먼저 꺼냄

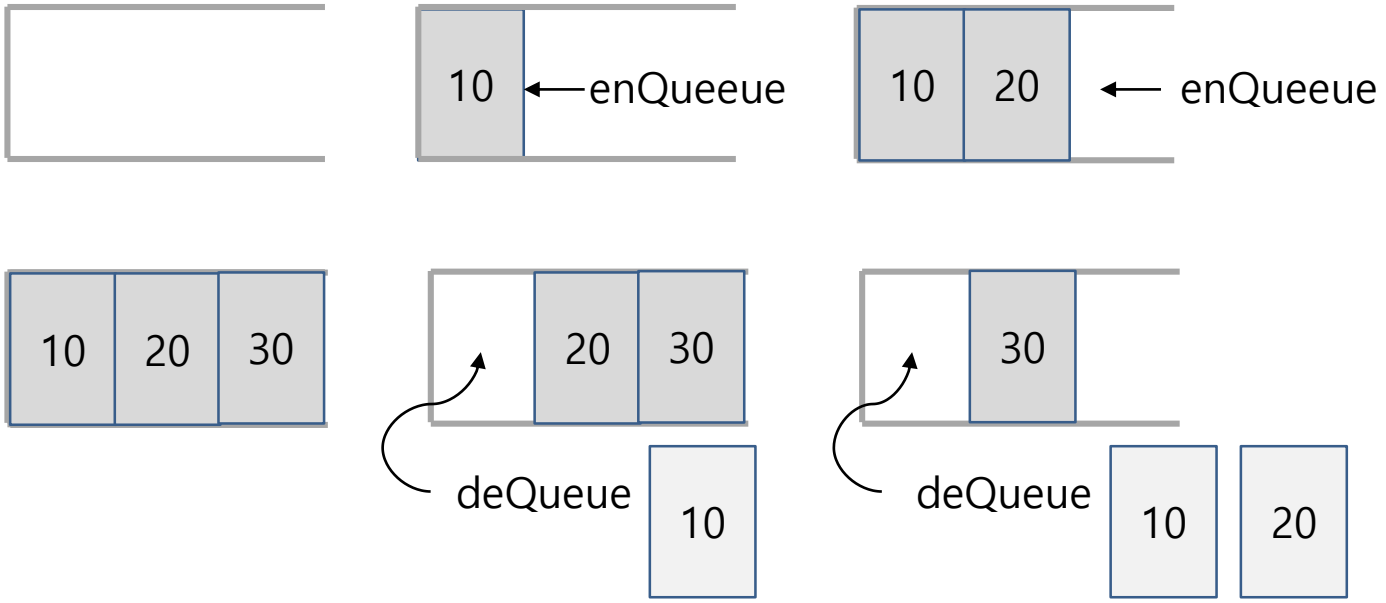
    (응용 예: 프린터 대기열, CPU 스케줄링)

메소드명	설명
enQueue()	주어진 객체를 큐에 넣는다.
dequeue()	객체 하나를 가져온다. 객체를 큐에서 제거한다.



큐(Queue) 자료 구조

- 큐(Queue)



큐(Queue) 자료 구조

- 원형 큐 상태 변화

1. 초기 상태(비어 있음)
   [ ][ ][ ][ ] front=0 rear=0
    ^front
    ^rear

2. enQueue("A")
   [A][ ][ ][ ] front=0 rear=1
    ^front
      ^rear

3. enQueue("B")
   [A][B][ ][ ] front=0 rear=2
    ^front
        ^rear

MAX_QUEUE = 4

front: 데이터 꺼낼 위치

rear: 데이터 넣을 위치

비어있음: front == rear

가득 참: (rear + 1) % MAX_QUEUE == front

rear는 (rear+1) % MAX_QUEUE,

front는 (front+1) % MAX_QUEUE로 이동하면서

배열을 원처럼 사용하는 것이 원형 큐의 핵심이다.

front는 배열 끝에 도달하면 다시 0으로 돌아오게

연산함

큐(Queue) 자료 구조

- 원형 큐 상태 변화

4. enQueue("C")
   [A][B][C][ ] front=0 rear=3
    ^front
          ^rear

5. deQueue() → "A" 꺼냄
   [ ][B][C][ ] front=1 rear=3
      ^front
         ^rear

6. enQueue("D")
   [ ][B][C][D] front=1 rear=0(회전)
      ^front
   ^rear

7. enQueue("E") → 불가능 (가득 참)
   큐가 가득 찼습니다!

