**<기본적인 자료구조들>**

1) 선형 구조 - 배열 - 연결 리스트 - 스택 - 큐

2) 비선형 구조 - 트리(Tree) - 그래프(Graph)

**시간복잡도: 알고리즘에 사용되는 연산 횟수**

**공간복잡도: 알고리즘에 사용되는 메모리 양**

효율적인 알고리즘을 사용한다고 가정했을 때 일반적으로 시간과 공간은 반비례 관계

**<시간 복잡도>**

**시간 복잡도는 Big-O 표기법을 사용함**

최악의 경우를 나타냄

* **O(𝑛) 시간 복잡도**

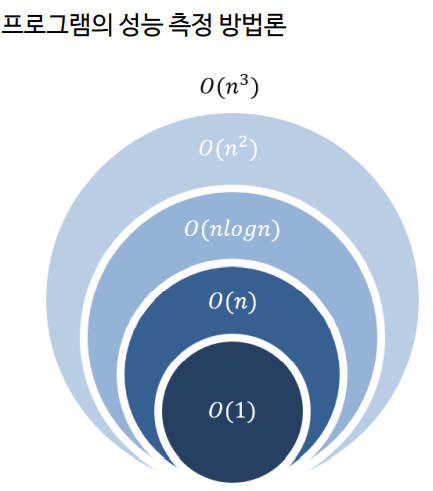
텍스트이(가) 표시된 사진

자동 생성된 설명

**O(n\*\*2)의 시간 복잡도 -> 이중 for문**

텍스트이(가) 표시된 사진

자동 생성된 설명

 텍스트이(가) 표시된 사진

자동 생성된 설명

왠만하면 10억번 이상이 아니어야함.

시간 복잡도를 표기할 때는 항상 큰 항과 계수만 표

𝑂(3𝑛^2 + 𝑛) = 𝑂(𝑛^2 )

현실의 다양한 문제에서는 시간 제한이 1초정도

<공간 복잡도>

공간 복잡도를 표기할 때는 일반적으로 MB 단위로 표기함.

텍스트이(가) 표시된 사진

자동 생성된 설명

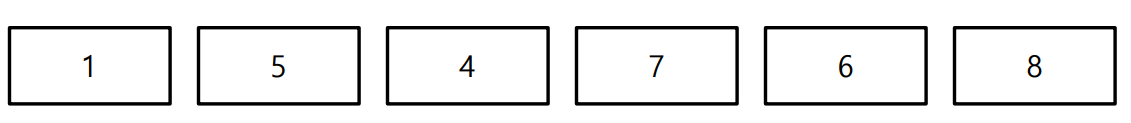
<연결 리스트>

연결 리스트의 필요성

1) 일반적으로 배열을 사용하여 데이터를 순차적으로 저장하고, 나열할 수 있음.

2) 배열을 사용하는 경우 메모리 공간이 불필요하게 낭비될 수 있음.

<배열 기반의 리스트>



1) 배열로 만들었으므로 특정한 위치의 원소에 즉시 접근할 수 있다는 장점이 있습니다.

2) 데이터가 들어갈 공간을 미리 메모리에 할당해야 한다는 단점이 있습니다.

3) 원하는 위치로의 삽입이나 삭제가 비효율적입니다.

연결 리스트

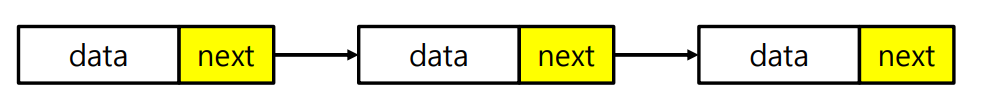
1) **일반적으로 연결 리스트는 구조체와 포인터를 함께 사용하여 구현**합니다.

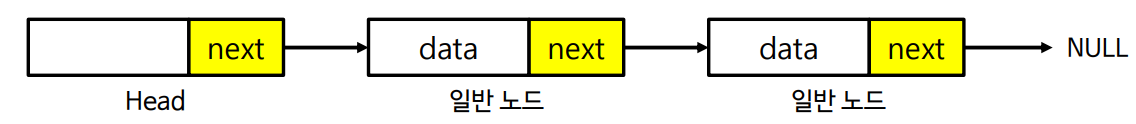
2) 연결 리스트는 **리스트의 중간 지점에 노드를 추가하거나 삭제할 수 있어야 합니다.**

3) **필요할 때마다 메모리 공간을 할당 받습니다. 동적 할당**

**<단일 연결리스트>**

포인터를 이용해 단방향적으로 다음 노드를 가리킴

****

****

1) 단일 연결 리스트는 다음과 같은 형태로 나타낼 수 있습니다.

