자료구조(정진홍)

Junhyeok Lee (wnsx0000@gmail.com)

August 18, 2024

목차

Part I

자료구조 기초

1. 서론

1.1. DS

1.1.1. DS

Definition 1 *효율적으로 데이터를 접근/사용/수정하기 위한 데이터 관리/보관 방법을 자료구조(DS, Data Structure)라고 함.*

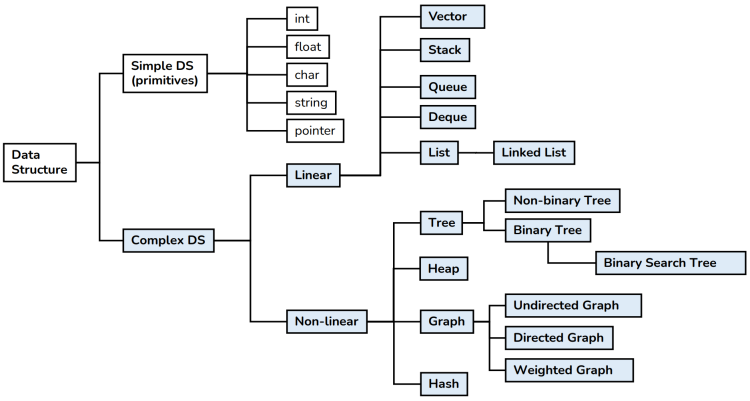
적절한 DS를 선택해 사용함으로써 시간/공간적으로 효율적인 프로그램을 만들 수 있음.

프로그램은 DS + 알고리즘으로 구성됨. DS가 데이터를 다루는 것이라면 알고리즘은 DS를 이용한 논리적 문제해결 과정이라고 할 수 있음.

1.1.2. DS의 종류

프로그래밍 언어가 알아서 구성하는 simple DS와, 사용자가 직접 구성해야 하는 complex DS로 나눌 수 있음. 또한 complex DS는 선형과 비선형으로 나눌 수 있음.

대표적인 DS에는 아래와 같은 것들이 있음.

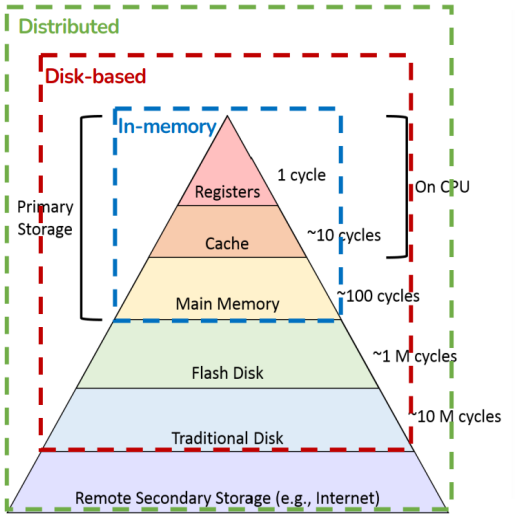


1.1.3. in-memory model vs disk-based model

DS에는 in-memory model과 disk-based model이 있음.

in-memory model은 데이터를 메인 메모리에 올리고 모든 데이터가 메인 메모리(RAM) 위에서만 작동한다고 가정한 상태의 DS임. 이 필기에서 정리하는 내용들은 in-memory model 기반임.

dis-based model은 데이터가 disk에도 존재하는 것으로 가정한 상태의 DS임. 데이터의 양이 메인 메모리를 벗어나게 될 경우 disk까지 내려가 작업하는 것.



1.1.4. 효율과 비용

시간적(실행 시간), 공간적(사용하는 메모리의 양)으로 어떤 자료구조가 효율적인지를 판단할 수 있음. 이때 사용되는 시간적, 공간적 자원(time cost, memory cost)을 비용이라고 함.

1.1.5. 자료구조의 선택

적절한 자료구조를 선택하기 위한 단계는 아래와 같음.

Step 1. 문제를 분석하고 어떤 연산들이 필요한지를 확인.
Step 2. 각 연산이 가지게 되는 시간적/공간적 한계를 확인.
Step 3. 요구사항을 만족시키는 최선의 DS 선택.

이때 복잡도 모델을 사용하여 판단할 수 있음. 각 DS는 장단점을 가지기 때문에 장점을 부각시키고 단점을 최소화할 수 있어야 함.

1.1.6. STL

Definition 2 *C++에서 사용하는 DS와 알고리즘을 템플릿 기반(Template-based)으로 구현해 둔 것을 STL(Standard Template Library)이라고 함.*

템플릿 기반이란 것은 코드 작성자가 자유롭게 자료형을 지정할 수 있게 한 것을 의미함.

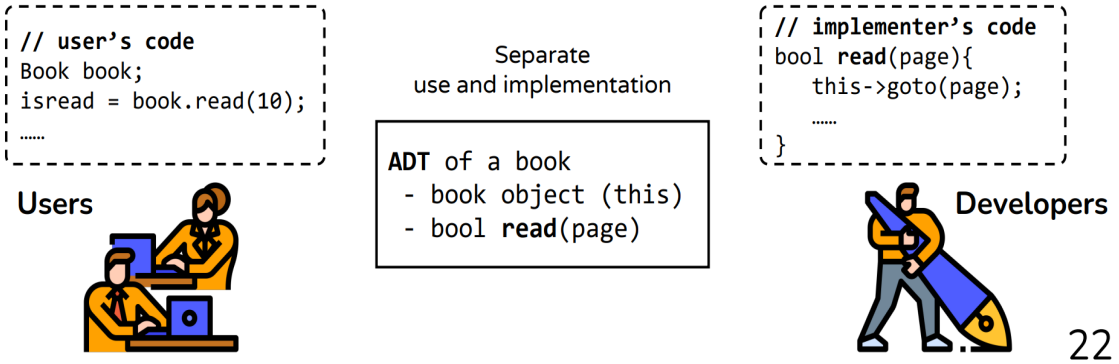
1.2. ADT

1.2.1. ADT

Definition 3 *데이터의 집합과 연산(operation)의 집합으로 구성된 추상적인 자료형을 추상자료형(ADT, Abstract Data Type)이라고 함. ADT는 연산에 대한 구체적인 정의를 내리지 않고, 방법론적인 정의만 한 것임.*

사용자와 자료형 사이의 인터페이스인 것. 이때 사용자는 ADT에 대한 방법론적 설명만을 듣고 가져다 쓸 수 있고, ADT를 선택할 때 구체적인 구현은 신경쓸 필요가 없음.

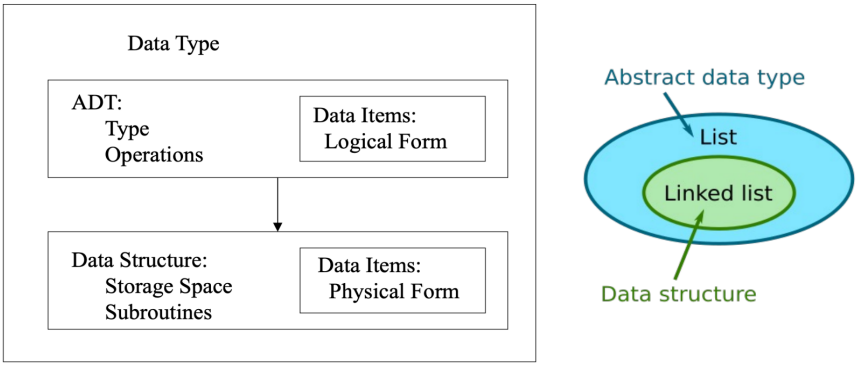
ADT는 객체지향언어(OOP, Object-Oriented Programming)의 class 개념을 사용하여 제작이 가능함.



1.2.2. ADT와 DS

자료구조는 ADT의 실질적인 구현임.
(ex. list는 ADT, linked list나 array list는 DS.)

어떤 데이터는 논리적(logical) 형태와 물리적(physical) 형태를 가지게 됨. ADT에서의 추상적 정의가 논리적 형태이고, DS에서의 구체적 구현이 물리적 형태임.



1.3. DS 학습의 방식

1.3.1. DS 학습의 방식

본 필기(수업)에서는 top-down view의 순서로 개념에 접근함. 즉, 추상적인 부분(high level)으로부터 구체적인 부분(low level)을 설명함. 구체적 순서는 아래와 같음.

1. DS에 대한 개요
2. DS의 ADT
3. ADT에서의 연산이 가지는 알고리즘
4. ADT를 기반으로 한 C++ 구현
5. 복잡도 분석
+ 6. STL 사용법 확인

학습 과정에서 신경써야 할 부분은, DS의 장단점에 따른 선택을 통해 효율적인 프로그램을 만드는 것, ADT의 개념과 쓸모를 이해하는 것, DS의 작동 원리를 학습하는 것임.

2. 알고리즘 분석

2.1. 알고리즘

2.1.1. 알고리즘

Definition 4 *컴퓨터를 활용한 문제 해결에서, 입력(input)으로부터 출력(output)으로 가는 논리적/절차적 단계를 알고리즘(Algorithm)이라고 함.*

즉, 알고리즘을 생각하기 이전에 입력과 출력을 정확하게 파악해야 함.

2.1.2. DS와의 관계

DS의 연산(operation)을 알고리즘으로 표현할 수 있음.

또한 알고리즘 분석을 통해 연산의 성능을 확인할 수 있음.

2.1.3. 알고리즘 표현 방법

알고리즘의 대표적인 표현 방법에는 3가지 방식이 있음.

1. NL(Natural language, 자연어)
: high-level language(일상 언어)로 알고리즘의 메인 아이디어와 단계를 표현하는 것.
알아보기 쉽게 그림 등으로 표현하기도 함.
사람이 알아보기에는 편리하지만 컴퓨터는 알아들을 수 없고, 개념이나 단어를 명확하게 정의하지 않은 경우 내용이 잘못 전달될 수 있음.

2. Pseudo-code
: NL과 유사 PL을 사용해 표현하는 것. 핵심만 표기하고 세부사항 표기는 반려함.
개발자를 위한 표현 방법.
여기서는 Python 스타일로 작성함.

3. PL(Programming language)
: 직접 소스 코드를 작성하는 것.
핵심을 이해하기 불편할 수 있음.

2.2. 알고리즘의 효율

한 문제에 대해 여러 알고리즘이 존재할 수 있음. 이 중 효율적인 알고리즘을 선택함.

2.2.1. 알고리즘 효율 측정법

효율은 시간/공간적 비용을 고려하여 계산함.

1. Empirical measurement : 실제로 측정하는 방식.
시간은 Wall-clock time을 측정하여 알 수 있음. Wall-clock은 벽에 걸린 시계를 의미하는데, 그냥 시간을 잰다는 것임. 프로그램 앞뒤로 시간을 재고 두 값을 빼서 프로그램 실행 시간을 측정할 수 있음.
공간은 메모리 사용량을 측정함.

실제 성능을 확인하기에 용이하지만, 환경에 따라 측정값이 달라질 수 있음. 또한 입력값에 따른 성능변화를 알 수 없고, 실험 자체도 오래 걸릴 수 있음.

2. Theoretical measurement : 이론적으로 확인하는 방식.
시간, 공간 효율을 시간복잡도(Time Complexity)와 공간복잡도(Space Complexity)로 확인할 수 있음. 시간복잡도는 기본연산이 수행된 횟수이고, 공간복잡도는 사용된 메모리의 양임.

복잡도는 입력 데이터의 크기에 따른 값을 가짐.
시간복잡도는 $T(n)$으로 표기하고, 공간복잡도는 $S(n)$으로 표기함.
대체로 공간복잡도는 입력 데이터의 크기에 선형적으로 비례하는 값을 가짐.
시간복잡도는 아래에 따로 정리함.

2.2.2. 기본연산

Definition 5 *입력 크기에 상관없이 수행 시간이 상수값(constant time)을 가지는 연산을 기본연산(Basic operation)이라고 함.*

기본연산으로는 사칙연산, 대입 연산(Assignment), 비교 연산(Comparison), 그리고 PL로부터 제공받은 연산들이 존재함.

2.2.3. 시간복잡도 분석

기본연산이 수행된 횟수를 세서 시간복잡도를 구할 수 있음.
이때 너무 엄격하게 모든 횟수를 셀 필요는 없음. 예를 들어, for문 내부에서 돌아가는 i 변수 등에서도 덧셈 연산이 계속 이뤄지지만, 경향성을 보는 것이라면 굳이 포함시키지 않아도 됨.

아래는 그 예시임.

Algorithm A	Algorithm B	Algorithm C

`sum ← 0`
`for i in range(0, n):`
`  sum ← sum + n`
	`sum ← 0`
	`for i in range(0, n):`
	`  for j in range(0, n):`
	`  sum ← sum + 1`
		`sum ← n × n`

- Count the number of basic operations := $T(n)$

	Algorithm A	Algorithm B	Algorithm C
Assignments	$n+1$	$n \times n+1$	1
Additions	n	$n \times n$	
Multiplications			1
Total	$T_A(n) = 2n+1$	$T_B(n) = 2n^2+1$	$T_C(n) = 2$

2.3. 최선/평균/최악의 경우

2.3.1. 최선/평균/최악의 경우

입력값에 따라 연산량이 다를 수 있는데, 제일 연산량이 적은 경우를 최선의 경우(Best Cases), 평균적인 연산량을 가지는 경우를 평균적 경우(Average Cases), 제일 연산량이 많은 경우를 최악의 경우(Worst Cases)라고 함.

2.3.2. 입력의 특수성을 고려한 효율 측정 방법

DS에서 실제 성능 비교 시에는 최악의 경우를 사용함. 적어도 해당 경우보단 좋은 경우만이 존재하므로 성능을 보장할 수 있음.

최선의 경우는 성능적 지표로서 사용하기에는 무리가 있음.

평균적 경우가 프로그램 사용의 체감 성능이 됨. 이때 실제로 평균적인 경우에 대한 정확한 값을 구하려면 모든 경우를 다 시도하고 평균값을 구해야 하기 때문에, 통계적/확률적으로 접근하여 추정하는 것이 일반적임.

정말 엄밀하게 할 때에는 평균적 경우도 확인하지만 대체로 그렇게까지는 하지 않음. 다만 데이터의 특성에 따라 분석 결과가 실제와는 괴리가 있을 수 있기 때문에(대부분이 최선인 경우인 데이터 등), 특히 데이터마이닝 같은 분야에서는 더 자세히 분석을 함. DS 상에서는 최악의 경우로 비교를 함.

2.4. Asymptotic analysis

각 시간복잡도가 가지는 성능을 나타내는 방법.

2.4.1. Asymptotic analysis

Definition 6 *데이터의 크기가 무한히 커지는 경우를 생각하는 것을 점근적 분석(Asymptotic analysis)라고 함.*

n이 매우 클 때, 복잡도 함수의 그래프로 점근석 분석을 할 수 있음. 이때의 함수 변화를 Asymp-

totic/Limiting/Tail behavior라고 한다.

Asymptotic analysis에서, n이 무한히 커질 때 복잡도 함수의 그래프에 가장 큰 영향을 미치는 항 또는 요소를 Dominant factor라고 함.

어떤 알고리즘이 특정 구간에서 더 효율적이어도, n이 충분히 커지는 경우를 생각하면 결과적으로 어떤 알고리즘의 성능이 더 좋은지를 판단할 수 있음.

Asymptotic behavior로는 linearly(직선), quadratically(이차함수) 등이 있음.

이때 무한으로 갈수록 Dominant factor의 영향력이 계속 커지고, 나머지 항들과 계수 등은 큰 의미가 없어짐.

2.4.2. Asymptotic notation

Definition 7 *임의의 복잡도 함수의 Asymptotic behavior를 간단히 묘사하기 위한 표기를 점근적 표기(Asymptotic notation)라고 함.*

Asymptotic notation에는 Big-O, Big-Omega, Big-Theta 표기법 등이 있음.

2.5. Big-O 표기법

2.5.1. Big-O 표기법

Definition 8 *아래의 식과 같은 특정 복잡도 함수들의 집합을 $O(f(n))$이라고 하고, 이 집합으로 복잡도 함수를 점근적으로 표기하는 방법을 Big-O 표기법이라고 함. 이때의 $f(n)$을 Basic operation/function이라고 함. Big-O 표기법은 $cf(n)$로 해당 복잡도 함수에 대한 점근적 상한(asymptotically upper bound)을 제공함.*

$O(f(n)) = \{g(n) \mid$ *there is two positive constants* c *and* n_0 *such that* $g(n) \leq cf(n)$ *for all* $n \geq n_0\}$

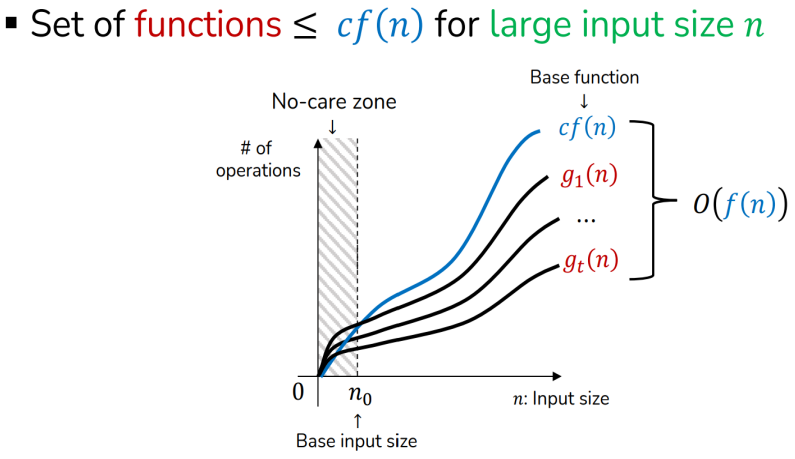
dominant factor를 표기하는 것이 Big-O 표기법임.

즉, 어떤 함수 $g(n)$에 대해, $n \geq n_0$의 범위에서 $g(n) \leq cf(n)$를 만족시키는 두 양의 상수 c와 n_0가 존재하면 $g(n)$은 $O(f(n))$에 포함되는 함수인 것. 정의를 만족하는 상수 c와 n_0에 대한 존재 여부만을 확인하면 해당 big-O에 포함되는지를 알 수 있고, 당연하게도 이때 그 값이 최솟값일 필요는 없음.

Big-O는 집합이므로 해당 알고리즘의 시간복잡도가 어떤 $O(f(n))$에 속한다는 것은 $f(n) \in O(f(n))$로 표기하는데, =나 ≠가 관습적으로 사용되기도 함.

참고로 n은 0이 될 수 없음. 0이면 입력되지 않은 것.

Big-O가 O(1)이면 상수 시간이 걸린다고 함.



2.5.2. 다항함수의 Big-O

Definition 9 *복잡도 함수 $T(n)$이 다항함수(polynomial)인 경우, 차수를 k라고 하면 $k \leq r$인 자연수 r에 대해 $T(n) \in O(n^r)$임.*

다항함수는 차수만으로 Big-O를 구할 수 있음.

2.5.3. loose or tight

Definition 10 *점근적 상한과 복잡도 함수 $T(n)$ 사이의 gap이 비교적 크게 설정되었을 때, Big-O 표기법이 loose하게 상한을 지정한다고 함.*

점근적 상한과 복잡도 함수 $T(n)$ 사이의 gap이 비교적 작게 설정되었을 때, Big-O 표기법이 tight하게 상한을 지정한다고 함.

$O(n^2)$에 속하는 복잡도 함수는 $O(n^3), O(n^4), \cdots$에도 속하므로, 점근적 상한이 해당 복잡도 함수에 비교적 딱 맞게 지정됐을 수도 있고 그렇지 않을 수도 있음.

점근적 상한은 가능한 한 tight하게 지정하는 것이 합리적이고, 실제로 "As tight as possible"이라는 워딩이 자주 사용됨.

당연하게도 두 알고리즘을 분석하는데 하나는 tight하게, 다른 하나는 loose하게 지정하는 것은 합리적이지 않음.

2.5.4. 실제 사용

최악의 경우에 대한 Big-O 표기는 rough하게 알고리즘의 효율을 비교하는 데에 적합함.

참고로, 2^k 등에 대해서는 변수를 치환하면 깔끔하게 정리할 수 있음.

2.5.5. 주의점

big-O를 이용한 알고리즘의 비교에서는 둘 중 하나가 '아마 우세할 것'이기 때문에 어떤 것의 성능이 더 좋다고 이야기함. 즉, big-O만 사용하는 것은 rough하게 비교하는 것으로 실제로는 결과가 반대일수도 있음.