8. deQueue() → "B" 꺼냄
   [ ][ ][C][D] front=2 rear=0
         ^front
    ^rear

원형 큐에서 MAX_QUEUE가 4이라면, 실제로 저장할 수 있는 원소 개수는 3개이다. (왜냐하면 front==rear일 때 "빈 상태"와 구분해야 하므로 한 칸은 항상 비워둔다.)

원형 큐(Queue)

- 배열 기반 원형 큐(Queue)

```c
#define MAX_QUEUE 4  //큐의 최대 크기

//전역 변수 선언
int queue[MAX_QUEUE];
int front = 0;  //데이터 꺼낼 위치
int rear = 0;   //데이터 넣을 위치

void enQueue(int x) {
    if ((rear + 1) % MAX_QUEUE == front) {
        printf("큐가 가득 찼습니다.\n");
        return;
    }
    queue[rear] = x; //뒤에서 데이터 넣기
    rear = (rear + 1) % MAX_QUEUE; //rear를 다음 위치로 이동
    //printf("front=%d, rear=%d, x=%d\n", queue[front], queue[rear], x);
    printf("%d ", x);
}
```

원형 큐(Queue)

- 배열 기반 원형 큐(Queue)

```
int deQueue() {
    if (front == rear) {
        printf("큐가 비었습니다.\n");
        return -1;
    }
    int data = queue[front];  //앞(front) 데이터를 저장
    front = (front + 1) % MAX_QUEUE;  //front 다음 위치로 이동
    return data;  //데이터를 내보냄
}

int main()
{
    puts("=== 큐에 데이터 넣기 ===");
    enQueue(10);
    enQueue(20);
    enQueue(30);
    //enQueue(40);  //오류, 큐가 가득 참
```

원형 큐(Queue)

- 배열 기반 원형 큐(Queue)

```
puts("\n=== 큐에서 데이터 꺼내기 ===");
int val;  //큐에서 꺼낸 값(데이터)

/*val = deQueue();
printf("%d ", val);

val = deQueue();
printf("%d ", val);

val = deQueue();
printf("%d ", val);

val = deQueue();
printf("%d ", val); */ // 비었을 때 처리 확인

while (front != rear) {
    val = deQueue();
    printf("%d ", val);
}
```

```
=== 큐에 데이터 넣기 ===
10 20 30
=== 큐에서 데이터 꺼내기 ===
10 20 30
```

원형 큐(Queue)

- 고객 대기열 큐(Queue)

```
#define MAX_QUEUE 4
#define NAME_LEN 20

/*
queue[MAX_QUEUE][NAME_LEN]; //이차원 배열
queue[4][20] = {"고객A", "고객B", "고객C", ""};

queue[0] → {'고','객','A','\0', … }  // 고객A
queue[1] → {'고','객','B','\0', … }  // 고객B
queue[2] → {'고','객','C','\0', … }  // 고객C
queue[3] → { ?, ?, ?, … }            // 비어있음

*/

// 큐 정의
char queue[MAX_QUEUE][NAME_LEN];
int front = 0;
int rear = 0;
```

원형 큐(Queue)

- 고객 대기열 큐(Queue)

```
// 큐가 비었는지 확인
int isEmpty() {
    return front == rear;
}

// 큐가 가득 찼는지 확인
int isFull() {
    return (rear + 1) % MAX_QUEUE == front;
}

void enQueue(const char* name) { //고객 추가
    if (isFull()) {
        printf("큐가 가득 찼습니다!\n");
        return;
    }
    strcpy(queue[rear], name); //뒤에 고객 이름 저장(복사)
    rear = (rear + 1) % MAX_QUEUE;
}
```

원형 큐(Queue)

- 고객 대기열 큐(Queue)

```
int deQueue(char* name) { // 고객 꺼내기
    if (isEmpty()) {
        printf("큐가 비었습니다!\n");
        name[0] = '\0'; //name을 빈 문자로 초기화 {'고' '객' 'A' '\0'}
        return -1;  //실패
    }
    strcpy(name, queue[front]); //앞의 이름을 복사
    front = (front + 1) % MAX_QUEUE;
    return 0; //성공
}
```

원형 큐(Queue)

