

## < 자료구조 1 >

### 자료구조란?

자료구조(data structure)는 컴퓨터 과학에서 효율적인 접근 및 수정을 가능케 하는 자료의 집합을 의미하며 각 원소들 사이의 관계가 논리적으로 정의된 일정한 규칙에 의하여 나눌되어 자료에 대한 처리를 효율적으로 수행할 수 있도록 자료를 조직적, 체계적으로 구분하여 표현한 것을 말한다.

### 자료구조가 필요한 이유

데이터를 효율적으로 저장, 관리하여 메모리를 효율적으로 사용하기 위해서이다. 적절한 자료구조의 사용은 메모리의 용량을 절약해주고, 실행시간을 단축 시켜줄 수 있다.

### 자료구조의 선택 기준

작업의 효율성, 추상화, 재사용성을 증가시키기 위하여 상황에 따른 적절한 자료구조의 사용이 필요하다. 따라서, 아래의 사항을 고려하여 자료를 좀 더 효율적으로 처리할 수 있도록 한다.

- 자료의 처리시간
- 자료의 활용빈도
- 프로그램의 총이성
- 자료의 크기
- 자료의 간접정도

### 자료구조의 특징

#### 1. 효율성

자료구조를 사용하는 목적은 효율적인 데이터의 관리 및 사용이다. 적절한 자료구조의 선택은 업무의 효율에도 영향을 준다.

#### 2. 추상화

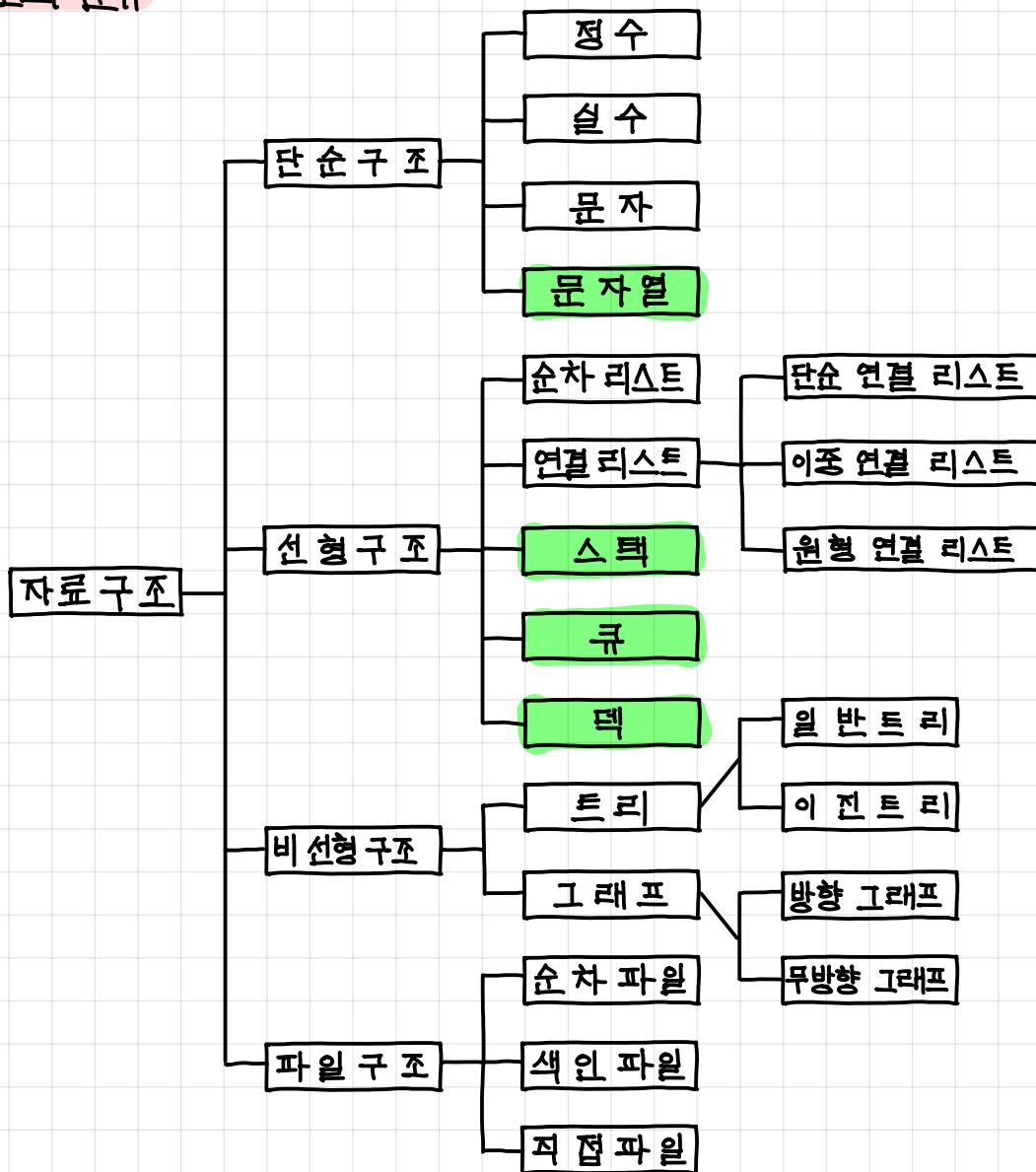
추상화란 복잡한 자료, 모듈, 시스템 등으로부터 핵심적인 개념만 추출해내는 것이다. 자료구조를 구현할 때 어느 시점에 데이터를 삽입할 것이며, 어느 시점에 이러한 데이터를 어떻게 사용할 것인지에 대해서 초기값을 맞출 수 있기 때문에 구현 외적인 부분에 더 시간을 쓸 수 있다.