2.6. Big-Omega 표기법

2.6.1. Big-Omega 표기법

Definition 11 *아래의 식과 같은 특정 복잡도 함수들의 집합을 $\Omega(f(n))$이라고 하고, 이 집합으로 복잡도 함수를 점근적으로 표기하는 방법을 Big-Omega 표기법이라고 함. 이때의 $f(n)$을 Basic operation/-function이라고 함. Big-Omega 표기법은 $cf(n)$로 해당 복잡도 함수에 대한 점근적 하한(asymptotically lower bound)을 제공함.*

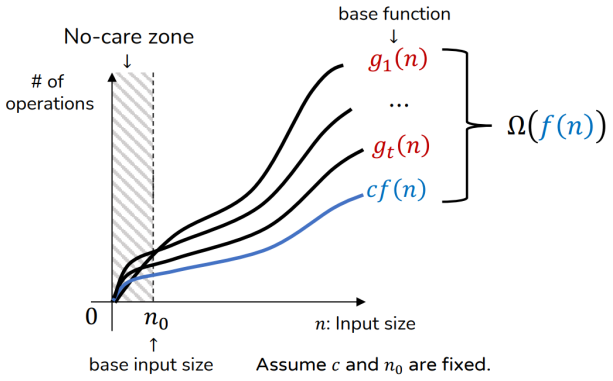
$\Omega(f(n)) = \{g(n) \mid$ *there is two positive constants* c *and* n_0 *such that* $g(n) \geq cf(n)$ *for all* $n \geq n_0\}$

즉, 어떤 함수 $g(n)$에 대해, $n \geq n_0$의 범위에서 $g(n) \geq cf(n)$를 만족시키는 두 양의 상수 c와 n_0가 존재하면 $g(n)$은 $\Omega(f(n))$에 포함되는 함수인 것. 정의를 만족하는 상수 c와 n_0에 대한 존재 여부만을 확인하면 해당 big-Omega에 포함되는지를 알 수 있고, 당연하게도 이때 그 값이 최솟값일 필요는 없음.

해당 알고리즘의 시간복잡도가 어떤 $\Omega(f(n))$에 속한다는 것은 $f(n) \in O(\Omega(n))$로 표기하는데, $=$나 $\neq$가 관습적으로 사용되기도 함.

참고로 n은 0이 될 수 없음. 0이면 입력되지 않은 것.

- Set of functions $\geq cf(n)$ for large input size n



2.6.2. 다항함수의 Big-Omega

Definition 12 *복잡도 함수 $T(n)$이 다항함수(polynomial)인 경우, 차수를 k라고 하면 $k \geq r$인 자연수 r에 대해 $T(n) \in \Omega(n^r)$임.*

다항함수는 차수만으로 Big-Omega를 구할 수 있음.

2.6.3. loose or tight

Definition 13 *점근적 하한과 복잡도 함수 $T(n)$ 사이의 gap이 비교적 크게 설정되었을 때, Big-Omega 표기법이 loose하게 상한을 지정한다고 함.*

점근적 하한과 복잡도 함수 $T(n)$ 사이의 gap이 비교적 작게 설정되었을 때, Big-Omega 표기법이 tight하게 상한을 지정한다고 함.

$\Omega(n^{10})$에 속하는 복잡도 함수는 $\Omega(n^9), \Omega(n^8), \cdots$에도 속하므로, 점근적 하한이 해당 복잡도 함수에 비교적 딱 맞게 지정됐을 수도 있고 그렇지 않을 수도 있음.

점근적 하한은 가능한 한 tight하게 지정하는 것이 합리적이고, 실제로 "As tight as possible"이라는 워딩 자주 사용함.

2.6.4. 실제 사용

최악의 경우 분석 시에 big-Omega는 유의미한 정보를 제공할 수 없기 때문에, 실제로 big-Omega를 가지고 비교를 진행하지는 않음.

2.7. Big-Theta 표기법

Big-O와 Big-omega의 개념을 모두 사용하여 상한과 하한 모두를 지정하는 방법. big-Theta는 big-O와 big-omega의 교집합이고, 이 둘의 교집합이 존재하려면 각각이 tight해야 함.

2.7.1. Big-Theta 표기법

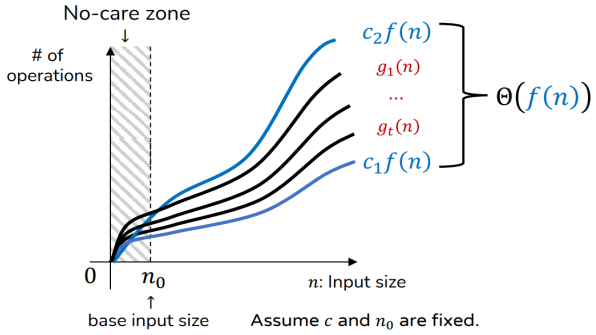
Definition 14 *아래의 식과 같은 특정 복잡도 함수들의 집합을 $\Theta(f(n))$이라고 하고, 이 집합으로 복잡도 함수를 점근적으로 표기하는 방법을 Big-Theta 표기법이라고 함. 이때의 $f(n)$을 Basic operation/function이라고 함. Big-Theta 표기법은 $c_1 f(n)$, $c_2 f(n)$로 해당 복잡도 함수에 대한 점근적 상한(asymptotically upper bound)과 점근적 하한(asymptotically lower bound)을 제공함.*

$\Theta(f(n)) = \{g(n) \mid$ *there is three positive constants* c_1, c_2 *and* n_0 *such that* $c_1 f(n) \leq g(n) \leq c_2 f(n)$ *for all* $n \geq n_0\}$

즉, 어떤 함수 $g(n)$에 대해, $n \geq n_0$의 범위에서 $c_1 f(n) \leq g(n) \leq c_2 f(n)$를 만족시키는 세 양의 상수 c_1, c_2와 n_0가 존재하면 $g(n)$은 $\Theta(f(n))$에 포함되는 함수인 것. 정의를 만족하는 상수 c와 n_0에 대한 존재 여부만을

확인하면 해당 big-Theta에 포함되는지를 알 수 있고, 이때 그 값이 최솟값일 필요는 없음.

- Set of $c_1 f(n) \leq$ functions $\leq c_2 f(n)$ for all $n \geq n_0$
  $g(n)$



2.7.2. 다항함수의 Big-Theta

Definition 15 *복잡도 함수 $T(n)$이 다항함수(polynomial)인 경우, 차수를 k라고 하면 $k \geq r$인 자연수 r에 대해 $T(n) \in \Theta(n^r)$임.*

다항함수는 차수만으로 Big-Theta를 구할 수 있음.

2.7.3. 실제 사용

최악의 경우에 대한 Big-Theta 표기는 정확하게 알고리즘의 효율을 비교하는 데에 적합함. 즉, Big-Theta는 tight한 상한과 하한을 제공함.

Big-Theta를 이용하여 엄밀한 비교를 할 수 있음. 다만 Big-O와 Big-omega가 다르거나 하한을 정하기 어려운 경우 등이 있을 수 있는데, 이때는 Big-O를 사용하는 것이 적절함. 본 수업에서는 Big-Theta를 사용할 수 있는 복잡도 함수들을 주로 다룸.

즉, 프로그램의 성능을 분석할 때, 최악의 경우에 대한 Big-Theta를 확인하면 됨. Big-Theta를 구해 보고 여의치 않은 경우 Big-O로 rough한 분석을 함.

프로그램의 코드를 짜면 바로 시간복잡도를 계산해 보는 것이 일련의 고정된 과정임.

2.7.4. 주의점

점근적 분석은 n이 무한대로 갈 때는 유효하지만, 프로그램을 데이터를 무한히 입력하며 사용하지는 않기 때문에 정확하지 않을 수 있음.

특히 n이 무한대로 가지 않는 경우에는 시간복잡도의 계수와 상수의 역할을 완전히 배제할 수 없음. 실제로 정말 제대로 된 최적화를 하려면 복잡도 함수의 계수까지 신경써야 할 수 있음. 물론 본 수업에서 그렇게까지 하지는 않음.

2.7.5. 매개변수가 여러 개인 경우

입력의 크기가 두 개의 매개변수(parameter)를 가지고 있는 경우에는 해당 매개변수들에 대해 복잡도 함수를 작성해야 함.

예를 들어 복잡도 함수가 n, m에 대해 작성된 경우 $T(n, m) \in \Theta(n \times m)$와 같이 두 변수에 대한 함수를 작성함.

2.8. 복잡도 카테고리

2.8.1. 복잡도 카테고리

알고리즘이 가지는 복잡도 함수를 점근적 기법으로 나타내면 대체로 복잡도 카테고리 안에 포함됨. 아래와 같이 각 카테고리별로 이름도 붙어 있음.

기본연산이 적게 수행될수록 성능이 좋고, 시간복잡도가 낮다고 이야기함.

보통 실제로 사용할 만한 알고리즘은 $nlogn$까지이고, 여기까지가 프로그램 성능의 허용 가능 범위.

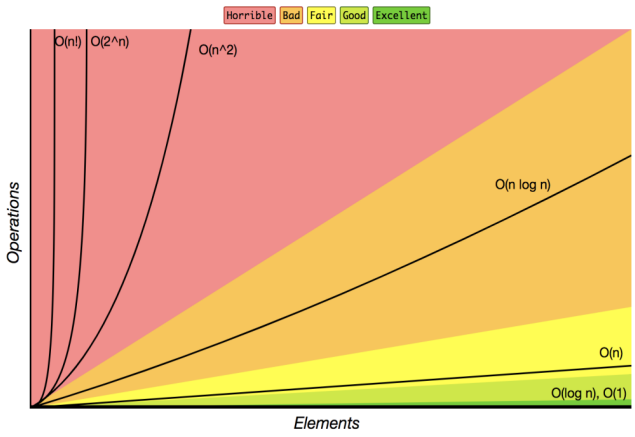
$log(logn)$이나 $nlog(logn)$ 등에 대해서도 직접 대입해 보거나, 그래프를 확인하는 등의 대소비교로 시간복잡도를 비교할 수 있음.

Base Func.	Name
1	Contant
$\log n$	Logarithmic
n	Linear
$n \log n$	Log-linear
n^2	Quadratic
n^3	Cubic
n^p	Polynomial
2^n	Exponential
$n!$	Factorial

Scalability

Good

Poor



2.9. 단순화 규칙

어떤 코드를 최악의 경우에 대해 복잡도를 분석할 때, 그 과정을 아래의 5가지 규칙에 의해 그 과정을 단순화할 수 있음. O에 대해 작성했지만 Ω, Θ에도 적용 가능함.

2.9.1. Rule 1 : 포함관계 성립

Definition 16 *If* $T(n) \in O(f(n))$ *and* $f(n) \in O(g(n))$, *then* $T(n) \in O(g(n))$

$T(n)$이 $O(f(n))$에 속하고, $f(n)$이 $O(g(n)$에 속하면 $T(n)$ 또한 $O(g(n))$에 속한다는 것. big-O가 상한을 지정한다는 것을 생각하면 당연함. $T(n)$ 위에 $c_1 f(n)$이 있고, $f(n)$ 위에 $c_2 g(n)$이 있음.

2.9.2. Rule 2 : 계수 삭제 가능

Definition 17 *If* $T(n) \in O(kf(n))$ *for constant* $k > 0$, *then* $T(n) \in O(f(n))$

$T(n)$이 속하는 big-O에 대해 생각할 때, $O()$ 내부 함수의 계수는 삭제가 가능함. big-O의 정의를 생각하면 당연함.

2.9.3. Rule 3 : 두 병렬적 시간복잡도의 병합

Definition 18 *If* $T_1(n) \in O(f_1(n))$ *and* $T_2(n) \in O(f_2(n))$, *then* $T_1(n) + T_2(n) = (T_1 + T_2)(n) \in O(max(f_1(n), f_2(n)))$

두 시간복잡도 모두에 대한 big-O는, 두 시간복잡도 각각의 big-O 중 basic funciton이 더 큰 것임. 주로 프로그램 내에서 순서대로 실행되는 두 코드에 대해서 적용함.

참고로, 시간복잡도가 속하는 big-O의 basic function 변수가 i, j, n 등으로 다르더라도 문자를 통일시켜서 볼 수 있음. 예를 들어, $O(i)$와 $O(n^2)$에 각각 해당되는 두 복잡도 함수를 더하는 경우 $O(i)$를 $O(n)$으로 보고 $O(n^2)$라고 하면 됨. 이건 다른 규칙들에서도 마찬가지임.

```
def main(n):
    # Part 1                T1(n) ∈ O(n)
    sum = 0
    for i in range(0, n):
        sum += 1

    # Part 2                T2(n) ∈ O(n^2)
    for i in range(0, n):
        for j in range(0, n):
            sum += 1
```

$(T_1 + T_2)(n) \in O(\max(n, n^2)) = O(n^2)$

2.9.4. Rule 4 : 조건문 분기 처리

Definition 19 *If $T(n)$ varies by conditions, then take greater complexity.*

조건문 분기에 따라 다른 시간복잡도를 가지게 될 때, 둘 중에 더 큰 것을 적용함. 최악의 경우에 대해 살펴보고 있으므로 당연함.

```
def main(n):

    if some condition:
        do something in O(n)

    else:
        do something in O(n^2)
```

$T(n) \in O(\max(n, n^2)) = O(n^2)$

2.9.5. Rule 5 : 두 직렬적 시간복잡도의 병합

Definition 20 *If $T_1(n) \in O(f_1(n))$ and $T_2(n) \in O(f_2(n))$, then $T_1(n) \times T_2(n) \in O(f_1(n) \times f_2(n))$*

쉽게 말해, 반복문 등에서 외부와 내부 모두에 대한 시간복잡도의 big-O는, 내부와 외부의 big-O basic function을 곱한 것임.

```
for i in range(0, n):
    for j in range(0, n):
        sum += 1          T2(n) ∈ O(n)
```

$T_1(n) \in O(n)$

$T_1(n) \times T_2(n) \in O(n \times n) = O(n^2)$

Suppose func(n) takes $O(n^2)$ times

```
for i in range(0, n):
    call func(n)
                          T2(n) ∈ O(n^2)
```

$T_1(n) \in O(n)$

$T_1(n) \times T_2(n) \in O(n \times n^2) = O(n^3)$

이때 주의해야 할 점은, 반복문 내부의 반복이 외부 변수에 영향을 받는 경우 단순 곱으로 계산해 버리면 loose한 upper bound가 만들어진다는 것임. 이 경우 직접 세서 계산하는 것이 바람직함. 특히 2개의 loop가 중첩되어 있고, 내부 loop의 반복 횟수가 바깥 loop의 변수로 지정되는 경우 단순히 $O(n^2)$이라고 하면 loose하게 upper bound를 잡은 것임.

아래는 그 예시임. 여기서 j는 1번, 2번, ... , i번 실행되는데 내부 시간복잡도 함수를 O(n)로 해 버리면 j가 매번 n번 수행되는 것으로 과대평가되어 loose해지는 것.

Example of loose upper bound

```
for(i = 1; i <= n; i *= 2)
    for(j = 1; j <= i; j++)
        sum += 1
```

$T_2(n) \in O(i) \in O(n)$

$T_1(n) \in O(\log_2 n)$

$T_1(n) \times T_2(n) \in O(n \log_2 n)$

Example of tight upper bound

Assume $n = 2^K$ where K is # of loops of the outer for-loop†

$$T(n) = 1 + 2 + 2^2 + \cdots + 2^K$$
$$= \frac{2^{K+1} - 1}{2 - 1}$$
$$= 2n - 1 \in O(n)$$

Part II

C++ 기초 문법

1. 기초 문법

c++은 c 기반 언어로 c의 문법을 기본적으로 사용하는데, c에 해당하는 문법은 제외하고 정리함. 시험 대비 등에는 c 문법도 같이 공부하자.

1.1. 표준 입출력

1.1.1. 표준 입출력

Definition 21 *c++에서는 std::cin 객체(character input)와 std::cout 객체(character output)를 사용하여 표준 입출력을 함. 이때 «, » 연산자를 사용하는데 그 형식은 아래와 같음.*

```
std::cin >> 변수명
std:cout << 입력값
```

«는 삽입(insertion) 연산자, »는 추출(extraction) 연산자임. cout과 cin은 입출력 위한 객체로, 프로그램 실행 시에 자동으로 생성됨. «/»은 cout/cin의 class에서 연산자 오버로딩을 한 것임.

여러 개의 «, » 연산자를 사용하면 가장 왼쪽 부터 오른쪽으로 하나씩 입출력함.

std::in에서 입력 데이터는 whitespace로 구분함. 공백, 개행, 탭 등이 whitespace로 취급됨. std::in은 scanf()와 달리 주소가 아니라 변수명을 작성함.

여기서 std::는 std라는 namespace를 지정하는 것임. namespace는 이름을 저장하는 별도의 공간임. using namespace std;를 전처리기 위치에 작성하면 std namespace를 사용한다는 것을 지정할 수 있음. 이러면 std::cout 등에서 std::를 생략할 수 있음.

c와 달리 c++의 std::cint과 std::cout에서는 입출력 데이터의 자료형이나 크기를 지정해 줄 필요가 없음. 문자열 등도 여러 개로 끊어서 출력하는 등 간편한 입출력이 가능함.

stream은 입력과 출력에 대한 흐름으로 이해하자.

1.1.2. iostream

표준 입출력 라이브러리.

c++애서 제공하는 표준 헤더파일에 대해서는 .h를 작성하지 않음. 사용자 정의 헤더파일에 대해서는 .h를 작성함.

1.1.3. 개행

c++에서는 std:endl(end line)로 개행함. «로 std::endl을 입력하면 해당 위치에 개행이 적용되는 것.

1.2. 자료형

1.2.1. bool형

Definition 22 *c++에는 논리값(boolean)에 대한 자료형으로 bool형을 사용함.*

bool형으로는 참거짓을 나타내는 1바이트 크기의 데이터를 저장함. 값으로는 true와 false를 사용할 수 있음.

bool형 변수에 0에 해당되는 값(0, 0.0, NULL 등)을 대입하면 0으로 저장되고, 그렇지 않은 값들은 모두 1로 저장됨.

참거짓을 판별하는 함수 반환값의 자료형으로 자주 사용함.

1.2.2. reference

Definition 23 *reference(참조자, 참조형 변수)는 변수에 가명(alias)를 붙이는 문법임. 아래와 같이 &(ampersand)를 사용하여 정의함.*

```
int &test2 = test;
int& test2 = test;
```

즉, test2가 test에 대한 새로운 이름, test의 주소를 가진 변수가 되는 것.

reference를 사용하면 해당 reference가 할당한 변수에 대해 또 다른 식별자로서 기능함. 포인터 개념을 더 강화한 문법.

1.2.3. nullptr

c에서 사용하던 null은 포인터가 아니라 단순히 0으로 대체되는 키워드였음. nullptr은 0을 값으로 가지는 포인터임.

c++에서는 null 대신 nullptr을 사용하는 것을 권장함.

1.3. 문자열 처리

1.3.1. string 객체

Definition 24 *<string> 라이브러리를 사용하여 string 객체를 사용할 수 있음. string 객체는 아래와 같이 선언함.*

```
std::string s = "game over";
```

string 객체에는 char형 배열과 동일하게 데이터가 저장되지만, 오버로딩된 연산과 함수를 사용하여 문자열을 편리하게 다룰 수 있음.

== : 두 문자열을 비교함.
+= : 왼쪽 문자열의 null을 제거하고 뒤에 오른쪽 문자열을 추가함.
s.size() : s의 길이를 반환함.

일부 헤더파일에서 다른 헤더파일의 내용을 포함하는 경우가 있기 때문에, <iostream> 등을 사용한다면 <string> 라이브러리를 사용하지 않아도 string 객체를 사용할 수는 있음. 하지만 가독성, 이식성 등의 측면에서 <string> 라이브러리를 사용하는 것이 권고됨.

1.4. 동적 메모리 할당

1.4.1. 할당과 반납

Definition 25 *c++에서는 new와 delete 키워드로 아래와 같이 메모리를 동적으로 할당하고 반납할 수 있음.*

```
int* ptr1 = new int;        // int만큼 할당
delete prt1;                // 반납
int* arr1 = new int[10];    // int[10]만큼 할당
delete []arr1;              // 반납
```

c에서 malloc/calloc과 free를 사용했다면, c++에서는 new와 delete를 사용할 수 있음.

동적으로 할당한 메모리는 자동 반납되지 않는데, 이로 인해 낭비되는 메모리가 생기는 것을 메모리 누수라고 함. 다 쓴 메모리는 반납하는 것이 좋음.

1.5. 반복문

1.5.1. 범위기반 for

Definition 26 *c++에서는 배열, vector 등에 대해 값을 더 쉽게 사용할 수 있도록 하는 for문인 범위 기반 for를 사용할 수 있음.*

아래와 같이 작성함. 앞에 작성한 변수에 배열, vecotr등의 값이 차례대로 복사됨.

```
for(data_type val : data_list)
{
    ...
}
```

1.6. 함수

참고로, 본 수업에서는 매개변수를 formal arguments, 인자를 actual arguments로 명명함.

1.6.1. 함수 오버로딩

Definition 27 *c++에서는 함수 오버로딩(function overloading)이라는 개념으로, 같은 식별자를 가진 함수를 여러 개 정의할 수 있음. 이때 식별자가 겹치는 함수들은 매개변수 목록이 서로 달라야 함.*

이때 매개변수의 개수가 다르거나, 개수가 같다면 자료형이 달라야 함.