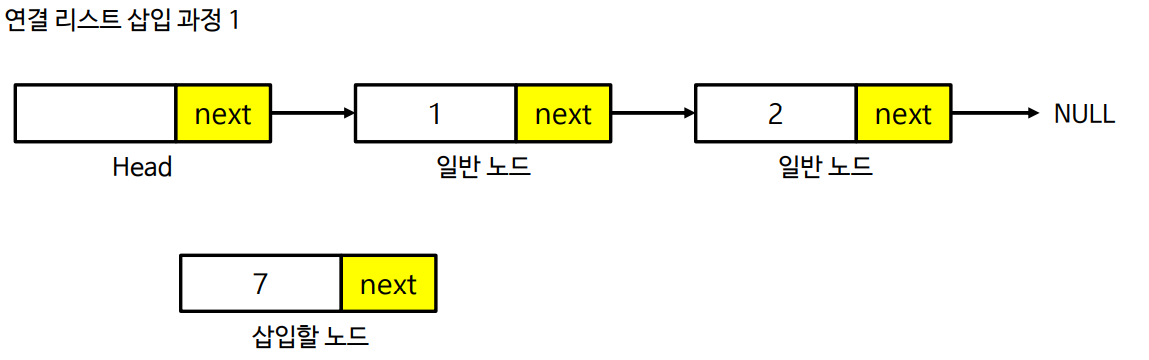
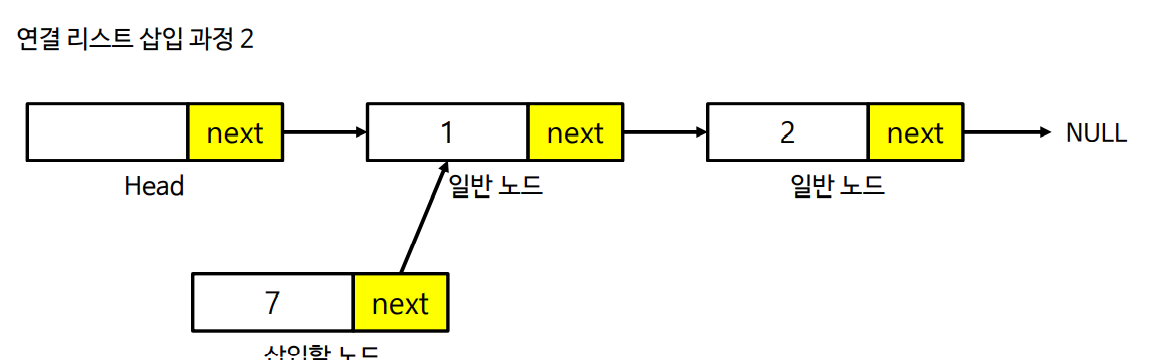
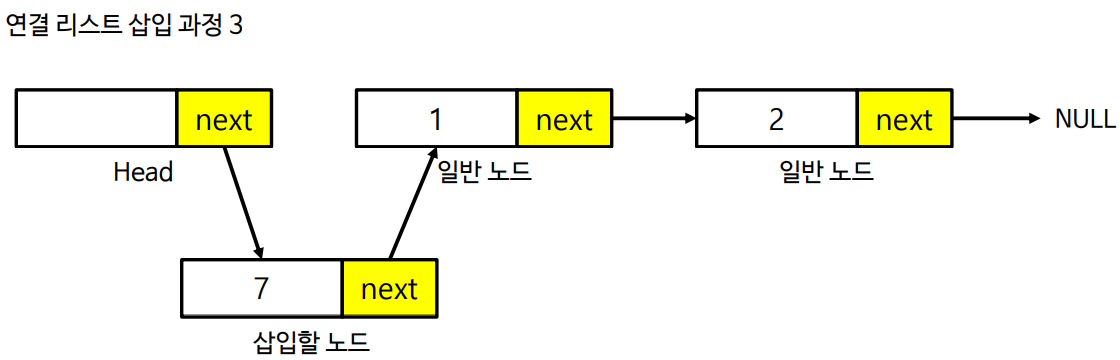
2) 포인터를 이용해 단방향적으로 다음 노드를 가리킵니다.

3) 일반적으로 연결 리스트의 **시작 노드를 헤드(Head)**라고 하며 별도로 관리합니다.

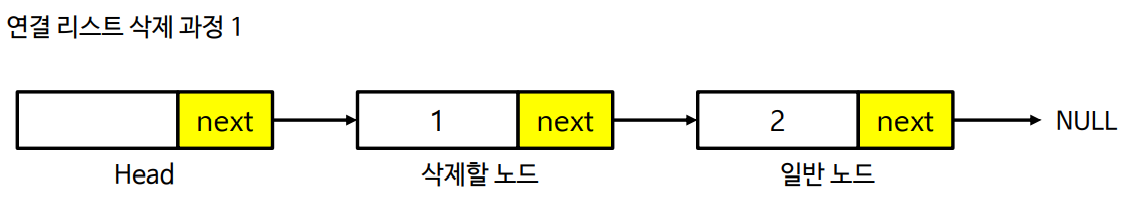
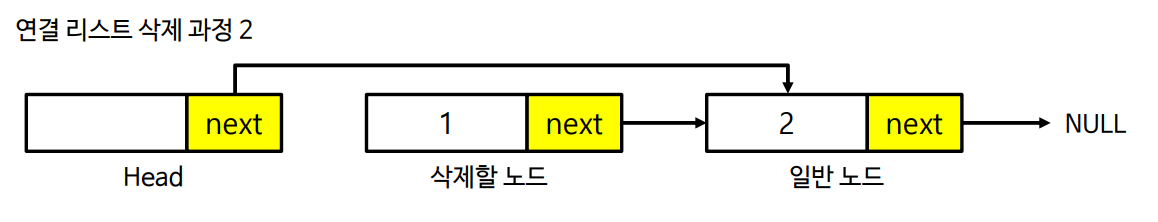
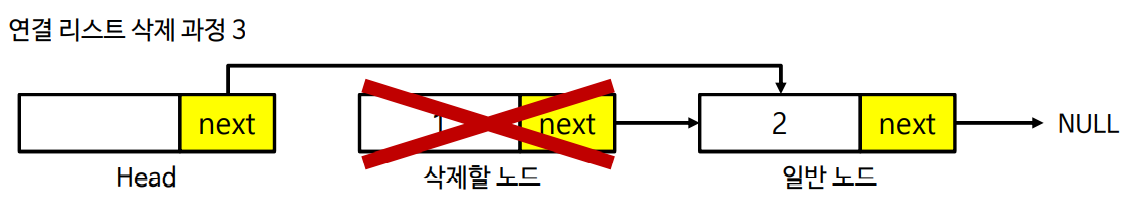
4) 다음 노드가 없는 **끝 노드의 다음 위치 값으로는 NULL을 넣습니다.**

단일 연결리스트에는 하나의 구조체 안에 두 개의 변수가 들어감.