예를 들어 스택(Stack)의 경우 먼저 들어간 것이 나중에 나오는 FILO(First In Last Out)의 형태를 가지고 있다. 그리고 Push() 함수와 Pop() 함수를 이용해 데이터를 삽입, 추출할 수 있다. 이 두 함수에 대해 내부 구현이 어떻게 되었는지는 별로 중요하지 않다. 상황별로 다른 코드가 나올 것기에 추상적인 개념에 대해서만 이해하고 사용할 수 있다.

#### 3. 재사용성

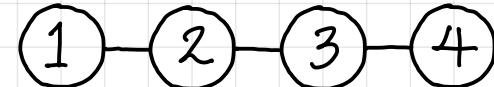
자료구조를 설계할 때 특정 프로그램에서만 동작하게 설계하지 않는다. 다양한 프로그램에서 동작할 수 있도록 별도로 설계하기 때문에 다른 프로젝트에서도 사용할 수 있다.

## 자료구조의 분류

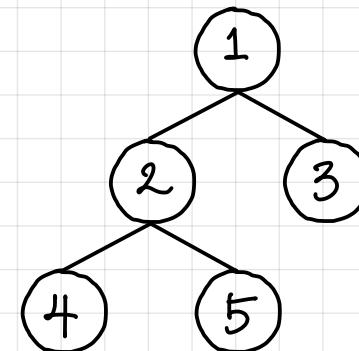


자료구조 1에서는 스택, 큐, 데크, 문자열을 볼 것이다.

### 선형구조



### 비선형구조



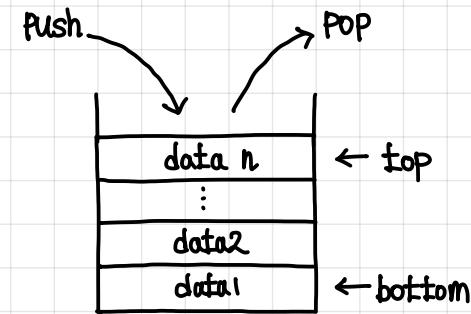
## 스택

스택은 마지막에 들어온 것이 먼저 나가는 LIFO( Last In First Out ) 구조를 가진 자료구조이다.