함수 반환값의 자료형은 관계 없음. 매개변수 목록이 같고 반환값의 자료형만 다르다면 오류가 발생함.

1.6.2. 값에 의한 호출과 참조에 의한 호출

c와 c++에서는 값에 의한 호출(call by value)가 기본이지만, 포인터와 reference로 참조에 의한 호출(call by reference)을 구현할 수 있음.

값에 의한 호출에서는 값을 복사해서 전달해야 하므로 copy cost가 발생함.

1.6.3. reference를 사용한 호출

함수의 매개변수로 reference를 사용하면 참조에 의한 호출을 구현할 수 있음. 아래는 그 예시임.

```
int func(int &a, int &b)
{
```

```
    a = 20;
    b = 10;

    return a + b;
}
```

1.6.4. reference로 반환

reference로 리턴값을 반환할 경우 해당 변수에 대한 참조가 반환됨. 즉, 함수 내부 변수를 전달하여 다른 식별자로 계속 사용할 수 있음.

이런 식의 구현은 []의 연산자 오버로딩 등에서 사용함. 아래는 그 예시임.

```
int& operator[](int idx)
{
    ...
    return array[idx];
}
```

물론 포인터에서처럼 함수의 지역변수를 reference로 반환하는 것은 의미가 없음. 함수 종료 시 메모리가 사라지기 때문.

1.7. vector

1.7.1. vector

Definition 28 *vector는 동적으로 사용 가능한 배열임.*

할당된 공간을 전부 사용할 때마다 해당 시점에서의 크기만큼의 두 배만큼 새로운 공간을 할당하는 방식을 사용함.

<vector> 라이브러리를 사용함.

아래와 같이 선언하여 생성할 수 있음. 크기도 지정 가능.

```
std::vector<data_type> name;
std::vector<data_type> name(size);
```

vector의 원소에 대해 접근, 삽입, 삭제하는 작업은 [], insert(), erase(), push_back() 등으로 함. []로 배열과 동일하게 사용할 수 있음.

선언 시에 중괄호로 초기화하는 것도 가능한데, class의 배열로 vector를 사용할 때 아래와 같이 중괄호로 각 class의 멤버 변수를 초기화할 수 있음.

```
class TestClass
{
private:
    int a;
    int b;
}
...
std::vector<TestClass> name1 = {{1,0}, {-1,0}, {0,1}, {0,-1}};
std::vector<TestClass> name2 = {TestClass(1,0), TestClass(-1,0),
                                TestClass(0,1), TestClass(0,-1)};
```

[]로 요소에 접근하는 것은 최악의 경우 1의 복잡도를 가짐.

vector는 삭제될 때 자동으로 메모리를 반납함.

1.7.2. 메소드, 연산자

vector 객체에 사용할 수 있는 메소드는 아래와 같음.

위치(position)은 iterator로 지정함. (추후에 list에서 설명함.)

insert(position, item) : 특정 위치에 item을 추가. 공간 확보를 위해 요소들을 옮겨야 하므로 최악의 경우 n의 복잡도를 가짐.
erase(position) : 특정 위치의 item을 삭제. 빈 공간을 채우기 위해 요소들을 옮겨야 하므로 최악의 경우 n의 복잡도를 가짐.
size() : 요소의 개수를 반환.
empty() : vector가 비어 있는 경우 true를, 차 있는 경우 false를 반환.
push_back(item) : vecotr의 끝에 item을 추가.

1.7.3. 다차원 vector

템플릿에 <vector<double»와 같이 자료형을 지정할 수 있음. vector<double>을 가리키는 요소들의 집합이므로, 요소에 참조값을 대입하여 이차원 벡터를 생성할 수 있음.

2. 객체지향 관련 문법

이 문법들은 대체로 생산성과 관련이 있음.

2.1. OOP

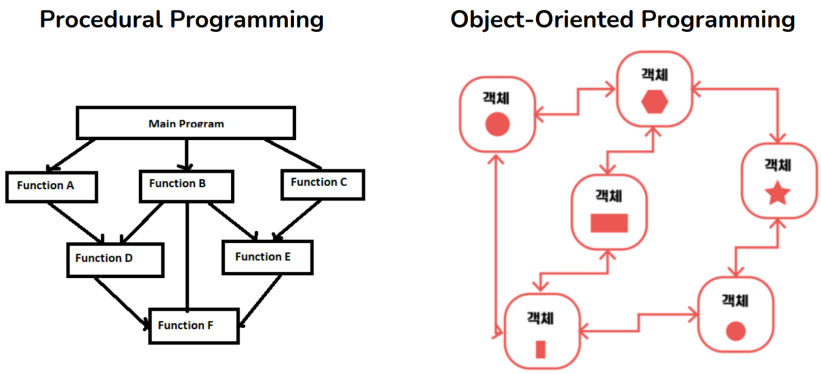
2.1.1. OOP

c에서는 절차지향 프로그래밍(PP, Procedural Programming)을 함. 절차지향 프로그래밍은 함수가 기본 단위이고, 데이터는 따로 관리하기 때문에 데이터를 유기적으로 관리하기에 어려움이 있음.

절차지향 프로그래밍에서 생산성을 더 강화하기 위해 등장한 것이 객체지향 프로그래밍(OOP, Object Oriented Programming)임. 절차지향언어에서는 함수가 기본 단위였다면, 객체지향언어에서는 객체(object)가 기본 단위임. 객체들 사이의 상호작용으로 프로그램이 작동함.

object는 데이터 부분인 state와 작업 부분인 behavior로 나뉨. c(PP)에서는 state를 구조체로, behavior를 함수로 따로 구분하여 구현해야 했음. c++(OOP)에서는 state와 behavior를 class라는 개념으로 합쳐서 구현함.

더 자세히 설명하면, class 내의 멤버 변수(member variable)로 state를, class 내의 멤버 함수(member function, 메소드)로 behavior를 구현함.



2.2. class

2.2.1. class

Definition 29 *state(데이터)와 behavior(함수)를 하나로 묶어서 정의하는 사용자 정의 자료형.*

state에 해당하는 부분을 를 멤버 변수(member variable), behavior를 멤버 함수(member function, 메소드)라고 함.

class는 class 키워드를 사용하여 정의함.

class 내부의 멤버에는 멤버접근자(member access specifier)를 지정함. 멤버접근자로는 private, public, protected가 있음. private은 외부에서 접근이 불가능하도록, public은 내외부 모두에서 접근이 가능하도록, protected는 내부와 자식 class에서는 접근이 가능하지만 외부에서는 접근이 불가능하도록 지정하는 것.

결과적으로 class는 아래와 같이 정의함.

```
class Name      // name으로 class 정의
{
private:        // member access specifier
    var1;       // member variable
    ...
public:         // member access specifier
    method1;    // member function
    ...
};
```

아래와 같이 클래스의 맨 윗줄에 friend 키워드로 특정 class를 지정하면 해당 class가 현재 class의 모든 멤버에 접근할 수 있도록 할 수 있음.

```
class B
{
    friend class A; // A에서 해당 class의 모든 멤버에 접근 가능
    ...
}
```

멤버 변수/함수는 일반적인 변수/함수와 동일한 형태로 작성함.

일반적으로 멤버 변수는 private, 멤버 함수는 public으로 지정함. 변수가 노출되면 의도치 않게 동작할 수도 있기 때문에 멤버 함수를 통해 멤버 변수에 접근하도록 하는 것. private으로 지정한 멤버 변수는 해당 class의 멤버 함수를 통해서 접근함.

멤버 함수가 멤버 변수를 조작할 때는 매개변수로 받지 않아도 됨. 멤버 변수가 class 내에서 전역적으로 선언된 것처럼 작동함.

접근이 불가능한 변수나 함수에 접근하려고 하면 컴파일 에러가 발생함.

class 정의 마지막에는 구조체에서처럼 ;를 작성해 줘야 함.

2.2.2. 멤버 함수 분리하기

멤버 함수의 선언부와 구현부를 분리해 작성할 수 있음.

클래스에는 함수 원형(선언부)만을 작성하고, 클래스 외부에 멤버 함수(구현부)를 작성하는 것. 이때 외부에 작성하는 메서드 앞에는 클래스명:: 을 붙여서 어떤 class의 메서드인지를 알려줘야 함.

아래는 그 예시임.

```
class SimpleClass{
    private:
        int num1;
        int num2;
    public:
```

```
        void print();
};

void SimpleClass::print()
{
    cout << num1 << " " << num2 << endl;
}
```

2.2.3. object

class는 state와 behavior를 하나로 묶은 자료형이고, 이 자료형을 사용하여 선언한 변수가 object임. class는 설계도이고, 설계도로 만든 실체가 object인 것. 그래서 object는 instance(인스턴스, 실체)라고도 하고, instance를 만드는 것을 실체화라고도 함.

object를 생성하는 것은 멤버 함수가 포함된 구조체 변수를 생성하는 것과 크게 다르지 않음. 실제로 그 메모리 구조도 동일한데, 해당 object의 멤버 변수는 스택에, 멤버 함수는 코드 세그먼트에 저장됨.

object의 멤버 변수/함수로의 접근은 구조체와 유사함. object에 .을 사용하여 접근하거나, object의 주소를 저장한 포인터에 ->을 사용하여 접근할 수 있음.

object의 주소를 포인터에 저장할 때에는 단순히 다른 object의 주소를 넣을 수도 있지만, new 키워드로 간단하게 동적할당을 할 수도 있음.

2.2.4. 생성자

Definition 30 *object가 생성되었을 때 실행되는 멤버 함수를 생성자(Constructor)라고 함.*

반환 자료형을 작성하지 않고 class 식별자와 동일한 식별자를 사용하여 정의함. 이때 서로 매개변수 목록이 다른 생성자를 여러 개 사용하여 들어오는 인자의 개수와 종류에 따른 경우를 나눌 수 있음. (function overloading)

생성자는 class 변수 선언 시에 실행되므로, 아래와 같이 선언과 동시에 인자를 전달해줌.

포인터에 new를 사용해서 객체를 생성할 때는 ()를 작성할 수 있지만, 변수로 객체를 생성할 때는 ()를 작성할 수 없음. 작성하면 컴파일러가 이를 함수로 오인하여 컴파일 오류가 남. 매개변수 목록이 ()인 생성자에 대해서도 마찬가지임.

```
Car car1;          // Car()이 실행됨.
Car car2(20);        // Car(int size)가 실행됨.
```

생성자 매개변수 목록에서 매개변수에 값을 대입하는 코드를 작성하면 해당 값이 default값으로 사용됨. 매개변수의 개수보다 인자를 적게 입력하면 인자는 앞쪽 매개변수부터 들어가고, 나머지는 default값이 사용됨.

생성자 또한 멤버 함수이므로 정의를 분리하여 class 외부에 작성할 수 있음.

생성자의 정의를 분리 작성한 경우 default는 원형에 작성할 수도, 정의에 작성할 수도 있음. 왜인지는 모르겠는데 원형과 정의 중 한 곳에만 default를 작성하는 것은 정상적으로 동작하지만, 원형과 정의 모두에 default를 작성하면 컴파일 에러가 발생함.

아래는 생성자 코드 예시임.

```
class Car
{
    private:
        int size;
    public:
        Car()                    // Car car1(); 또는 Car car1; 등인 경우 실행
        {
            this->size = 0;
```

```
        }
        Car(int size)           // Car car2(20); 등인 경우 실행
        {
            this->size = size;
        }
};
```

2.2.5. 소멸자

Definition 31 *object가 삭제될 때 실행되는 멤버 함수를 소멸자(Destructor)라고 함.*

반환 자료형을 작성하지 않고 class 식별자와 동일한 식별자와 그 앞에 ~을 붙여 사용하여 정의할 수 있음.

주로 동적할당으로 할당받은 메모리를 반납하기 위해 사용함.

아래는 소멸자 코드 예시임.

```
class Car
{
    ...
    public:
        ~Car()
        {
            delete[] this->name;
        }
};
```

2.2.6. 멤버 변수 초기화

기본적으로 멤버 변수는 class에 작성 시에 초기화 값을 지정해 놓을 수 있지만, 이 방법으로는 동적으로 초기화하지는 못함. 생성자의 인자를 사용하여 동적으로 초기화할 수 있음. 아래와 같이 '초기화할 변수'('인자') 꼴로 작성함.

```
class SimpleClass
{
    private:
        int num1;
        int num2;
    public:
        SimpleClass(int n1, int n2);
};

SimpleClass::SimpleClass(int n1, int n2) : num1(n1), num2(n2)
{
    ;
}
```

2.2.7. this 포인터

Definition 32 *this 포인터는 this 포인터가 사용되는 object를 가리키는 포인터임.*

주로 멤버 변수와 매개변수를 구분하기 위해 사용함. 예를 들어, this->num1 = num1;과 같이 멤버 변수와 멤버 함수의 매개변수의 식별자가 겹치는 경우, this가 가리키는 변수가 멤버 변수인 것을 알 수 있음.

2.3. 상속

이 부분에 대해서는 자세히 설명해주지 않았음. 인터넷에서 긁어온 내용이 좀 있으므로 추후에 다시 찾아보고 개선하자.

2.3.1. 상속

Definition 33 *부모 class로부터 멤버 변수/함수를 가져와 사용하는 것을 상속(inheritance)이라고 함.*

아래와 같이 부모로 지정할 class를 자식으로 지정할 class의 식별자 오른쪽에 public 키워드와 함께 작성함.

```
class SportCar : public Car
{
    ...
}
```

자식 객체를 생성하면 접근 가능한 부모/자식 class의 멤버 변수 모두가 해당 객체 메모리에 생성되고, 접근 가능한 부모/자식 class의 메소드 전부를 해당 객체에서 사용할 수 있음.

자식 class들에서 공통된 부분을 묶어서 부모 class에 올리고, 그 내용을 자식 class에 상속하여 사용하면 생산성을 높일 수 있음.

상속을 다이어그램으로 나타낼 때는 화살표 방향이 자식 class -> 부모 class임.

2.3.2. 다형성

Definition 34 *하나의 object가 여러 개의 형태를 가질 수 있도록 한 것이 다형성(Polymorphism)임.*

c++에서는 함수 오버라이드와 가상 함수로 다형성을 구현할 수 있음.

참고로 poly는 많다, morphism은 모양을 뜻함. 즉, 다형성은 class가 상황에 따른 다양한 모양을 가질 수 있음을 의미함.

2.3.3. 함수 오버라이드

Definition 35 *상속받은 부모 class의 멤버 함수를 자식 class에서 재정의하는 것을 함수 오버라이드(function overriding)라고 함.*

아래와 같이 자식 class의 멤버 함수 인자 오른쪽에 override 키워드를 사용하여 해당 멤버 함수를 오버라이드하도록 지정할 수 있음. 이때 이 멤버 함수의 식별자는 재정의할 부모 멤버 함수의 식별자와 동일해야 함.

```
void speedup() override
{
    ...
}
```

멤버 함수를 class에서 분리해 작성할 경우, override를 함수 원형에만 작성함. 원형과 정의 모두에 작성하거나 정의에만 작성하면 컴파일 에러가 발생함.

함수 오버로딩(function overloading)이랑은 다른 개념. 함수 오버로딩은 동일한 식별자의 함수를 여러 개 정의하는 것이고, 함수 오버라이드는 함수를 재정의하는 것.

2.3.4. 가상 함수

Definition 36 *실제로 지정된 함수를 고려하여 호출되는 함수를 가상 함수(virtual function)라고 함. 가상 함수로 지정된 메소드에 대해서만 override할 수 있음.*

아래와 같이 부모 class의 멤버 함수의 자료형 앞에 virtual을 작성하여 해당 멤버 함수를 가상 함수로 지정할 수 있음.

```
virtual void speedup()
{
    ...
}
```

멤버 함수를 class와 분리해 작성할 경우 virtual은 함수 원형에만 작성함. 원형과 정의 모두에 작성하거나 정의에만 작성하면 컴파일 에러가 발생함.

virtual으로 지정된 함수는 override로 지정된 함수로 재정의되어 호출됨.

2.3.5. 부모 class 포인터로 자식 class 가리키기

어떤 class의 부모 class를 지정하고 해당 class로 object를 만들면, 해당 class와 부모 class 모두에 대한 메모리가 할당됨. 이때 부모 class 포인터로 해당 object를 가리키면 부모 class에 대한 멤버에 접근할 수 있음. 단, 부모 포인터로 자식 class의 멤버에 접근할 수는 없음.

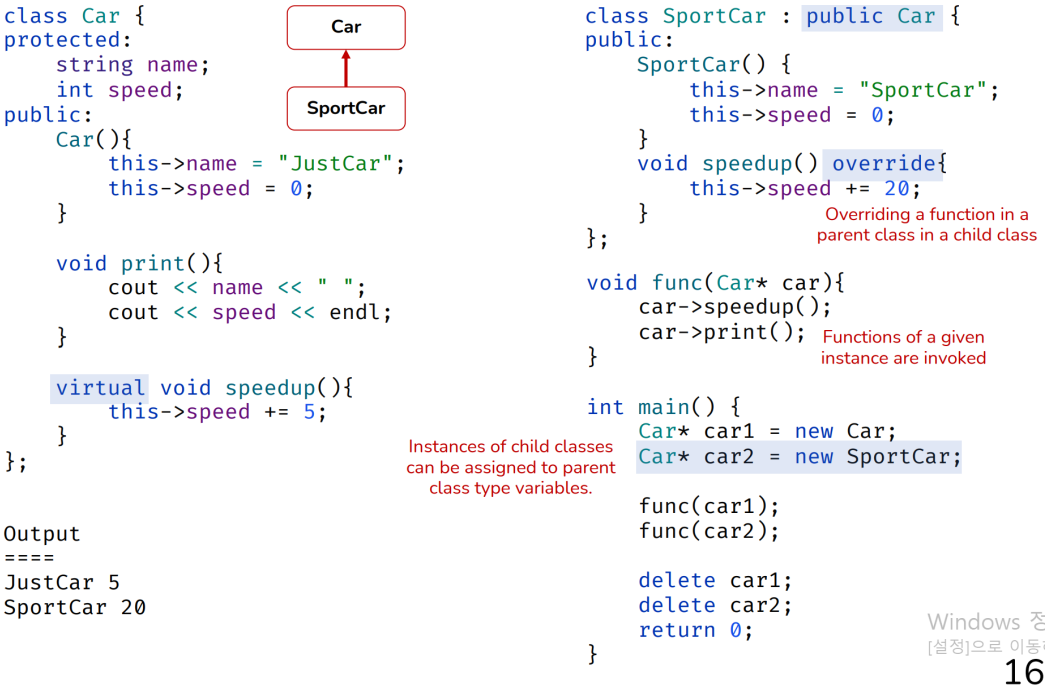
이때 부모 class의 메소드 중 virtual로 지정한 메소드에 대해서는, 해당 자식 class에서 override한 함수가 호출됨.

실제로 왜 이런 식으로 사용하는지는 더 찾아봐야 하지만, 아무튼 이런 식으로 부모 class 포인터로 자식 class에 접근할 수 있음.

즉, 아래와 같이 작성이 가능함. Car가 부모 class, SportCar가 자식 class임.

```
// ->로 SportCar가 상속받은 Car 멤버 사용 가능.
// virtual 함수에 대해서는 SportCar에서 override한 메소드가 호출됨.
Car* car2 = new SportCar;
```

아래는 그 예시임.



16

2.4. 연산자 오버로딩

2.4.1. 연산자 오버로딩

Definition 37 *연산자도 함수처럼 오버로딩하여 사용자 지정 연산자로 사용하는 것을 연산자 오버로딩(Operator Overloading)이라고 함.*

멤버 함수로 아래와 같이 operator 키워드와 연산자 오버로딩을 할 연산자를 작성함. 해당 class의 객체에 대해 해당 연산자는 정의된 작업을 수행함.

```
Point operator+(Point& p)
{
    ...
}
```

해당 class의 object a, b에 대해 a + b가 a.operator+(b)처럼 작동함.

아래는 그 예시임. 이 경우 P3 = P2 + P1이 마치 P3 = P2.operator+(P1)처럼 작동함.

```
class Point{
    private:
        double x;
        double y;
    public:
        Point(double x, double y)
        {
            this->x = x;
            this->y = y;
        }
        Point operator+(Point& p)   // 연산자 오버로딩
        {
            Point point(x + p.x, y + p.y);
            return point;
        }
        void show()
        {
            cout << x << " " << y << endl;
        }
};
```

2.4.2. []의 연산자 오버로딩

[]은 [] 내부의 값이 매개변수로 전달됨. 이때, []를 사용한 배열에 값을 대입하는 등의 연산을 수행하려면 반환 자료형을 reference로 지정해야 함.