하나는 데이터, 하나는 다음 데이터를 가르킴.

<삭제과정>

<연결 리스트 구현에 있어서 주의할 점>

1) 위 소스코드에 덧붙여서 삽입 및 삭제 기능에서의 예외 사항을 처리할 필요가 있음

2) 삭제할 원소가 없는데 삭제하는 경우, 머리(Head) 노드 자체를 잘못 넣은 경우 등을 체크

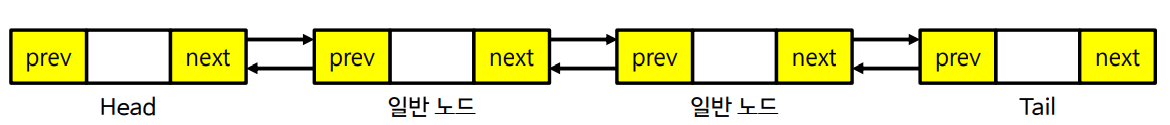
<특징>

1) 삽입과 삭제가 배열에 비해서 간단하다는 장점이 있음

2) 배열과 다르게 특정 인덱스로 즉시 접근하지 못하며, 원소를 차례대로 검색해야 함.

3) 추가적인 포인터 변수가 사용되므로 메모리 공간이 낭비됨.

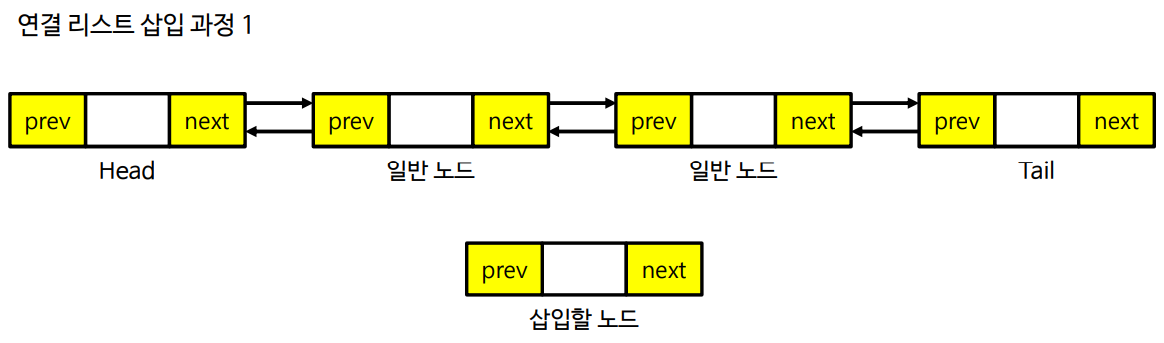
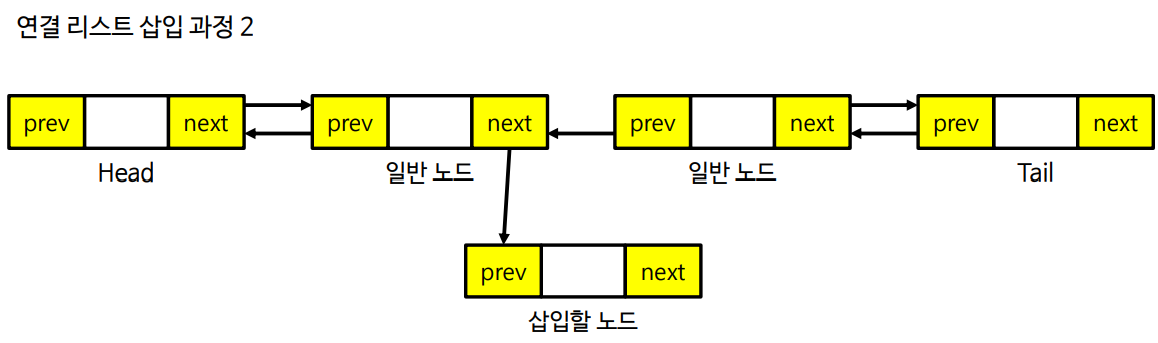
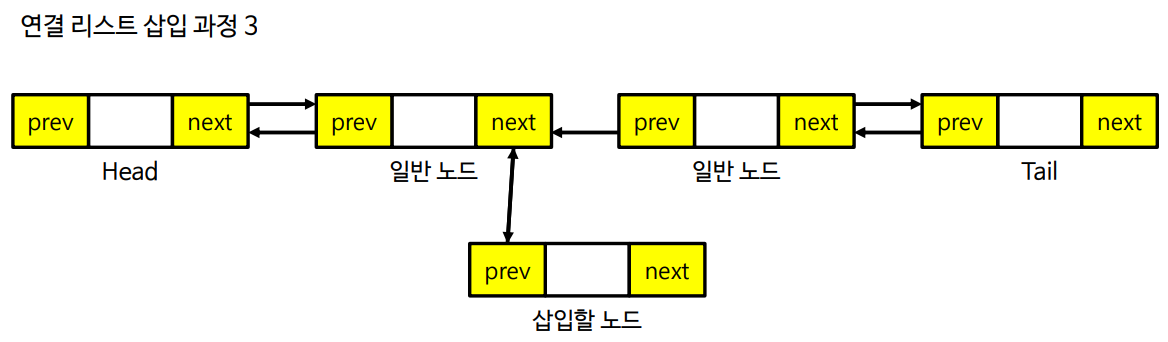
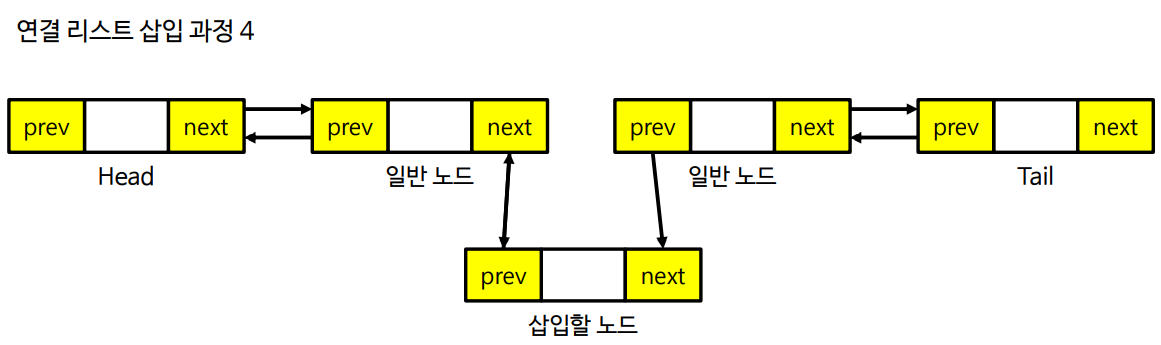
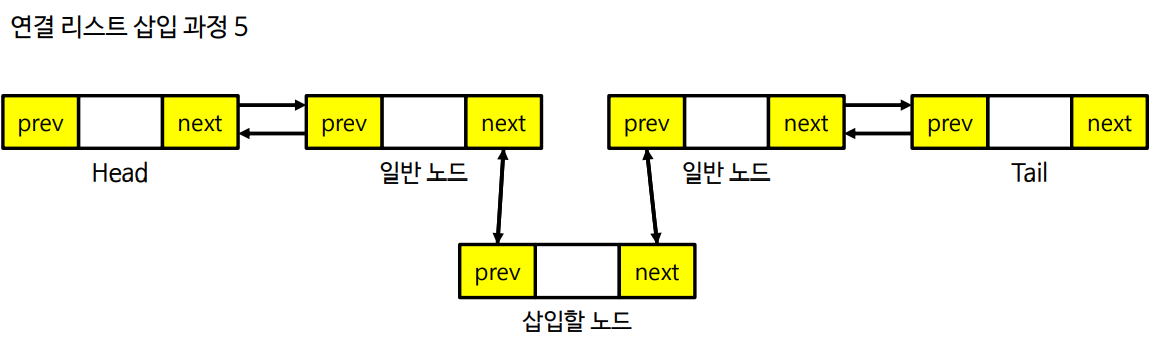
<양방향 연결 리스트>