- 고객 대기열 큐(Queue)

```
char name[NAME_LEN];

// 고객 대기열 추가
enQueue("고객A");
enQueue("고객B");
enQueue("고객C");

// 대기열 처리
while (!isEmpty()) {
    deQueue(name);
    printf("%s님 업무 처리 중...\n", name);
}

printf("모든 고객의 업무가 완료되었습니다.\n");
```

```
고객A님 업무 처리 중...
고객B님 업무 처리 중...
고객C님 업무 처리 중...
모든 고객의 업무가 완료되었습니다.
```

원형 큐(Queue)

- 고객 대기열 큐(Queue) – 포인터 배열로 정의

```c
#include <stdbool.h>

#define MAX_QUEUE 4
#define NAME_LEN 20

// 큐 정의 (포인터 배열)
char* queue[MAX_QUEUE];
int front = 0;
int rear = 0;

// 큐가 비었는지 확인
bool isEmpty() {
    return front == rear;
}

// 큐가 가득 찼는지 확인
bool isFull() {
    return (rear + 1) % MAX_QUEUE == front;
}
```

원형 큐(Queue)

- 고객 대기열 큐(Queue) – 포인터 배열로 정의

```
// 고객 추가
void enQueue(const char* name) {
    if (isFull()) {
        printf("큐가 가득 찼습니다!\n");
        return;
    }
    // 문자열 길이만큼 메모리 동적 할당
    queue[rear] = (char*)malloc(strlen(name) + 1); //'\0' 포함
    if (queue[rear] == NULL) {
        printf("메모리 할당 실패!\n");
        exit(1);
    }
    strcpy(queue[rear], name);  // 문자열 복사
    rear = (rear + 1) % MAX_QUEUE;
}
```

원형 큐(Queue)

- 고객 대기열 큐(Queue) – 포인터 배열로 정의

```
// 고객 꺼내기
int deQueue(char* name) {
    if (isEmpty()) {
        printf("큐가 비었습니다!\n");
        name[0] = '\0';
        return -1;
    }
    strcpy(name, queue[front]);    // 문자열 복사
    free(queue[front]);            // 메모리 해제
    queue[front] = NULL;           // 안전하게 초기화
    front = (front + 1) % MAX_QUEUE;
    return 0;
}
```

원형 큐(Queue)

- 구조체 기반 원형 큐(Queue)

```c
#define QUEUE_SIZE 10

// 구조체 정의
typedef struct {
    int data[QUEUE_SIZE];
    int front;
    int rear;
} CircularQueue;

// 큐 초기화
void initQueue(CircularQueue* q) {
    q->front = 0;
    q->rear = 0;
}
```

원형 큐(Queue)

- 구조체 기반 원형 큐(Queue)

```c
// 큐가 비었는지 확인
bool isEmpty(CircularQueue* q) {
    return q->front == q->rear;
}

// 큐가 가득 찼는지 확인
bool isFull(CircularQueue* q) {
    return (q->rear + 1) % QUEUE_SIZE == q->front;
}

// 데이터 추가
void enQueue(CircularQueue* q, int value) {
    if (isFull(q)) {
        printf("큐가 가득 찼습니다!!\n");
        return;
    }
    q->data[q->rear] = value;
    q->rear = (q->rear + 1) % QUEUE_SIZE;
}
```

원형 큐(Queue)

- 구조체 기반 원형 큐(Queue)

```
int deQueue(CircularQueue* q) { // 데이터 삭제
    if (isEmpty(q)) {
        printf("큐가 비었습니다!!\n");
        return -1;
    }
    int value = q->data[q->front];
    q->front = (q->front + 1) % QUEUE_SIZE;
    return value;
}

void printQueue(CircularQueue* q) { // 큐 상태 출력
    printf("큐 상태: ");
    if (isEmpty(q)) {
        printf("(비어 있음)\n");
        return;
    }
    int i = q->front;
    while (i != q->rear) {
        printf("%d ", q->data[i]);
        i = (i + 1) % QUEUE_SIZE;
    }
    printf("\n");
}
```

원형 큐(Queue)

