



스택, 큐 등 자료구조

Stack and queue, etc. Data structure

스택 (stack)



스택이란?

- 쌓아놓는 더미
- 후입선출(LIFO: Last-In First-Out)
- 가장 최근에 들어온 데이터가 가장 먼저 나감

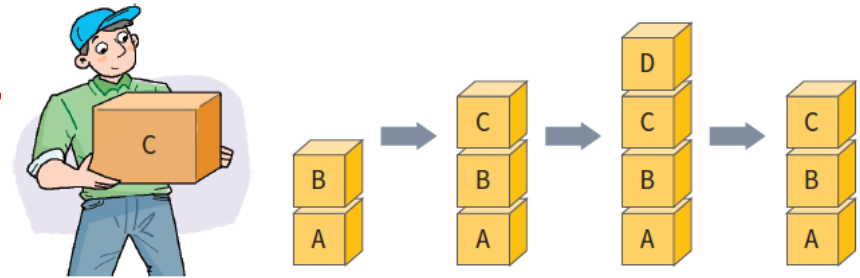


스택 (stack)



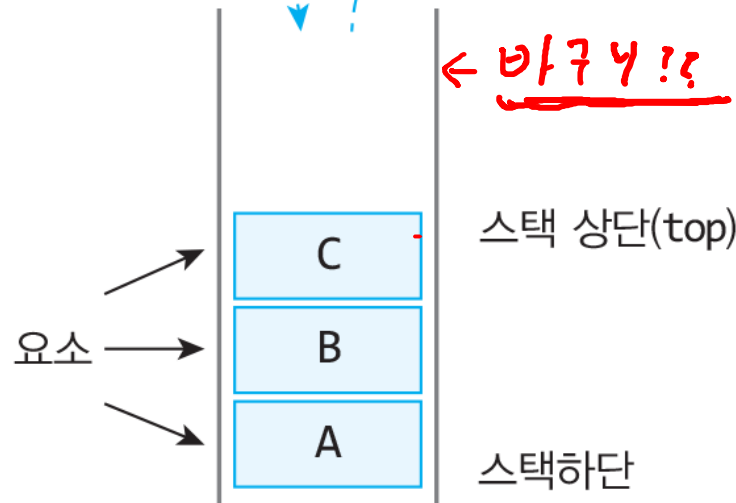
특징 및 구조

- 스택 상단: top
- 스택 하단: 따로 확인 필요 없음!
- 요소, 항목
- 공백상태, 포화상태
- 삽입, 삭제 연산 필요



삽입 연산(push)

삭제 연산(pop)

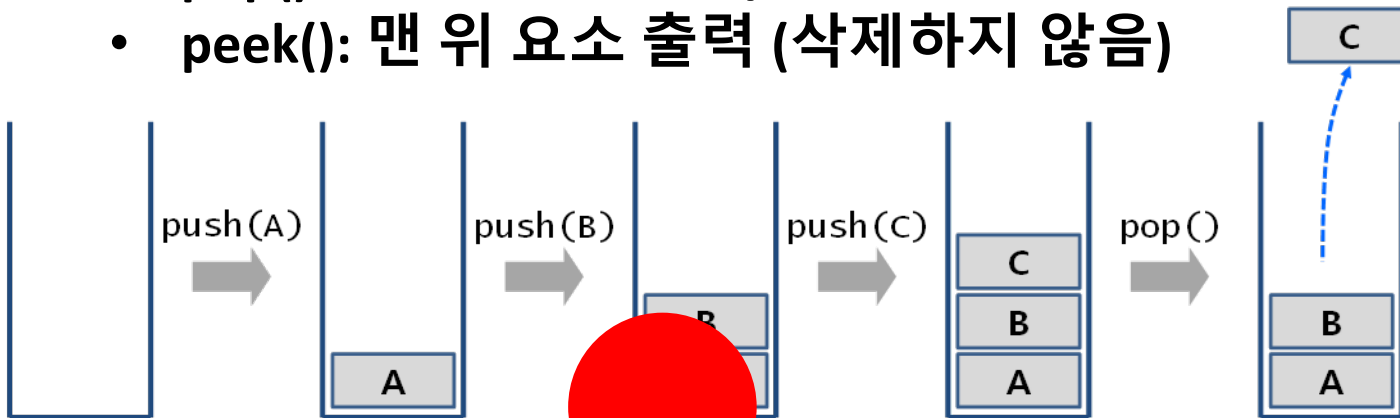


스택 (stack)



일반화

- 데이터: 후입선출의 접근 방법을 유지하는 요소들의 모임
- 연산:
 - `init()` : 스택 초기화
 - `is_empty()`: 스택의 비어 있는 유무 확인 (True, False)
 - `is_full()`: 스택이 가득 차 있는지 확인 (True, False)
 - `size()`: 스택 내 모든 요소들의 개수 반환
 - `push(x)`: 요소 `x`를 스택 맨 위 추가
 - `pop()`: 맨 위 요소 삭제 / 반환
 - `peek()`: 맨 위 요소 출력 (삭제하지 않음)



스택 (stack)

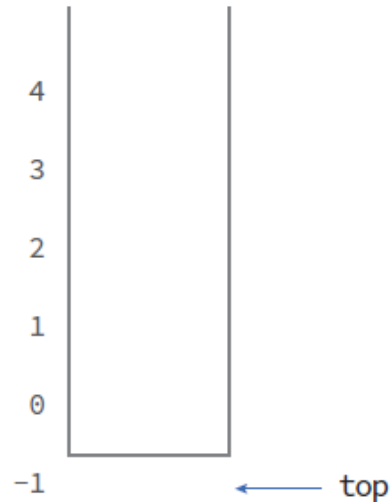


배열을 이용한 스택 구현

- 빈 공간 / 가득 차 있을 때에 대한 연산

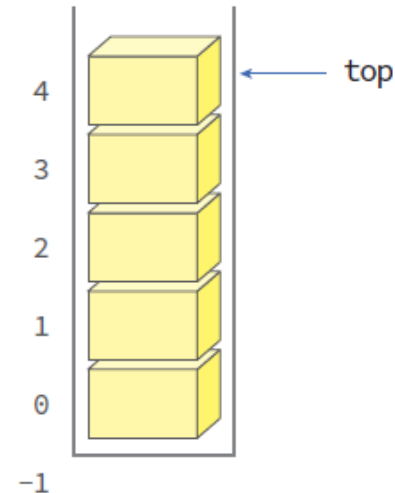
```
is_empty(S):
```

```
if top == -1  
    then return TRUE  
    else return FALSE
```



```
is_full(S):
```

```
if top == (MAX_STACK_SIZE-1)  
    then return TRUE  
    else return FALSE
```



스택 (stack)



배열을 이용한 스택 구현

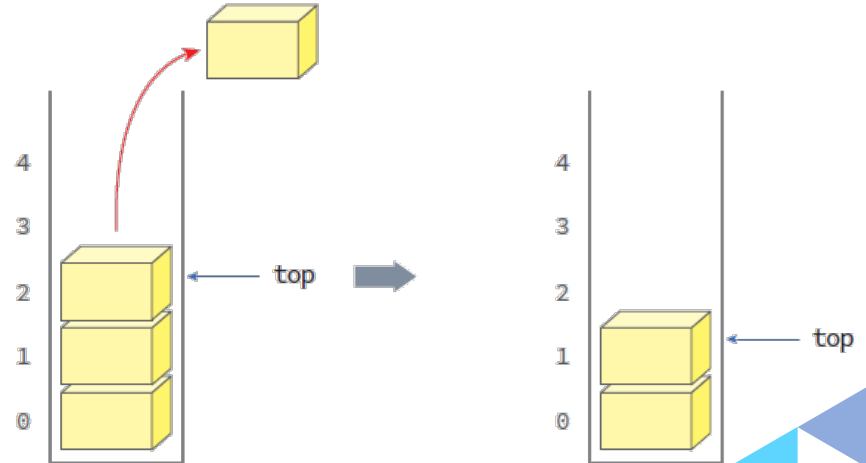
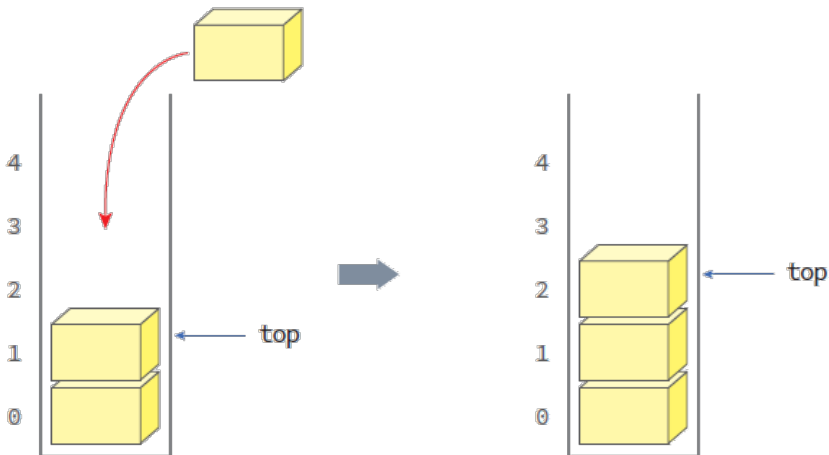
- pop / push 연산

```
push(S, x):
```

```
if is_full(S)  
    then error "overflow"  
else top←top+1  
    stack[top]←x
```

```
pop(S, x):
```

```
if is_empty(S)  
    then error "underflow"  
else e←stack[top]  
    top←top-1  
    return e
```



스택 (stack)



배열을 이용한 스택 구현

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_STACK_SIZE 100// 스택의 최대 크기
typedef int element;// 데이터의 자료형
element stack[MAX_STACK_SIZE]; // 1차원 배열
int top = -1;

int is_empty() {
    return (top == -1);
}
int is_full() {
    return (top == (MAX_STACK_SIZE - 1));
}
void push(element item) {
    if (is_full()) {
        fprintf(stderr, "스택 포화 에러\n");
        return;
    }
    else stack[++top] = item;
}
```

```
element pop() {
    if (is_empty()) {
        fprintf(stderr, "스택 공백 에러\n");
        exit(1);
    }
    else return stack[top--];
}

int main(void) {
    push(1);
    push(2);
    push(3);
    printf("%d\n", pop());
    printf("%d\n", pop());
    printf("%d\n", pop());
    return 0;
}
```

스택 (stack)



구조체를 이용한 스택 구현

```
#define MAX_STACK_SIZE 100
typedef int element;
typedef struct {
    element data[MAX_STACK_SIZE];
    int top;
} StackType;

void init_stack(StackType *s){
    s->top = -1;
}

...
int main(void){
    StackType s;

    init_stack(&s);
    push(&s, 1);
    push(&s, 2);
    push(&s, 3);
    printf("%d\n", pop(&s));
    printf("%d\n", pop(&s));
    printf("%d\n", pop(&s));
}
```

```
...
int main(void){
    StackType *s;
    s = (StackType *)malloc(sizeof(StackType));
    init_stack(s);
    push(s, 1);
    push(s, 2);
    push(s, 3);
    printf("%d\n", pop(s));
    printf("%d\n", pop(s));
    printf("%d\n", pop(s));
    free(s);
}
```

동적

스택 (stack)



응용 (괄호 검사)

- 문자열에 있는 괄호를 차례대로 조사
 - . 이 때, 왼쪽 괄호를 만나면 스택에 push
 - . 오른쪽 괄호를 만나면 스택에서 top 괄호를 pop 후 오른쪽 괄호와 짝이 맞는지 검사 (만약, 스택이 비어있으면 괄호 짝이 맞지 않으므로 오류)
- 마지막 괄호까지 조사한 후에도 스택에 괄호가 남아 있으면 이 또한 오류 반환

스택 (stack)



응용 (괄호 검사)

```
...
int check_matching(const char *in){
    StackType s;    char ch, open_ch;
    int i, n = strlen(in);  init_stack(&s);
    for (i = 0; i < n; i++) {
        ch = in[i];
        switch (ch) {
            case '(': case '[': case '{ ':
                push(&s, ch);
                break;
            case ')': case ']': case '} ':
                if (is_empty(&s)) return 0;
                else {
                    open_ch = pop(&s);
                    if ((open_ch == '(' && ch != ')') ||
                        (open_ch == '[' && ch != ']') ||
                        (open_ch == '{' && ch != '}')) {
                        return 0;
                    }
                }
                break;
        }
    }
    if (!is_empty(&s)) return 0; // 스택에 남아있으면 오류
    return 1;
}
...
```

```
int main(void){
    char* p = "{ A[(i+1)]=0; }";
    if (check_matching(p) == 1)
        printf("%s 괄호검사성공\n", p);
    else
        printf("%s 괄호검사실패\n", p);
    return 0;
}
```

스택 (stack)



응용 (괄호 검사 - 수식 계산)

- 수식 표기 방법: 전위(prefix), 중위(infix), 후위(postfix)

중위 표기법	전위 표기법	후위 표기법
$2+3*4$	$+2*34$	$234*+$
$a*b+5$	$+5*ab$	$ab*5+$
$(1+2)+7$	$+7+12$	$12+7+$

- 수식을 왼쪽에서 오른쪽으로 스캔
 - 피연산자일 경우 스택 저장
 - 연산자이면 필요한 수만큼 피연산자를 스택에서 꺼내 연산 실행
- 그 결과를 다시 스택에 저장

스택 (stack)



응용 (괄호 검사 - 수식 계산)

Calc_postfix (expr)

스택 초기화;

for 항목 in expr

do if (항목이 피연산자이면)

s.push(item);

if (항목이 연산자 op이면)

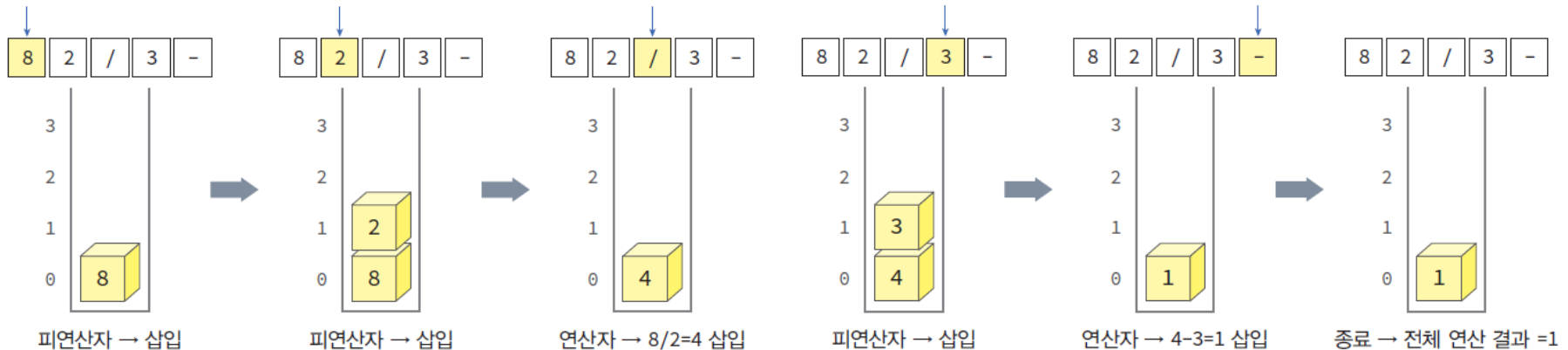
then second ← pop();

first ← pop();

temp ← first op second; // op 는 +-* / 중의 하나

push(temp);

result ← pop();



스택 (stack)



응용 (괄호 검사 - 수식 계산)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_STACK_SIZE 100

typedef char element;
#define MAX_STACK_SIZE 100

typedef struct {
    element data[MAX_STACK_SIZE];
    int top;
} StackType;

...

int main(void){
    int result;
    printf("후위표기식은 82/3-32*+Wn");
    result = eval("82/3-32*+");
    printf("결과값은 %dWn", result);
    return 0;
}
```