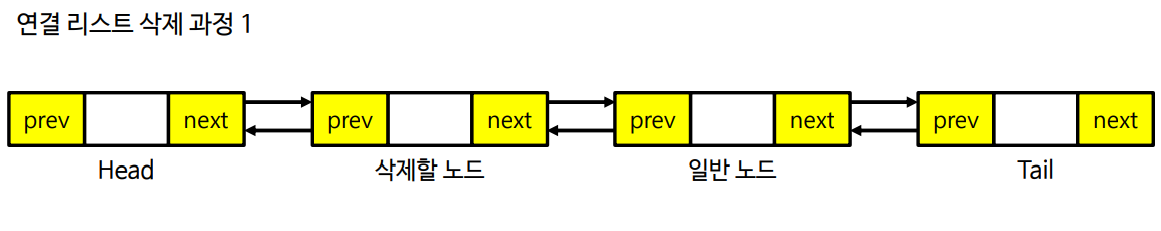
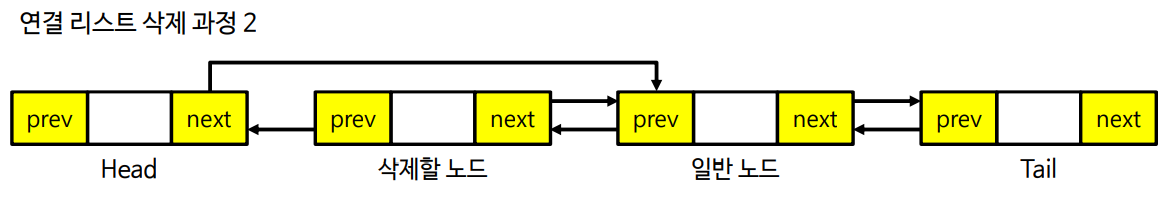
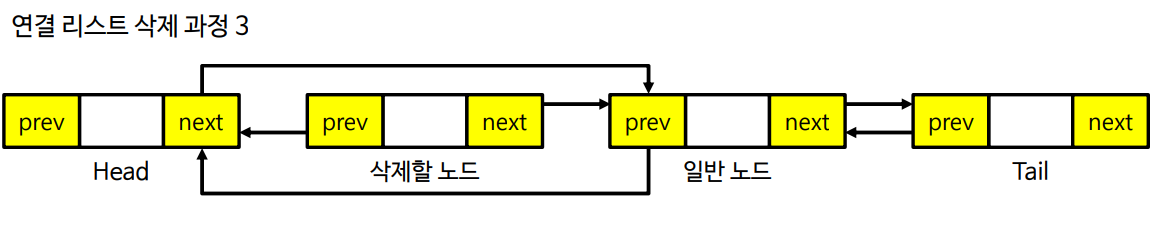
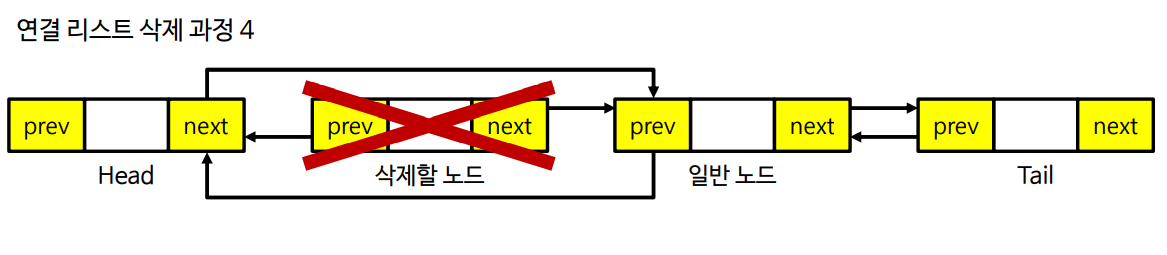