스택에서 데이터를 넣는 것을 Push, 데이터를 꺼내는 것을 POP이라고 한다.

Push와 Pop을 하는 위치를 top이라 하고, 스택의 가장 아래 부분을 bottom이라고 한다.

스택은 배열이나 연결리스트로 구현할 수 있다.



## 배열로 스택 구현

```
#define STACK_SIZE 10
typedef struct Stack
{
    int buf[STACK_SIZE]; // 저장소
    int top; // 가장 최근에 저장한 자료 번호
} stack;
```

```
void InitStack(Stack *stack)
```

```
{
    stack->top = -1;
}
```

```
int IsFull(Stack *stack)
```

```
{
    return (stack->top+1) == STACK_SIZE;
}
```

```
int IsEmpty(Stack *stack)
```

```
{
    return stack->top == -1;
}
```

```
void push(Stack *stack, int data)
```

```
{
    if (IsFull(stack))
    {
        printf("스택 꽉 찬");
        return;
    }
    stack->top++;
    stack->buf[stack->top] = data;
}
```

```
int Pop(Stack *stack)
```

```
{
    int re = 0;
    if (IsEmpty(stack))
    {
        printf("스택 빈");
        return;
    }
    re = stack->buf[stack->top];
    stack->top--;
    return re;
}
```

## 큐

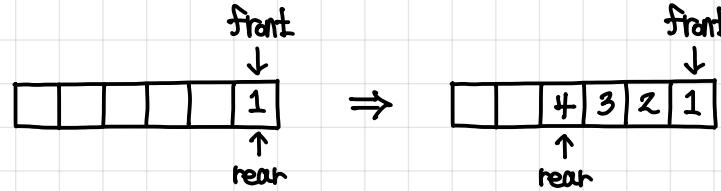
큐는 순차적으로 자료를 보관하고 가장 먼저 들어온 데이터가 먼저 나오는 FIFO(First In First Out) 구조로 저장하는 형식을 말한다. 큐에서 보관할 위치는 맨 뒤에 보관해서 **rear**라고 부르고, 꺼낼 위치는 맨 앞에서 **front**라고 부른다. 큐에 자료를 보관하는 행위를 **EnQueue(Put)**, 꺼내는 행위를 **DeQueue(Get)**이라고 한다. 큐는 다양한 방법으로 구현할 수 있으며, 큐에는 여러 종류가 있다.



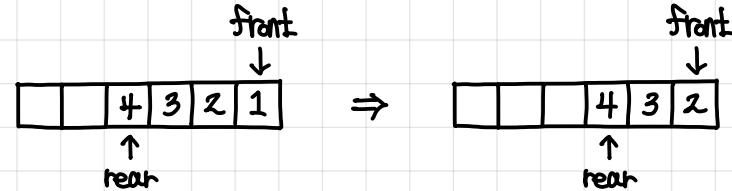
### 선형 큐(Linear Queue)

기본적인 큐의 형태이다. 막대 모양으로 된 큐로, 크기가 제한되어 있고 빈 공간을 사용하려면 모든 자료를 꺼내거나 자료를 한 칸씩 옮겨야 한다는 단점이 있다.

#### EnQueue



#### DeQueue



### 선형 큐의 문제점

일반적인 선형 큐는 **rear**가 마지막 index를 가르키면서 데이터의 삽입이 이루어진다. 문제는 **rear**가 배열의 마지막 인덱스를 가르키게 되면 앞에 남아있는 (삽입 공간에 DeQueue 되어 빌어있는 공간) 공간을 활용할 수 없게 된다. 이 방식을 해결하기 위해서는 DeQueue를 할 때 **front**를 고정시킨 채 남아 있는 데이터를 한 칸씩 앞당겨야 한다. 그래서 이를 보완하기 위해 나온 것이 원형 큐이다.

