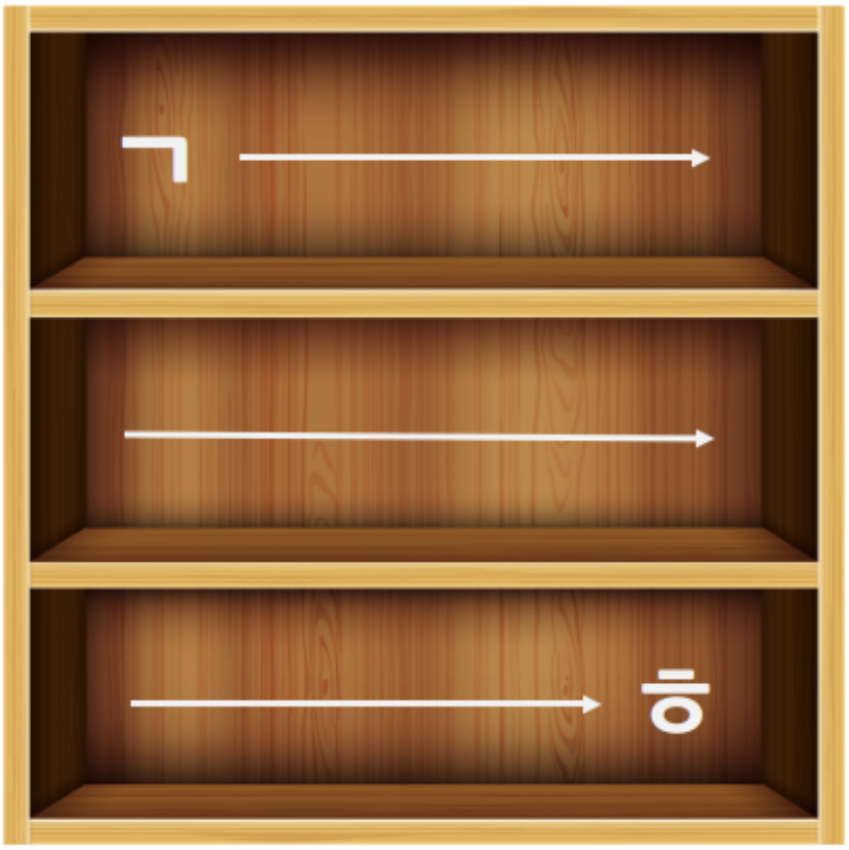
5.1 Complexity

💡 **자료구조**
- 데이터 값의 모임
-
데이터를 효율적으로 관리하고 수정, 삭제, 탐색, 저장 할 수 있는 **데이터 집합**

빈 책장



책 제목을 오름차순으로 정렬해 진열

책을 찾을 때 제목을 이용해 어디 꽂혀있는지 찾기 쉽다.
책의 모든 공간을 사용할 수 있다.
⇒ 처음 책을 꽂거나 이후 책을 추가할 때 수고스러움 발생
⇒ IT 관련 서적을 찾고 싶은 사람에겐 도움이 되지 않음

각 책장의 층을 분야별로 구분

관련 서적을 찾을 수 있다.
⇒ IT 서적 추가 자리 부족, 공간의 비효율 발생

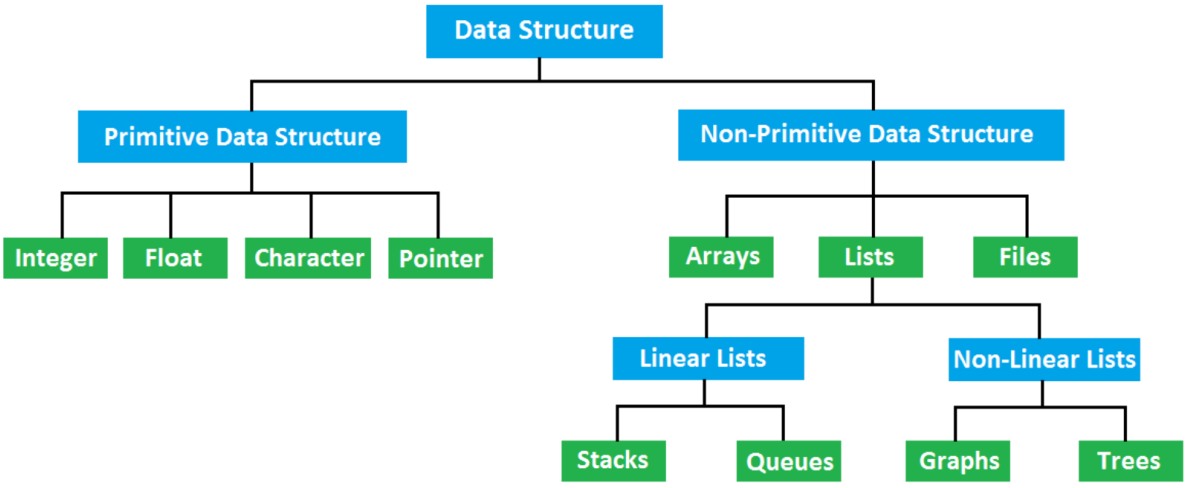
가림막 설치

가림막을 사용해 효율적으로 공간 사용
⇒ 가림막이 불필요한 공간 차지

> 책장 = 메모리
> 책 = 데이터

💡 **정리**
- 자료구조 :
메모리 공간을 효율적으로 사용해야 하는데 필요한 것
- 모든 목적에 맞는 자료구조는 없다.
- 각 자료구조의 장점, 한계 파악하는 것이 중요
- 메모리 공간의 효율, 실행시간의 효율성을 따진다.