1) 양방향 연결 리스트는 머리(Head)와 꼬리(Tail)를 모두 가진다는 특징이 있음.

2) 양방향 연결 리스트의 각 노드는 앞 노드와 뒤 노드의 정보를 모두 저장

3) 데이터를 ‘오름차순’으로 저장하는 양방향 연결 리스트를 구현

**<양방향 연결 리스트 구현에 있어서 주의할 점>**

1) 위 소스코드에 덧붙여서 삽입 및 삭제 기능에서의 예외 사항을 처리할 필요가 있음

2) 더 이상 삭제할 원소가 없는데 삭제하는 경우 등을 체크 해야함.

1) 양방향 연결 리스트에서는 각 노드가 앞 노드와 뒤 노드의 정보를 저장하고 있음.

2) 양방향 연결 리스트를 이용하면 리스트의 앞에서부터 혹은 뒤에서부터 모두 접근할 수 있음.

<스택>

스택(Stack)은 한쪽으로 들어가서 한쪽으로 나오는 자료 구조(Data Structure)

특성 때문에 수식 계산 등의 알고리즘에서 다방면으로 활용

마치 배열처럼 활용 가능

- PUSH: 스택에 데이터를 넣습니다.

- POP: 스택에서 데이터를 빼냅니다

음악, 피아노이(가) 표시된 사진

자동 생성된 설명 테이블이(가) 표시된 사진

자동 생성된 설명 테이블이(가) 표시된 사진

자동 생성된 설명 테이블이(가) 표시된 사진

자동 생성된 설명 테이블이(가) 표시된 사진

자동 생성된 설명



스택(Stack)은 배열을 이용한 구현 방법과 연결 리스트를 이용한 구현 방법으로 나누어짐.

가장 기본적인 형태의 자료구조로 구현 난이도는 낮은 편.

<배열을 이용한 구현 방법>

#include <stdio.h>

#define SIZE 10000

#define INF 99999999

int stack[SIZE];

int top = -1;

void push(int data) {

    if (top == SIZE - 1) {

    printf("스택 오버플로우가 발생했습니다.\n");

    return;

    }

    stack[++top] = data;

}

int pop() {

    if (top == -1) {

        printf("스택 언더플로우가 발생했습니다.\n");

        return -INF;

        }

    return stack[top--];

}

void show() {

    printf("--- 스택의 최상단 ---\n");

    for (int i = top; i >= 0; i--) {

    printf("%d \n", stack[i]);

    }

    printf("--- 스택의 최하단 ---\n");

}

int main(void) {

    push(7);

    push(5);

    push(4);

    pop();

    push(6);

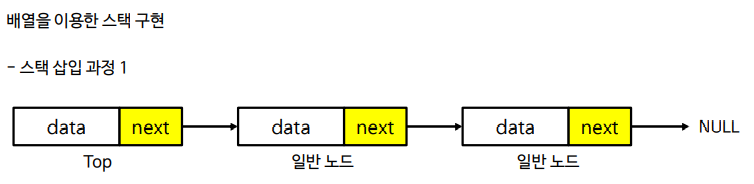
    pop();