(추후에 다시 정리)

아래는 그 예시임.

```
int& operator[](int idx)
{
    if(idx < 0 || idx >= n)
    {
        cout << "out of range" << endl;
        exit(1);
    }
    return array[idx];
```

```
}
```

2.5. 템플릿

2.5.1. 템플릿

Definition 38 *자료형에 따라 class를 생성할 수 있도록 하는 문법을 템플릿(Template)이라고 함. 아래와 같이 클래스 정의 위에 template 키워드를 사용해 지정할 수 있음. 이때 주로 T(Template)를 사용함. 이 T 위치에 추후에 지정할 자료형이 들어감.*

```
template <typename T>
class Point{
    ...
};
```

해당 class를 사용할 때는 항상 T 위치에 들어갈 자료형을 아래와 같이 지정해야 함.

```
Point<int> p1(3, 4);
```

2.5.2. 선언부와 구현부

멤버 함수의 선언부와 구현부를 나누어 작성할 때는 둘 다에 template <typename T>를 작성하고, 구현부의 class 식별자 오른쪽에 <T>를 작성해야 함.

아래는 그 예시임.

```
template <typename T>
class SimpleTemplate{
public:
    T func();
    ...
};

...

template <typename T>
T SimpleTemplate<T>::func()
{
    ...
}
```

2.6. 예외 처리

2.6.1. 예외 처리

프로그램 실행 중에 발생하는 상황에 대한 예외 처리. try, catch, throw 키워드 등을 사용함.

컴파일 시에 발생하는 문법적 오류는 예외 처리에서 취급하지 않음.

예외 처리를 제대로 하지 않으면 프로그램이 예기치 못하게 종료되거나 이상하게 작동할 수 있음.

2.6.2. 사용법

try 블록으로 예외를 검사할 범위를 지정하고, catch 블록으로 예외 감지 시 수행할 작업을 지정함. throw로 예외가 발생했음을 알리고 데이터를 전송할 수 있음.

catch 블록은 여러 개 작성할 수 있는데, throw로 보낸 데이터의 자료형과 가장 가까운 자료형을 가지는 catch 블록이 선택되어 실행됨.

호출한 함수에서 throw문을 만나면 데이터를 전송하면서 제어도 함께 넘어감. 즉, 해당 함수의 나머지 부분은 실행되지 않음.

해당되는 catch 블록의 코드가 실행된 이후에는 try 블록 다음으로 제어가 넘어가 프로그램이 정상적으로 계속 실행됨.

아래는 사용 예시임.

```
try
{
    func();
}
catch(type1 exception1)
{
    // handling
}
catch(type2 exception2)
{
    // handling
}

...

void func()
{
    if(an error)
    {
        throw exception1;
    }
    ...
}
```

2.6.3. exception class 사용하기

예외 처리를 위한 class를 정의하여 object를 throw하는 방법을 사용하기도 함.

이때 throw Exception() 등으로 객체를 생성하여 던지고, catch(Exception& ex) 등으로 받음. catch(Exception ex)로 받으면 값이 복사됨.

아래는 그 예시임.

```
class MyArray{
private:
    int *array;
    int n;
public:
    MyArray(int n){
        this->n = n;
        array = new int[n];
    }
    ~MyArray() {
        delete[] array;
    }
    int &operator[](int idx) {
        if (idx < 0 || idx >= n)
            throw Exception();
        return array[idx];
    }
};
```

```
class Exception{
public:
    void report(){
        cout << "exception report" << endl;
    }
};


int main(){
    MyArray array(10);
    try{
        array[100];
    } catch (Exception& ex) {
        ex.report();
    }
    return 0;
}
```

31

Part III

자료구조

1. 스택

1.1. 스택

1.1.1. 스택

Definition 39 *스택(Stack)은 LIFO(Last In First Out)를 따르는 선형적 자료구조임.*

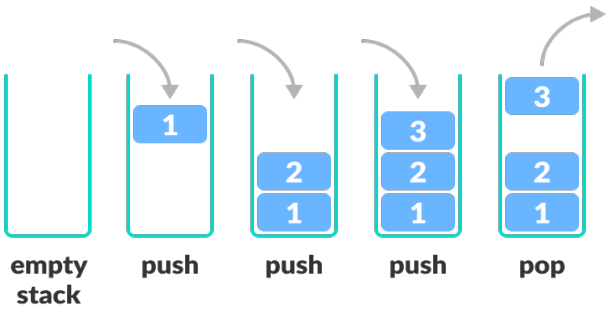
스택의 맨 위를 top, 아래를 bottom이라고 함. 기본적으로 top에서만 연산이 수행되고 bottom은 사용되지 않음.

대표 연산으로는 push(삽입)와 pop(추출)이 있음.

LIFO로 데이터를 다뤄야 할 때 스택을 사용함. 구현 자체는 어렵지 않은데 문제에서 특정 자료구조를 사용해야겠다는 아이디어를 떠올리는 것이 어려움. 문제를 많이 풀어봐야 함.

주로 함수 스택프레임, expression 평가, 미로찾기, 괄호 검사 등에 사용함.

LIFO는 라이포라고 읽음.



1.1.2. 스택 ADT

1. Data
LIFO를 따르는 데이터 집합.

2. Main Operations
push(x) : x를 top에 삽입.
pop() : top의 요소를 제거하고 그 값을 반환.

3. Auxiliary Operations
peek() : top의 요소 리턴. (삭제x)
empty() : stack이 비어있으면 true를 리턴하고, 비어있지 않으면 false를 리턴.
full() : stack이 꽉 차있으면 true를 리턴하고, 꽉 차 있지 않으면 false를 리턴.
size() : 요소의 개수를 리턴.

1.2. 구현

기본적인 방식은 수도코드를 짜 보고, 시간복잡도를 분석해 보고, 실제로 구현하는 것임.

1.2.1. 배열로 구현

top의 인덱스를 나타내는 top 변수와, 스택을 저장할 배열만 있으면 구현이 가능함.

구현이 간단하다는 장점이 있지만, 배열 길이를 넘어가는 길이의 데이터는 저장하지 못함. 동적 배열(vector)이나 linked list를 사용하여 단점 개선 가능.

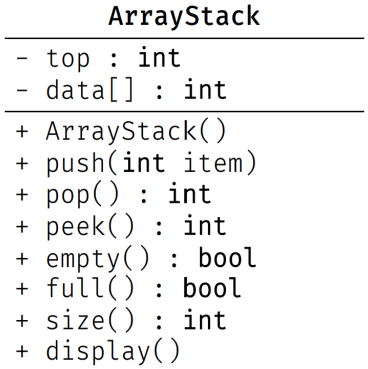
class로 데이터와 함수를 묶어서 정의하면 더 간편함. 객체를 생성하면 그 안에 데이터가 들어 있고, 데이터를 조작하기 위한 함수를 사용하는 것. 템플릿까지 추가하면 객체 선언 시에 자료형을 지정할 수 있으므로 생산성이 더 올라감.

1.2.2. class diagram

Definition 40 *class의 구현 이전에 class diagram이라는 diagram을 작성하여 그 형태를 확인할 수 있음. 수도코드와 동일한 기능을 함.*

class 이름, 멤버 변수 영역, 멤버 함수 영역을 작성함. 이때 멤버 옆에 붙이는 -는 private, +는 public, #은 protected를 뜻함. :로 반환 자료형도 명시함.

아래는 배열로 구현한 stack을 class로 만들기 위해 작성한 class diagram임.



1.2.3. 예시 : 사칙연산 수식 평가

스택을 이용해 괄호를 포함하는 사칙연산의 값을 계산할 수 있음.

사칙연산을 표현하는 방법은 infix, prefix, postfix가 있음. 각각 연산자가 중간에(2+3), 앞에(+23), 뒤에(23+) 오는 것을 뜻함. 일상에서는 infix를 사용하지만, prefix와 postfix는 우선순위가 이미 반영되어 있기 때문에 컴퓨터가 연산하기에는 더 편리함. 그래서 아래의 단계를 거쳐 사칙연산의 수식을 평가할 수 있음.

1. infix를 postfix로 변환.
2. postfix 수식을 평가.

2번에서는 다음과 같은 알고리즘을 사용함.
1. 피연산자이면 스택에 넣음.
2. 연산자이면 스택에서 피연산자를 가져와 연산을 한 후, 그 결과를 스택에 넣음. 결과적으로 마지막에 stack에 남은 값이 최종 값임.

1번에서는 다음과 같은 알고리즘을 사용함. 여기서 '출력'은 다른 메모리로의 출력을 의미함.
1. 피연산자이면 출력.
2. 연산자이면 뒤에 오는 연산자와의 우선순위 비교를 통해 출력하거나 스택에 넣음. 이때 왼쪽 괄호를 가장 낮은 우선순위의 연산자로 취급하고, 오른쪽 괄호를 발견하면 왼쪽 괄호까지 전부 출력하는 방식으로 처리함.

문자열의 길이에 따른 push와 pop의 횟수로 시간복잡도를 생각해 볼 수 있음. 문자열의 길이에 비례하는, n 정도의 복잡도를 가짐을 알 수 있음.

1.3. STL

1.3.1. stack STL

Definition 41 *<stack> 라이브러리를 사용함.*

아래와 같이 선언함.

std::stack<data_type> name;

stack STL에서 사용 가능한 메소드는 아래와 같음.

push(item) : item을 push함. item은 선언 시에 지정한 자료형으로 복사됨.
pop() : pop함. pop한 값은 반환하지 않고 삭제만 함.
top() : top에 있는 값을 선언 시에 지정한 자료형으로 반환함.
size() : 요소의 개수를 반환함.
empty() : stack이 비어 있으면 true, 차 있으면 false를 bool형으로 반환함.

pop()으로 pop을 하면 값을 삭제하기만 하므로, 값을 사용하려면 top()으로 확인한 뒤 pop()으로 삭제해야 함.

2. 큐

2.1. 큐

2.1.1. 큐

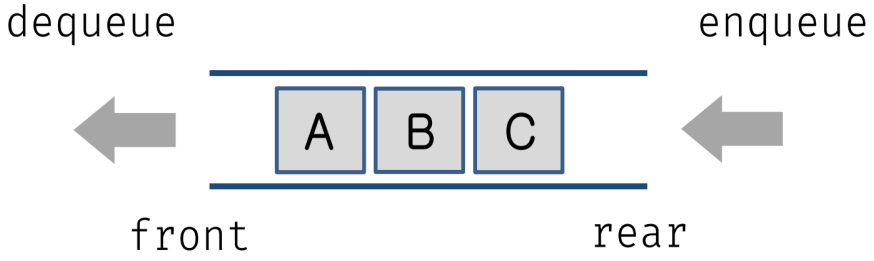
Definition 42 *큐(Queue)은 FIFO(First In First Out)를 따르는 선형적 자료구조임.*

큐의 맨 앞을 front, 맨 뒤를 rear라고 함. 기본적으로 데이터 삽입은 rear에서, 추출은 front에서 함.

대표 연산으로는 enqueue(삽입)와 dequeue(추출)가 있음.

FIFO로 데이터를 다뤄야 할 때 스택을 사용함. 즉, 순서에 맞게 처리하는 상황에 사용함.

주로 데이터 패킷, 서버 접근 등을 처리할 때 사용함.



2.1.2. 큐 ADT

1. Data
FIFO를 따르는 데이터 집합.

2. Main Operations
enqueue(x) : x를 rear에 삽입.
dequeue() : front의 값을 삭제하고 반환.

3. Auxiliary Operations
peek() : front의 요소 리턴. (삭제x)
empty() : 큐가 비어있으면 true를 리턴하고, 비어있지 않으면 false를 리턴.
full() : 큐가 꽉 차있으면 true를 리턴하고, 꽉 차 있지 않으면 false를 리턴.
size() : 요소의 개수를 리턴.

2.2. 구현

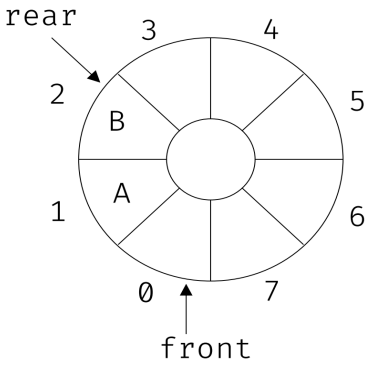
2.2.1. 배열로 구현

front의 인덱스를 나타내는 front 변수와, rear의 인덱스를 나타내는 rear 변수를 사용함. 이때 front 변수는 실제 데이터가 있는 위치보다 한 칸 앞으로 지정하기도 하는데, 구현상의 편의를 위해서임. 이렇게 사용하는 빈 공간을 dummy라고 함. front와 rear가 모두 실제 데이터 위치를 가리키도록 설정한 경우, front에서 추가한 데이터들과 rear에서 추가한 데이터들 사이에 비는 공간 하나가 생기게 되어 처리가 불편해짐.

일반적인 배열로 구현하면 front에서 데이터를 추출한 이후 나머지 데이터들을 옮겨 주거나, front를 가리키는 포인터를 하나 땡겨 줘야 하는데, 두 방식 모두 낭비임. 그래서 Circular Queue의 형태로 구현함.

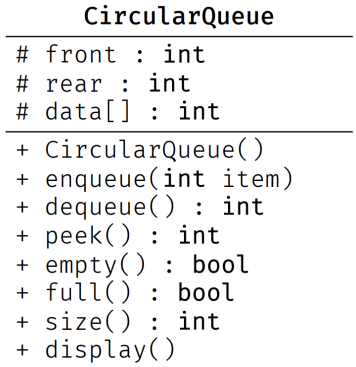
이때 특정 위치에는 인덱스를 전체 길이로 modular 연산을 해서 얻은 나머지로 접근함.

empty()와 full()은 front와 empty 값을 비교하여 구현할 수 있음.



2.2.2. class diagram

배열을 사용한 circular queue의 구현을 위한 class diagram은 아래와 같음.



2.2.3. 예시 : 미로 찾기

미로가 2차원 배열(벽은 1, 통로는 0)로 주어지면, 해당 미로가 해결될 수 있는지를 판단하는 문제. DFS 또는 BFS로 해결이 가능함.

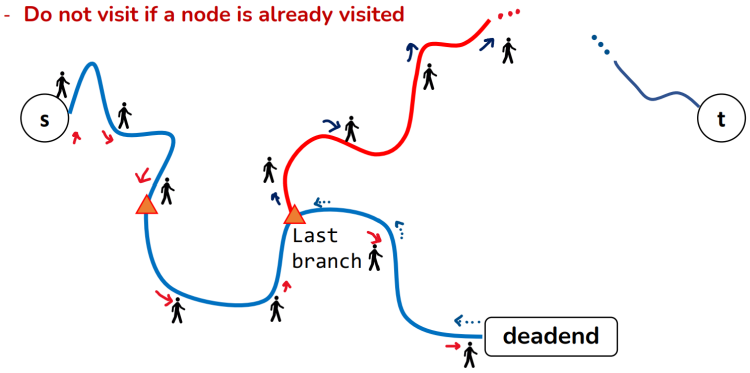
각 방식에 대한 시간복잡도는 스택과 큐에 좌표가 들어오고 나가는 횟수로 생각할 수 있음. nXm 미로에 대한 최악의 경우(벽이 없는 경우) nm번 좌표가 들어오고 나가므로, $\Theta(nm)$이라고 할 수 있음.

1. DFS(Depth First Search)
: 막다른 길까지 이동하고, 마지막 분기점으로 되돌아오는 방식. Stack 사용.

주위의 빈 칸을 검사하여 다음 이동 가능 좌표를 스택에 저장하고, 스택에서 좌표를 하나 빼서 해당 위치로 이동함. 갈림길을 만나서 이동 가능 좌표가 여러 개인 경우 모두 스택에 넣음.

막다른 길을 만날 때까지 이동하고, 막다른 길을 만나면 저장한 갈림길로 이동하여 다른 방향으로 이동함. 스택에 갈림길에 대한 좌표까지 저장했으므로 단순히 스택에서 추출하기만 하면 마지막 갈림길로 이동하는 것.

이미 지나온 경로는 다시 가지 않도록 처리함.

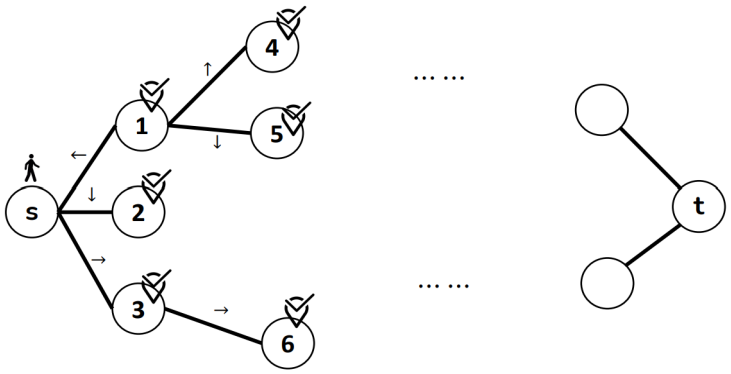


2. BFS(Breadth First Search)
: 가능한 모든 경로에 대해 각각 한 칸씩 전부 가는 방식. Queue 사용.

주위의 빈 칸을 검사하여 다음 이동 가능 좌표를 큐에 저장하고, 큐에서 좌표를 하나 빼서 해당 위치로 이동함. 갈림길을 만나서 이동 가능 좌표가 여러 개인 경우 모두 큐에 넣음.

각 분기마다 한 칸씩 전부 이동하므로, 막다른 길에 다다른 분기에 대한 좌표는 더 이상 큐에 들어오지 않아 자연스럽게 처리됨.

이때 각 계층에 있는 원소의 개수를 너비라고 함.



2.3. STL

2.3.1. Queue STL

Definition 43 *<queue> 라이브러리를 사용함.*

아래와 같이 선언함.

std::queue<data_type> name;

queue STL에서 사용 가능한 메소드는 아래와 같음.

push(item) : item을 enqueue함. item은 선언 시에 지정한 자료형으로 복사됨.
pop() : dequeue함. pop한 값은 반환하지 않고 삭제만 함.
front() : peek함. 이때 값은 선언 시에 지정한 자료형으로 반환함.
size() : 요소의 개수를 반환함.
empty() : stack이 비어 있으면 true, 차 있으면 false를 bool형으로 반환함.

pop()으로 dequeue를 하면 값을 삭제하기만 하므로, 값을 사용하려면 front()으로 확인한 뒤 pop()으로 삭제해야 함.

3. Deque

3.1. Deque

3.1.1. Deque

Definition 44 *Deque는 front, rear 모두에서 삽입과 추출이 가능한 큐임.*

front에서의 삽입, 추출, 삭제를 addFront, getFront, deleteFront라고 함. rear에서의 삽입, 추출, 삭제를 addRear, getRear, deletetRear라고 함.

특정 연산만 사용하여 deque를 스택과 큐처럼 사용할 수 있음. 어떤 연산을 해도 자료구조가 유지될 때 해당 연산이 자료구조에 대해 닫혀 있다고 함. 자료구조를 유지하려면 적절한 연산만을 사용해야 함.

3.1.2. ADT

1. Data
front와 rear 모두에서의 접근을 허용하는 데이터 집합.

2. Main Operations
addFront(x) : x를 front에 삽입.
deleteFront() : front의 값을 삭제하고 반환.
addRear(x) : x를 rear에 삽입.
deletetRear() : rear의 값을 삭제하고 반환.

3. Auxiliary Operations
getFront() : front의 요소 리턴. (삭제x)
getRear() : rear의 요소 리턴. (삭제x)
empty() : deque가 비어있으면 true를 리턴하고, 비어있지 않으면 false를 리턴.
full() : deque가 꽉 차있으면 true를 리턴하고, 꽉 차 있지 않으면 false를 리턴.
size() : 요소의 개수를 리턴.

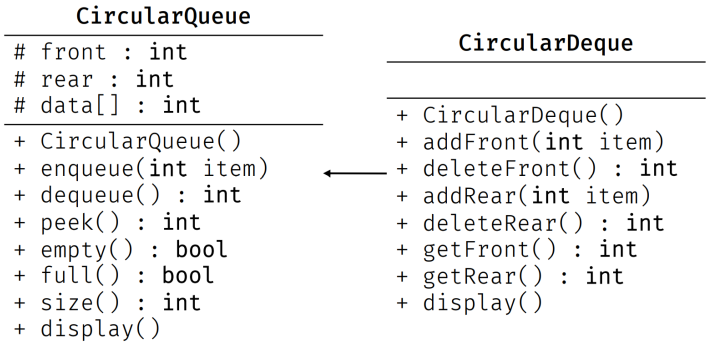
3.2. 구현

3.2.1. 배열로 구현

배열로 구현할 경우 큐에서처럼 circular하게 만들 수 있음. 큐를 구현할 때 사용한 class를 상속하고 연산자를 추가하여 구현하는 것이 편리함.

3.2.2. class diagram

배열을 사용한 circular deque의 구현을 위한 class diagram은 아래와 같음. circular queue의 class를 상속받아 deque 연산 구현 시에 그대로 사용하면 편리함.



3.3. STL

3.3.1. Deque STL

Definition 45 *<deque> 라이브러리를 사용함.*

아래와 같이 선언함.

std::deque<data_type> name;

deque STL에서 사용 가능한 메소드는 아래와 같음.