스택 (stack)

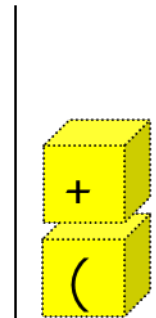
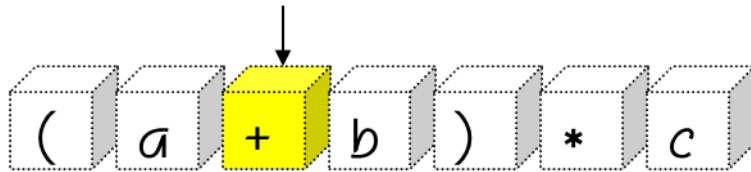
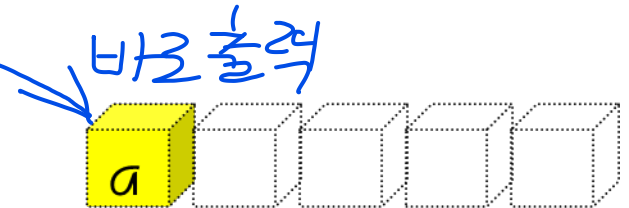
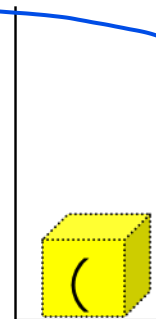
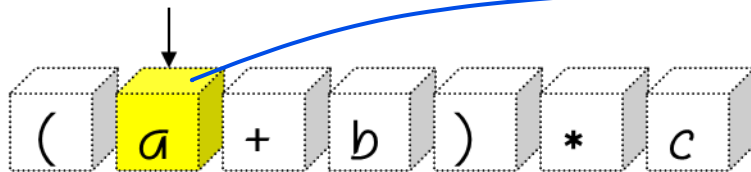
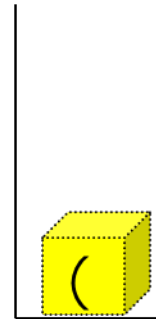
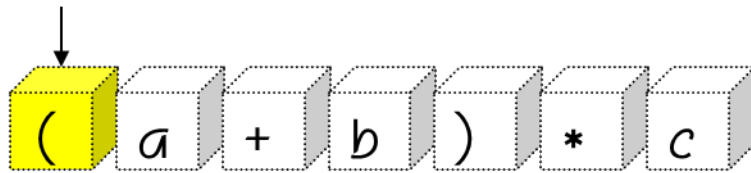


응용 (중위 표기 수식을 후위 표기 변환)

- 중위 표기와 후위 표기는 피연산자의 순서가 동일하며, 연산자들 순서만 다름(우선순위 순서)
- 알고리즘
 1. 피연산자를 만나면 그대로 출력
 2. 연산자를 만나면 스택에 저장했다가 스택보다 우선 순위가 낮은 연산자가 나오면 그 때 출력
 3. 왼쪽 괄호는 우선순위가 가장 낮은 연산자로 취급
 4. 오른쪽 괄호가 나타나면 스택에서 왼쪽 괄호 위에 쌓여있는 모든 연산자 출력

스택 (stack)

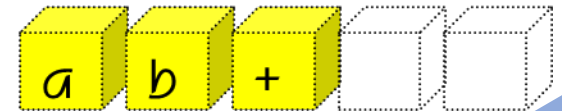
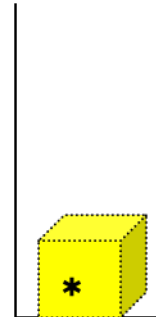
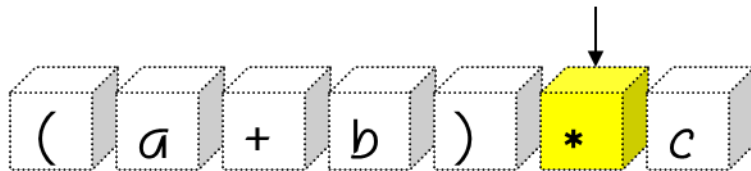
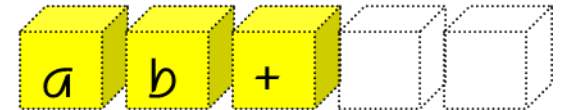
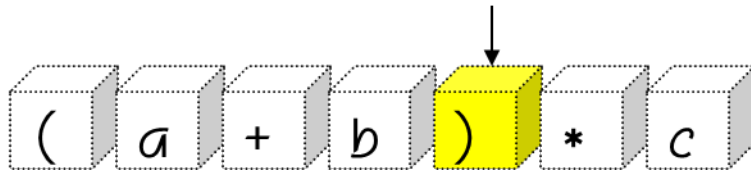
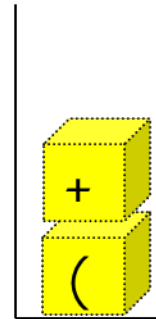
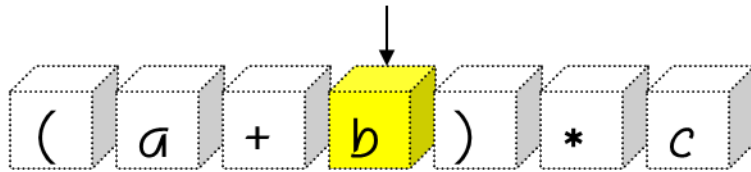
🔍 응용 (중위 표기 수식을 후위 표기 변환)



스택 (stack)



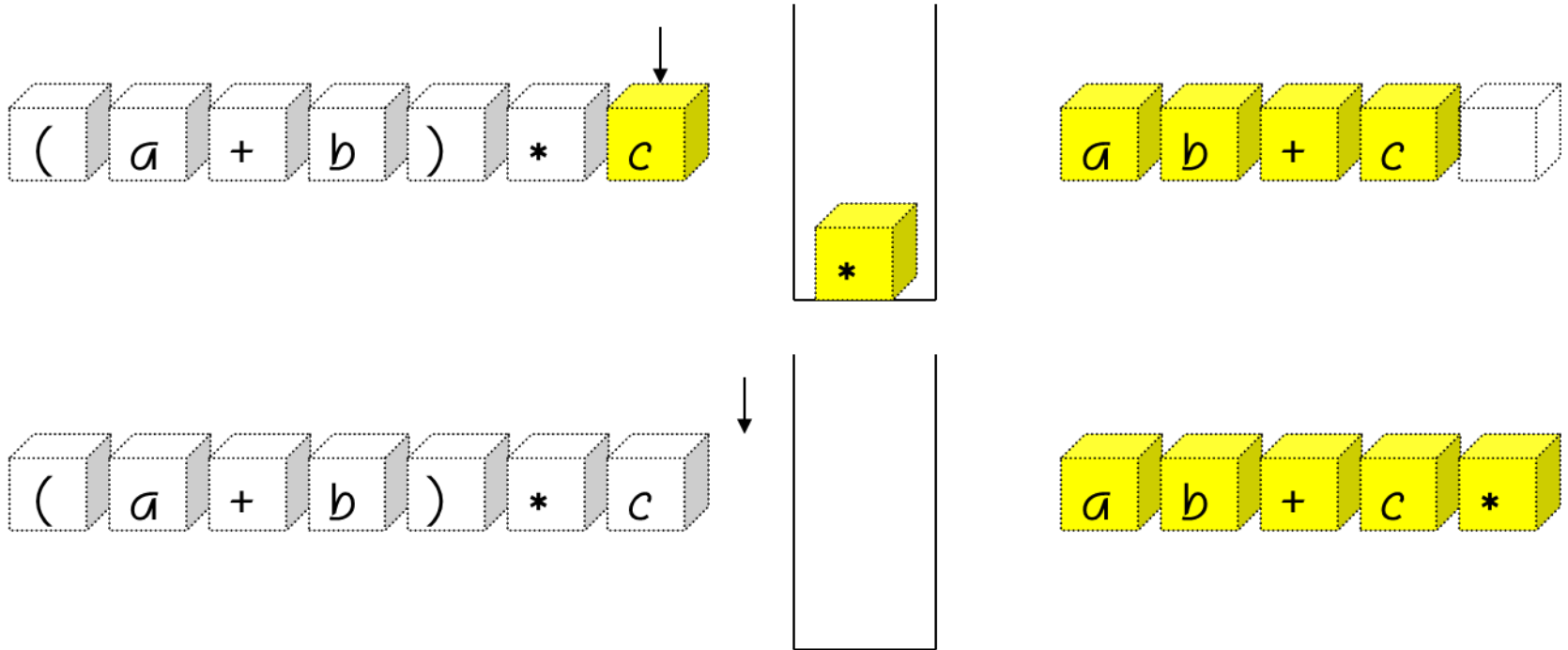
응용 (중위 표기 수식을 후위 표기 변환)



스택 (stack)



응용 (중위 표기 수식을 후위 표기 변환)



스택 (stack)



응용 (중위 표기 수식을 후위 표기 변환)

구조체를 이용한 스택 코드 사용

```
int prec(char op){
    switch (op) {
        case '(': case ')': return 0;
        case '+': case '-': return 1;
        case '*': case '/': return 2;
    }
    return -1;
}

void infix_to_postfix(char exp[]){
    int i = 0;
    char ch, top_op;
    int len = strlen(exp);
    StackType s;
    init_stack(&s);                // 스택 초기화
    for (i = 0; i < len; i++) {
        ch = exp[i];
        switch (ch) {
            case '+': case '-': case '*': case '/': // 연산자
                while (!is_empty(&s) && (prec(ch) <= prec(peek(&s))))
                    printf("%c", pop(&s));
                push(&s, ch);
                break;
```

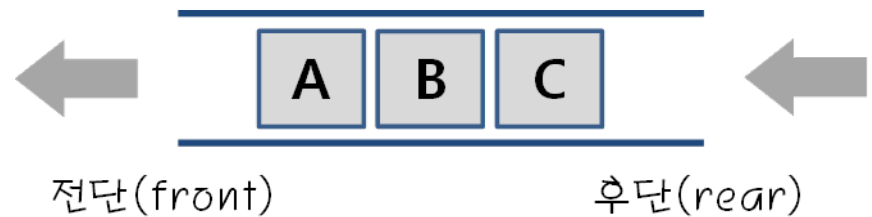
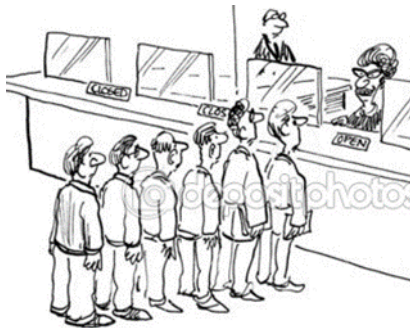
```
        case '(': // 왼쪽 괄호
            push(&s, ch);
            break;
        case ')': // 오른쪽 괄호
            top_op = pop(&s);
            while (top_op != '(') {
                printf("%c", top_op);
                top_op = pop(&s);
            }
            break;
        default: // 피연산자
            printf("%c", ch);
            break;
        }
    }
    while (!is_empty(&s))
        printf("%c", pop(&s));
}
```

큐 (queue)



큐 란?