- 필수 자료구조 8가지



5. 1 복잡도

- 시간 복잡도
- 공간 복잡도

5.1.1 시간 복잡도

💡 **시간 복잡도**

- 알고리즘을 수행하기 위해 프로세스가 수행해야 하는 연산을 수치화 한 것
- 효율적인 코드로 개선하는 데 쓰이는 척도

💡 **빅오 표기법**
입력범위 n을 기준으로 로직이 몇 번 반복되는지 나타내는 것

```
# 빅오 표기법 예시

1             # O(1)
2n + 20       # O(n)
3n^2          # O(n^2)
```

O(1) : 상수

```
def hello_world():
        print('Hello, world!')
```

O(N) : 선형

```
def print_each(lst):
        for i in lst:
                print(i)
```

O(N^2) : Square

```
def print_each_n_times(lst):
        for i in lst:
                for j in lst:
                        print(n, m)
```

O(log n), O(n log n)

```
def binary_search(lst, item, first=0, last=None):
        if not last:
                last = len(lst)

        midpoint = (last - first) / 2 + first

        if lst[midpoint] == item:
                reuturn midpoint
```

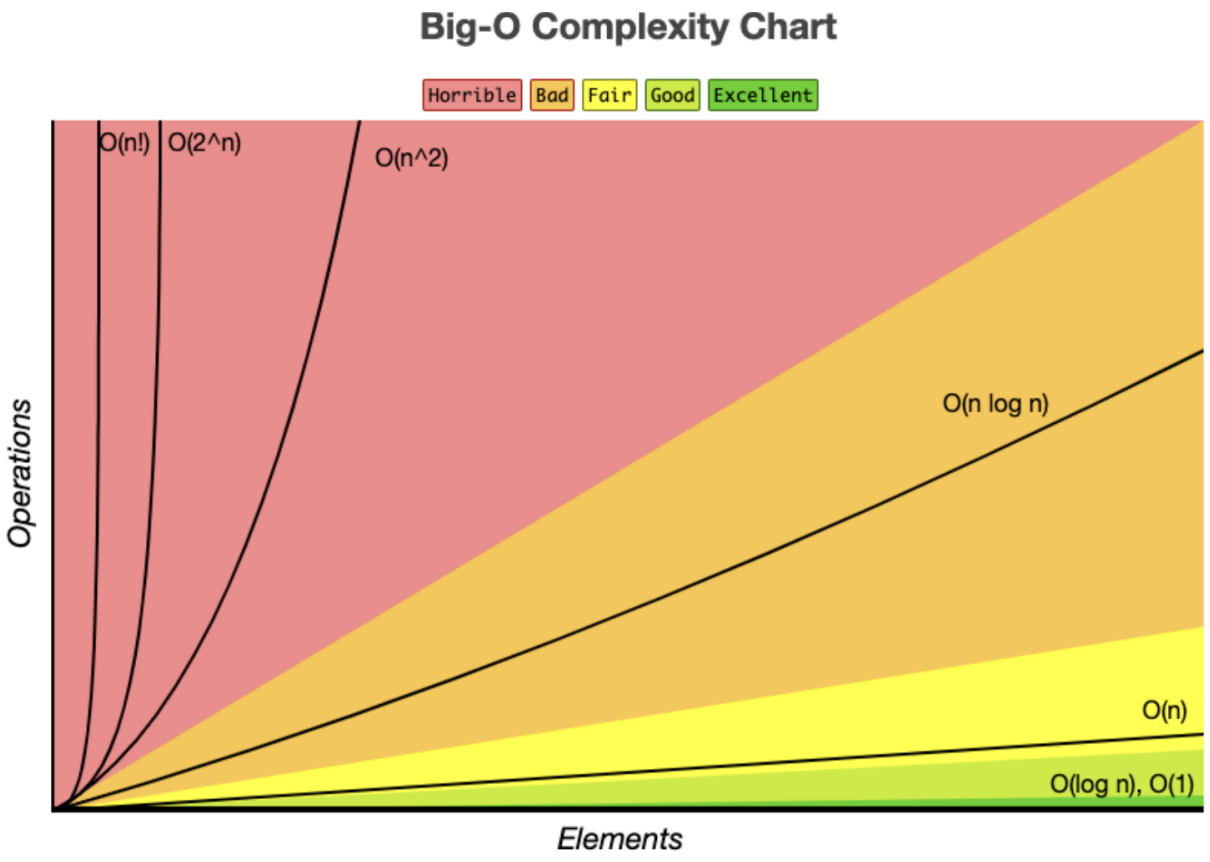
```
        elif lst[midpoint] > item:
                return binary_search(lst, item, first, midpoi

        else:
                return binary_search(lst, item, minpoint, las
```

O(2^n) : 지수 시간

```
def fibo(n):
        if n <= 1:
                return n
        return fibo(n-1) + fibo(n-2)
```

- 빅오 복잡도 차트



5.1.2 공간 복잡도

💡 **공간 복잡도**
- 프로그램을 실행시켰을 때 필요로 하는 지원 공간의 양
- 알고리즘에서 사용하는 메모리의 양
- 컴퓨터 성능의 발달로 중요도가 떨어지나 빅데이터 분야에서는 여전히 중요

O(1) : 상수

```
a = 1
```

```
result = 0
for i in range(1, 100):
        result += i
```

O(N) : 선형

```
def factorial(n):
        if n == 1:
                return 1
        return n * factorial(n-1)
```

O(N^2) : 이중 루프

```
def create_2d_array(n):
    matrix = [[0] * n for _ in range(n)]
    return matrix
```

- **자료구조 별 시간 복잡도와 공간 복잡도**

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)