사이즈 관련 메소드들 : size(), max_size(), resize(), empty(), shrink_to_fit()
요소 접근 관련 메소드들 : operator[], at(), front(), back()
요소 삽입/추출 메소드들 : assign(), push_back(), push_front(), pop_back(), pop_front(), insert()

assign()은 새로운 값으로 deque를 새로 작성함. 이때 반복된 값으로 삽입할 수도 있고, 다른 deque의 값을 가져올 수도 있는데 정확한 내용은 따로 찾아보자. (ex. assign(4, 3)이면 3을 4개 삽입함.)

insert()는 중간에 값을 삽입함. (ex. insert(4, 3)이면 4 위치에 3을 삽입함.)

요소를 추출하면 값을 삭제하기만 하므로, 값을 사용하려면 우선 접근해서 값을 확인한 뒤 추출해서 삭제해야 함.

4. Linked list

4.1. Linked list

4.1.1. Linked list

Definition 46 *데이터 요소들을 노드(node)로 묶고 서로 연결하여 구성된 자료구조를 Linked list(연결 리스트)라고 함.*

Linked list에는 singly linked list(SLL), doubly linked list(DLL), circular linked list(CLL), circular doubly linked list(CDLL) 등 다양한 종류가 있음.

노드는 data field와 pointer field로 구성되어 있음. pointer는 다음 노드를 가리킴.

c에서는 구조체와 malloc() 등으로 Linked list를 구현했지만, c++에서는 아래와 같이 class와 new를 사용할 수 있음.

```
Student* test_ptr = new Student(202, "james");
cout << test_ptr->ID; // 202
```

일반적으로 맨 앞 노드 또는 맨 뒤 노드를 가리키는 head/front 포인터와 tail/rear 포인터를 사용하고, 노드를 가리키지 않는 포인터에는 nullptr값을 넣음.

Linked list는 중간 데이터를 쉽게 수정할 수 있고, 동적으로 메모리를 계속 할당할 수 있음. 다만 포인터를 위한 메모리 공간을 추가로 사용해야 하고, 계속해서 내부적으로 많은 연산이 수행되는 동적할당을 해야 함. 시간복잡도의 측면에서 보면 동적할당은 계수가 큰 상수인 것.

생성 시에 생성자로 값을 바로 지정하면 편리함.

4.1.2. 자기참조 class

Definition 47 *자기참조 구조체와 동일한 형태로 자기참조 class를 사용할 수 있음. Linked list에는 다음 노드의 주소를 저장할 수 있어야 하므로 아래와 같이 자기참조 class를 사용함.*

```
class LinkedNode
{
    int data;
    LinkedNode* next_ptr;
}
```

4.2. Linked list를 사용한 구현

4.2.1. Stack

배열로 구현할 때는 변수 하나에 top에 대한 인덱스를 저장했다면, Linked list로 구현할 때는 포인터로 top을 가리킴.

stack이 비어있는 경우와 비어있지 않은 경우로 나누어 push, pop 등을 구현할 수 있음.

4.2.2. Queue

배열로 구현할 때는 Circular하다고 가정하고 rear와 front를 가리킬 변수를 사용했다면, Linked list로 구현할 때는 rear와 front를 가리킬 포인터 두 개를 사용함. 이때 front->rear 방향으로 구현해야 enqueu와 dequeue의 구현이 간단함.

Queue가 비어있는 경우와 비어있지 않은 경우로 나누어 enqueue, dequeue 등을 구현할 수 있음.

5. List

5.1. List

5.1.1. List

Definition 48 *List는 임의의 위치에서 삽입과 추출이 가능한 선형적 자료구조임.*

데이터 접근 위치를 0으로 시작하는 인덱스 등으로 표기함.

5.1.2. List ADT

1. Data
임의의 위치에서의 삽입과 추출이 필요한 순서 있는 데이터 집합.

2. Main Operations
insert(i, item): i번째 위치에 item 삽입.
remove(i): i번째 위치의 데이터 삭제.
get(i): i번째 위치의 데이터 반환.
replace(i, item): i번째 위치의 데이터를 item으로 대체.
find(item): item을 값으로 가지는 위치 반환.

3. Auxiliary Operations
empty() : List가 비어있으면 true를 리턴하고, 비어있지 않으면 false를 리턴.

full() : List가 꽉 차있으면 true를 리턴하고, 꽉 차 있지 않으면 false를 리턴.
size() : 요소의 개수를 리턴.

5.2. 구현

5.2.1. 배열로 구현 : Array List

구현이 간단하지만 동적으로 공간을 사용하지 못함.

MLS(MAX_LIST_SIZE)를 정의하여 배열 크기를 지정하고, length 변수를 사용하여 구현이 가능함.

insert(), remove()로 중간에 삽입/삭제할 때는 요소들을 한 칸씩 움직여줘야 함. List에 접근할 때 잘못된 인덱스를 사용했을 경우에 대해 예외처리를 하면 좋음.

find()는 관습적으로 가장 앞쪽의 요소를 반환함. 일치하는 요소가 없는 경우에 대해 예외처리를 하면 좋음.

5.2.2. singly linked list로 구현

동적으로 공간을 사용할 수 있지만 포인터의 사용으로 인한 추가 공간이 필요함.

일반적인 singly linked list로 구현이 가능함. head ptr 대신 head node를 만들어 link 필드만 사용하는 것으로 구현할 수도 있음. head node는 인덱스를 -1로 생각함. 노드 하나에 대한 연산과, linked list에 대한 연산을 각각 클래스로 구현할 수 있음.

linked list의 연산은 search와 task로 그 작업이 구분됨. search로 순회하여 해당 인덱스의 위치로 이동하고, task로 삽입/삭제를 하는 것.

5.2.3. circular linked list로 구현

rear node를 가리키는 rear ptr를 사용하고, rear node는 front node를 가리키게 한 것.

front, rear에서의 삽입/삭제가 빨라짐. 물론 이렇게 구현하면 rear에서의 삭제는 여전히 느림.

5.2.4. doubly linked list로 구현

링크를 다음 노드, 이전 노드 모두와 연결하여 좌우 이동을 빠르게 함. 다만 포인터 공간을 추가로 사용해야 함.

circular doubly linked list로도 구현이 가능함. front의 이전 노드를 rear, rear의 다음 노드를 front로 생각한 것. 이 경우에 circular linked list처럼 rear ptr 하나를 사용하여 rear node를 가리킴.

5.2.5. array list vs linked list

front에서 삽입/삭제하는 것은 linked list가 array list보다 빠르고, rear에서 삽입/삭제하는 것은 arrary list가 linked list보다 빠름.

get(), replace()만 할 것이라면 array list가 유리함.

데이터를 계속해서 삽입/삭제해야 되는 경우 linked list가 유리함. 다만 포인터만큼의 공간을 추가로 사용해야 함.

어떤 형태의 자료구조를 사용할 것인지는, 데이터의 특성에 따라 어떤 연산이 사용되고, 각 연산들의 시간/공간복잡도를 생각해 결정함.

5.3. STL

5.3.1. List STL

Definition 49 *<list> 라이브러리를 사용함.*

아래와 같이 선언함.

```
std::list<data_type> list_name;
```

search는 iterator로 함.

task 메소드들 : clear, insert, erase, push_back, pop_back, push_front, pop_front, emplace, emplace_back, emplace_front

list 라이브러리는 기본적으로 DLL(doubly linked list)로 되어 있음. SLL(singly linked list)를 사용하려면 std::forward_list를 사용해야 함.

5.3.2. Iterator

Definition 50 *STL에서 iterator란, 자료구조를 순회하여 특정 위치의 요소를 찾아주는(search) 것을 말함. iterator는 해당 노드를 가리키는 포인터와 유사하게 기능하고, 실제로 포인터처럼 사용이 가능함.*

아래와 같이 선언하고 사용함. begin() 메소드로 list의 시작 주소를 전달해줘야 함.

```
std::list<data_type> list_name;
list<data_type>::iterator iterator_name = list_name.begin();
list_name.insert(iterator_name, value);
```

*증감 연산자가 오버로딩되어 있어, 아래와 같이 iterator 값을 조정하여 특정 노드를 가리킬 수 있음. 이때 전위 증감 연산자를 사용해야 함. (후위 증감 연산자는 오버로딩되어 있지 않음.) * 연산자 또한 오버로딩되어 있어서, iterator가 가리키는 값을 직접 사용할 수 있음.*

```
list<data_type>::iterator it;
for(it = list_name.begin(); it != list_name.end(); ++it)
{
    cout << *it << ' ';
}
```

List에서는 search와 task가 분리되어 있는데, search는 iterator를 사용하고 task는 STL의 메소드들을 사용함. 그래서 list STL에서 push/pop front/back은 그냥 사용이 가능하지만 중간 지점에서의 삽입/삭제는 iterator를 사용해야 함.

iterator는 인덱스라기보다는 포인터처럼 기능함. 새로운 요소를 insert할 때 iterator가 가리키는 위치에 요소가 들어가지만, 요소 삽입 이후 iterator는 새로 추가된 요소가 아닌 원래 가리키던 요소를 가리킴.

erase는 요소 삭제 후 바로 다음 위치의 노드 주소를 반환함. 즉, it = mylist.erase(it);와 같이 사용이 가능함.

6. 이진 트리

6.1. 트리

6.1.1. 트리

Definition 51 *트리(Tree)는 부모-자식 관계의 노드들로 이루어져 계층적 구조를 나타내는 비선형적 자료구조임.*

트리는 root에서 어떤 노드까지의 경로가 하나여야 함. 즉, 각 노드는 하나의 부모만을 가져야 함. 경로가 여러 개이면 그래프라고 함.

6.1.2. 트리 관련 용어

1. 기본 용어
노드(node) : 트리를 구성하는 기본 요소.
간선 : 노드와 노드를 연결하는 선.

서브트리(subtree) : 한 노드와 그 자손들로 이뤄진 하위 트리.
노드의 차수(degree) : 노드의 자식 노드 개수.
노드의 깊이(depth) : root에서 해당 노드까지의 간선 개수.
노드의 레벨(level) : 트리의 계층 번호. depth+1한 값.
노드의 높이(height) : 노드에서 가장 깊은 leaf까지 간선의 개수
트리의 높이 : root 노드의 높이. 구현의 편의성을 위해 주로 빈 트리의 높이는 -1로 함.

2. 자식 유무에 따른 구분
단말 노드(terminal/leaf/external node) : 자식이 없는 노드
비단말 노드(non-terminal/non-leaf/internal/branch node) : 자식이 있는 노드

3. 위치 기준 가족관계
조부모(grandparent), 부모(parent), 형제(sibling), 자식(child), 손주(grandchild) 등의 관계 존재.

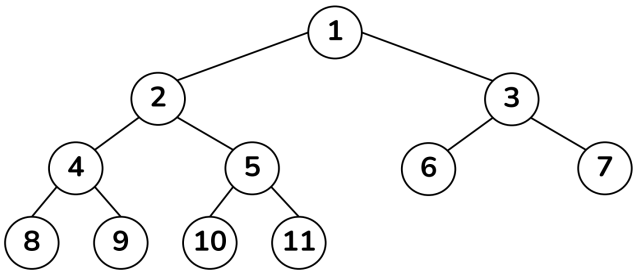
6.1.3. 트리의 종류

1. Generic tree (non-binary tree)
: 각 노드가 임의의 개수만큼의 자식 노드를 가질 수 있는 트리.
데이터베이스 등에서 사용하는 자료구조로, 본 수업에서는 다루지 않음.

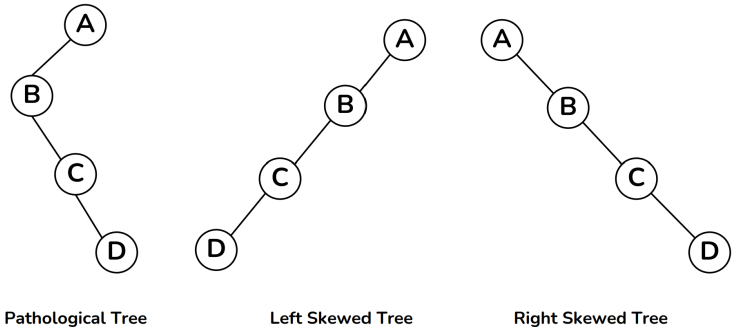
2. 이진 트리(binary tree)
: 각 노드가 최대 2개의 서브트리를 가질 수 있는 트리.
한 노드의 두 자식 노드는 순서가 존재하여 왼쪽(left), 오른쪽(right)으로 구분됨.
이진 트리에는 포화 이진 트리, 완전 이진 트리, degenerate 트리 등이 있음.

3. 포화 이진 트리(full binary tree)
: 트리의 각 레벨에 노드가 완전히 꽉 차 있는 트리. 이때 각 노드의 번호는 위에서 아래로, 왼쪽에서 오른쪽으로 매김.
2의 거듭제곱 만큼의 노드들을 가짐.

4. 완전 이진 트리(complete binary tree)
: 레벨이 l일 때, 레벨 1 l-1까지는 노드가 꽉 차있고, 마지막 레벨 l에서는 노드가 왼쪽에서 오른쪽으로 순서대로 채워져 있는 트리.



5. degenerate tree
: 노드가 n개 있을 때 높이가 n-1인 트리. 즉, 일자 형태의 이진 트리.
왼쪽으로만 내려가는 것은 left skewed tree, 오른쪽으로만 내려가는 것은 right skewed tree, 중간에 방향이 바뀌는 것은 pathological tree라고 함.
list로 볼 수도 있음. 즉, 트리는 list를 포함하는 개념임.



6.1.4. 이진 트리의 성질

실제로 이진 트리의 성질은 굉장히 다양하지만, 대표적으로 자주 사용되는 것들만 정리함.

1. 노드의 개수가 n개이면 간선의 개수는 n-1개임.
root를 제외하고는 전부 부모가 하나씩 존재하므로 당연함.

2. 이진 트리의 높이가 h일 때 노드의 개수는 $h+1 \sim 2^{h+1}-1$개임.
degenerate tree일 때 개수가 최소이고, complete binary tree일 때 개수가 최대이므로 당연함.

3. 이진 트리에 노드가 n개 있을 때 높이는 $n-1 \sim \lceil \log_2(n+1) - 1 \rceil$임.
2번을 정리하면 당연함. 이때 높이는 자연수이므로 올림을 함.

6.1.5. 이진 트리 ADT

이진 트리는 그 자체로 사용한다기보다는 다른 자료구조의 기본 개념으로 사용함.

1. Data
이진 트리로 구성할 수 있는 데이터 집합.

2. Operations
empty(): 이진 트리가 비어있는지 체크.
getRoot(): 이진 트리의 루트노드를 반환.
getCount(): 이진 트리의 노드 개수를 반환.
getHeight(): 이진 트리의 높이 반환.
getLeafCount(): 이진 트리의 leaf 노드의 개수 반환.

값을 insert, remove하는 것은 이진 탐색 트리에서 다룸.

6.2. 순회

6.2.1. 순회

Definition 52 *자료구조 내에서 데이터를 특정 순서대로 방문하는 것을 순회(traversal)라고 함.*

선형적인 자료구조에서는 순회의 순서가 명확하지만, 비선형적 자료구조에서는 그렇지 않기 때문에 여러 가지 방식으로 순회가 가능함.

이진 트리의 순회 방법으로는 작업 순서에 따라 전위/중위/후위 순회와 레벨 순회가 있음.

6.2.2. 전위/중위/후위 순회

Definition 53 *트리를 재귀적으로 순회할 수 있는데, 이때 작업과 재귀 호출의 실행 순서에 따라 전위(pre-order)/중위(in-order)/후위(post-order) 순회로 나눌 수 있음.*

이를 pseudo-code로 나타내면 아래와 같음.

1. 전위 순회

```
def preorder(node) :
    if !(node == NULL) :
        # 작업 수행
        preorder(node->left)
        preorder(node->right)
```

2. 중위 순회

```
def inorder(node) :
    if !(node == NULL) :
        inorder(node->left)
        # 작업 수행
        inorder(node->right)
```

3. 후위 순회

```
def postorder(node) :
    if !(node == NULL) :
        postorder(node->left)
        postorder(node->right)
        # 작업 수행
```

각 호출을 function call stack에 나타내 볼 수도 있음.

6.2.3. 레벨 순회

Definition 54 *레벨(level) 순서로 순회하는 것을 레벨(level-order) 순회라고 함. 루트를 시작점으로 하는 BFS를 통해 레벨 순회를 구현할 수 있음.*

이를 pseudo-code로 나타내면 아래와 같음.

```
def levelorder() :
    queue q
    q.enqueue(root)
    while !q.empty() :
        cur = q.dequeue()
        if cur != NULL :
            # 작업 수행
            q.enqueue(cur->left)
            q.enqueue(cur->right)
```

물론 NULL이면 아예 큐에 넣지 않도록 하여 연산량을 줄일 수도 있음.

6.3. 구현

이진 트리의 구현 방법. 이진 트리는 기본적으로 포인터로 구현하고, 힙 등에서는 배열로 구현함.

6.3.1. 배열로 구현

1. 구현 방법
각 노드에 번호를 붙이고, 해당 번호에 해당하는 인덱스에 노드의 데이터를 저장함. 이때 구현의 편의성을 위해 배열의 0번째를 사용하지 않고 1번째부터 사용함. 번호는 순서가 아니라 위치에 따라 부여함. 예를 들어, 5번째에 값이 존재하지 않아도 위치상 적절하다면 값을 6에 지정함.

i번 노드의 부모 노드 번호는 $\lfloor i/2 \rfloor$이고, left 자식 노드 번호는 $2i$, right 자식 노드 번호는 $2i+1$임. 이를 이용해 간단히 구현할 수 있음.

배열로 구현하는 것도 binary heap 등에서 유용하게 사용됨.

2. 장단점
트리를 쉽고 간단하게 구현할 수 있지만, 배열 크기 이상으로 트리를 만들 수는 없음. 또한 트리의 높이에 비해 노드 수가 적은 경우 빈 노드 자리 때문에 메모리가 낭비됨.

6.3.2. linked list로 구현

1. 구현 방법
노드별로 두 개의 포인터를 사용하고 각각 left/right 자식 노드를 가리키도록 하여 구현할 수 있음.

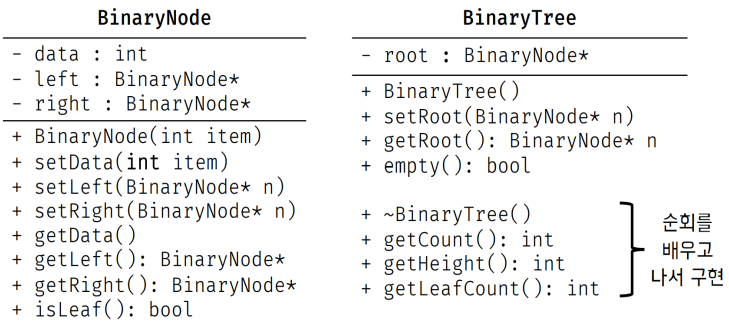
2. 장단점
노드를 (메모리 공간 내에서) 제한 없이 추가할 수 있고, 필요한 만큼 할당해 사용하므로 빈 노드 자리에 의한 메모리 낭비가 없음. 다만 추가적인 포인터 사용에 의한 메모리 낭비가 존재하고, 동적 할당 연산 자체에 대한 부담이 있음.

6.3.3. class diagram

linked list를 사용한 트리의 구현을 위한 class diagram은 아래와 같음. 메소드가 많지만 대부분이 setter와 getter임.

node에 대한 class의 필드로는 데이터를 저장하는 부분과 두 자식에 대한 포인터가 있고, 메소드로는 생성자, getter/setter, leaf 노드인지를 반환하는 메소드 등이 있음.

이진 트리에 대한 class의 필드로는 루트 노드를 가리키는 포인터가 있고, 메소드로는 생성자와 getter/setter, 이진 트리가 비어있는지를 반환하는 메소드, 순회를 이용한 메소드 등이 있음.



6.3.4. 순회를 이용하는 메소드의 구현

이진 트리에 대한 class에서 트리를 순회하여 값을 반환하는 메소드들로는 소멸자, 노드 개수를 구하는 메소드, 트리의 높이를 구하는 메소드, leaf 노드의 개수를 구하는 메소드 등이 있음.

1. 소멸자
후위 순회를 사용하여 각 노드를 동적 해제할 수 있음. 재귀 호출이 종료된 후에 동적 해제를 해야 하므로 후위 순회로 처리해야 함.

2. 노드 개수를 구하는 메소드
재귀적으로 순회하며 left/right 자식 노드의 값을 반환받고 1을 더하여 반환하는 방식으로 구현할 수 있음. 당연하게도 전위/중위/후위 순회 중 어떤 방식을 사용하더라도 상관 없음.
물론 이런 메소드를 쓰기보다는 count 변수를 사용하여 노드가 insert/delete될 때마다 개수를 관리하는 것이 효율적임.