- 먼저 들어온 데이터가 먼저 나가는 자료구조
- 선입 선출 (FIFO: First-In First-Out)

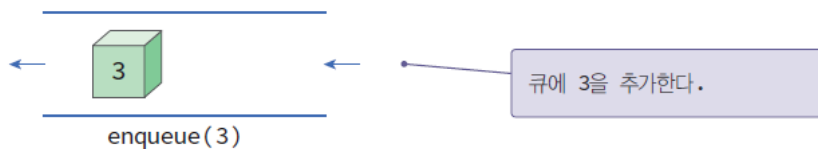


큐 (queue)



일반화

- 데이터: 선입 선출의 접근 방법을 유지하는 요소들의 모임
- 연산:
 - `init()` : 큐 초기화
 - `is_empty()`: 큐가 비어 있는 유무 확인 (True, False)
 - `is_full()`: 큐가 가득 차 있는지 확인 (True, False)
 - `size()`: 큐 내 모든 요소들의 개수 반환
 - `peek()`: 큐가 비어있지 않으면 맨 앞 요소 출력 (삭제하지 않음)
 - `enqueue(e)`: 주어진 요소 `e`를 큐 맨 뒤에 추가
 - `dequeue()`: 큐가 비어있지 않으면 맨 앞 요소 삭제

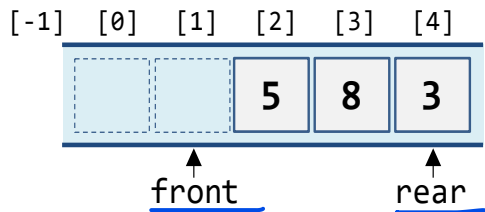


큐 (queue)

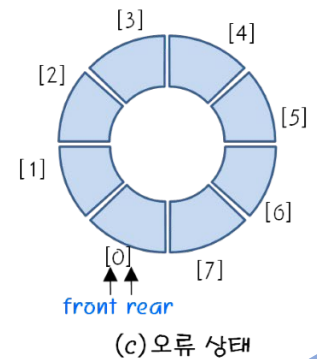
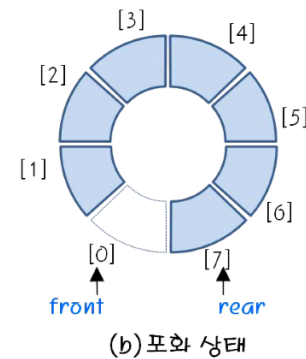
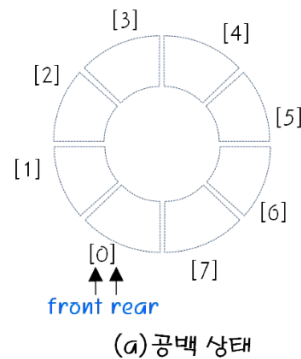
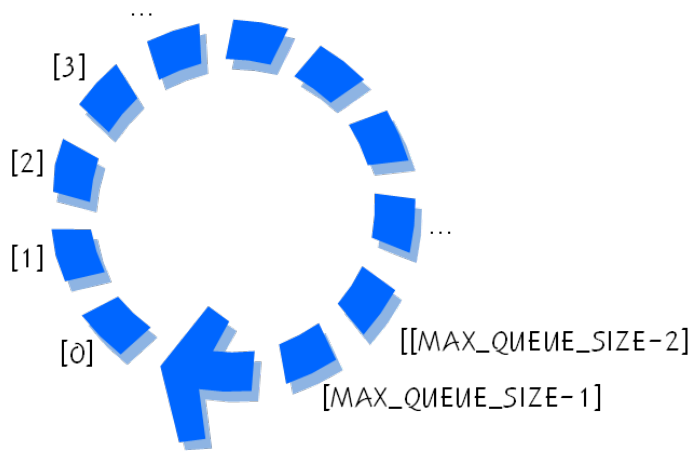
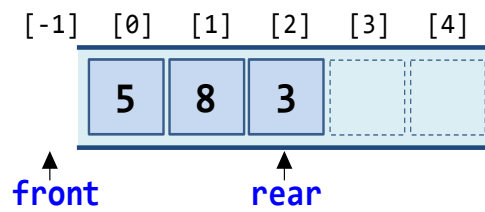


선형 큐 / 원형 큐

- 배열을 선형으로 사용하여 큐 구현
 - 단점: 삽입을 계속하기 위해서는 요소들을 이동해야함
- 이를 해결하기 위해 원형 형태로 큐를 구현



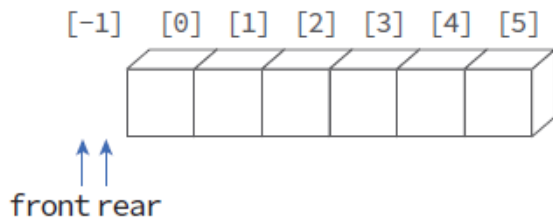
→
항목들을
전단쪽으로
모두 이동



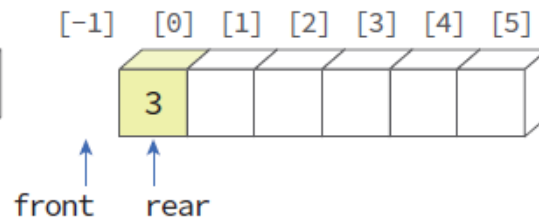
큐 (queue)



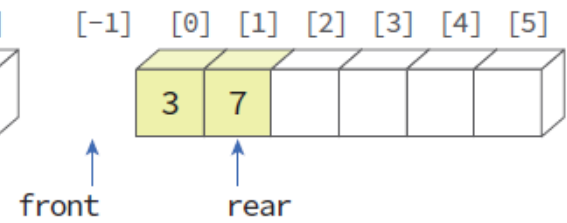
선형 큐



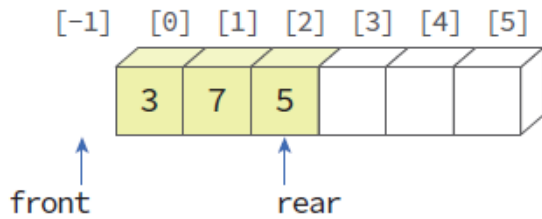
(a) 초기상태



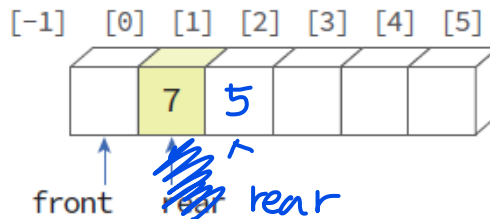
(b) enqueue(3)



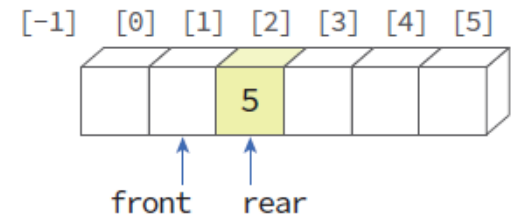
(c) enqueue(7)



(d) enqueue(5)



(e) dequeue()



(f) dequeue()

큐 (queue)



선형 큐

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_QUEUE_SIZE 5

typedef int element;
typedef struct { // 큐 타입
    int front;
    int rear;
    element data[MAX_QUEUE_SIZE];
} QueueType;

void error(char* message){
    fprintf(stderr, "%s\n", message);
    exit(1);
}

void init_queue(QueueType* q){
    q->rear = -1;
    q->front = -1;
}
```

```
void queue_print(QueueType* q){
    for (int i = 0; i < MAX_QUEUE_SIZE; i++) {
        if (i <= q->front || i > q->rear)
            printf("    | ");
        else
            printf("%d | ", q->data[i]);
    }
    printf("\n");
}

int is_full(QueueType* q){
    if (q->rear == MAX_QUEUE_SIZE - 1)
        return 1;
    else
        return 0;
}

int is_empty(QueueType* q){
    if (q->front == q->rear)
        return 1;
    else
        return 0;
}
```

큐 (queue)



선형 큐

```
void enqueue(QueueType* q, int item){
    if (is_full(q)) {
        error("큐가 포화상태입니다.");
        return;
    }
    q->data[++(q->rear)] = item;
}

int dequeue(QueueType* q){
    if (is_empty(q)) {
        error("큐가 공백상태입니다.");
        return -1;
    }
    int item = q->data[++(q->front)];
    return item;
}
```

```
int main(void){
    int item = 0;
    QueueType q;

    init_queue(&q);

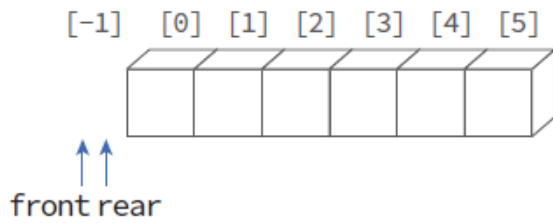
    enqueue(&q, 10);
    queue_print(&q);
    enqueue(&q, 20);
    queue_print(&q);
    enqueue(&q, 30);
    queue_print(&q);

    item = dequeue(&q);
    queue_print(&q);
    item = dequeue(&q);
    queue_print(&q);
    item = dequeue(&q);
    queue_print(&q);
    return 0;
}
```

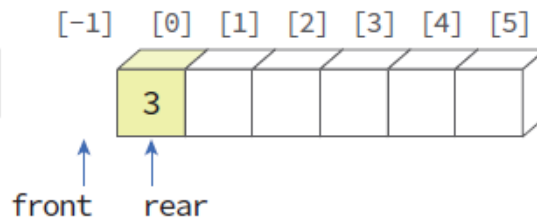

큐 (queue)



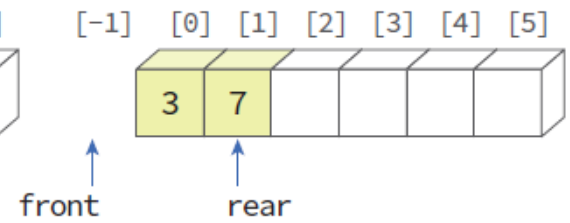
원형 큐



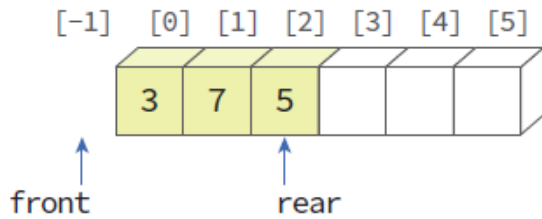
(a) 초기상태



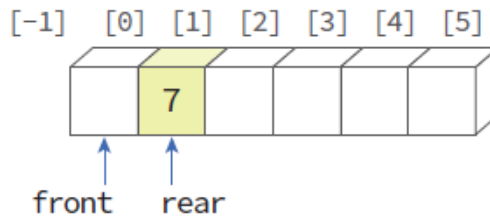
(b) enqueue(3)



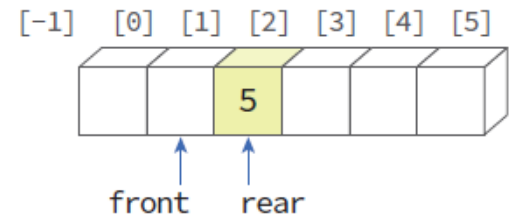
(c) enqueue(7)



(d) enqueue(5)



(e) dequeue()



(f) dequeue()

큐 (queue)



원형 큐

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_QUEUE_SIZE 5
typedef int element;
typedef struct { // 큐 타입
    element data[MAX_QUEUE_SIZE];
    int front, rear;
} QueueType;

void error(char* message){
    fprintf(stderr, "%s\n", message);
    exit(1);
}

void init_queue(QueueType* q){
    q->front = q->rear = 0;
}

int is_empty(QueueType* q){
    return (q->front == q->rear);
}

int is_full(QueueType* q){
    return ((q->rear + 1) % MAX_QUEUE_SIZE == q->front);
}
```

```
void queue_print(QueueType* q){
    printf("QUEUE(front=%d rear=%d) = ", q->front, q->rear);
    if (!is_empty(q)) {
        int i = q->front;
        do {
            i = (i + 1) % (MAX_QUEUE_SIZE);
            printf("%d | ", q->data[i]);
            if (i == q->rear)
                break;
        } while (i != q->front);
    }
    printf("\n");
}
```

나머지 연산 미응

큐 (queue)



원형 큐

```
void enqueue(QueueType* q, element item){
    if (is_full(q))
        error("큐가 포화상태입니다");
    q->rear = (q->rear + 1) % MAX_QUEUE_SIZE;
    q->data[q->rear] = item;
}

element dequeue(QueueType* q){
    if (is_empty(q))
        error("큐가 공백상태입니다");
    q->front = (q->front + 1) % MAX_QUEUE_SIZE;
    return q->data[q->front];
}

element peek(QueueType* q){
    if (is_empty(q))
        error("큐가 공백상태입니다");
    return q->data[(q->front + 1) % MAX_QUEUE_SIZE];
}
```

```
int main(void){
    QueueType queue;
    int element;

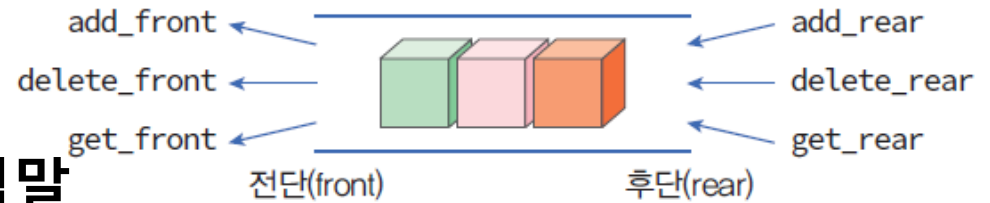
    init_queue(&queue);
    printf("--데이터 추가 단계--\n");
    while (!is_full(&queue)){
        printf("정수를 입력하시오: ");
        s_scanf("%d", &element);
        enqueue(&queue, element);
        queue_print(&queue);
    }
    printf("큐는 포화상태입니다.\n\n");

    printf("--데이터 삭제 단계--\n");
    while (!is_empty(&queue)){
        element = dequeue(&queue);
        printf("꺼내진 정수: %d\n", element);
        queue_print(&queue);
    }
    printf("큐는 공백상태입니다.\n");
    return 0;
}
```

큐 (queue)



덱(deque)



- Double-ended queue 줄임말
- 전단(front)과 후단(rear)에서 모두 삽입과 삭제가 가능한 큐
- 연산:
 - create(): 덱 생성 init(): 초기화
 - is_empty(): 덱 비어 있는 유무 확인 (True, False)
 - is_full(): 스택이 가득 차 있는지 확인 (True, False)
 - add_front(): 덱 앞 요소 추가
 - add_rear(): 덱 뒤에 요소 추가
 - delete_front: 덱 앞에 있는 요소를 반환하고 다음 삭제
 - delete_rear: 덱 뒤에 있는 요소를 반환하고 다음 삭제
 - get_front: 덱 앞에 삭제하지 않고 앞에 있는 요소 반환
 - get_rear: 덱 뒤에 삭제하지 않고 뒤에 있는 요소 반환