### 배열로 선형 큐 구현

```
#define QUEUE_SIZE 10
typedef struct Queue
{
    int front;
    int rear;
    int data[QUEUE_SIZE];
} Queue;

void InitQueue(Queue *q)
{
    q->rear = -1;
    q->front = -1;
}
```

```
int IsFull(Queue *q)
{
    if(q->front == QUEUE_SIZE - 1)
        return 1;
    else
        return 0;
}

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

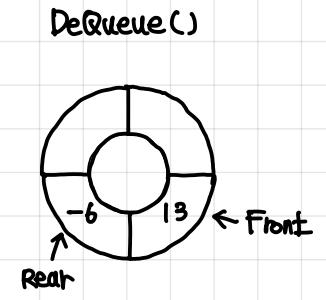
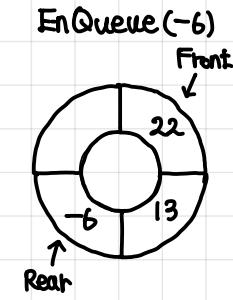
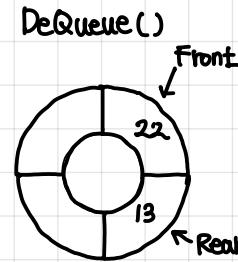
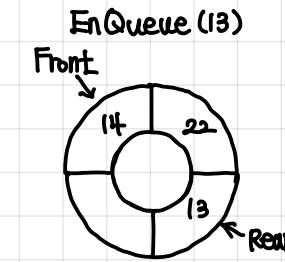
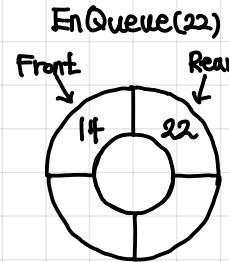
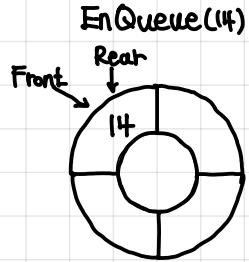
```
void EnQueue(Queue *q, int item)
{
    if(IsFull(q))
        printf("꽉 찬");
    else
        q->data[++(q->rear)] = item;
}

int DeQueue(Queue *q)
{
    if(IsEmpty(q))
        printf("빈");
    else
        int item = q->data[++(q->front)];
        return item;
}
```

## 원형 큐 (Circular Queue)

선형 큐의 문제점을 보완하고자 나왔다. 배열의 마지막 인덱스에서 다음으로 넘어갈 때  $(index + 1) \% (\text{배열 사이즈})$  를 이용하여 Out of Bounds Exception이 일어나지 않고 인덱스 0으로 순환되는 구조를 가진다.

## EnQueue / DeQueue



## 원형 큐의 문제점

한 칸을 바우지 않고 전부 다 채우면 공백과 포화상태를 구별할 수 없다. 물론 다 채우는 방법도 있다. 원형에 들어갈 수 있는 데이터 수를 size라고 한다면,

EnQueue 연산 시(데이터 추가)  $rear = (rear + 1) \% \text{size}$  하여 데이터를 새로 넣는다.  $(rear + 1) \% \text{size} = \text{front}$  이면 꽉 찬다고 판단한다.

DeQueue 연산 시(데이터 삭제) front가 1칸 움직인다. front도  $(front + 1) \% \text{size}$  만큼 움직인다.

## 배열로 원형 큐 구현

```
#define NEXT(index, QSIZE) ((index + 1) % QSIZE)
```

## typedef struct Queue

```
{
    int *buf; // 저장소
    int qsize;
    int front; // 개별 인덱스
    int rear; // 보관할 인덱스
    int count; // 보관 개수
}
```

Queue;