3. 트리의 높이를 구하는 메소드
left/right 자식 노드의 높이 중 큰 것에 1을 더한 값이 해당 노드의 높이임. 루트 노드에 대해 해당 과정을 수행하여 트리의 높이를 구할 수 있음. null에 대한 높이는 -1로 하여 구현하면 편리함.
left/right 값이 모두 필요하기 때문에 후위 순회를 사용해야 함.

4. leaf 노드의 개수를 구하는 메소드
노드 개수를 구하는 메소드처럼 leaf 노드의 개수를 셀 수 있음.
당연하게도 전위/중위/후위 순회 중 어떤 방식을 사용하더라도 상관 없음.

6.3.5. API vs worker

class 구현 시에는 user에게 보여지는 public 메소드와, 실제 작업을 수행하는 private 메소드를 분리하여 정의하기도 함. 이때 전자를 API, 후자를 worker라고 함.

아래와 같이 메소드 오버로딩으로 구현하면 편리함.

```
public:
    void preorder(){
        cout << "preorder: ";
        preorder(this->root);
        cout << endl;
    }
private:
    void preorder(BinaryNode* node){
        if(node != nullptr)
        {
            cout << node->getData() << " ";
            preorder(node->getLeft());
            preorder(node->getRight());
        }
    }
```

7. BST

7.1. BST

7.1.1. 탐색

Definition 55 *자료구조 내에서 원하는 정보를 효율적으로 찾는 것을 탐색(search)이라고 함.*

단순한 선형 자료구조(배열 등)에서의 탐색은 최악의 경우 $\Theta(n)$의 시간복잡도를 가짐. BST를 사용하여 데이터를 저장하면 $\Theta(n)$보다 더 빠르게 탐색이 가능함.

7.1.2. key와 value

Definition 56 *필요한 경우 트리에서 노드는 key와 value를 가짐. key는 노드끼리의 구분에 사용되는 식별자이고, value는 key에 대응되는 데이터임.*

BST에서의 탐색은 key를 기준으로 함.

7.1.3. 이진 탐색 트리

Definition 57 *아래의 규칙을 만족시키는 이진 트리를 이진 탐색 트리(binary search tree, BST)라고 함.*

1. 모든 노드는 유일한 key를 가지고, key는 순서가 존재해 비교가 가능함.
2. 왼쪽 서브트리의 모든 노드의 key들은 루트의 key보다 작음.
3. 오른쪽 서브트리의 모든 노드의 key들은 루트의 key보다 큼.
4. 왼쪽/오른쪽 서브트리는 재귀적으로 BST임.

key로 정수/실수를 사용할 수 있고, 문자열 또한 사전순으로 비교하는 등의 방식으로 사용할 수 있음.

참고로 유일하지 않은 key들을 가지는 트리도 존재는 함.

7.1.4. BST ADT

1. Data
BST로 구성할 수 있는 데이터 집합.

2. Operatons
search(key): 해당 key를 가지는 노드나 그 데이터를 반환.
insert(key, value): BST를 유지하며 해당 key/value를 가지는 노드를 삽입하고, 해당 key가 이미 존재한다면 값을 value로 갱신.
remove(key): BST를 유지하며 해당 key를 가지는 노드를 삭제.

가장 많이 수행되는 연산이 search이므로 search는 가능한 한 효율적으로 작성하는 것이 좋음.

BST의 연산에서는 1. BST가 유지되는지 2. $\Theta(n)$(선형 자료구조)보다 효율적인지 확인해야 함.

7.1.5. self-balancing BST

Definition 58 *아래의 조건을 만족하는 트리를 균형 잡힌 트리(balanced tree)라고 함.*

1. |왼쪽 서브트리 높이 - 오른쪽 서브트리 높이| ≤ 1
2. 서브트리가 재귀적으로 균형 잡힌 트리임.

후술하겠지만 BST는 최선의 경우 $\Theta(1)$, 평균적 경우 $\Theta(logn)$, 최악의 경우 $\Theta(n)$인데, self-balancing BST는 BST를 개선하여 최악의 경우에도 $\Theta(logn)$이도록 하는 트리임.

self-balancing BST는 노드에 변화가 있을 때마다 높이를 검사하여 balanced tree가 아닌 경우 노드 구조를 회전시켜 높이를 맞추는 작업을 수행함.

self-balancing tree로는 대표적으로 red-black tree가 있음. c++의 set/map STL은 내부적으로 red-black tree로 구현되어 있음.

7.2. 구현

7.2.1. search 연산의 구현

탐색 key(찾으려는 노드의 key)와 현재 노드를 비교하고 그 결과에 따라 반환/왼쪽 탐색/오른쪽 탐색을 수행하여 search 연산을 구현할 수 있음. 해당 key의 노드가 존재하지 않으면 결과적으로 NULL을 반환함.

아래는 search 연산에 대한 pseudo-code임.

```
def searchBST(node, key) :
    if node->key == key || node == NULL :
        return node # 탐색 성공/실패 위치
    else if node->key > key :
        return searchBST(node->left, key)
    else # if node->key < key :
        return searchBST(node->right, key)
```

7.2.2. insert 연산의 구현

search 연산을 수행하고, search 연산이 실패한 지점에 해당 key/value의 노드를 삽입함. 만약 해당 key의 노드가 존재하면 search된 노드의 값을 value로 갱신함. 트리가 비어 있는 경우는 따로 처리해줘야 함.

아래는 insert 연산에 대한 pseudo-code임.

```
def insertBST(node, key, value) :
    if node->key == key :
        node->value = value;
```

```
    else if node->key > key :
        if node->left == NULL :
            node->left = Node(key, value, NULL, NULL);
        else :
            insertBST(node->left, key, value)
    else # if node->key < key :
        if node->right == NULL :
            node->right = Node(key, value, NULL, NULL);
        else :
            insertBST(node->right, key, value)
```

7.2.3. remove 연산의 구현

remove할 노드를 찾는 것은 search로 수행하는 것으로 동일하지만, 삭제하는 과정에서 remove 연산은 총 3가지 경우를 고려해야 함.

1. 삭제하려는 노드가 leaf인 경우
삭제하려는 노드의 부모노드에서 NULL 처리를 해 주고, 해당 노드를 동적해제함. 이때 삭제하려는 노드가 루트이면 부모가 없으므로 동적해제만 함.

2. 삭제하려는 노드가 왼쪽/오른쪽 서브트리 중 하나만 자식으로 갖는 경우
삭제하려는 노드의 서브트리의 루트를 삭제하려는 노드의 부모 노드에 붙이고, 해당 노드를 동적해제함. 이때 삭제하려는 노드가 루트이면 부모가 없으므로 서브트리의 루트를 루트로 함.

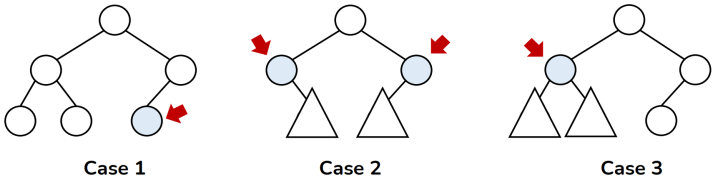
3. 삭제하려는 노드가 왼쪽/오른쪽 서브트리 모두를 자식으로 갖는 경우
삭제하려는 노드를 대체할 후계자(successor)를 찾아서 삭제하려는 노드 자리에 넣어줘야 함. 후계자로는 1. 왼쪽 서브트리에서 key가 가장 큰 노드 2. 오른쪽 서브트리에서 key가 가장 작은 노드 중 하나를 선택하면 됨. 여기서는 2번으로 구현함. 후계자의 key와 value를 삭제하려는 노드에 복사하고 후계자 노드는 동적해제함.

이때 트리에서 가장 큰 노드는 트리에서 가장 오른쪽에 있는 노드이고, 가장 작은 노드는 트리에서 가장 왼쪽에 있는 노드임. 그렇지 않다면 BST의 구조가 깨짐.

remove 연산을 pseudo-code로 나타내면 아래와 같음.

```
def removeBST(node, parentNode, key) :
    if node == NULL :
        return
    if node->key > key :
        return removeBST(node->left, node, key)
    else if node->key < key :
        return removeBST(node->right, node, key)
    else # node->key == key :
        if node has right/left children :
            # 오른쪽 서브트리에서 가장 왼쪽 노드를 후계자로 node를 대체
            # 후계자 노드 삭제. 이때 후계자가 오른쪽 자식을 가지고 있을 수 있으므로
            # removeBST()로 삭제해 줘야 함.
        else if node only has right children ;
            # 오른쪽 서브트리의 root로 node를 대체
        else if node only has left children ;
            # 왼쪽 서브트리의 root로 node를 대체
        else :
            # 부모에 NULL을 넣고 node를 삭제
```

removeBST(worker)는 동적해제 외의 모든 작업을 수행한 이후 삭제할 노드를 반환하도록 하고, 반환받은 노드를 API에 해당하는 메소드에서 동적해제하도록 구현할 수 있음.



7.2.4. search/insert/remove의 시간복잡도

1. search의 시간복잡도
최선의 경우는 루트에 탐색 key가 있는 경우로, $\Theta(1)$임.
최악의 경우는 탐색 key가 트리의 가장 깊은 노드에 있는 경우로, 높이만큼 이동하게 되므로 $\Theta(h)$임. 트리에 노드가 n개 있을 때 degenerate tree이면 높이가 최대인데, 이 경우 $\Theta(n)$임.

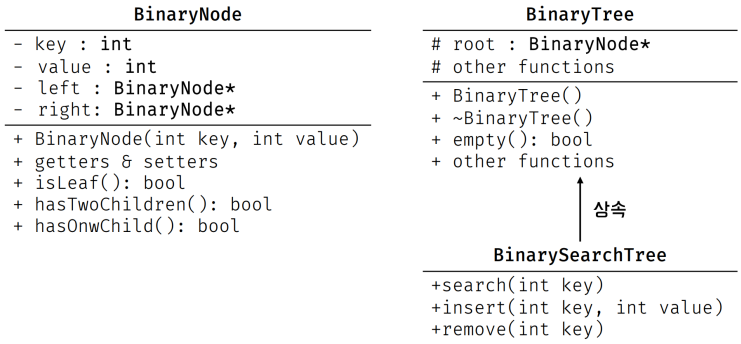
2. insert/remove의 시간복잡도
insert에서는 탐색 이후 삽입만 수행하면 되므로 search와 복잡도가 같음. remove에서는 해당 노드를 찾는 데에는 동일한 복잡도를 가지지만, 이후 해당 노드의 유형에 따라 복잡도가 달라짐. 후계자를 사용하지 않으면 search/insert와 동일한 복잡도를 가지지만, 후계자를 사용하는 경우 해당 후계자를 찾고 remove를 재귀적으로 수행하게 됨. 이때 최악의 경우에는 해당 트리의 높이만큼의 연산을 수행해야 함.

결과적으로 트리의 높이가 h일 때 최악의 경우라면 $\Theta(h)$이고, 임의의 트리에 대한 최악의 경우(degenerate tree 등)에는 각 연산이 $\Theta(n)$임. 최악의 경우에 대해서만 생각하면 선형적 자료구조(배열)에서의 탐색보다 성능이 좋지는 않지만, 평균적인 경우에 대해서는 BST의 성능이 더 좋음.

비어 있는 BST에 무작위 key값이 임의 순서로 insert된다고 하면, BST는 결국 균형 잡힌 트리(balanced tree)가 됨. 따라서, 평균적인 경우 BST의 search/insert/remove 연산은 $\Theta(logn)$의 시간 복잡도를 가짐. (엄밀한 증명은 생략.)

7.2.5. class diagram

이진 트리에 search/insert/remove만 추가하면 되므로, 이진 트리 class를 상속받으면 됨. 코드의 간결함을 위해 node class에 hasTwoChildren(), hasOneChildren() 등의 메소드도 추가하면 좋음.



8. PQ

8.1. PQ

8.1.1. PQ

Definition 59 *우선순위 큐(PQ, priority queue)는 우선순위 개념을 추가한 큐임. 우선순위를 가지는 데이터를 저장하고 우선순위가 높은 데이터부터 출력하는 것.*

PQ는 priority 값이 클수록 우선순위가 높은 것으로 취급하는 최대 우선순위 큐와, priority 값이 작을수록 우선순위가 높을 것으로 취급하는 최소 우선순위 큐로 분류됨.

PQ는 네트워크, OS 스케줄링, 다익스트라 알고리즘 등에 사용됨.

늦게 들어온 데이터의 우선순위를 높게 하여 스택을 구현할 수 있고, 먼저 들어온 데이터의 우선순위를 높게 하여 큐를 구현할 수 있음.

8.1.2. PQ ADT

1. Data
우선순위를 가지는 데이터 집합.
주로 key, value, priority 값을 각각 가지도록 함.
단순함을 위해 key=value=priority로 지정하여 priority만을 사용하기도 함.

2. Main Operations
insert(item) : PQ에 item을 삽입.
remove() : 우선순위가 가장 높은 요소를 삭제하고 반환.

3. Auxiliary Operations
find() : 우선순위가 가장 높은 요소를 반환.

8.2. 구현

8.2.1. 배열을 이용한 구현

1. priority에 대해 정렬하여 저장하는 경우
우선순위가 높은 원소가 tail 쪽에 들어가도록 정렬하여 저장함.
insert의 경우, 원소를 살펴 해당 item이 들어갈 위치를 찾고 원소를 삽입한 후 이후의 원소들을 옮겨줘야 함. 단순 탐색을 한다면 최악의 경우 $\Theta(n)$이고, 이분(이진) 탐색을 한다면 최선의 경우에는 $\Theta(\log n)$(마지막에 넣는 경우)이고 최악의 경우에는 $\Theta(n)$임.
remove의 경우, 단순히 가장 마지막 원소를 가져오면 됨. 최악의 경우 $\Theta(1)$임.

2. 정렬하지 않는 경우
맨 앞 자리부터 빈 공간 없이 넣는다고 가정함. 물론 빈 공간이 존재할 수 있도록 구현할 수도 있지만 비효율적임.
insert의 경우, 단순히 맨 뒤에 있는 자리에 원소를 넣어주면 됨. 최악의 경우 $\Theta(1)$임.
remove의 경우, 모든 원소를 살펴 우선순위가 가장 높은 원소를 가져오고, 이후의 원소들을 이동시켜 빈 자리를 채워줘야 함. 최악의 경우 $\Theta(n)$임.

8.2.2. linked list를 이용한 구현

1. priority에 대해 정렬하여 저장하는 경우
우선순위가 높은 원소가 head 쪽에 있도록 정렬하여 저장함.
insert의 경우, 모든 원소를 살펴 적절한 위치에 넣어줘야 함. 최악의 경우 $\Theta(n)$임.
remove의 경우, head 포인터가 가리키는 원소를 가져오면 됨. 최악의 경우 $\Theta(1)$임.

2. 정렬하지 않는 경우
insert의 경우, 단순히 맨 앞에 추가해주면 됨. 최악의 경우 $\Theta(1)$임.
remove의 경우, 모든 원소를 살펴 우선순위가 가장 높은 것을 찾음. 최악의 경우 $\Theta(n)$임.

8.2.3. binary heap을 이용한 구현

추후에 설명할 binary heap을 사용하여 PQ를 구현할 수 있음.

8.3. STL

8.3.1. PQ STL

Definition 60 *<priority_queue> 라이브러리를 사용함.*

내부적으로 heap을 사용하여 구현되어 있음.

아래와 같이 선언함. 본 STL은 기본적으로 최대 우선순위 큐에 대응됨.

```
1. 최대 우선순위 큐
priority_queue<int> pq;

2. 최소 우선순위 큐
priority_queue<int, std::vector<int>, std::greater<int>> pq;
```

PQ STL에서 사용 가능한 메소드는 아래와 같음.

push(item) : insert에 대응됨. item을 PQ에 넣음.
pop() : remove에 대응됨. pop한 값은 반환하지 않고 삭제만 함.
top : find에 대응됨. 가장 우선순위가 높은 원소를 반환함.

9. binary heap

9.1. binary heap

9.1.1. binary heap

Definition 61 *binary heap은 heap property를 만족시키는 complete binary tree를 사용하여 데이터를 저장하는 자료구조임.*

아래와 같이 어떤 종류의 PQ를 구현하는지에 따라 heap property가 존재하고, binary heap의 종류가 나뉨.

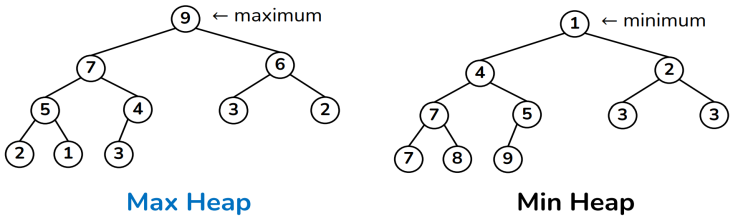
1. 최대 우선순위 큐를 구현하는 경우
부모 노드의 priority가 자식 노드의 priority보다 크거나 같아야 함.
이때의 heap을 max heap이라고 함.

2. 최소 우선순위 큐를 구현하는 경우
부모 노드의 priority가 자식 노드의 priority보다 작거나 같아야 함.
이때의 heap을 min heap이라고 함.

complete binary tree를 사용하는 것은 insert/remove가 트리의 높이만큼의 연산량을 가지게 되기 때문에 높이를 최소로 하기 위함임. 또한 트리에 노드가 순서대로 채워지기 때문에 배열로 구현했을 때 빈 공간 없이 구현할 수 있음.

heap은 더미라는 뜻임. 자료구조에서 heap은 완전 이진 트리를 사용하는 자료구조를 의미함.

목적과 방법에 따라 PQ를 더 효율적으로 구현하는 자료구조는 다양하지만, 가장 대표적인 것이 binary heap 임. 단순히 배열/linked list로 구현하면 insert/remove 중 하나가 $\Theta(1)$, 다른 하나가 $\Theta(n)$의 효율을 가지지만, binary heap으로 구현하면 둘 다 $\Theta(\log n)$의 효율을 가지도록 할 수 있음.



9.2. 구현

여기서는 max heap을 구현함.

9.2.1. 배열을 이용한 구현

binary heap은 대체로 배열로 구현함. 각 배열의 인덱스에 대한 값은 트리에서 해당 위치에 존재하는 값임. 이때 구현의 편의성을 위해 루트 노드의 인덱스를 1로 하고 0번째 인덱스는 사용하지 않음. 배열로 구현한 트리에서 다뤘던 것처럼, 부모/자식의 인덱스는 단순한 사칙연산으로 알 수 있음.

9.2.2. insert/remove

1. insert(item)
up-heap(bubble-up, heapify-up) 방식으로 구현할 수 있음.
heap의 가장 마지막에 item을 추가하고, 그 부모와 priority를 비교하여 필요한 경우 swap하는 과정을 반복하여 적절한 위치까지 item을 이동시킬 수 있음. heap을 거슬러 올라가는 형태이므로 up-heap이라고 함.

아래는 그 pseudo-code임.

```
def insert(item) :
    size++
    i = size
    A[i] = item

    while i != 1 && node[i] > node[parent(i)] :
        swap(i, parent(i))
        i = parent(i)
```

2. remove()
down-heap(bubble-down, heapify-down) 방식으로 구현할 수 있음.
필요한 경우 루트의 값을 반환하도록 지정함. 이후 루트 노드의 값을 가장 마지막 노드(마지막 level의 가장 오른쪽 노드)로 대체하고, 양쪽 자식과 비교하여 가장 큰 것을 위로 올리는 과정(min heap이면 가장 작은 것을 위로.)을 반복하여 적절한 위치까지 마지막 노드를 이동시킬 수 있음. 루트부터 아래 방향으로 내려가는 형태이므로 down-heap이라고 함.

아래는 그 pseudo-code임.

```
def remove() :
    A[1] = A[size]
    largest = 0
    i = 1

    while true :
        largest = i
        left = getLeft(i)
        right = getRight(i)

        if largest < left && left < size :
            largest = left
        if largest < right && right < size :
            largest = right
        swap(i, largest)
        i = largest

        if largest == i :
            break;
    size--
```

9.2.3. decrease/increase

binary heap의 추가적인 Operation들로는 decrease-key, increase-key 등이 있음. 여기서 key는 노드 별로 유일하고, key는 size보다 작거나 같은 양의 정수라고 가정함. 이런 연산을 구현하기 위해 PQ에서의 data가

key를 가지도록 정의된 듯함.

key는 단순히 노드를 구분하기 위한 것으로, key에 대해서는 정렬되어 있다거나 하지 않음. 이에 따라 key에 대한 노드를 매번 탐색하는 것은 시간이 많이 걸리므로, (size + 1) 크기의 배열을 하나 선언하여 특정 key에 대한 위치를 저장하면 $\Theta(1)$만에 위치를 얻을 수 있음. insert/remove/decrease/increase에서의 swap 등 노드의 위치가 바뀌거나 추가/삭제되는 경우에는 해당 배열에도 값을 반영해 줘야 함.