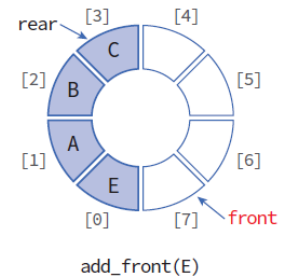
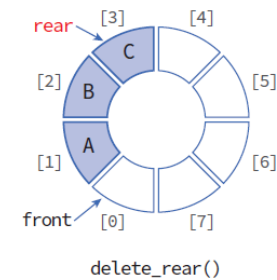
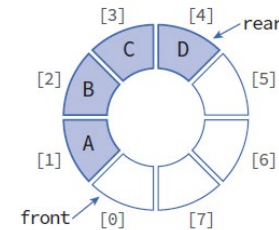
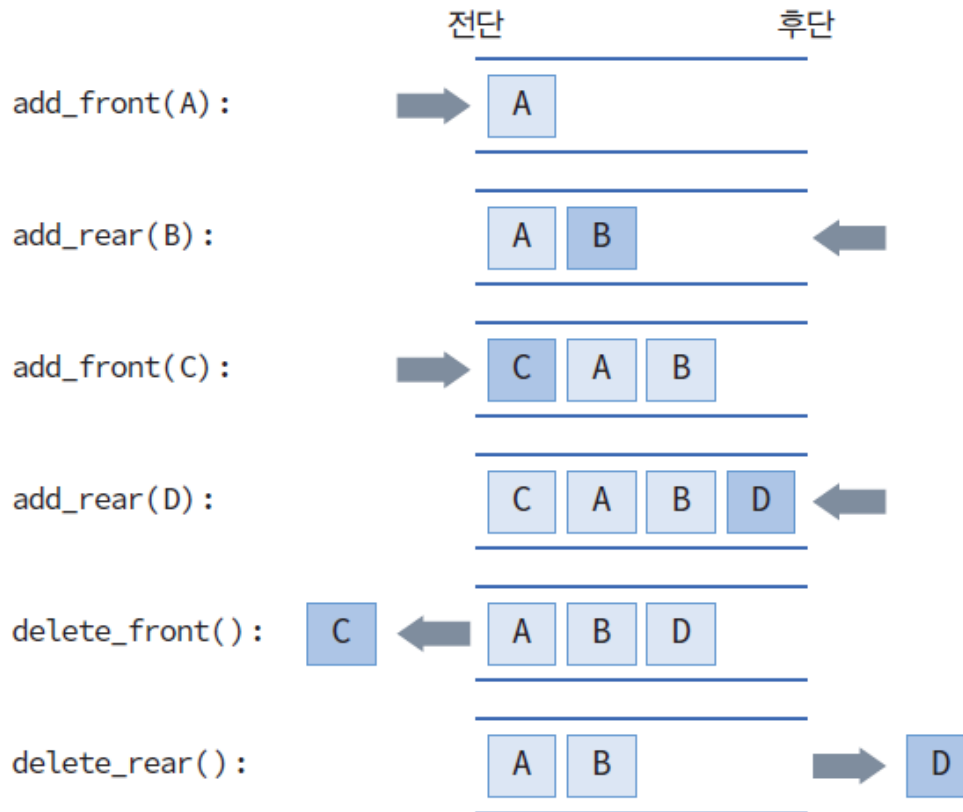
큐 (queue)



덱(deque)

```
front ← (front-1 + MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE;  
rear ← (rear-1 + MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE;
```

- Double-ended queue 줄임말
- 전단(front)과 후단(rear)에서 모두 삽입과 삭제가 가능한 큐



큐 (queue)



덱(deque)

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_QUEUE_SIZE 5
typedef int element;
typedef struct { // 큐 타입
    element data[MAX_QUEUE_SIZE];
    int front, rear;
} DequeType;
void error(char* message){
    fprintf(stderr, "%s\n", message);
    exit(1);
}
void init_deque(DequeType* q){
    q->front = q->rear = 0;
}
int is_empty(DequeType* q){
    return (q->front == q->rear);
}
int is_full(DequeType* q){
    return ((q->rear + 1) % MAX_QUEUE_SIZE == q->front);
}
```

```
void deque_print(DequeType* q){
    printf("DEQUE(front=%d rear=%d) = ", q->front, q->rear);
    if (!is_empty(q)) {
        int i = q->front;
        do {
            i = (i + 1) % (MAX_QUEUE_SIZE);
            printf("%d | ", q->data[i]);
            if (i == q->rear)
                break;
        } while (i != q->front);
    }
    printf("\n");
}
```

큐 (queue)



덱(dequeue)

```
void add_rear(DequeType* q, element item){
    if (is_full(q))
        error("큐가 포화상태입니다");
    q->rear = (q->rear + 1) % MAX_QUEUE_SIZE;
    q->data[q->rear] = item;
}

void add_front(DequeType* q, element val){
    if (is_full(q))
        error("큐가 포화상태입니다");
    q->data[q->front] = val;
    q->front = (q->front - 1 + MAX_QUEUE_SIZE) %
MAX_QUEUE_SIZE;
}

element get_rear(DequeType* q){
    if (is_empty(q))
        error("큐가 공백상태입니다");
    return q->data[q->rear];
}
```

```
element delete_rear(DequeType* q){
    int prev = q->rear;
    if (is_empty(q))
        error("큐가 공백상태입니다");
    q->rear = (q->rear - 1 + MAX_QUEUE_SIZE) %
MAX_QUEUE_SIZE;
    return q->data[prev];
}

element delete_front(DequeType* q){
    if (is_empty(q))
        error("큐가 공백상태입니다");
    q->front = (q->front + 1) % MAX_QUEUE_SIZE;
    return q->data[q->front];
}

element get_front(DequeType* q){
    if (is_empty(q))
        error("큐가 공백상태입니다");
    return q->data[(q->front + 1) % MAX_QUEUE_SIZE];
}
```

큐 (queue)



덱(deque)

```
int main(void){
    DequeType queue;

    init_deque(&queue);
    for (int i = 0; i < 3; i++) {
        add_front(&queue, i);
        deque_print(&queue);
    }
    for (int i = 0; i < 3; i++) {
        delete_rear(&queue);
        deque_print(&queue);
    }
    return 0;
}
```


연결 리스트 (Linked List)



리스트

- 순서를 가진 항목들의 모임
- 예) 요일, 해야할 일, 카드, 다항식 각 항들 ...

My To-Do List	
Date	Item
<input type="checkbox"/>	
<input type="checkbox"/>	
<input type="checkbox"/>	
<input type="checkbox"/>	
<input type="checkbox"/>	
<input type="checkbox"/>	
<input type="checkbox"/>	
<input type="checkbox"/>	
<input type="checkbox"/>	
<input type="checkbox"/>	

Bucket List
• 유럽가기
• 오토바이 타기
• 에버레스트 등반
• 유화 그리기
• 발레 배우기
• 테니스 대회 우승하기
• 사자 기르기
• 스카이 다이빙



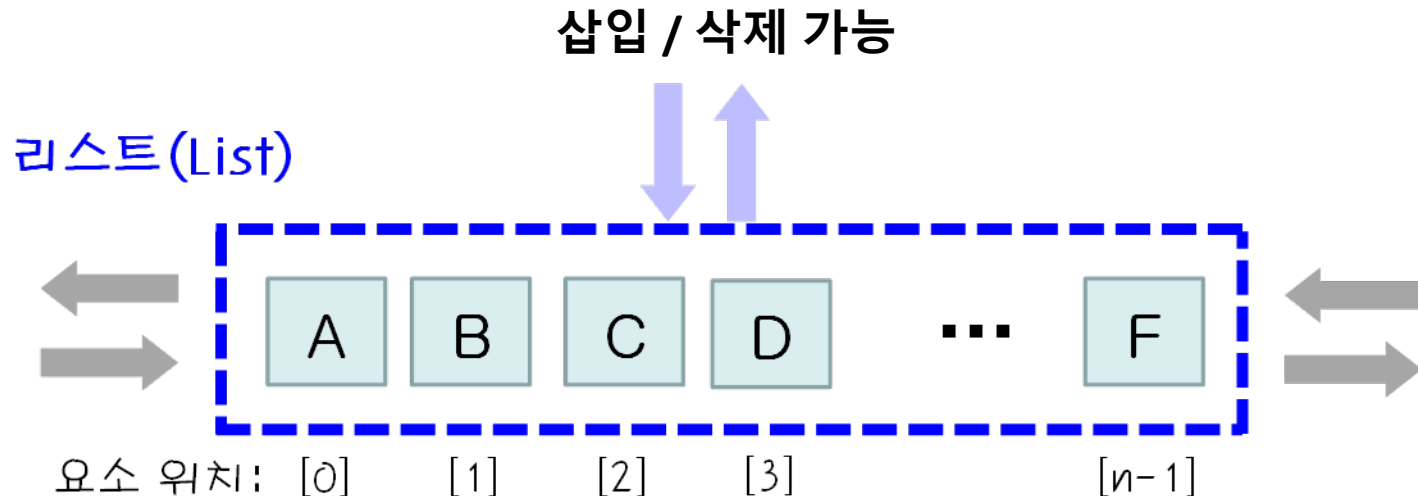
$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

연결 리스트 (Linked List) 검색 가능



리스트

- 스택, 큐와의 공통점 / 차이점
 - 공통점: 선형 자료구조
 - 차이점: 자료 접근 위치
(리스트는 임의의 위치에서도 삽입 삭제가 가능)



연결 리스트 (Linked List)



리스트 연산

- 기본 연산
 - 특정 위치 새로운 요소 삽입
 - 특정 위치 요소 삭제
 - 특정 위치 요소 반환
 - 리스트 데이터 비어있는지 확인
- 고급 연산
 - 리스트에 어떤 요소가 있는지 확인
 - 특정 위치에 요소를 새로운 요소로 대체
 - 리스트 안 요소 개수 확인
 - 리스트 안 모든 요소 출력

연결 리스트 (Linked List)



일반화

- 데이터: 임의의 접근 방법을 제공하는 같은 타입 요소들의 순서 있는 모임
- 연산:
 - Init() : 리스트 초기화
 - Is_empty(): 리스트가 비어 있는 유무 확인 (True, False)
 - Is_full(): 리스트가 가득 차 있는지 확인 (True, False)
 - size(): 리스트 안 모든 요소들의 개수 반환
 - insert(pos, item): pos 위치에 새로운 요소 item 삽입
 - delete(pos): pos 위치에 있는 요소 삭제
 - get_entry(pos): pos 위치에 있는 요소 반환
 - find(item): 리스트에 요소 item이 있는지 확인
 - replace(pos, item): pos 위치를 새로운 요소 item으로 바꿈

연결 리스트 (Linked List)



배열 / 연결리스트

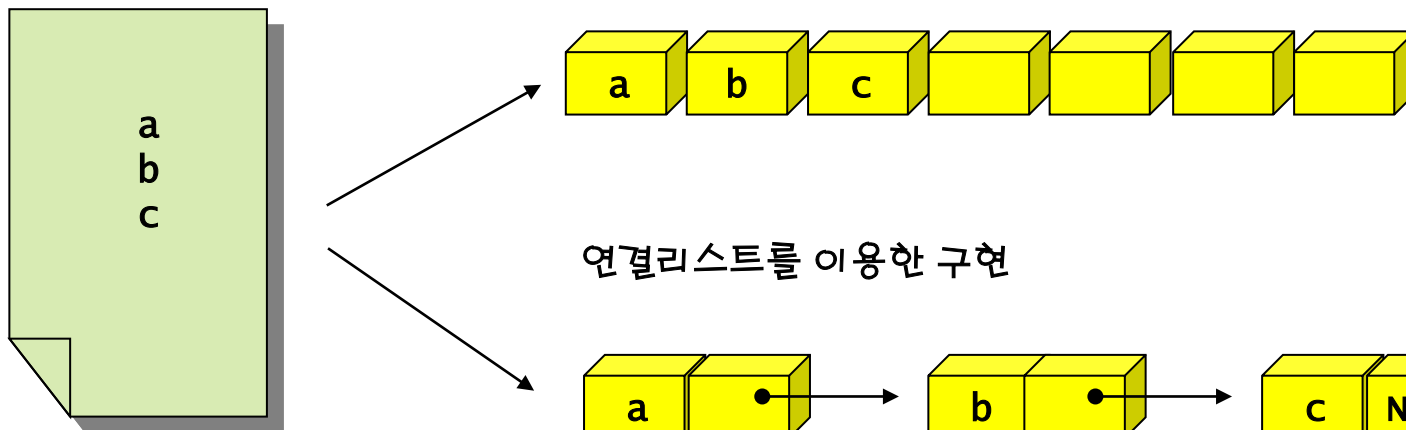
- 배열 이용

- 구현 간단
- 삽입, 삭제 시 오버헤드
- 항목 개수 제한

- 연결리스트 이용

- 구현 복잡
- 삽입, 삭제 효율
- 크기가 제한 없음
(메모리 동적할당)

배열을 이용한 구현

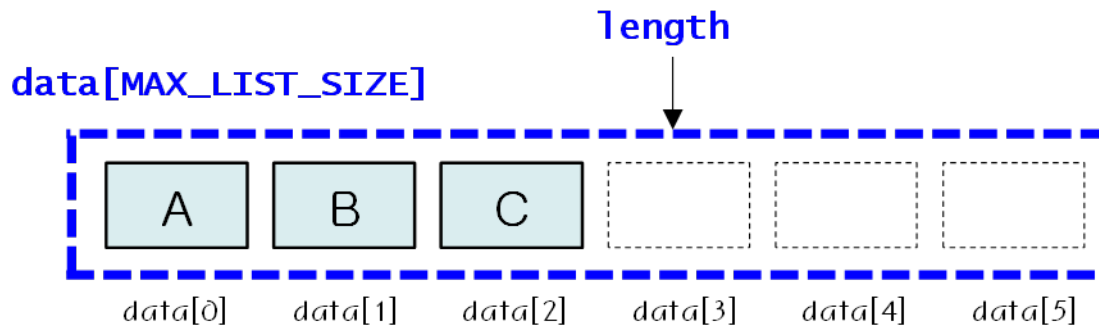


연결 리스트 (Linked List)



구현 방법 (배열)

- 배열을 이용하여 리스트를 구현하면 순차적인 메모리 공간이 할당(리스트 순차적 표현)
- 1차원 배열에 항목들을 순서대로 저장



```
#include <stdio.h>
#include <stdlib.h>

#define MAX_LIST_SIZE 100 // 리스트의 최대크기

typedef int element; // 항목의 정의
typedef struct {
    element array[MAX_LIST_SIZE]; // 배열 정의
    int size; // 현재 리스트에 저장된 항목들의 개수
} ArrayListType;
```

연결 리스트 (Linked List)



구현 방법 (배열)

```
void error(char *message){
    fprintf(stderr, "%s\n", message);
    exit(1);
}

void init(ArrayListType *L){
    L->size = 0;
}

int is_empty(ArrayListType *L){
    return L->size == 0;
}

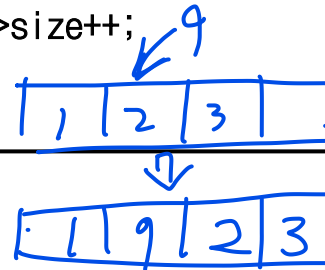
int is_full(ArrayListType *L){
    return L->size == MAX_LIST_SIZE;
}

element get_entry(ArrayListType *L, int pos){
    if (pos < 0 || pos >= L->size)
        error("위치 오류");
    return L->array[pos];
}

void print_list(ArrayListType *L){
    int i;
    for (i = 0; i < L->size; i++)
        printf("%d->", L->array[i]);
    printf("\n");
}
```