- 구조체 기반 원형 큐(Queue)

```
CircularQueue q1, q2;

// 큐 초기화
initQueue(&q1);
initQueue(&q2);

printf("=== 큐 1에 데이터 넣기 ===\n");
enQueue(&q1, 10);
enQueue(&q1, 20);
enQueue(&q1, 30);
printQueue(&q1);

printf("\n=== 큐 1에서 데이터 빼기 ===\n");
printf("%d ", deQueue(&q1));
printf("%d ", deQueue(&q1));
printQueue(&q1);

printf("\n=== 큐 2 테스트 ===\n");
enQueue(&q2, 100);
enQueue(&q2, 200);
printQueue(&q2);
```

```
=== 큐 1에 데이터 넣기 ===
큐 상태 : 10 20 30

=== 큐 1에서 데이터 빼기 ===
10 20 큐 상태 : 30

=== 큐 2 테스트 ===
큐 상태 : 100 200
```

연결 리스트(Linked List)

- 연결 리스트(Linked List)의 필요성

  ✓ 배열 자료구조의 문제점

  배열에서 특정 요소를 삽입하거나 삭제하려면 빈 방을 왼쪽으로 이동하지 않으면, 빈 방을 찾아야하는 번거로움이 있다.

  이러한 문제를 보완하는 자료 구조가 연결리스트이다.

연결 리스트(Linked List)

- 배열에서 요소 삭제

```c
//학생 구조체 정의
typedef struct {
    int num;  //번호
}Student;

int main()
{
    Student st[10]; //구조체 배열 생성
    int i;

    for (i = 0; i < 10; i++) {
        st[i].num = i + 1;  //요소 저장
    }

    for (i = 0; i < 10; i++) {
        printf("%d ", st[i].num);  //요소 출력
    }
    printf("\n");
```

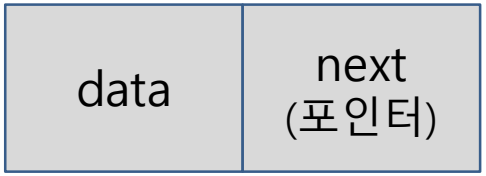
연결 리스트

- 배열에서 요소 삭제

```
    printf("4번 학생 전학\n");
    st[3].num = 0;  //요소 삭제

    printf("방을 왼쪽으로 이동\n");
    for (i = 3; i < 9; i++) {
        st[i].num = st[i + 1].num;  //한 칸 왼쪽으로 밀기
    }
    st[9].num = 0;

    for (i = 0; i < 10; i++) {
        printf("%d ", st[i].num);  //1 2 3 5 6 7 8 9 10 0
    }

    return 0;
}
```

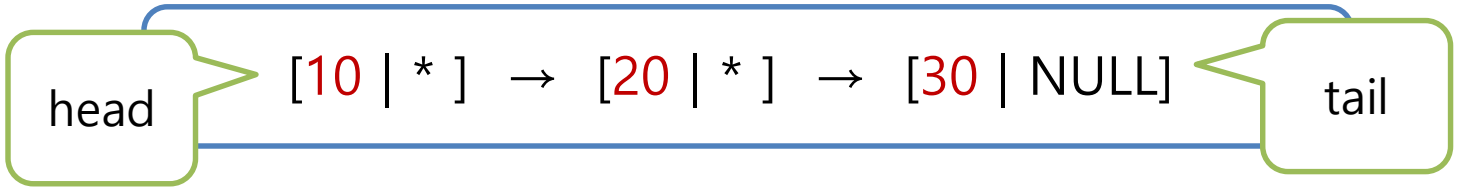
연결 리스트

- 연결 리스트(Linked List)란?
  - 데이터를 연속된 메모리에 저장하는 배열과 달리, **노드(node)**라는 독립적인 메모리 블록들이 포인터로 연결된 자료 구조이다
  - 각 노드는 데이터 + 다음 노드의 주소를 저장합니다.



노드(node)

- data: 노드가 가지는 데이터 값
- next: 포인터, 다음 노드의 주소 저장

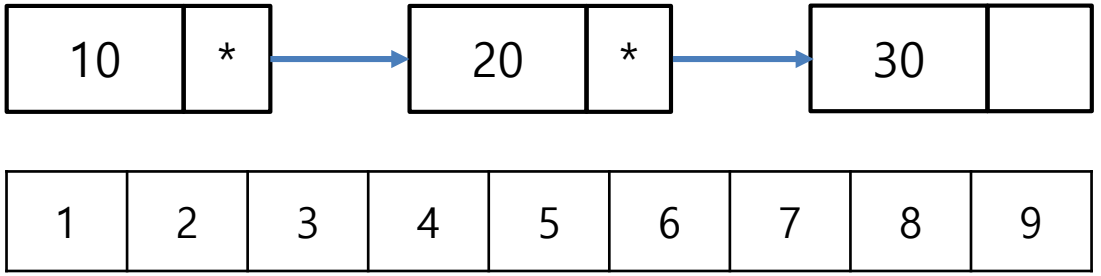


✓ 각 노드가 자신의 데이터와 다음 노드의 주소를 저장.