1. decrease(key, decrement) : key에 해당하는 노드의 priority를 decrement만큼 감소.
우선 key에 해당하는 노드를 찾아 priority를 decrement만큼 감소시킴. priority가 감소되었으므로 해당 노드의 아래쪽으로 heap property가 깨질 가능성이 생기게 됨. 해당 노드부터 down-heap을 하여 적절한 위치로 이동시켜 줄 수 있음.

2. increase(key, decrement) : key에 해당하는 노드의 priority를 decrement만큼 증가.
우선 key에 해당하는 노드를 찾아 priority를 decrement만큼 증가시킴. priority가 증가되었으므로 해당 노드의 위쪽으로 heap property가 깨질 가능성이 생기게 됨. 해당 노드부터 up-heap을 하여 적절한 위치로 이동시켜 줄 수 있음.

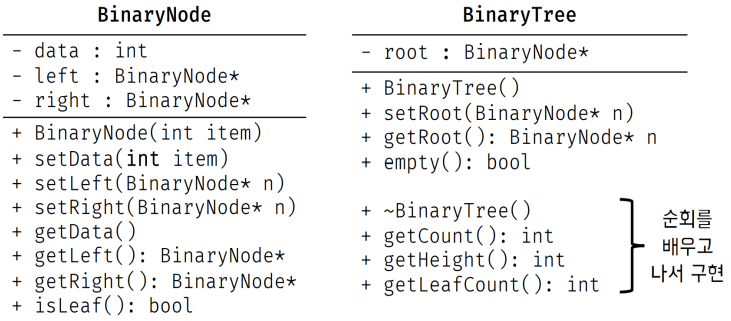
9.2.4. 시간복잡도

insert/remove의 경우 level을 하나씩 오르거나 내려가며 연산을 수행하므로, 트리 높이 만큼의 연산량을 가짐. binary heap의 경우 complete binary tree이므로 둘 다 $\Theta(\log n)$의 성능을 가짐.

decrease/increase의 경우 노드를 찾고 priority를 증가/감소시키는 연산은 상수 시간에 안에 끝나지만, 이후 down-heap/up-heap을 하는 과정에서 최악의 경우 $\Theta(\log n)$의 복잡도를 가지게 됨.

9.2.5. class diagram

max heap의 구현을 위한 class diagram은 아래와 같음. 부모/자식 인덱스를 계산하는 연산을 getLeft(), getRight, getParent() 등으로 작성하여 가독성을 높일 수 있음.



9.2.6. 예시 : heap sort

binary heap을 사용한 정렬을 heap sort라고 함. 각 원소의 값을 기준으로 정렬한다면 값을 priority로 생각할 수 있음.

n개의 원소를 binary heap에 전부 넣고 다시 꺼낸다면, 각 insert/remove가 $\Theta(\log n)$의 시간복잡도를 가지므로 heap sort는 $\Theta(n \log n)$의 복잡도를 가지게 됨.

10. disjoint set

10.1. disjoint set

10.1.1. disjoint set

Definition 62 *서로소 집합(disjoint set)은 서로 원소가 겹치지 않는 집합들을 관리하는 자료구조임. union과 find 연산을 주로 사용하기 때문에 union-find 자료구조라고도 함.*

특히 그래프에서 간선으로 연결된 노드들의 집합을 나타내는 경우 등에서 주로 사용함.

disjoint set은 non-binary tree로 간단히 구현할 수 있음.

10.1.2. disjoint set ADT

1. Data
서로 원소가 겹치지 않는 데이터 집합들.

2. Main Operations
make-set(u) : u만을 가지고 있는 새로운 집합 생성.
find-set(u) : u가 속한 집합 반환.
union(u, v) : u가 속한 집합과 v가 속한 집합을 합침.

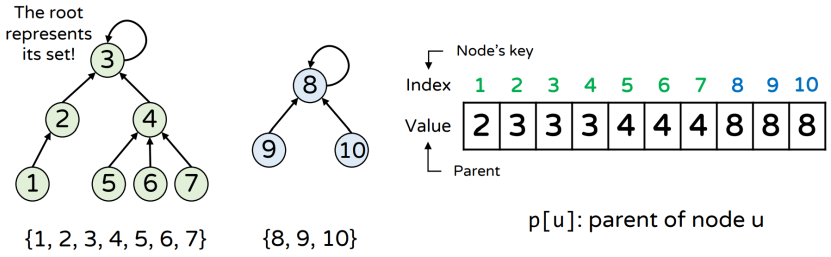
10.2. 구현

10.2.1. 배열을 이용한 non-binary tree

disjoint set은 배열을 이용한 non-binary tree로 구현할 수 있음.

자식 노드가 부모 노드를 가리키도록 하는 parent pointer tree를 사용함. 각 노드들은 자신의 부모를 가리키고, 루트 노드는 자기 자신을 가리키도록 함. point에 대한 정보는 일차원 배열에 저장함. k에 해당되는 노드가 가리키는 부모 노드는 배열에서 인덱스 k에 해당하는 공간에 저장함.

각 집합은 하나의 트리에 저장되고, 각 집합은 루트 노드로 구분함. 특정 원소에 대해서 배열의 값을 확인하고, 그 값을 인덱스로 배열의 값을 확인하는 과정을 반복하면 해당 집합의 루트 노드를 얻을 수 있음.



10.2.2. make/find/union

make/find/union은 아래와 같이 구현함. 각 노드별로 자신이 가리키는 부모 노드의 정보를 저장하므로 노드의 개수가 n개인 경우 공간복잡도는 $\Theta(n)$임.

1. make-set(u)
make-set(u)은 u를 루트 노드로 하는 집합을 생성함. 즉, A[u]=u로 지정함.

$\Theta(1)$만큼 걸림.

2. find-set(u)
해당 find-set(u)은 u가 속하는 집합의 루트 노드를 반환하도록 함. A에서 u의 값을 확인하고, 이 값을 인덱스로 하여 값을 얻는 과정을 반복하여 루트 노드를 얻을 수 있음.

해당 집합의 높이를 h_u라고 했을 때 $\Theta(h_u)$만큼 걸림.

3. union()
union(u, v)은 두 원소에 해당하는 루트 노드를 찾고(각각 find-set() 수행) 한 루트 노드가 다른 루트 노드를 가리키게 함.

각 집합의 높이를 h_u, h_v라고 했을 때 $\Theta(h_u + h_v)$만큼 걸림.

10.2.3. 개선

make-set()은 상수 시간이 걸리지만 find-set()/union()의 경우 집합의 높이만큼의 시간복잡도를 가지게 됨. 최악의 경우는 해당 트리가 degenerate tree인 경우인데, 이 경우 전부 $\Theta(n)$만큼이 걸리게 됨.

이를 개선하려면 트리의 높이를 줄여야 하는데, 여기에서는 Union by rank와 Path compression을 사용할 수 있음.

정확히는 rank는 트리 높이의 상한인데, 여기서는 단순히 트리의 높이로 생각해도 됨.

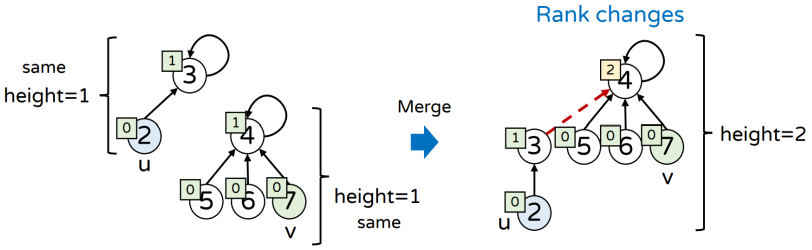
1. Union by rank
make-set()은 루트 노드 하나를 만들고, find-set()은 집합을 확인하기만 하므로 트리의 높이는 항상 union()에서 높아지게 됨. union()시에 트리의 rank에 따라 더 작은 트리를 더 큰 트리에 연결되도록 하는 것이 Union by rank임.

이를 위해 각 노드별로 별도의 변수를 사용해 각각의 높이를 rank로 저장함. rank는 해당 노드를 루트로 하는 서브트리의 높이임. 높이가 높아지는 경우에만 rank를 늘려 주면 됨. 자식이 없는 노드는 rank가 0이므로 make-set() 시에 rank는 0으로 지정해야 함.

union() 시에 높이가 다르면 rank는 변화하지 않지만, 높이가 같으면 rank는 변화하게 됨.

아래와 같이 구현할 수 있음.

```
def Union(u,v) :
    root_u = find-set(u)
    root_v = find-set(v)
    if root_u.rank > root_v.rank :
        A[root_v] = root_u
    else if root_u.rank < root_v.rank :
        A[root_u] = root_v
    else : # root_u.rank == root_v.rank
        A[root_v] = root_u
        root_u.rank++ # 두 트리의 높이가 같으므로 u에 추가하면 u의 rank가 1 증가함.
```



2. Path compression
find-set()시에 해당 원소로부터 루트 노드까지 거슬러 올라가게 되는데, 이때 거치게 되는 노드들을 해당 트리의 루트 노드에 연결해 트리의 높이를 낮추는 것이 path compression(압축)임.

union by rank를 사용해도 동일한 높이의 트리를 계속 합칠 경우 높이가 증가하게 됨. path compression까지 사용하여 높이를 더 줄일 수 있음. path compression을 적용하면 find-set()을 수행할 때마다 해당 경로에 대해서 트리의 높이를 최대한으로 낮출 수 있음.

path compression이 적용된 경우 해당 disjoint set은 사용할수록 성능이 향상됨.

아래와 같이 재귀적으로 구현할 수 있음. (재귀적으로 구성하기에 헷갈릴 수 있으므로 기억해두자.)

```
def find-set(u) :
    if A[u] != u :
        A[u] = find-set(u)
    return A[u]
```

10.2.4. 개선된 성능

(이 부분 추후에 확실히 이해하고 넘어가자)

요약하면, 굉장히 빨라짐.

1. union by rank에 의한 성능 개선
트리의 높이로 성능을 확인할 수 있음. 트리의 높이는 수학적 귀납법으로 보일 수 있음.

루트의 높이를 k, 해당 집합이 가지는 원소의 개수를 n이라고 하면 $2^k \leq n$임. 정리하면 $k \leq \log_2(n)$이므로 원소 개수에 따른 루트 높이의 상한은 $\log_2(n) \in O(\log n)$임. 트리의 높이에 따라 disjoint set의 성능이 결정되므로, find-set()과 union()은 $O(\log n)$의 성능을 가지게 됨.

$2^k \leq n$임은 다음과 같이 증명할 수 있음. k가 0인 경우 원소의 개수가 1개이므로 성립함. $k = m$인 경우 성립한다고 가정하자. $k = m$일 때 원소의 개수는 최소 2^k이므로, $k = m + 1$일 때 원소의 개수가 최소인 경우는 $k = m$이고 원소의 개수가 2^k인 두 집합을 합쳤을 때임. 이때의 원소의 개수는 2^{k+1}이므로 $k = m+1$일 때에도 성립함.

2. path compression에 의한 성능 개선
n개의 원소가 존재하는 경우, find-set()과 union()의 시간 복잡도는 $O(m \log^*(n))$임. $\log^*(n)$을 상수 취급하면 각 연산은 $O(1)$만큼이 걸리는 것으로 취급할 수 있음.

$\log^*(n)$는 n에 log를 계속 취하다가 1보다 작아지는 시점의 값을 말함. 즉, n에 대해 굉장히 작은 수임.

증명은 복잡하므로 다루지 않음.

11. 그래프

11.1. 그래프

11.1.1. 그래프

Definition 63 *임의의 두 대상 사이의 연결 관계를 표현하는 자료구조.*

그래프에서의 각 대상을 정점(vertex), 대상 사이의 연결을 간선(edge)이라고 함. 그래프(G=(V,E))는 정점 집합(V)과 간선 집합(E)의 쌍으로 정의됨. 이는 아래와 같이 작성함.

지도, 대상 사이의 관계, 추천/검색 시스템 등에 자주 사용됨.

대표적인 예시로는 오일러 문제(한붓그리기)가 있음. 한붓그리기처럼 존재하는 경로를 오일러 경로라고 하는데, 오일러 경로는 각 정점에 연결된 간선의 수가 짝수이면 존재함.

V와 E는 V=A,B,C,D, E=(A,B),(A,C),(C,D) 등으로 나타낼 수 있음.

11.1.2. graph의 종류

Definition 64 *1. 방향 그래프(directed graph) : 간선에 방향이 존재하여 해당 방향으로만 이동이 가능한 그래프. 간선을 $(u \to u)$로 표기함.*

2. 무방향 그래프(undirected graph) : 간선에 방향이 존재하지 않는 그래프. 간선을 (u, u) 또는 $(u \leftrightarrow u)$로 표기함.

3. 가중치 그래프(weighted graph) : 간선에 가중치가 존재하는 그래프. 간선에 숫자를 작성하는 식으로 표기함.

4. 비가중치 그래프(unweighted graph) : 간선에 가중치가 존재하지 않는 그래프.

방향과 가중치라는 두 변수가 존재하므로 총 4가지 종류의 그래프를 생각할 수 있음.

11.1.3. graph 관련 용어

1. 인접 정점(adjacent vertex/node)
: 어떤 정점에 대해 해당 정점과 직접 연결된 정점들.

정점 A의 인접 정점들의 집합은 무방향 그래프의 경우 N_A로 표기함. 방향 그래프의 경우 해당 정점에서 나가는 방향으로 연결된 인접 정점은 out-neighbors라고 하며 $\overrightarrow{N_A}$로 표기하고, 들어오는 방향으로 연결된 인접 정점은 in-neighbors라고 하며 $\overleftarrow{N_A}$로 표기함.

예를 들어, $\overrightarrow{N_A} = \{B, C, D\}$와 같이 작성함.

2. 정점의 차수(vertex/node degree)
: 해당 정점에 연결된 이웃 노드의 수. 즉, 인접 정점의 개수.

||는 해당 집합의 원소 개수를 나타냄. 간선의 개수는 $|E|$로 표기하고, 정점의 차수는 $|N_A|$, $|\overrightarrow{N_A}|$, $|\overleftarrow{N_A}|$로 표기함.

무방향 그래프의 경우 $\sum |N_A| = 2|E|$이고, 방향 그래프의 경우 $\sum |\overrightarrow{N_k}| = \sum |\overleftarrow{N_k}| = |E|$임.

3. 경로(path)
: 그래프에서 간선을 따라 이동할 수 있는 길.

경로의 길이(length)는 경로를 구성하는 간선의 개수임.

단순 경로(simple path)는 정점의 반복이 존재하지 않는 경로를 말함. 단순 사이클(simple cycle)은 시작 정점과 끝 정점이 동일하고, 이를 제외한 정점의 반복이 존재하지 않는 경우를 말함. 당연하게도 non-simple path와 non-simple cycle 또한 존재함.

트리는 cycle을 가지지 않는 특수한 그래프임.

예를 들어, 정점 $v_1, v_2, \cdots, v_n$를 순서대로 거치는 경로는 $\{v_1, v_2, \cdots, v_n\}$로 표기하고, 이때의 길이는 $n-1$임.

4. 연결 그래프(connected graph)
: 그래프의 모든 정점이 하나 이상의 간선으로 연결되어 있는 그래프. 즉, 간선으로 연결되지 않은 정점이 없는 그래프.

당연하게도 비연결 그래프(non-connected graph)도 존재함.

5. 완전 그래프(complete graph)
: 모든 정점이 서로 연결되어 있는 무방향 그래프.

정점이 n개라고 하면 하나의 정점이 $n-1$개의 간선과 연결되어 있으므로, 전체 간선의 개수는 $\frac{n(n-1)}{2}$임. n개 중에 2개를 뽑는 조합으로도 생각할 수 있음.

11.1.4. graph ADT

1. Data
정점의 집합과 간선의 집합으로 표현될 수 있는 데이터 집합.

2. Main Operations
insert-node(u) : key값을 u로 가지는 노드를 생성하여 그래프에 삽입.
insert-edge(u,v,w) : u와 v 사이에 가중치 w를 가지는 간선 삽입.
neighbors(u) : u의 인접 정점 반환.
remove-node(u) : key값을 u로 가지는 노드 삭제. u와 연결된 노드 또한 전부 삭제.
remove-edge(u,v) : u와 v사이의 간선 삭제.

11.2. 구현

그래프는 각 정점에 대한 인접 정점을 저장하여 구현할 수 있음. 이를 행렬 또는 list에 저장할 수 있음.

11.2.1. 인접 행렬

n개의 노드가 존재한다면 $n \times n$ 행렬을 사용하여 그래프를 저장하는데, v행 u열에는 u에서 v로 가는 간선에 대한 정보를 저장하는 방식임. 이에 따라 같은 크기의 이차원 배열 또는 vector을 사용하고, A[u][v]에 u에서 v로 가는 간선에 대한 정보를 저장함.

무방향 그래프라면 [u][v]와 [v][u] 모두에 동일한 간선에 대한 정보를 저장하고, 방향 그래프라면 해당 방향에만 정보를 저장함. 비가중치 그래프라면 true/false를 나타내는 데이터만을 저장하면 되고(가중치를 0 또는 1로 하기도 함.), 가중치 그래프라면 가중치를 저장하면 됨. 당연하게도 저장해야 하는 값이 여러 개라면 구조체나 클래스를 사용함.

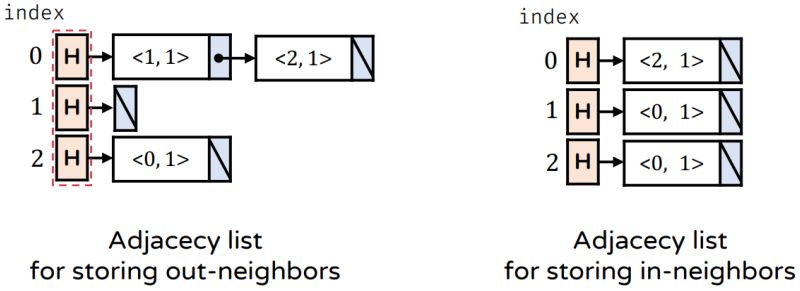
인접 행렬을 사용하는 경우는 주로 노드의 개수가 정해져 있고, 그 개수가 변하지 않는 상황임. insert-node()와 remove-node()는 구현이 까다로움.

11.2.2. 인접 리스트

각 노드마다 버킷(bucket)을 가지도록 하여 노드 별 인접 정점들을 리스트의 형태로 저장함. bucket의 집합은 주로 배열이나 vector로 만들고, 해당 배열/vector의 원소가 리스트를 가리키게 구현함.

무방향 그래프라면 두 노드의 bucket 모두에 서로와 간선에 대한 정보를 저장하고, 방향 그래프라면 해당 방향에만 정보를 저장함. 또는 아예 정점의 집합에 대해 두 개의 bucket 집합을 만들어서 각각을 in/out으로 사용할 수도 있음. 가중치 그래프의 경우 key와 가중치를 함께 저장할 수 있음.

인접 리스트를 사용하는 경우는 주로 노드와 간선이 계속해서 추가되는 경우임. remove-edge()와 remove-node()는 구현이 까다로움.



11.2.3. 열거형

아래와 같이 열거형으로 그래프의 종류(방향, 가중치)를 정의해 두고 종류에 따른 작업을 하도록 구현하여 범용성을 높일 수 있음.

```
// 정의
enum GraphType
{
    UNDIRECTED,
    DIRECTED,
    UNDIRECTED_WEIGHTED,
    DIRECTED_WEIGHTED
};

// 사용
AdjMatGraph(GraphType::UNDIRECTED_WEIGHTED, 4)
...
if(type == DIRECTED) { ... }
```

11.2.4. 인접 행렬 vs. 인접 리스트

인접 행렬과 인접 리스트는 각 연산에 대해 아래의 시간복잡도를 가짐.

인접 행렬은 간선의 추가/삭제가 간단하고 데이터가 행렬로 표현되기 때문에 수학적 분석에 유용하지만, 간선이 존재하지 않는 부분에 대해서도 메모리를 사용하기 때문에 공간복잡도가 높음.

인접 리스트는 간선/정점의 추가가 간단하고 실제로 존재하는 간전에 대해서만 메모리를 사용하지만, 포인터에 의한 메모리 소모가 존재함.

완전 그래프일수록 인접 리스트의 메모리 사용량이 더 많아지므로, 그래프가 dense할수록 인접 행렬을, sparse할수록 인접 리스트를 사용하는 것이 적합함.

Space/Operations	Adjacency matrix	Adjacency List
storage	$\Theta(n^2)$	$\Theta(n+m)$
insert_node	Not supported	$\Theta(1)$
insert_edge	$\Theta(1)$	$\Theta(1)$
remove_edge	$\Theta(1)$	Not supported
remove_node	Not supported	Not supported
(iterating) neighbors	$O(n)$	$\Theta(\lvert N_u \rvert)$

11.3. 그래프 탐색

11.3.1. DFS vs. BFS

그래프의 탐색 알고리즘에는 DFS와 BFS가 자주 사용됨. queue에서 등장했던 것과 동일한 방식임.

이때 방문한 정점을 다시 방문하지 않도록 하는 것이 중요함. 이거 놓치면 성능이 굉장히 안 좋아짐.