```
void insert_last(ArrayListType *L, element item){
    if( L->size >= MAX_LIST_SIZE ) {
        error("리스트 오버플로우");
    }
    L->array[L->size++] = item;
}

void insert(ArrayListType *L, int pos, element
item){
    if (!is_full(L) && (pos >= 0) && (pos <= L-
>size)) {
        for (int i = (L->size - 1); i >= pos; i--)
            L->array[i + 1] = L->array[i];
        L->array[pos] = item;
        L->size++;
    }
}
```



연결 리스트 (Linked List)



구현 방법 (배열)

```
element delete(ArrayListType *L, int pos){
    element item;

    if (pos < 0 || pos >= L->size)
        error("위치 오류");
    item = L->array[pos];
    for (int i = pos; i < (L->size - 1); i++)
        L->array[i] = L->array[i + 1];
    L->size--;
    return item;
}
```

```
int main(void){
    ArrayListType list;

    init(&list);
    insert(&list, 0, 10);
    print_list(&list); // 0번째 위치에 10 추가
    insert(&list, 0, 20);
    print_list(&list); // 0번째 위치에 20 추가
    insert(&list, 0, 30);
    print_list(&list); // 0번째 위치에 30 추가
    insert_last(&list, 40);
    print_list(&list); // 맨 끝에 40 추가
    delete(&list, 0);
    print_list(&list); // 0번째 항목 삭제
    return 0;
}
```

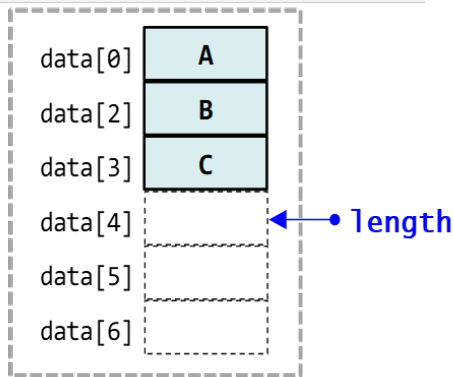

연결 리스트 (Linked List)



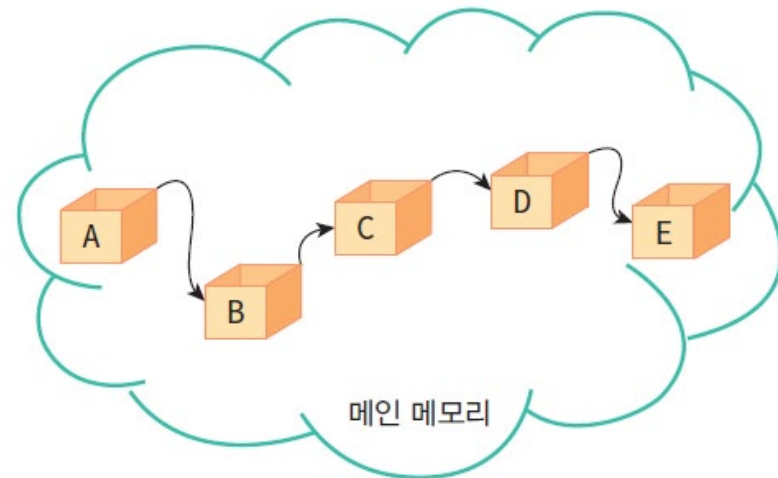
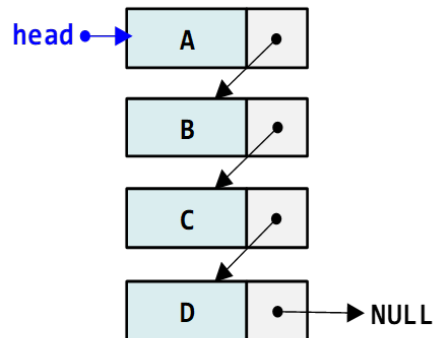
연결 리스트

- 단순 연결 리스트 (simply linked list)
 - 하나의 링크 필드를 이용하여 연결
 - 마지막 노드 링크 값은 NULL
- 데이터 필드 - 리스트 원소(데이터 값 저장)
- 링크 필드 - 다른 노드 주소 값을 저장 (포인터)

```
Element data[MAX_LIST_SIZE]  
int length;
```



```
Node* head;
```

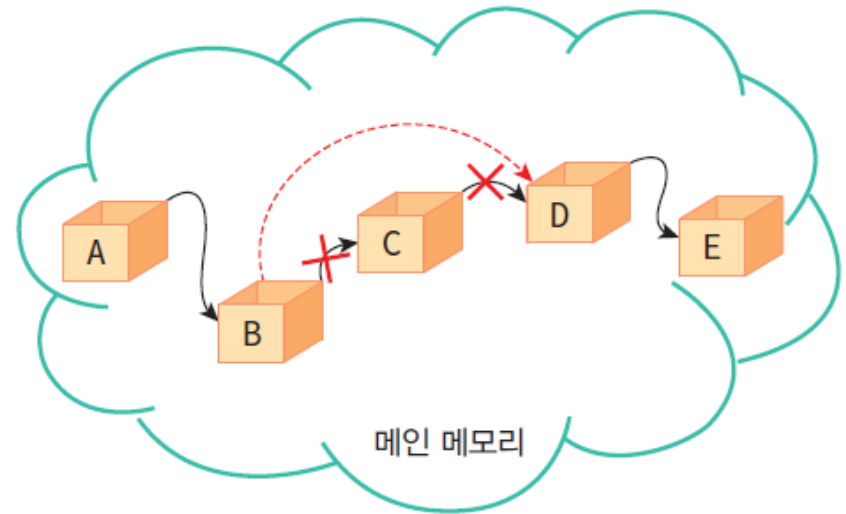
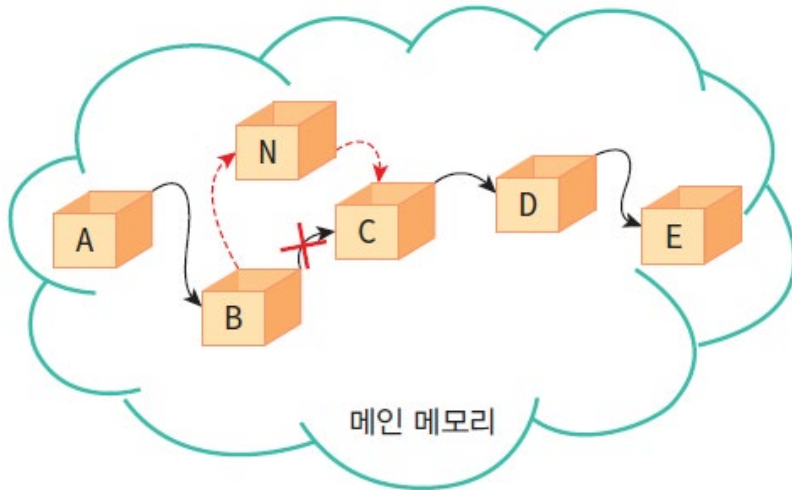


연결 리스트 (Linked List)



연결 리스트

- 삽입과 삭제



- 장점

- 삽입, 삭제 용이
- 연속된 메모리 공간이 필요 없음
- 크기 제한 없음

- 단점

- 구현 어려움
- 오류 발생 쉬움

연결 리스트 (Linked List)



구현 방법 (연결 리스트)

```
typedef int element;
```

```
typedef struct ListNode {  
    element data;  
    struct ListNode *link;  
} ListNode;
```

```
-----  
ListNode *head = NULL;
```

```
head = (ListNode *)malloc(sizeof(ListNode));
```

1번째
메모리

```
head->data = 10;  
head->link = NULL;
```

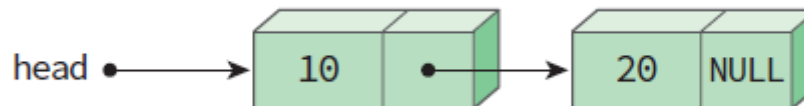
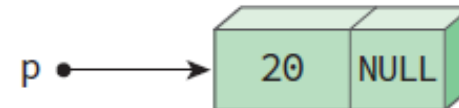
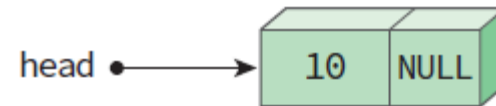
```
-----  
ListNode *p;
```

```
p = (ListNode *)malloc(sizeof(ListNode));
```

2번째
메모리

```
p->data = 20;  
p->link = NULL;
```

```
-----  
head->link = p;
```



연결 리스트 (Linked List)



구현 방법 (연결 리스트)

```
#include <stdio.h>
#include <stdlib.h>
typedef int element;
typedef struct ListNode {
    element data;
    struct ListNode* link;
} ListNode;
void error(char* message){
    fprintf(stderr, "%s\n", message);
    exit(1);
}
ListNode* insert_first(ListNode* head, int value){
    ListNode* p = (ListNode*)malloc(sizeof(ListNode));
    p->data = value;
    p->link = head;
    head = p;
    return head;
}
ListNode* insert(ListNode* head, ListNode* pre, element value){
    ListNode* p = (ListNode*)malloc(sizeof(ListNode));
    p->data = value;
    p->link = pre->link;
    pre->link = p;
    return head;
}
```

연결 리스트 (Linked List)



구현 방법 (연결 리스트)

```
ListNode* delete_first(ListNode* head){
    ListNode* removed;
    if (head == NULL)
        return NULL;
    removed = head; // (1)
    head = removed->link; // (2)
    free(removed); // (3)
    return head; // (4)
}

ListNode* delete(ListNode* head, ListNode* pre){
    ListNode* removed;
    removed = pre->link;
    pre->link = removed->link; // (2)
    free(removed); // (3)
    return head; // (4)
}

void print_list(ListNode* head){
    for (ListNode* p = head; p != NULL; p = p->link)
        printf("%d->", p->data);
    printf("NULL\n");
}
```

```
int main(void){
    ListNode* head = NULL;

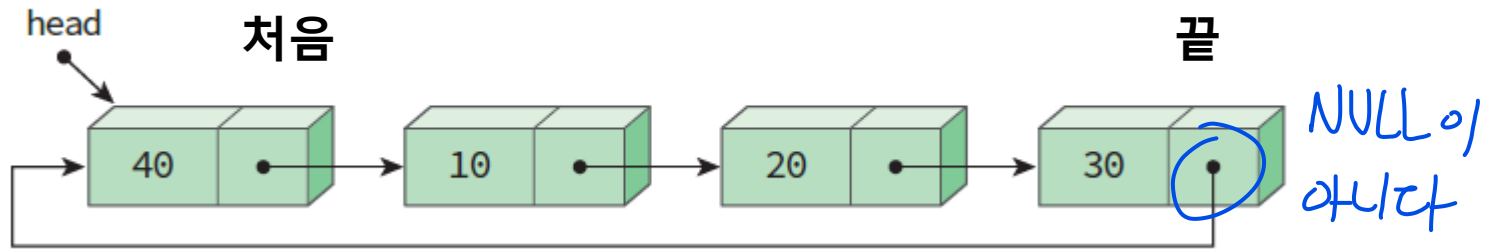
    for (int i = 0; i < 5; i++) {
        head = insert_first(head, i);
        print_list(head);
    }
    for (int i = 0; i < 5; i++) {
        head = delete_first(head);
        print_list(head);
    }
    return 0;
}
```

연결 리스트 (Linked List)

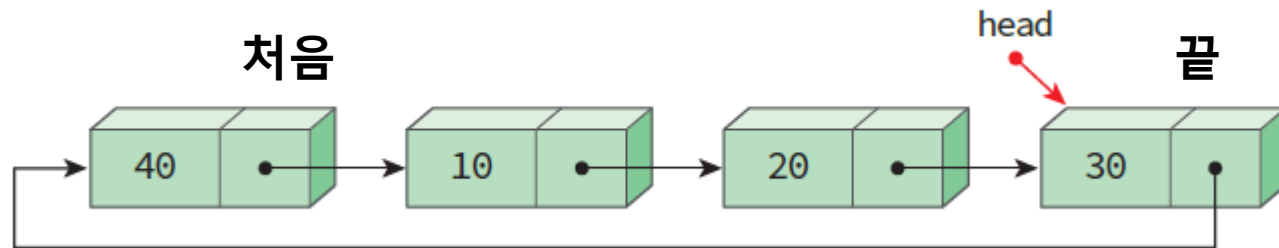


원형 연결 리스트

- 마지막 노드의 링크가 첫 번째 노드를 가리키는 리스트
- 한 노드에서 다른 모든 노드로 접근 가능



- 처음이나 마지막 노드를 삽입하는 연산이 단순 연결 리스트에 비하여 용이 (head를 끝에 위치)

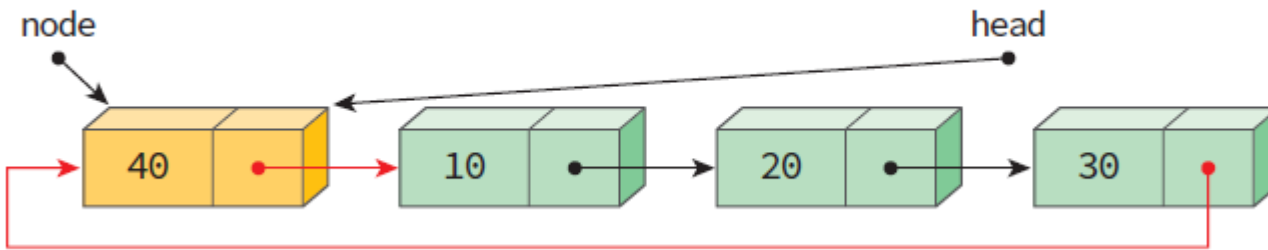
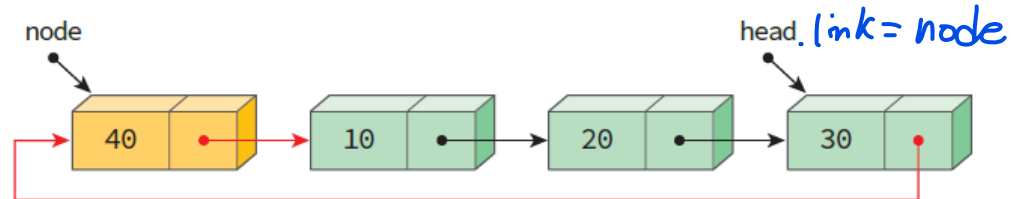
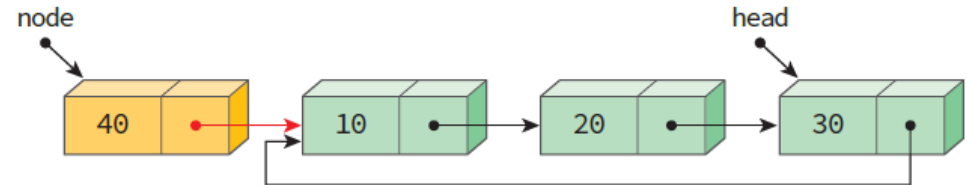
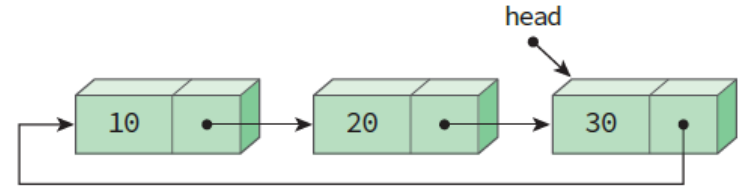


연결 리스트 (Linked List)



원형 연결 리스트

- 첫 번째 삽입
- 마지막 삽입



연결 리스트 (Linked List)



구현 방법 (원형 연결 리스트)