✓ NULL이 나오면 리스트의 끝.

연결 리스트(Linked List)

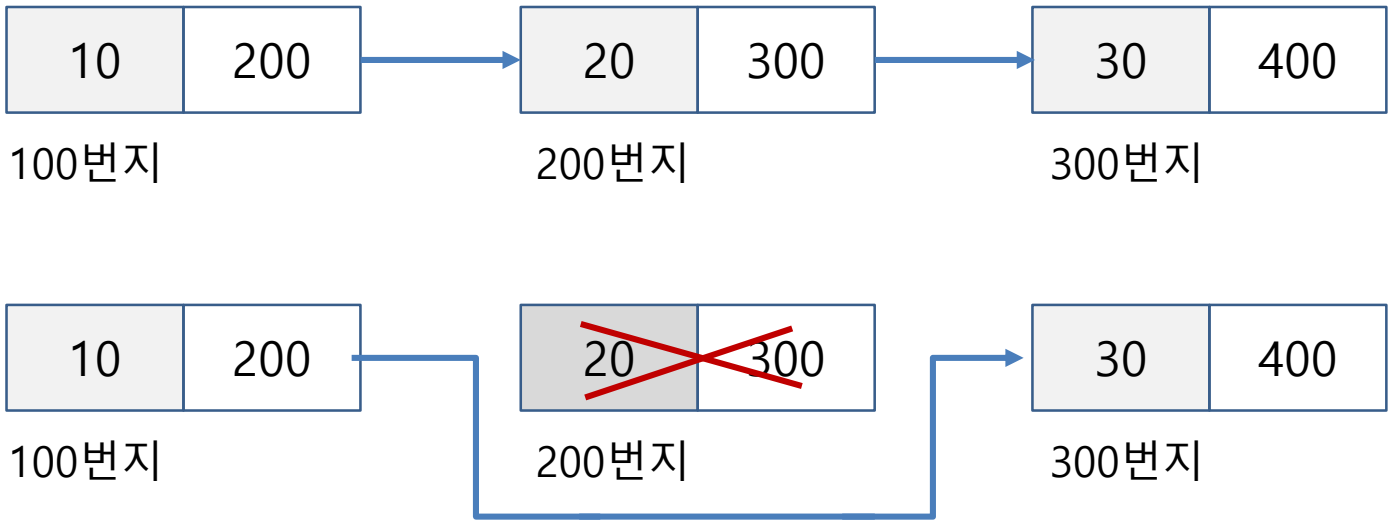
- 배열과 연결 리스트의 비교



비교 항목	배열	연결 리스트
메모리 구조	연속된 공간에 저장	흩어져 있는 노드를 포인터로 연결
크기 변경	어렵다 (재할당 필요)	자유롭게 추가/삭제 가능
접근 방법	random access 가능	random access 불가능
접근 속도	O(1) (인덱스로 접근)	O(n) (처음부터 순차 탐색)
삽입/삭제	느림 (데이터 이동 필요) O(n) 시간 소요	빠름 (포인터만 수정)

연결 리스트

- 노드 연결 및 삭제



연결 리스트

- 연결 리스트 구현

```c
typedef struct{
    int data;
    struct List* next;  //자기 참조 구조체
}List;

int main()
{
    List x, y, z; //노드 생성

    //요소 저장
    x.data = 10;
    y.data = 20;
    z.data = 30;

    //노드 연결
    x.next = &y;    //x -> y
    y.next = &z;    //y -> z
    z.next = NULL; //z는 마지막 노드
```

연결 리스트

- 연결 리스트(Linked List) 구현

```
//노드 출력
List* p;
p = &x;

printf("%d %x\n", x.data, p->next); //첫번째 노드 출력
p = p->next;
printf("%d %x\n", y.data, p->next); //두번째 노드 출력

//전체 노드 출력
for (p = &x; p != NULL; p = p->next) {
    printf("%d ", p->data);
}

printf("\n*** 노드 y 삭제 후 ***\n");
x.next = y.next; //x -> z(연결)
y.next = NULL; //y는 연결에서 제외

for (p = &x; p != NULL; p = p->next) {
    printf("%d ", p->data);
}
```

```
10 3beff868
20 3beff898
10 20 30
*** 노드 y 삭제 후 ***
10 30
```