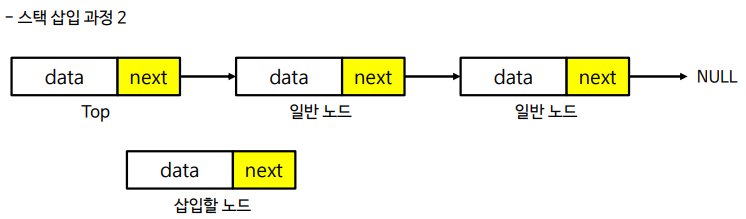
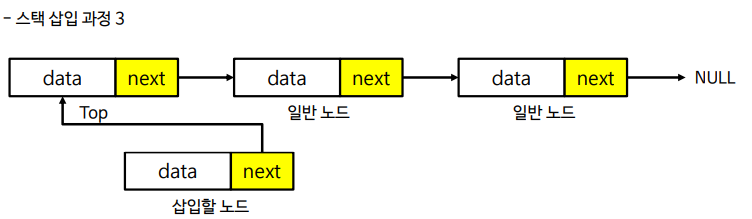
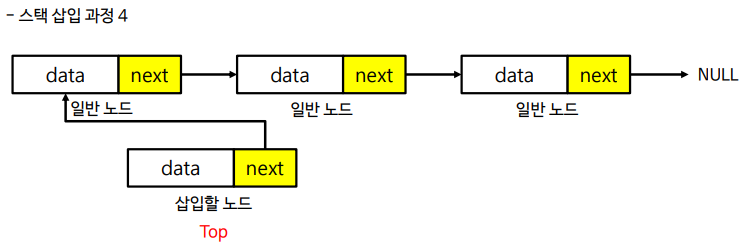
    show();

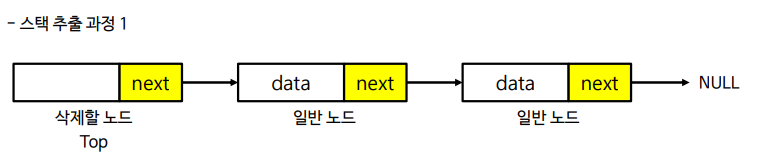
    return 0;

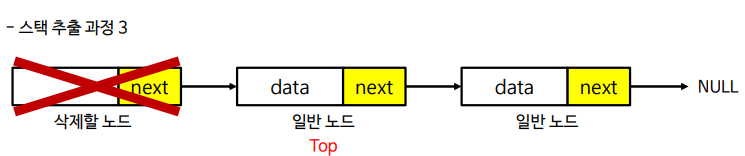
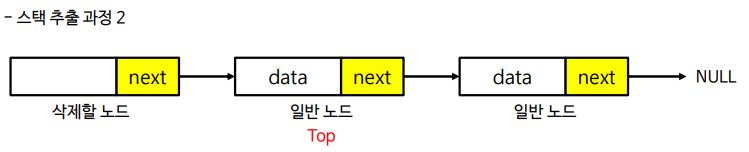
}

<연결리스트를 이용한 구현 방법>







#include <stdio.h>

#include <stdlib.h>

#define INF 99999999

typedef struct { //스택의 선언

    int data;

    struct Node \*next;

} Node;

typedef struct {

    Node \*top;

} Stack;

void push(Stack \*stack, int data) {

    Node \*node = (Node\*) malloc(sizeof(Node));

    node->data = data;

    node->next = stack->top;

    stack->top = node;

}

int pop(Stack \*stack) {

    if (stack->top == NULL) {

    printf("스택 언더플로우가 발생했습니다.\n");

    return -INF;

    }

    Node \*node = stack->top;

    int data = node->data;

    stack->top = node->next;

    free(node);

    return data;

}

void show(Stack \*stack) {

    Node \*cur = stack->top;

    printf("--- 스택의 최상단 ---\n");

    while (cur != NULL) {

        printf("%d\n", cur->data);

        cur = cur->next;

    }

    printf("--- 스택의 최하단 ---\n");

}

int main(void) {

    Stack stack;

    stack.top = NULL;

    show(&stack);

    push(&stack, 7);

    push(&stack, 5);

    push(&stack, 4);

    pop(&stack);

    push(&stack, 6);

    pop(&stack);

    show(&stack);

    system

    ("pause");

    return 0;

}

<스택을 활용한 계산기 만들기>

중위 표기법

1) 중위 표기법이란 일반적으로 사람이 수식을 표기할 때 사용하는 표기 방법입니다.

2) 중위 표기법의 예시: 7 \* 5 + 3

후위 표기법

1) 후위 표기법이란 컴퓨터가 계산하기에 편한 수식의 형태입니다.

2) 후위 표기법에서 연산자는 뒤쪽에 위치합니다.

3) 후위 표기법 예시: 7 5 \* 3

스택을 활용해 수식을 계산하는 방법은 다음과 같습니다.

① 수식을 후위 표기법으로 변환합니다.

중위 표기법을 후위 표기법으로 바꾸는 방법

1) 피연산자가 들어오면 바로 출력합니다.

2) 연산자가 들어오면 자기보다 우선순위가 높거나 같은 것들을 빼고 자신을 스택에 담습니다.

3) 여는 괄호 ‘(‘를 만나면 무조건 스택에 담습니다.

4) 닫는 괄호 ‘)’를 만나면 ‘(‘를 만날 때까지 스택에서 출력합니다.

② 후위 표기법을 계산하여 결과를 도출합니다.

후위 표기법을 계산하는 방법

1) 피연산자가 들어오면 스택에 담습니다.

2) 연산자를 만나면 스택에서 두 개의 연산자를 꺼내서 연산한 뒤에 그 결과를 스택에 담습니다.

3) 연산을 마친 뒤에 스택에 남아있는 하나의 피연산자가 연산 수행 결과입니다.

<큐>

1) 큐(Queue)는 뒤쪽으로 들어가서 앞쪽으로 나오는 자료 구조(Data Structure)

2) 이러한 특성 때문에 스케줄링, 탐색 알고리즘 등에서 다방면으로

- PUSH: 큐에 데이터를 넣습니다.