```
#include <stdio.h>
#include <stdlib.h>

typedef int element;
typedef struct ListNode {
    element data;
    struct ListNode* link;
} ListNode;

void print_list(ListNode* head){
    ListNode* p;

    if (head == NULL) return;
    p = head->link;
    do {
        printf("%d->", p->data);
        p = p->link;
    } while (p != head);
    printf("%d->", p->data);
}
```

```
ListNode* insert_first(ListNode* head, element data){
    ListNode* node = (ListNode*)malloc(sizeof(ListNode));
    node->data = data;
    if (head == NULL) {
        head = node;    node->link = head;
    }
    else {
        node->link = head->link;
        head->link = node;
    }
    return head;
}

ListNode* insert_last(ListNode* head, element data){
    ListNode* node = (ListNode*)malloc(sizeof(ListNode));
    node->data = data;
    if (head == NULL) {
        head = node;    node->link = head;
    }
    else {
        node->link = head->link;
        head->link = node;
        head = node;
    }
    return head;
}
```


연결 리스트 (Linked List)



구현 방법 (원형 연결 리스트)

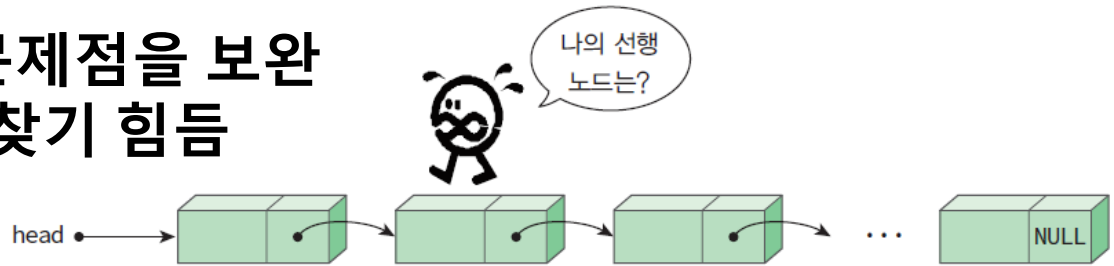
```
int main(void){
    ListNode* head = NULL;
    // list = 10->20->30->40
    head = insert_last(head, 20);
    head = insert_last(head, 30);
    head = insert_last(head, 40);
    head = insert_first(head, 10);
    print_list(head);
    return 0;
}
```

연결 리스트 (Linked List)

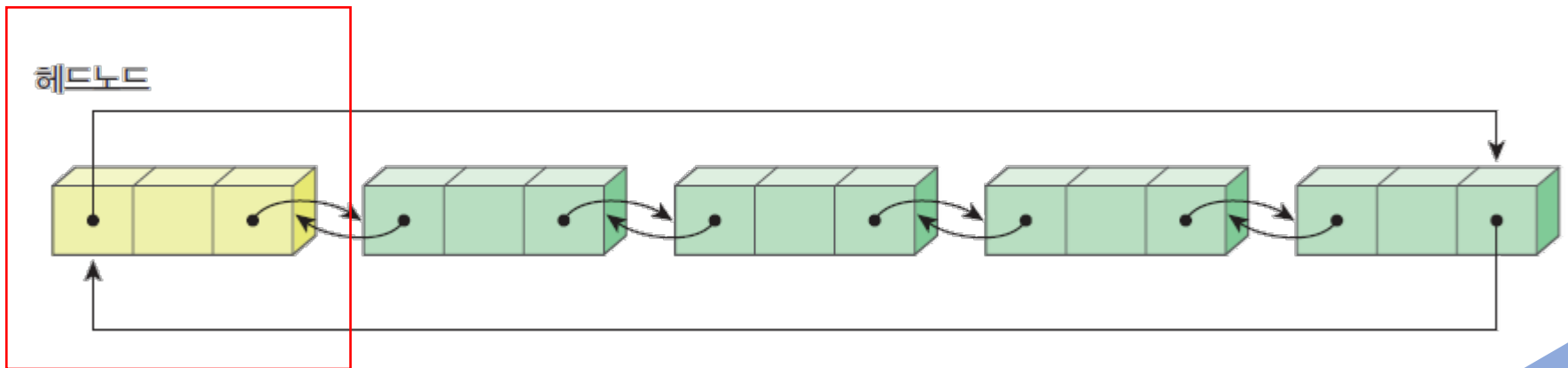


이중 연결 리스트

- 단순 연결 리스트의 문제점을 보완
- 문제점? 선행 노드를 찾기 힘들



- 하나의 노드가 선행 노드와 후속 노드에 대한 두 개의 링크를 가지는 리스트
- 단점: 공간을 많이 차지하고 코드가 복잡



데이터를 가지지 않고 단지 삽입, 삭제 코드를 간단하게 할 목적으로 만들어진 노드

연결 리스트 (Linked List)



구현 방법 (이중 연결 리스트)

```
#include <stdio.h>
#include <stdlib.h>
```

```
typedef int element;
typedef struct DListNode {
    element data;
    struct DListNode* llink;
    struct DListNode* rlink;
} DListNode;
```

```
void init(DListNode* phead){
    phead->llink = phead;
    phead->rlink = phead;
}
```

```
void print_dlist(DListNode* phead){
    DListNode* p;
    for (p = phead->rlink; p != phead; p = p->rlink) {
        printf("<-| %d| |-> ", p->data);
    }
    printf("\n");
}
```

```
void dinsert(DListNode* before, element data){
    DListNode* newnode =
        (DListNode*)malloc(sizeof(DListNode));
    strcpy(newnode->data, data);
    newnode->llink = before;
    newnode->rlink = before->rlink;
    before->rlink->llink = newnode;
    before->rlink = newnode;
}
```

```
void ddelete(DListNode* head, DListNode* removed){
    if (removed == head)
        return;
    removed->llink->rlink = removed->rlink;
    removed->rlink->llink = removed->llink;
    free(removed);
}
```

연결 리스트 (Linked List)



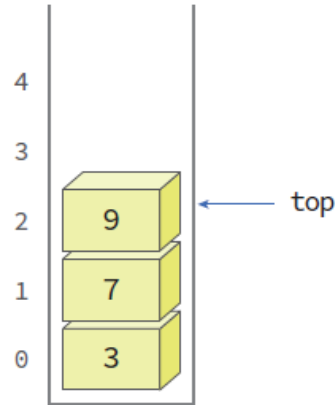
구현 방법 (이중 연결 리스트)