연결 리스트

- 내 친구를 리스트로 저장하기

```c
typedef struct{
    int age;
    char name[20];
    struct Person* next;  //자기 참조 구조체
}Person;

int main()
{
    //노드 생성 및 초기화
    Person a = { 33, "손흥민" };
    Person b = { 21, "신유빈" };
    Person c = { 22, "임시현" };
    Person d = { 26, "이정후" };

    a.next = &b;
    b.next = &c;
    c.next = &d;
    d.next = NULL;
```

연결 리스트

- 내 친구를 리스트로 저장하기

```
Person* p; //구조체 포인터
p = &a;

//첫째 노드 출력
//printf("%d %s\n", p->age, p->name); //33 손흥민

//노드 순회 및 출력
for (p = &a; p != NULL; p = p->next) {
    printf("%d %s\n", p->age, p->name);
}

printf("*** 노드 c 삭제 후 ***\n");
b.next = c.next; //b -> d(연결)
c.next = NULL;

//노드 순회 및 출력
for (p = &a; p != NULL; p = p->next) {
    printf("%d %s\n", p->age, p->name);
}
```

```
33 손흥민
21 신유빈
22 임시현
26 이정후
*** 노드 c 삭제 후 ***
33 손흥민
21 신유빈
26 이정후
```

연결 리스트

- 동적 메모리 기반 단순 연결 리스트

```c
// 노드 구조 정의
typedef struct{
    int data;            // 노드가 저장하는 값
    struct Node* next;   // 다음 노드의 주소(자기 참조)
} Node;

int main() {
    // 노드 3개 생성 - 동적 할당(힙 메모리 영역)
    Node* head, * second, * third;

    head = (Node*)malloc(sizeof(Node));  //첫 노드
    second = (Node*)malloc(sizeof(Node));
    third = (Node*)malloc(sizeof(Node));

    // 데이터 저장
    head->data = 10;
    head->next = second;
```

연결 리스트

- 동적 메모리 기반 단순 연결 리스트

```
second->data = 20;
second->next = third;

third->data = 30;
third->next = NULL; // 마지막 노드

// 출력
Node* current = head;
while (current != NULL) {
    printf("%d -> ", current->data);
    current = current->next;
}
printf("NULL\n"); //10 -> 20 -> 30->NULL

// 메모리 해제
free(third);
free(second);
free(head);
```

연결 리스트

- 동적 메모리 기반 단순 연결 리스트(함수로 구현)

```c
// 노드 구조 정의
typedef struct {
    int data;           // 노드가 저장하는 값
    struct Node* next;  // 다음 노드의 주소(자기 참조)
} Node;

// 노드 생성 함수
Node* createNode(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("메모리 할당 실패!\n");
        exit(1);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}
```

연결 리스트

- 동적 메모리 기반 단순 연결 리스트(함수로 구현)

```c
// 리스트 출력 함수
void printList(Node* head) {
    Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main() {
    // 노드 3개 생성 (동적 할당)
    Node* head = createNode(10);
    Node* second = createNode(20);
    Node* third = createNode(30);
```