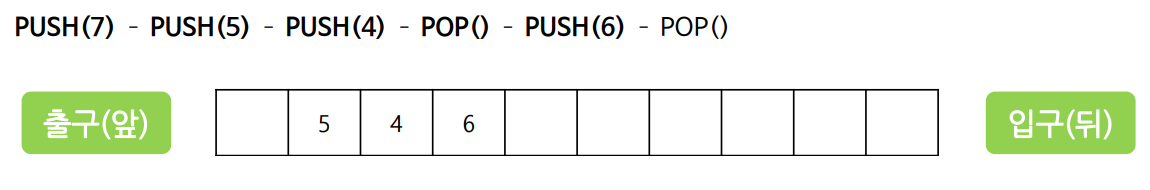
- POP: 큐에서 데이터를 빼냅니다.

텍스트, 쇼지이(가) 표시된 사진

자동 생성된 설명텍스트, 음악이(가) 표시된 사진

자동 생성된 설명테이블이(가) 표시된 사진

자동 생성된 설명음악이(가) 표시된 사진

자동 생성된 설명테이블이(가) 표시된 사진

자동 생성된 설명

큐(Queue)는 배열을 이용한 구현 방법과 연결 리스트를 이용한 구현 방법이 있음.

<배열을 이용한 큐 구현>

#include <stdio.h>

#define SIZE 10000

#define INF 99999999

int queue[SIZE];

int front = 0;

int rear = 0;

void push(int data) {

    if (rear >= SIZE) {

        printf("큐 오버플로우가 발생했습니다.\n");

    return;

    }

    queue[rear++] = data;

}

int pop() {

    if (front == rear) {

        printf("큐 언더플로우가 발생했습니다.\n");

    return -INF;

    }

    return queue[front++];

}

void show() {

    printf("--- 큐의 앞 ---\n");

    for (int i = front; i < rear; i++) {

        printf("%d\n", queue[i]);

    }

    printf("--- 큐의 뒤 ---\n");

}

int main(void) {

    push(7);

    push(5);

    push(4);

    pop();

    push(6);

    pop();

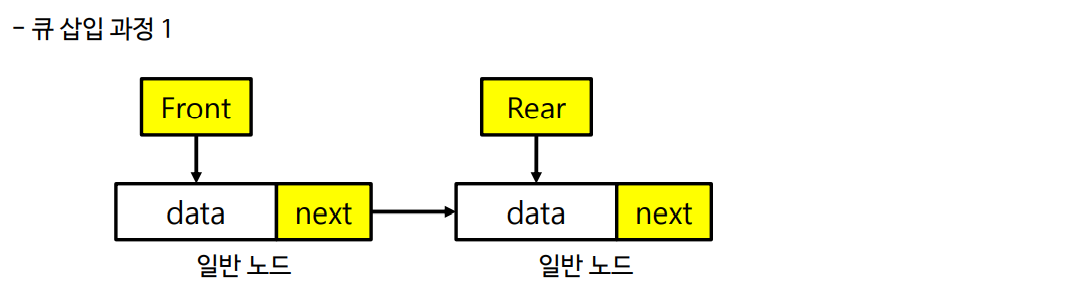
    show();

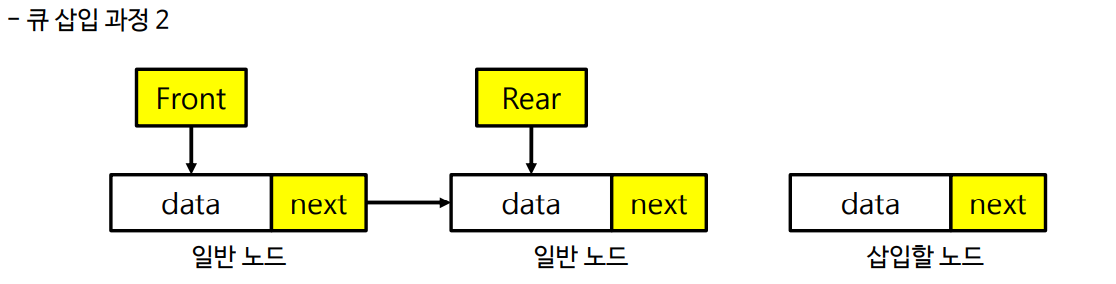
    system("pause");