```
int main(void){
    DListNode* head = (DListNode*)malloc(sizeof(DListNode));
    init(head);
    printf("추가 단계\n");
    for (int i = 0; i < 5; i++) {
        dinsert(head, i);
        print_dlist(head);
    }
    printf("\n삭제 단계\n");
    for (int i = 0; i < 5; i++) {
        print_dlist(head);
        ddelete(head, head->rlink);
    }
    free(head);
    return 0;
}
```

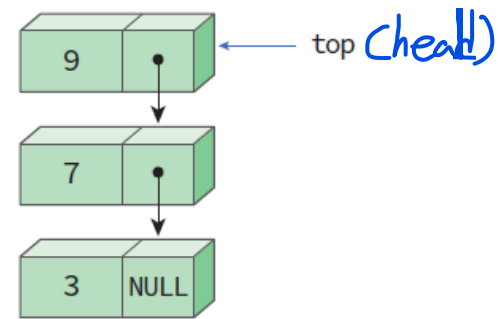
연결 리스트 (Linked List)



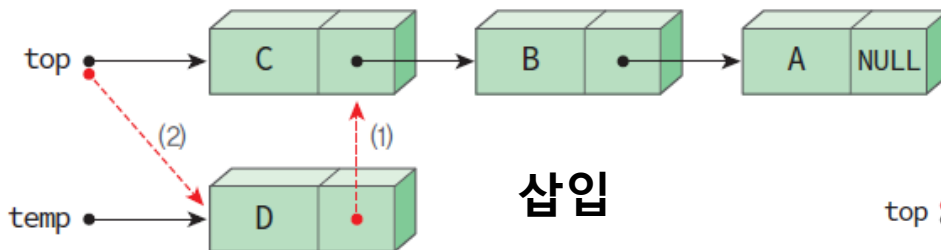
연결 리스트로 구현한 스택 / 큐



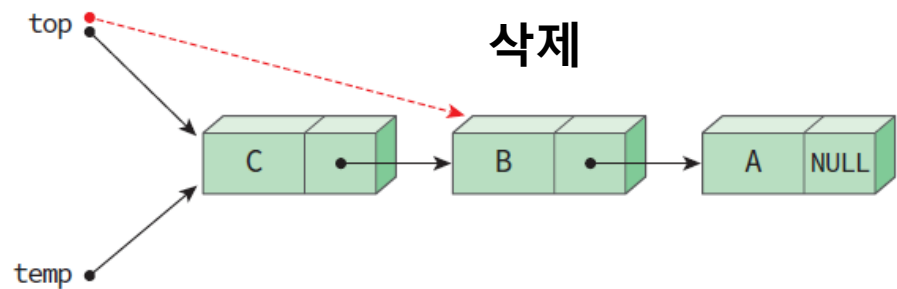
배열을 이용한 스택



연결 리스트를 이용한 스택



삽입

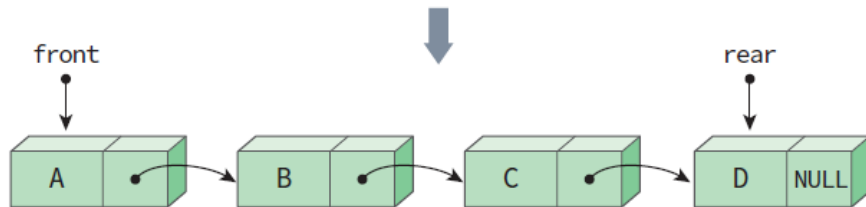
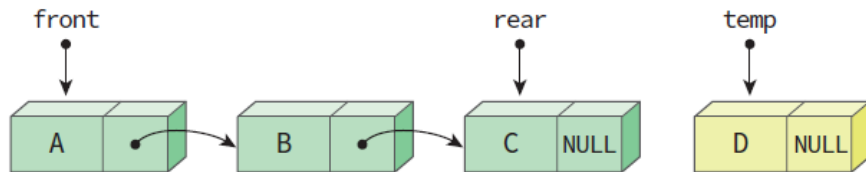
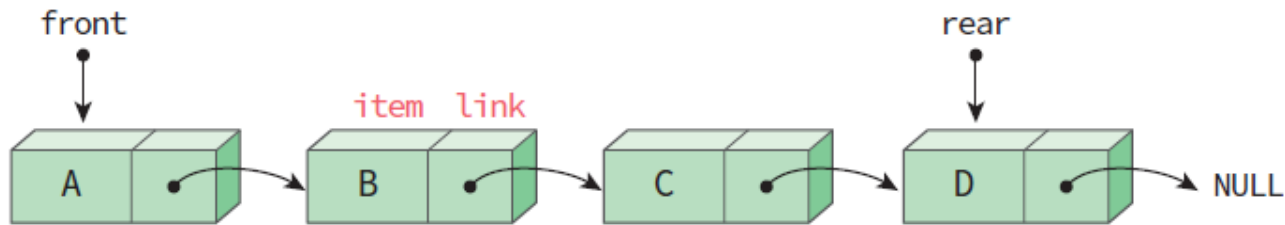


삭제

연결 리스트 (Linked List)

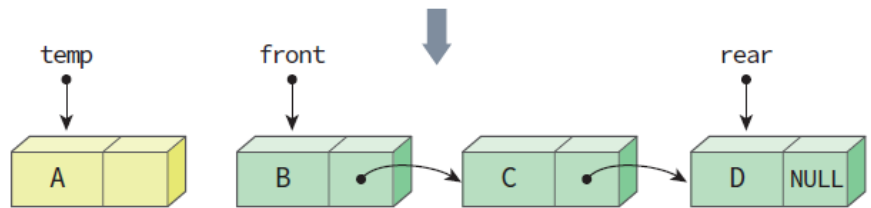
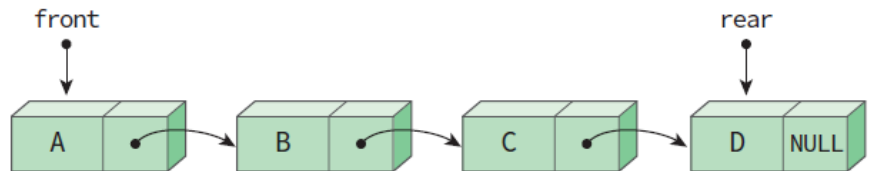


연결 리스트로 구현한 스택 / 큐



삽입

삭제

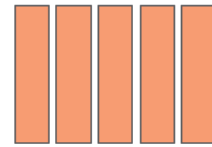


트리 (TREE)



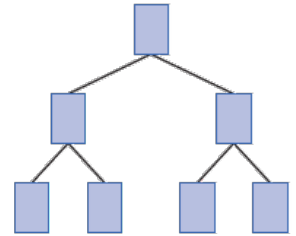
트리란?

- 리스트, 스택, 큐 등은 선형 구조
- 트리: 계층적인 구조를 나타내는 자료구조
- 부모-자식 관계의 노드들로 이루어짐

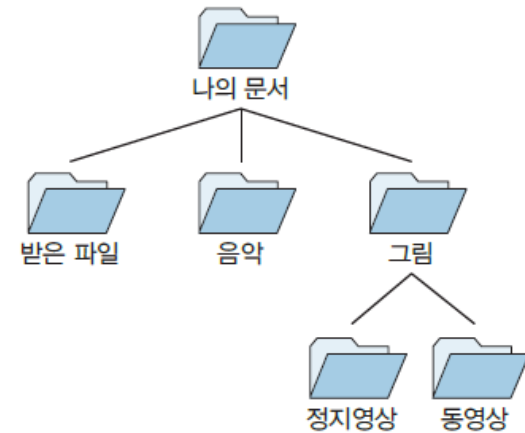
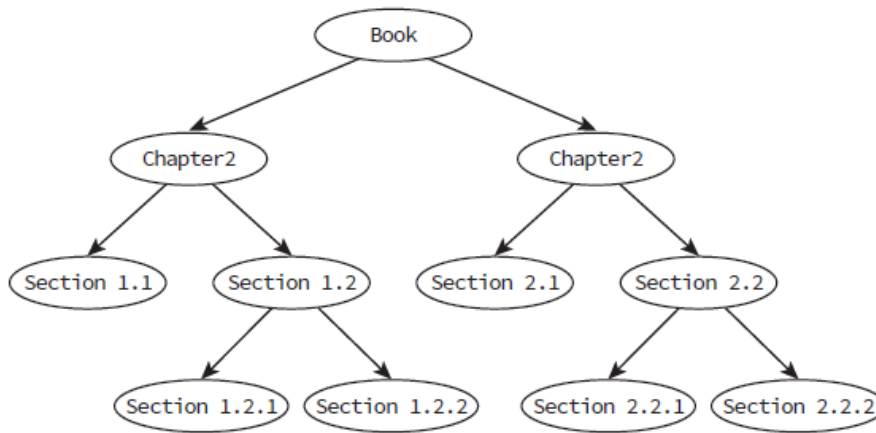


선형 자료구조

트리



비선형 자료구조

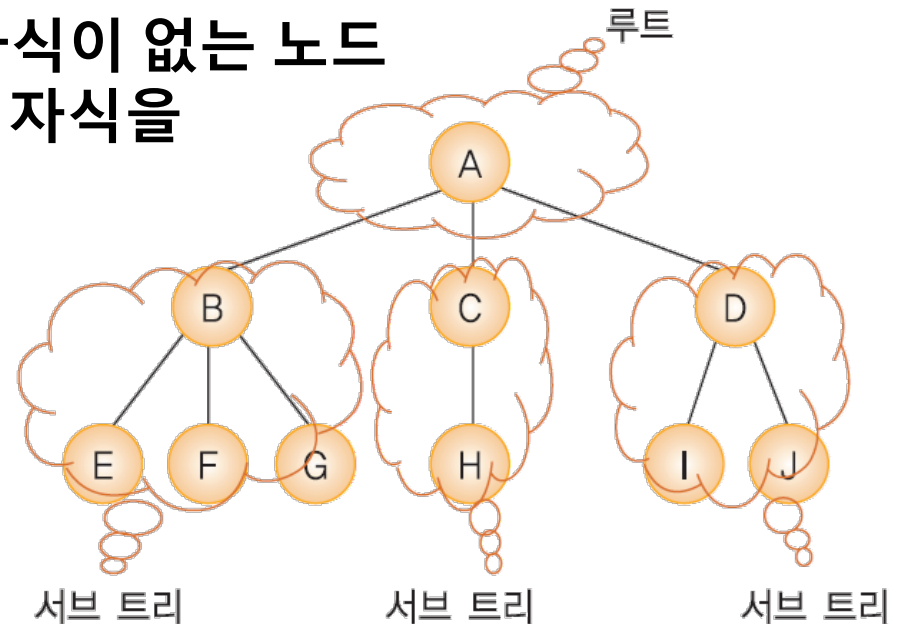


트리 (TREE)



트리에서 사용되는 용어

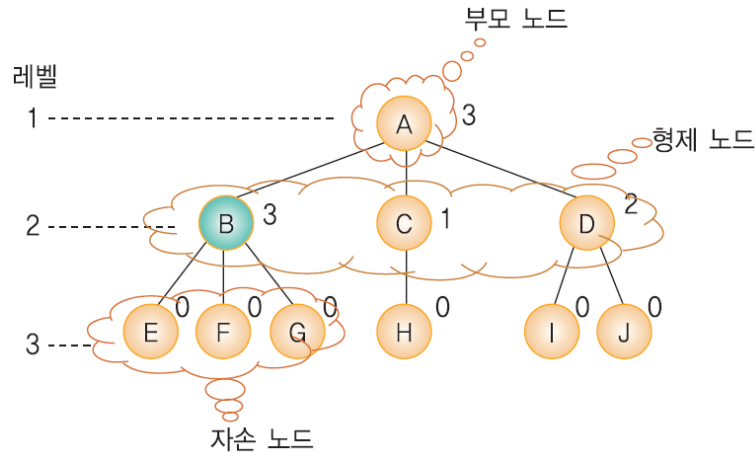
- 노드(node): 트리의 구성요소
- 루트(root): 부모가 없는 노드
- 서브트리(subtree): 하나의 노드와 그 노드들의 자손들로 이루어진 트리
- 단말노드(terminal node): 자식이 없는 노드
- 비단말노드: 적어도 하나의 자식을 가지는 노드



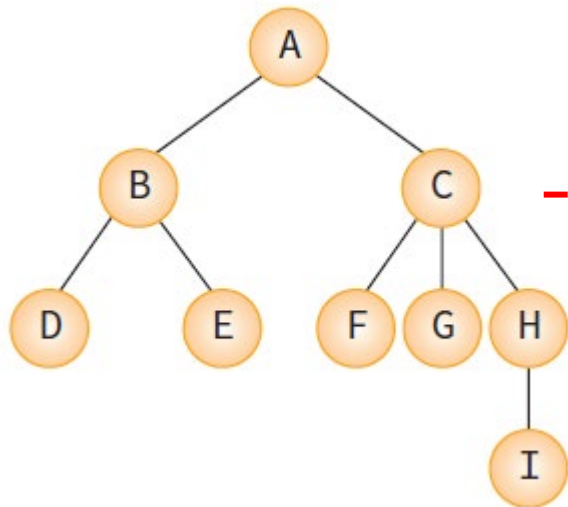
트리 (TREE)



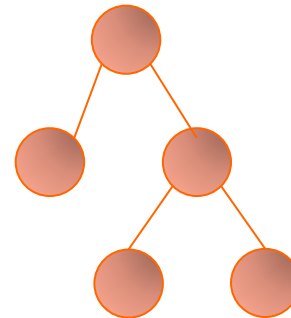
트리에서 사용되는 용어



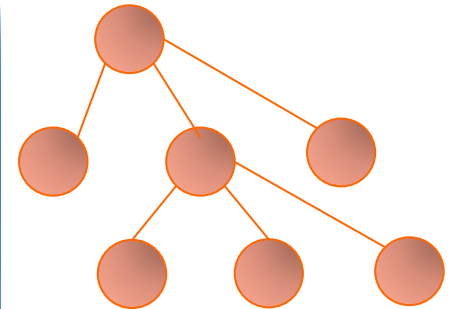
- 자식, 부모, 형제, 조상, 자손 노드
- 레벨: 트리의 각층 번호
- 높이: 트리의 최대 레벨
- 차수: 노드가 가지고 있는 자식 노드의 개수



--- ???



이진 트리
자식노드가 최대 2개



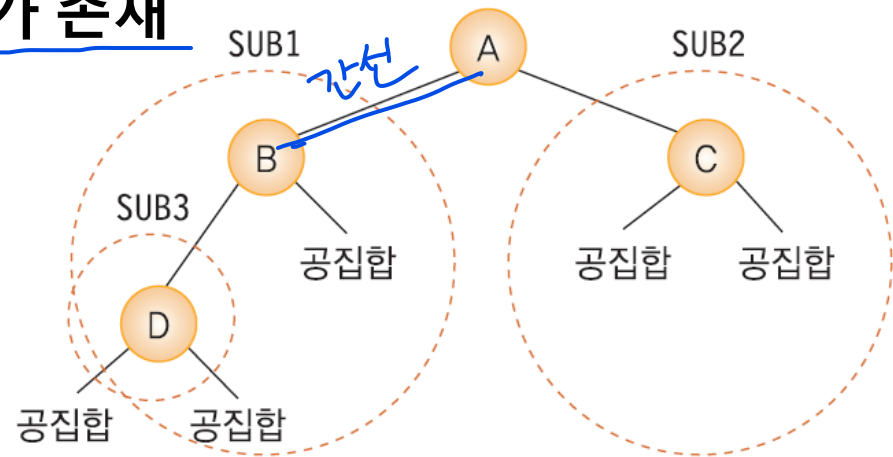
일반 트리
" 3개 이상

트리 (TREE)



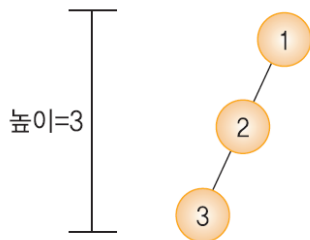
이진 트리

- 모든 노드가 2개의 서브 트리를 가지고 있는 트리
 - 최대 2개까지의 자식 노드가 존재
 - 모든 노드의 차수가 2 이하
 - 서브 트리간 순서 존재

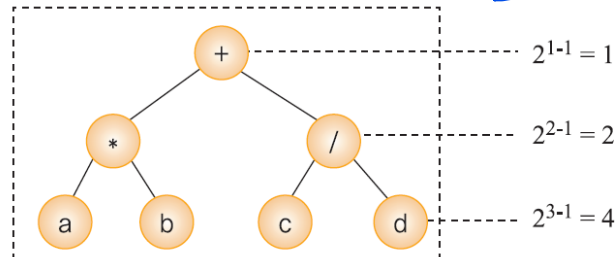


- 노드의 개수가 n 이면
간선의 개수는 $n-1$

- 높이가 x 인 이진 트리의 경우
최소 x 개의 노드를 가짐 and 최대 2^{x-1} 개의 노드를 가짐



최소 노드 개수 = 3



최대 노드 개수 = $2^{1-1} + 2^{2-1} + 2^{3-1} = 1 + 2 + 4 = 7$

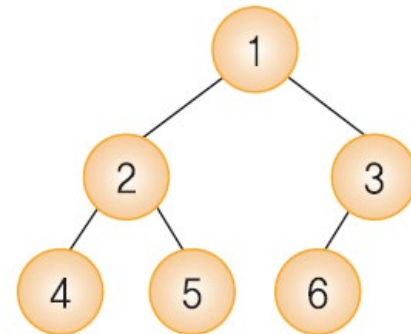
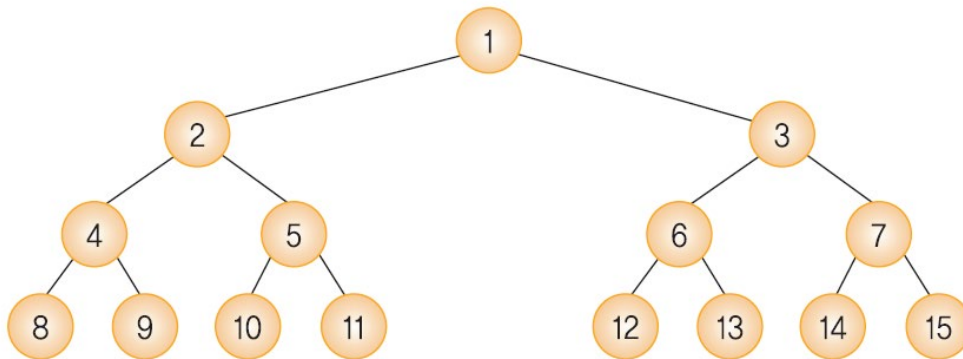
일반화

트리 (TREE)



이진 트리

- 포화, 완전, 기타 세 가지로 분류
- 포화: 각 레벨 노드가 꽉 차 있는 이진 트리
- 완전: 마지막 레벨에서 왼쪽부터 오른쪽으로 노드가 순서대로 채워져 있는 이진 트리
- 포화와 완전이 아닌 이진 트리



이진 트리 표현은 “배열”과 “포인터” 로 구현 가능

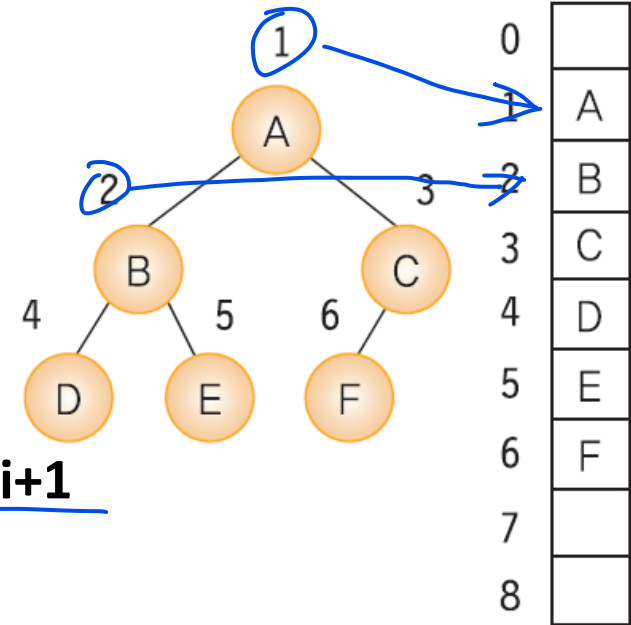
트리 (TREE)



배열 / 링크 표현법

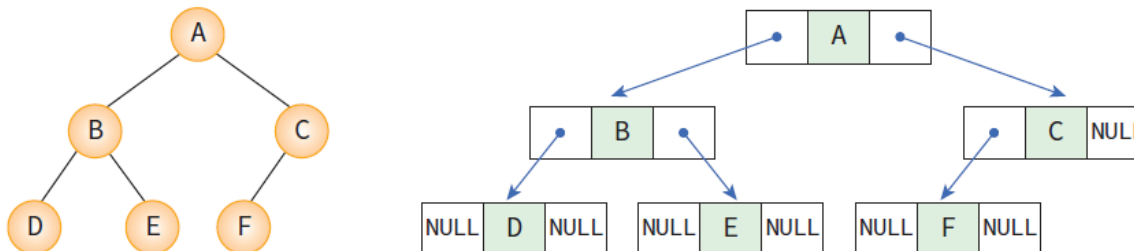
- 배열

- 완전 이진 트리라 가정
- 각 노드에 번호를 붙여 인덱스로 삼아 배열에 저장
- 노드 i 의 부모 노드 인덱스 = $i/2$
- 노드 i 의 왼쪽 자식 노드 인덱스 = $2i$
- 노드 i 의 오른쪽 자식 노드 인덱스 = $2i+1$



- 링크 (연결리스트)

- 포인터를 이용하여 부모/자식 노드를 가리키게 하는 방법

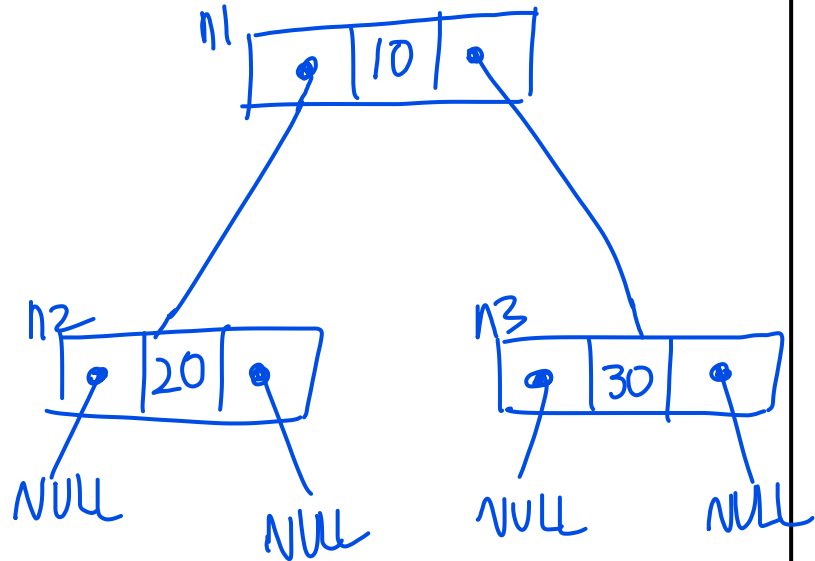


트리 (TREE)



간단 구현 방법 (트리 - 링크)

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
typedef struct TreeNode {
    int data;
    struct TreeNode* left, * right;
} TreeNode;
int main(void){
    TreeNode* n1, * n2, * n3;
    n1 = (TreeNode*)malloc(sizeof(TreeNode));
    n2 = (TreeNode*)malloc(sizeof(TreeNode));
    n3 = (TreeNode*)malloc(sizeof(TreeNode));
    n1->data = 10; // 첫 번째 노드를 설정한다.
    n1->left = n2;
    n1->right = n3;
    n2->data = 20; // 두 번째 노드를 설정한다.
    n2->left = NULL;
    n2->right = NULL;
    n3->data = 30; // 세 번째 노드를 설정한다.
    n3->left = NULL;
    n3->right = NULL;
    free(n1); free(n2); free(n3);
    return 0;
}
```