```
Void InitQueue(Queue *q, int qsize)
{
    q->buf = (int *) malloc(sizeof(int)*qsize);
    q->qsize = qsize;
    q->front = q->rear = 0;
    q->count = 0;
}

int IsFull(Queue *q){
    return q->count == q->qsize;
}

int IsEmpty(Queue *q){
    return q->count == 0;
}
```

```
Void EnQueue(Queue *q, int data){
    if(IsFull(q)){
        printf("꽉참");
        return;
    }
    q->buf[q->rear] = data;
    q->rear = NEXT(q->rear, q->qsize);
    q->count++;
}

int DeQueue(Queue *q){
    int re = 0;
    if(IsEmpty(q)){
        printf("텅");
        return re;
    }
    re = q->buf[q->front];
    q->front = NEXT(q->front, q->qsize);
    q->count--;
    return re;
}
```

덱

덱은 double-ended queue의 줄임말로, 큐의 앞, 뒤 모두에서 삽입 및 삭제가 가능한 큐를 의미한다.  
덱은 스택과 큐의 성질을 모두 갖고 있다. 따라서 덱은 상황에 따라서 스택도 될 수 있고, 큐도 될 수 있다.

### 원형 큐를 활용해 덱 구현

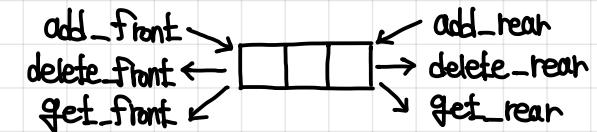
```
#define TRUE 1
#define FALSE 0
#define ERROR -1
#define MAX 5
typedef int boolean;
typedef int element;

typedef struct _dequeType {
    element *data;
    int rear, front;
} Deque;

Void init_deque(Deque *q) {
    q->data = (element *)malloc(sizeof(element)*MAX);
    q->front = q->rear = 0;
}

int is_empty(Deque *q) {
    if(q->front == q->rear)
        return TRUE;
    else
        return FALSE;
}

int is_full(Deque *q) {
    if(((q->rear + 1) % MAX) == q->front)
        return TRUE;
    else
        return FALSE;
}
```



```
Void add_front(Deque *q, element data) {
    if(is_full(q)){
        printf("꽉찬\n");
        return;
    }
    q->data[q->front] = data;
    q->front = (q->front - 1 + MAX) % MAX;
    return;
}

Void add_rear(Deque *q, element data) {
    if(is_full(q)){
        printf("꽉찬\n");
        return;
    }
    q->rear = (q->rear + 1) % MAX;
    q->data[q->rear] = data;
}
```

```
int delete_front(Deque *q) {
    if(is_empty(q)){
        printf("공백임\n");
        return ERROR;
    }
    element tmp = get_front(q);
    q->front = (q->front + 1) % MAX;
    return tmp;
}

element delete_rear(Deque *q) {
    if(is_empty(q)){
        printf("공백임\n");
        return ERROR;
    }
    element tmp = q->data[q->rear];
    q->rear = (q->rear - 1 + MAX) % MAX;
    return tmp;
}
```

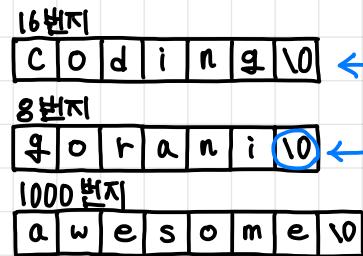
## 문자열

문자열(String)은 딱 그대로 문자들의 열이다. 따라서 문자열을 저장하기 위해서는 Char 배열이 필요하다. 예를 들어 5글자의 문자를 저장한다고 하면 크기가 5보다 큰 문자열을 만들면 된다.

문자열을 메모리에 저장한다고 해보자. 아래와 같이 될 것이다.

```
char C[] = "Coding";
char *d = "gorani";
char *e = malloc(sizeof(char)*100);
e = "awesome";
```

200번지	
C	16번지
300번지	
d	8번지
160번지	
e	1000번지



그림에서 볼 수 있듯 배열 자체는 어떠한 주소에 저장되어 있다.  
그러나 문자열들은 배열 자체의 주소가 아닌 다른 주소에 저장된다.  
또 문자열의 끝은 Null 문자로 처리되 있기 때문에 크기 5짜리 문자를 온전히 저장하려면 최소 6이상의 배열이 필요하다.