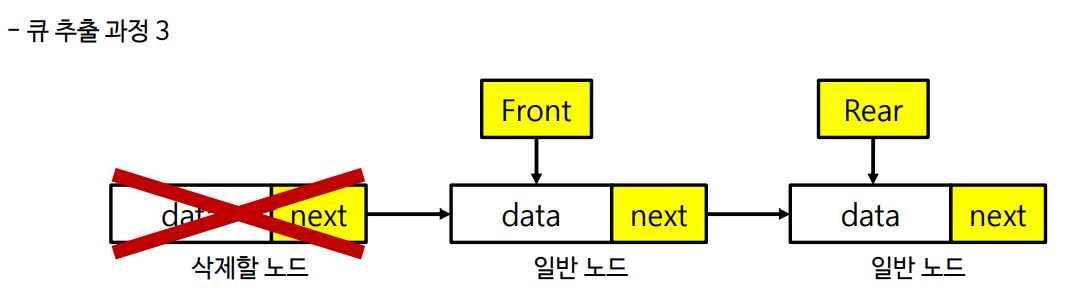
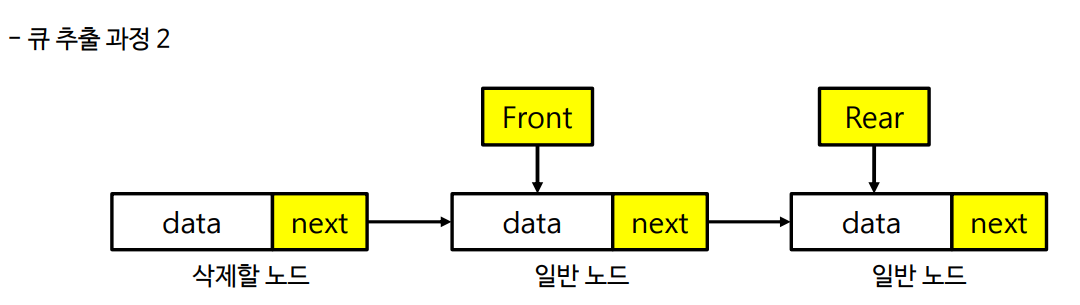
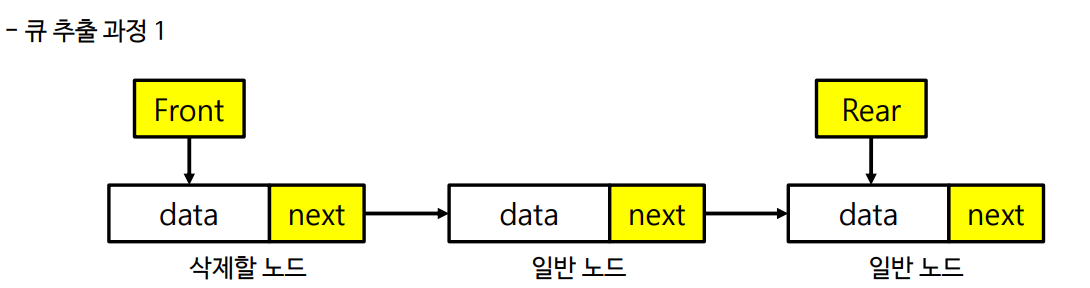
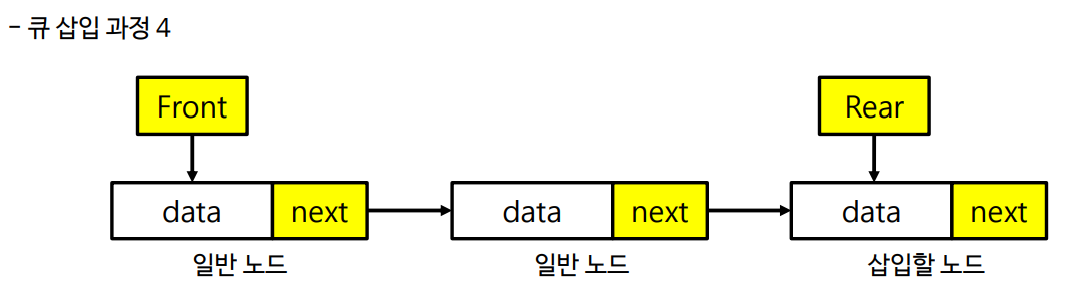
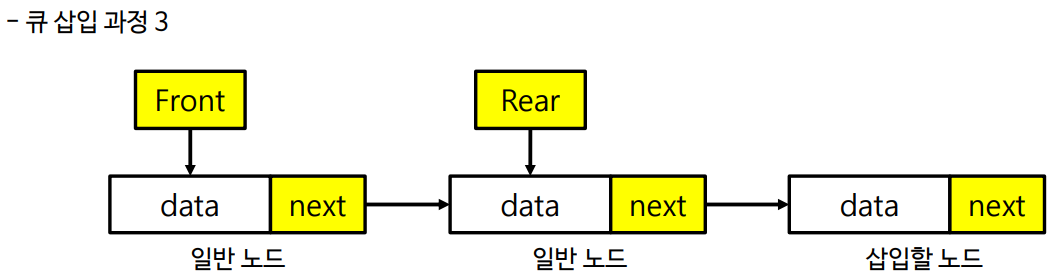
    return 0;

}

<연결 리스트를 이용한 큐 구현>







#include <stdio.h>

#include <stdlib.h>

#define INF 99999999

typedef struct{

    int data;

    struct Node \*next;

} Node

;

typedef struct{

    Node \*front;

    Node \*rear;

    int count;

} Queue;

void push(Queue \*queue, int data) {

    Node \*node = (Node\*)malloc(sizeof(Node));

    node->data = data;

    node->next = NULL;

    if (queue->count == 0) {

        queue->front = node;

    }

    else {

        queue->rear->next = node;

    }

    queue->rear = node;

    queue->count++;

}

int pop(Queue \*queue) {

    if (queue->count == 0) {

        printf("큐 언더플로우가 발생했습니다.\n");

        return -INF;

    }

    Node \*node = queue->front;

    int data = node->data;

    queue->front = node->next;

    free(node);

    queue->count--;

    return data;

}

void show(Queue \*queue) {

    Node \*cur = queue->front;

    printf("--- 큐의 앞 ---\n");

    while (cur != NULL) {

        printf("%d\n", cur->data);

        cur = cur->next;

    }

    printf("--- 큐의 뒤 ---\n");

}

int main(void) {

    Queue queue;

    queue.front = queue.rear = NULL;

    queue.count = 0;

    push(&queue, 7);

    push(&queue, 5);

    push(&queue, 4);

    pop(&queue);

    push(&queue, 6);

    pop(&queue);

    show(&queue);

    system("pause");

    return 0;

}