트리 (TREE)



순회 방법

- 순회: 트리 노드들을 체계적으로 방문하는 방법

- 전위순회(VLR): 자손 노드보다 루트 노드를 먼저 방문 
- 중위순회(LVR): 왼쪽 자손, 루트, 오른쪽 자손 순으로 방문 
- 후위순회(LRV): 루트 노드보다 자손을 먼저 방문 

트리 (TREE)



순회 방법

- 전위순회(VLR): 자손 노드보다 루트 노드를 먼저 방문

preorder(x)



if $x \neq \text{NULL}$

```
then print DATA(x);  
preorder(LEFT(x));  
preorder(RIGHT(x));
```

// x가 NULL이 아닐 때만 처리

// ① 루트(x) 노드 처리

// ② 왼쪽 서브트리 방문

// ③ 오른쪽 서브트리 방문



(이진트리라서)

트리 (TREE)



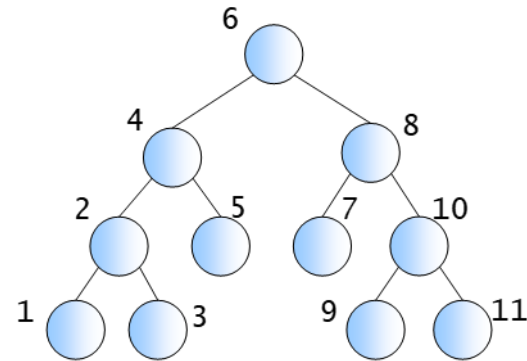
순회 방법

- 중위순회(LVR): 왼쪽 자손, 루트, 오른쪽 자손 순으로 방문

inorder(x)

if $x \neq \text{NULL}$

```
[ then inorder(LEFT(x));  
    print DATA(x);  
    inorder(RIGHT(x));
```

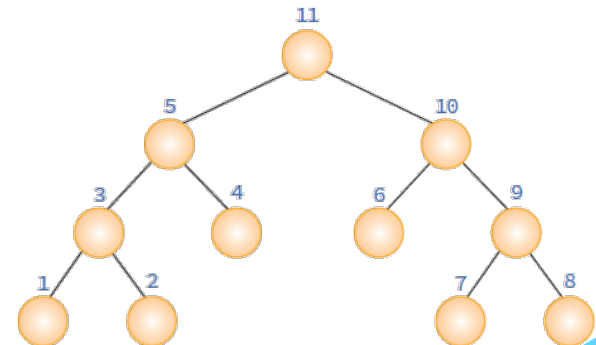


- 후위순회(LRV): 루트 노드보다 자손을 먼저 방문

postorder(x)

if $x \neq \text{NULL}$

```
[ then postorder(LEFT(x));  
    postorder(RIGHT(x));  
    print DATA(x);
```



트리 (TREE)

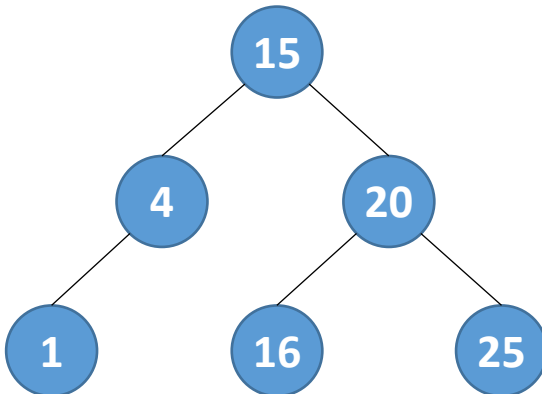


구현 방법 (이진 트리/순회 방법)

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>

typedef struct TreeNode {
    int data;
    struct TreeNode* left, * right;
} TreeNode;

TreeNode n1 = { 1, NULL, NULL };
TreeNode n2 = { 4, &n1, NULL };
TreeNode n3 = { 16, NULL, NULL };
TreeNode n4 = { 25, NULL, NULL };
TreeNode n5 = { 20, &n3, &n4 };
TreeNode n6 = { 15, &n2, &n5 };
TreeNode* root = &n6;
```



```
// 중위 순회
void inorder(TreeNode* root) {
    if (root) {
        inorder(root->left);
        printf("[%d] ", root->data);
        inorder(root->right);
    }
}
```

재귀함수

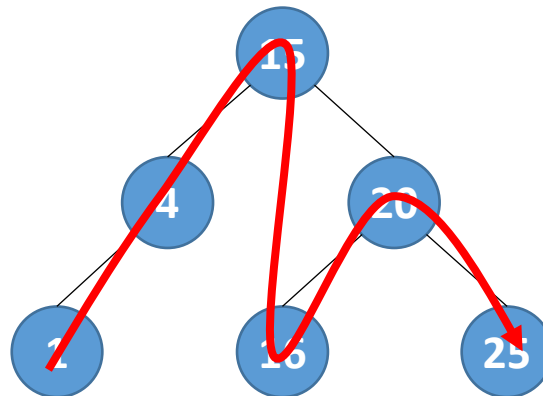
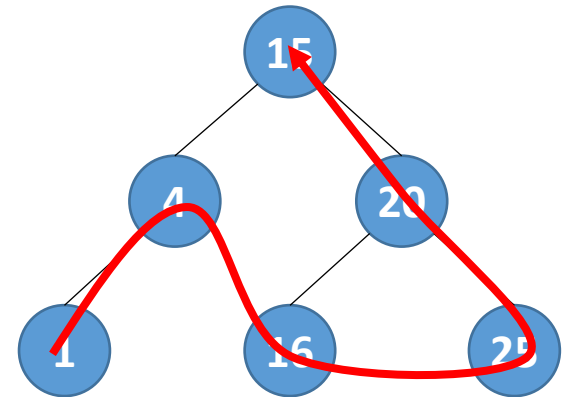
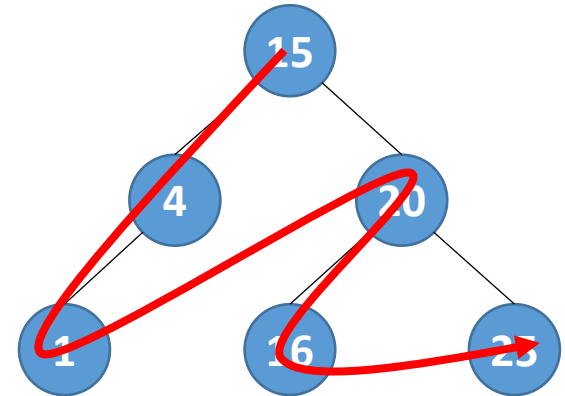
```
// 전위 순회
void preorder(TreeNode* root) {
    if (root != NULL) {
        printf("[%d] ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
```

```
// 후위 순회
void postorder(TreeNode* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("[%d] ", root->data);
    }
}
```

트리 (TREE)

구현 방법 (이진 트리/순회 방법)

```
int main(void){  
    printf("중위 순회=");  
    inorder(root);  
    printf("\n");  
  
    printf("전위 순회=");  
    preorder(root);  
    printf("\n");  
  
    printf("후위 순회=");  
    postorder(root);  
    printf("\n");  
    return 0;  
}
```



트리 (TREE)



반복적 순회

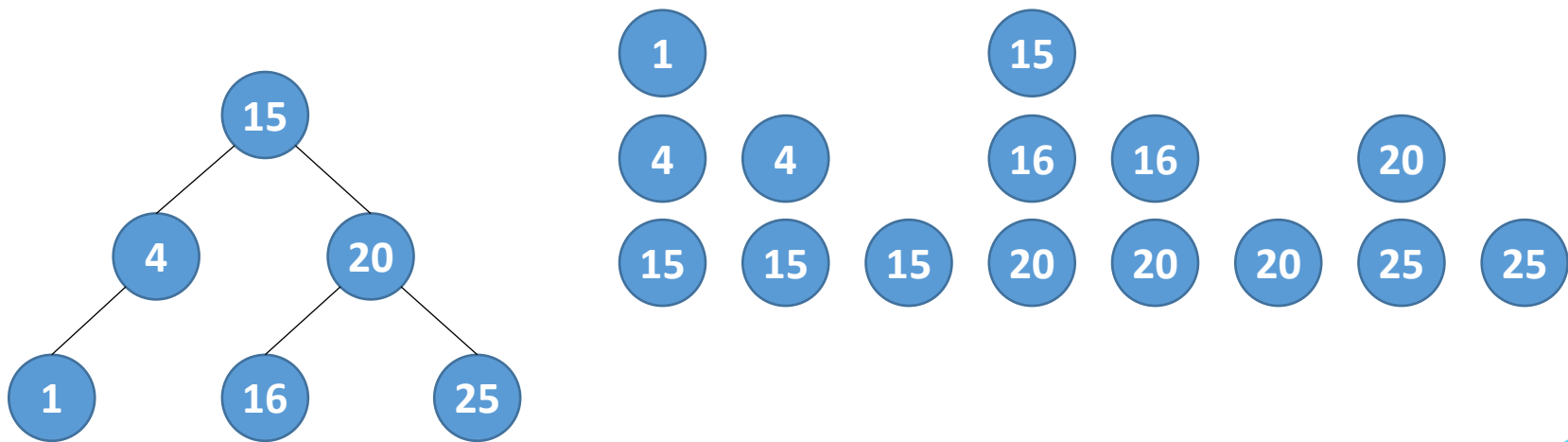
- 순환 호출의 순회가 표준적인 방법, 쉽게 이해

재귀

- > 반복을 이용해서도 트리 순회 가능 (스택 필요)

for

- 스택에 자식 노드들을 저장하고 꺼내면서 순회
(순환 호출 또한 시스템 스택을 사용)



트리 (TREE)



구현 방법 (이진 탐색 트리)

```
#include <stdio.h>
#include <stdlib.h>
typedef int element;
typedef struct TreeNode {
    element key;
    struct TreeNode *left, *right;
} TreeNode;

TreeNode * new_node(int item){
    TreeNode * temp = (TreeNode
*)malloc(sizeof(TreeNode));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

TreeNode * insert_node(TreeNode * node, int key){
    if (node == NULL)
        return new_node(key);
    if (key < node->key)
        node->left = insert_node(node->left, key);
    else if (key > node->key)
        node->right = insert_node(node->right, key);
    return node;
}
```

```
// 중위 순회
void inorder(TreeNode* root) {
    if (root) {
        inorder(root->left);
        printf("[%d] ", root->key);
        inorder(root->right);
    }
}

// 전위 순회
void preorder(TreeNode* root) {
    if (root != NULL) {
        printf("[%d] ", root->key);
        preorder(root->left);
        preorder(root->right);
    }
}

// 후위 순회
void postorder(TreeNode* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("[%d] ", root->key);
    }
}
```

트리 (TREE)



구현 방법 (이진 탐색 트리)

```
TreeNode* delete_node(TreeNode* root, int key){
    if (root == NULL)
        return root;
    if (key < root->key)
        root->left = delete_node(root->left, key);
    else if (key > root->key)
        root->right = delete_node(root->right, key);
    else {
        if (root->left == NULL) {
            TreeNode* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            TreeNode* temp = root->left;
            free(root);
            return temp;
        }
        TreeNode* temp = min_value_node(root->right);
        root->key = temp->key;
        root->right = delete_node(root->right, temp->key);
    }
    return root;
}
```

```
int main(void){
    TreeNode* root = NULL;
    TreeNode* tmp = NULL;

    root = insert_node(root, 30);
    root = insert_node(root, 20);
    root = insert_node(root, 10);
    root = insert_node(root, 40);
    root = insert_node(root, 50);
    root = insert_node(root, 60);

    //////////////////////////////////////
    printf("이진 탐색 트리 중위 순회 결과 \n");
    inorder(root);
    printf("\n\n");
    if (search(root, 30) != NULL)
        printf("이진 탐색 트리에서 30을 발견함 \n");
    else
        printf("이진 탐색 트리에서 30을 발견못함 \n");
    return 0;
}
```