연결 리스트

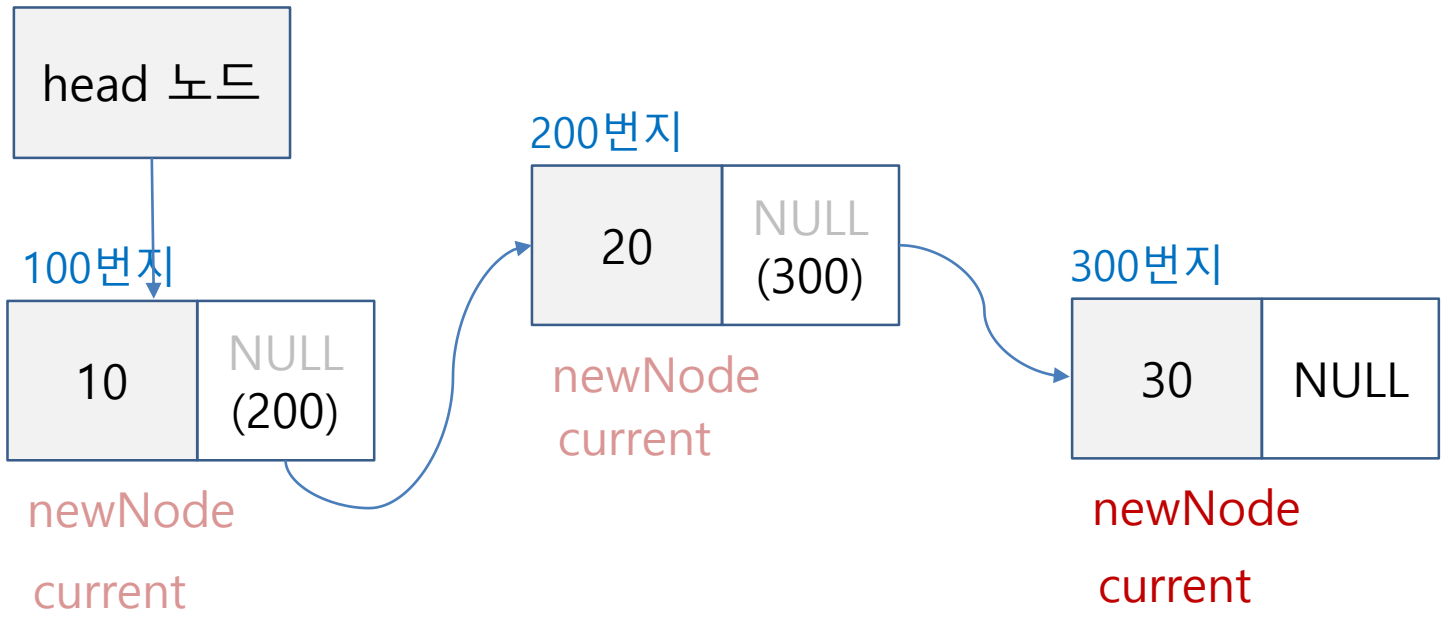
- 동적 메모리 기반 단순 연결 리스트(함수로 구현)

```
    // 노드 연결
    head->next = second;
    second->next = third;

    // 출력
    printf("연결 리스트 출력: ");
    printList(head); // 10 -> 20 -> 30 -> NULL

    // 메모리 해제
    free(third);
    free(second);
    free(head);

    return 0;
}
```

연결 리스트

- 동적 메모리 기반 단순 연결 리스트(사용자 입력)

```
노드 개수를 입력하세요: 3
1번째 노드 값 입력: 10
2번째 노드 값 입력: 20
3번째 노드 값 입력: 30

연결 리스트 출력: 10 -> 20 -> 30 -> NULL
```



연결 리스트

- 동적 메모리 기반 단순 연결 리스트(사용자 입력)

```c
typedef struct{
    int data;            // 노드가 저장하는 값
    struct Node* next;   // 다음 노드의 주소(자기 참조)
} Node;

int main() {
    int n;       //노드 개수
    int value;   //노드 값
    Node* head = NULL;
    Node* current = NULL;
    Node* newNode = NULL;

    printf("노드 개수를 입력하세요: ");
    scanf("%d", &n);
```

연결 리스트

- 동적 메모리 기반 단순 연결 리스트(사용자 입력)

```
for (int i = 0; i < n; i++) {
    newNode = (Node*)malloc(sizeof(Node)); // 새 노드 생성
    if (newNode == NULL) {
        printf("메모리 할당 실패\n");
        return 1;
    }

    printf("%d번째 노드 값 입력: ", i + 1);
    scanf("%d", &value);

    newNode->data = value;
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;  // 새 노드가 head가 됨
        current = newNode; //새 노드가 현재 노드가 됨
    }
    else {
        current->next = newNode; // 이전 노드와 연결
        current = newNode;       // 새 노드가 현재 노드가 됨
    }
}
```

연결 리스트

- 동적 메모리 기반 단순 연결 리스트(사용자 입력)

```
printf("\n연결 리스트 출력: ");
current = head;
while (current != NULL) {
    printf("%d -> ", current->data);
    current = current->next; //현재 노드를 다음 노드로 이동
}
printf("NULL\n");

// 리스트를 끝까지 순회하면서 동적 할당된 노드를 하나씩 메모리 해제(free)
current = head;
while (current != NULL) {
    Node* temp = current; // 현재 노드 주소를 저장
    current = current->next; //current를 다음 노드로 이동
    free(temp); //현재 노드 메모리 해제
}
```