1. DFS(depth-first search) → 재귀 또는 스택 사용.
DFS를 시용한 그래프 순회의 pseudo-code는 아래와 같음.

```
def DFS(u) : # 처음에는 시작 노드를 넣음
    visited[i] = true
    # 작업 수행
    for i in u가 가리키는 정점의 key :
        if visited[i] == false :
            DFS(i)
```

2, BFS(breath-first search) → 큐 사용. BFS를 시용한 그래프 순회의 pseudo-code는 아래와 같음.

```
def BFS(u) : # 처음에는 시작 노드를 넣음
    Queue q
    visited[u] = true
    q.push(u)
    # 작업 수행
    while !q.empty() :
        tmp = q.pop()
        for i in tmp가 가리키는 정점의 key :
            if visitied[i] == false :
                visitied[i] = true
                q.push(i)
                # 작업 수행
```

11.3.2. 시간복잡도 분석

1. DFS
방문한 정점은 다시 방문하지 않으므로 정점의 개수가 n이면 DFS()는 n번 호출됨.

인접 행렬의 경우 DFS() 내부의 for문에서 n만큼 돌며 전부 검사해야 하기 때문에 DFS()는 $\Theta(n)$의 성능을 가지게 되고, 총 n번 호출되므로 전체 시간복잡도는 $\Theta(n^2)$임.

인접 리스트의 경우 DFS() 내부의 for문에서 $|\overrightarrow{N_A}|$만큼 돌며 검사하므로 각 노드별로 $|\overrightarrow{N_A}|+C$의 연산량을 가짐. 간선의 개수가 m개라고 하면, 총 n번 호출되므로 $\sum|\overrightarrow{N_A}|+C = m+cn$으로 전체 시간복잡도는 $\Theta(m+n)$임.

2. BFS
방문한 정점은 다시 방문하지 않으므로(queue에 넣지 않음) 정점의 개수가 n이면 가장 바깥쪽의 loop는 n번 수행됨.

인접 행렬의 경우 각 loop마다 내부의 for문에서 n만큼 돌며 전부 검사해야 하기 때문에 $\Theta(n)$의 성능을 가지게 되고, 총 n번 수행되므로 전체 시간복잡도는 $\Theta(n^2)$임.

인접 리스트의 경우 loop마다의 for문에서 $|\overrightarrow{N_A}|$만큼 돌며 검사하므로 각 노드별로 $|\overrightarrow{N_A}|+C$의 연산량을 가짐. 간선의 개수가 m개라고 하면, 총 n번 호출되므로 $\sum|\overrightarrow{N_A}|+C=m+cn$으로 전체 시간복잡도는 $\Theta(m+n)$임.

12. map

12.1. map

12.1.1. map

Definition 65 *key와 그에 따른 value를 저장하는 자료구조.*

특정 key에 대한 value에 대해 탐색/저장/삭제를 수행함.

map은 다양한 방식으로 구현될 수 있음.

c++(gcc)의 std::unordered_map STL에서는 hash map으로 구현되어 있고, 해시 함수로 murmur hash를 사용하고, seperate chaining으로 collision resolution을 함.

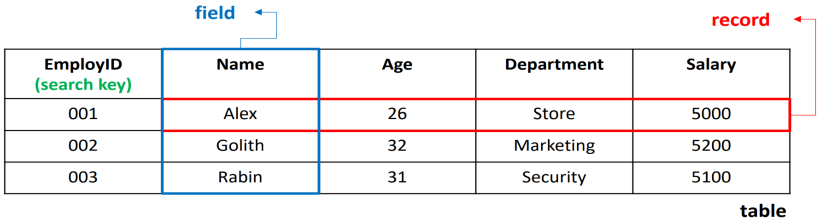
파이썬(python3)의 dictionary에서는 hash map으로 구현되어 있고, 해시 함수로 siphash를 사용하고, open addressing으로 collision resolution을 함.

12.1.2. map 관련 용어

map에 저장하는 데이터에 대해서 아래의 용어들을 사용함.

필드(field) : 변수.
레코드(record) : 필드 값들의 집합.
탐색키(search key) : 각 레코드에 대해 구분할 수 있도록 하는 유일한 값.
테이블(table) : 레코드의 집합.

이때 key를 가지는 레코드를 keyed record, key-value pair라고 함.



12.1.3. map ADT

1. Data
유일한 key를 가지는 레코드들로 구성된 데이터 집합.

2. Main Operations
search(key) : key에 해당되는 레코드 또는 그 위치를 반환.
insert(record) : 해당 레코드를 map에 삽입.
remove(key) : key에 해당되는 레코드를 찾아 삭제.

12.2. 구현

12.2.1. 정렬되지 않은 배열

배열의 각 원소에 key-value pair를 저장.

search는 배열의 각 원소를 전부 순회해야 하므로 최선의 경우 $\Theta(1)$, 최악/평균의 경우 $\Theta(n)$임.

insert는 배열의 맨 뒤에 원소를 추가하면 되므로 $\Theta(1)$임.

remove는 탐색하여 삭제한 후 더 뒤쪽에 있는 원소들을 전부 앞으로 당겨야 하므로 $\Theta(n)$임.

12.2.2. 정렬된 배열

배열의 각 원소에 key-value pair를 저장하는데, key에 따라 정렬된 상태로 저장함. 이 경우 탐색 시에 이진(이분) 탐색(binary search)를 할 수 있음.

search는 최선의 경우 $\Theta(1)$, 최악/평균의 경우 $\Theta(\log n)$임. 원소가 n개일 때의 시간복잡도를 $T(n)$이라고 하면 $T(n) = T(\frac{1}{n}) + C$이므로 정리할 수 있음.

insert시에 정렬 상태가 유지되어야 하므로 적절한 위치를 찾기 위해 탐색함. 탐색한 위치가 가장 마지막 위치인 경우 다른 원소들을 이동할 필요가 없지만, 그렇지 않다면 뒤쪽 원소들을 전부 옮겨줘야 함. 최선의 경우 $\Theta(\log n)$, 최악/평균의 경우 $\Theta(n)$임.

remove는 탐색 위치가 마지막이거나 삭제할 대상이 존재하지 않는 경우 다른 원소들을 이동할 필요가 없지만, 그렇지 않다면 뒤쪽 원소들을 전부 옮겨줘야 함. 최선의 경우 $\Theta(\log n)$, 최악/평균의 경우 $\Theta(n)$임.

12.2.3. BST

BST를 이용해 key-value pair를 저장할 수 있음.

1. 일반 BST
search는 최선의 경우 $\Theta(1)$, 최악의 경우 $\Theta(n)$, 평균적 경우 $\Theta(\log n)$임.

insert/remove는 최선의 경우 $\Theta(1)$, 최악의 경우 $\Theta(n)$, 평균적 경우 $\Theta(\log n)$임.

2. self-balancing BST
search는 최선의 경우 $\Theta(1)$, 최악/평균적 경우 $\Theta(\log n)$임.

insert/remove는 최선의 경우 $\Theta(1)$, 최악/평균적 경우 $\Theta(\log n)$임.

12.2.4. hash map

후술할 hash map을 이용해 구현할 수 있음.

13. hash map

13.1. hash map

13.1.1. hash

Definition 66 *해시(hash)는 hash function을 사용하여 주어진 데이터를 고정된 크기의 고유한 값으로 변환하는 것을 말함.*

13.1.2. hash map

Definition 67 *해시 맵(hash map)은 hash를 사용하여 구현한 map임. hash map은 hash table과 hash function을 사용하여 데이터를 관리함.*

1. 해시 테이블(hash table)
: hash function에 의해 결정되는 hash값을 인덱스로 하여 데이터를 저장하는 table.

특정 hash 값에 대한 공간을 bucket이라고 하는데, bucket에는 여러 개의 slot이 있어 각 slot에 데이터를 저장할 수 있음.

즉, 각 데이터는 그 hash 값에 해당하는 bucket에 저장되는데, hash 값이 겹치는 경우 해당 bucket 내에서 다른 slot에 저장됨.

2. hash function
: key을 입력받아 hash 값을 반환하는 함수.

hash map을 사용하면 collison과 overflow가 발생할 수 있음.

1. 충돌(collision)
: bucket 개수의 한계로 서로 다른 key에 대해 동일한 hash 값이 반환된 상황.

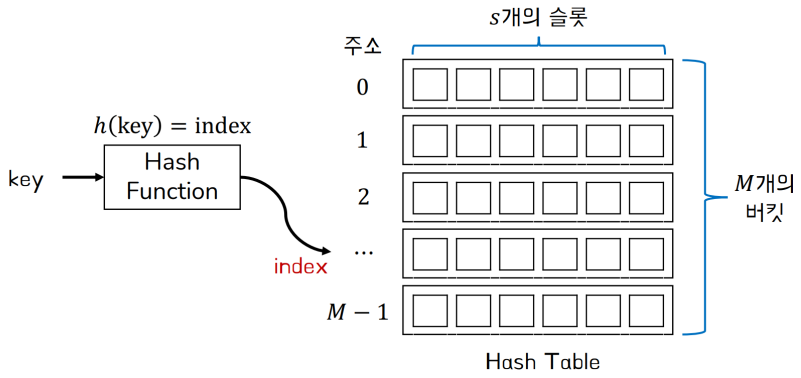
slot을 여러 개 사용하여 해결이 가능함.

2. 오버플로우(overflow)
: 특정 bucket에 대해 slot 여분이 부족해 데이터를 더 저장하지 못하는 상황.

key를 비교하는 대신 table을 사용해 빠르게 접근할 수 있도록 한 것.

bucket을 데이터 개수만큼 준비하여 각 bucket에 데이터가 하나씩 들어가도록 하면 탐색에는 $\Theta(1)$만큼이 걸리지만 너무 많은 공간이 필요함. 대신 bucket의 개수를 줄이고 적절한 hash function을 사용하며 collision/overflow 처리를 하도록 구현하는데, 당연하게도 이 경우 탐색이 $\Theta(1)$보다는 느려짐.

핵심은 hash function의 설계와 collision/overflow의 처리임.



13.2. hash function의 종류

다양한 hash function들이 있지만 여기서는 가장 교과서적인 것들을 살펴봄.

13.2.1. hash function의 조건

hash function은 아래의 요건을 만족하는 것이 이상적임.

1. 가능한 bucket(인덱스) 값 내의 hash 값만을 반환함.

2. 반환되는 hash 값들이 hash table에서 고르게 분포해야 함.
즉, collision을 최소화해야 함.

3. 계산이 빨라야 함.
탐색 때마다 수행되어야 하므로 상수 시간 안에 수행될 수 있어야 함.

13.2.2. division function

Definition 68 *제산 함수(division function)는 아래와 같이 key와 단순히 정해둔 수(M)의 modular 연산으로 hash 값을 반환함. 이때 M에 따라 bucket 개수가 정해짐(또는 bucket 개수에 따라 M을 정함).*

```
h(key) = key % M
```

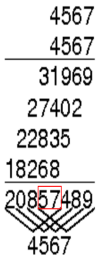
연산 한 번으로 해시 값을 얻을 수 있으므로 빠르지만, M을 잘못 설정하면 collision이 다수 발생할 수 있음.

일반적인 경우 M을 소수로 설정하면 hash값이 골고루 반환됨.

13.2.3. mid-squares function

Definition 69 *mid-squares function은 key를 제곱하여 얻은 수의 중간 수(bits)를 가져와 M과의 modular 연산을 수행함.*

division function은 key의 전체 숫자에서 M보다 큰 부분의 값은 hash값 도출에 활용하지 못하는데, key를 제곱하여 중간 숫자를 사용하면 모든 값을 활용할 수 있음.



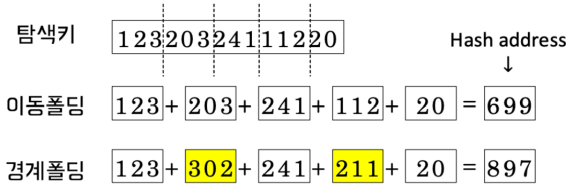
13.2.4. folding function

Definition 70 *폴딩 함수(folding function)는 key를 여러 개의 부분(fold)들로 쪼개서 각 부분으로 연산을 수행함. folding function에는 shift folding과 boundary folding 등이 있음.*

1. 이동 폴딩(shift folding)
각 fold의 값들을 전부 더한 값을 반환함.

2. 경계 폴딩(boundary folding)
각 fold에 대해서 alternative하게 수를 뒤집은 후, 전부 더한 값을 반환함.

folding function 또한 key의 모든 부분을 활용하기 위한 것임.



13.3. collision resolution

13.3.1. collision resolution

Definition 71 *collision resolution은 collision/overflow에 대한 처리 방식 정책임. collision resolution의 대표적인 방법으로는 separate chaining과 open addressing이 있음.*

1. separate chaining
bucket 별로 고정된 slot 대신 리스트를 사용해 데이터를 저장하는 방식. bucket 내부에서의 탐색은 단순하게 순차 탐색을 사용함.

포인터 사용에 따른 추가 비용이 발생하고, 하나의 bucket에 데이터가 많이 들어가면 성능이 떨어짐 (물론 overflow는 발생하지 않음.).

bucket에 들어가는 데이터가 많은 경우 단순 리스트 대신 BST를 사용하기도 함.

2. open addressing
고정된 크기의 hash table을 사용하는 대신, overflow가 발생하면 다른 bucket에 데이터를 저장하는 방식.

open addressing에는 linear/quadratic probing과 double hashing이 존재함.

13.3.2. linear probing

Definition 72 *선형 조사법(linear probing)은 bucket이 꽉 차있다면 바로 다음 인덱스의 bucket을 조사하는 방식임. 빈 bucket이 나올 때까지 조사하고, table의 끝에 도달했다면 처음 인덱스로 이동해 조사함.*

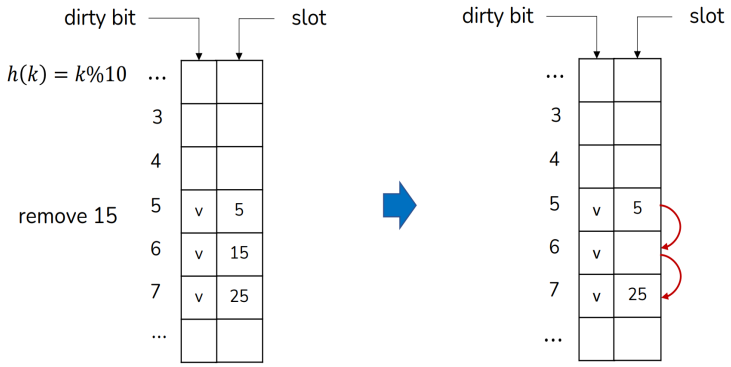
아래와 같이 반복문이 구성되며 bucket에 빈 공간이 있는지 조사함.

```
(h(k)+i)%M for i in 0~M-1
```

linear probing은 collision/overflow를 해결하지만, cluster가 형성되기 쉽고 이 경우 성능이 저하됨.

1. dirty bit 사용
map에 데이터를 삽입했다면 동일한 알고리즘으로 해당 데이터에 다시 접근할 수 있어야 함. 만약 어떤 원소가 여러 개의 bucket이 꽉 차있어서 이들을 건너뛴 위치에 저장되었다고 하자. remove 등을 통해 이 꽉 차 있던 bucket이 더 이상 꽉 차지 않게 되었다면, 동일한 알고리즘으로는 다시 이 원소에 접근이 불가능함. 이는 dirty bit을 사용하여 dirty하지 않은 부분은 만났을 때 probing을 멈추도록 구현하여 해결할 수 있음.

2. cluster의 형성
cluster는 꽉 찬 bucket들이 연속적으로 붙어 있는 것을 말함. linear probing에 의해 형성된 직선 형태의 cluster를 primary cluster라고 함. primary cluster가 형성되면 매번 수행해야 하는 probing 횟수가 증가하여 성능이 저하됨.



13.3.3. quadratic probing

Definition 73 *이차 조사법(quadratic probing)은 bucket이 꽉 차있다면 제곱수 만큼 이동한 인덱스의 bucket을 조사하는 방식임. 즉, linear probing과 유사한데, i 대신 i^2 만큼 이동하여 조사하는 방식인 것.*

아래와 같이 반복문이 구성되며 bucket에 빈 공간이 있는지 조사함.

```
(h(k)+i^2) % M for i in 0~M-1
```

quadratic probing에 의해 형성된 cluster를 secondary cluster라고 하는데, 이는 primary cluster처럼 연속적으로 형성되지는 않지만 부분적으로 형성될 수 있음. 이에 따라 primary cluster만큼은 아니지만 성능을 저하시킴.

당연하게도 원래의 hash 값이 같은 데이터들은 linear probing과 동일한 이유로 성능이 저하됨. 원래의

hash 값이 같지는 않은 데이터들의 경우에는 cluster에 의한 성능 저하가 적음.

유의해야 할 점은, $(h(k)+x^2)\%M$에서 실패했다면 $(h(k)+x^2+(x+1)^2)\%M$가 아니라 $(h(k)+(x+1)^2)\%M$을 조사함. 즉, 이동한 위치에서 더하는 것이 아니라 원래 위치에서 더하는 것임.

13.3.4. double hashing

Definition 74 *이중 해시법(double hashing, rehashing)은 overflow가 발생한 경우 해당 데이터를 저장할 위치를 결정하는 데에 별도의 hash function을 사용하는 방식임.*

1. step
step은 bucket이 꽉 차 있는 경우 기존의 hash값에 더하는 값을 말함.

linear/quadratic probing에서는 각각 i, i^2이 step이었음.

double hashing에서는 key에 대한 별도의 hash function을 사용하여 그 출력값을 step으로 함. hash function이 division function인 경우 아래와 같이 반복문이 구성됨. 이때 C는 modular 연산을 위한 별도의 값임. step은 1 ~ C인 수가 됨. (여기서 정확하게는 $i * step$을 step이라고 해야 될 것 같기는 함.)

```
step = 1 + (key % C)
(h(k) + i * step) % M for i in 0~M-1
```

실험적으로 확인해 보면 M는 소수로, C는 M보다 약간 작은 소수로 하는 것이 성능이 좋음.

13.4. 구현

13.4.1. 구현 방법

hash map은 어떤 hash function을 사용할 것인지, 어떤 collision resolution을 사용할 것인지에 따른 구현이 존재함. 이에 따라 구현이 간단하거나 복잡할 수 있고, 성능도 달라짐.

division function과 separate chaining을 사용하면 구현이 단순하지만, folding function과 rehashing/dirty bit을 사용한다면 구현이 복잡함.

13.4.2. 성능 분석

separate chaining, open addressing에 따라 최선/최악/평균의 경우에 대해 알아봄.

1. 최선의 경우
collision이 전혀 일어나지 않는다면 search/insert/remove 모두 $\Theta(1)$만에 수행됨.

2. 최악의 경우
separate chaining을 사용한 경우 하나의 bucket에 데이터가 전부 들어간 상황이 최악임. 이 경우 search/remove는 $\Theta(n)$이고, insert는 그냥 넣기만 하면 되므로 $\Theta(1)$임. 물론 bucket을 BST를 사용하여 구현한다면 BST에 의한 복잡도를 가지게 됨.

open addressing을 사용한 경우 cluster의 크기가 n이 될 때가 최악임. search/insert/remove 모두 탐색을 수행해야 하므로 $\Theta(n)$이 걸림.

3. 평균적인 경우
n개의 데이터와 M개의 bucket이 있을 때, 평균적인 경우는 각 M개의 bucket에 데이터가 전부 고르게 삽입된 상황임.

이때 각 bucket은 평균적으로 $\alpha = \frac{n}{M}$개의 데이터를 가지게 되는데, 이때의 α를 적재율(load factor)라고 함.

separate chaining에 대해서는 각 bucket에 α개의 원소가 존재하므로 search/remove는 $\Theta(\alpha)$, insert는 $\Theta(1)$임. 이 경우에도 BST라면 그에 따른 복잡도를 가짐. 만약 M이 n에 비해 충분히 크다면 전부 $\Theta(1)$이라고 할 수 있음.

open addressing에 대해서는 증명이 까다로우므로 그 결과만 살펴봄. 탐색에 실패하는 경우(해당 key가 없는 경우)에는 탐색 횟수의 기댓값은 $\frac{1}{1-\alpha}$이하임. 탐색에 성공하는 경우에는 탐색 횟수의 기댓값은 $\frac{1}{\alpha}\log\frac{1}{1-\alpha}$임.

α가 충분히 작은 경우 전부 $\Theta(1)$에 수행된다고 할 수 있음. 이때 α가 커질수록 성능이 급격히 떨어지므로 M과 n의 설정을 잘 해야 함.

13.4.3. separate chaining vs. open addressing

기본적으로 둘 다 적재율이 낮아야 평균적인 경우에 성능이 좋음.

1. separate chaining을 사용하는 경우
적재율이 높거나, 데이터의 개수가 정해지지 않았거나 insert/remove의 횟수가 정해지지 않은 경우.

2. open addressing을 사용하는 경우
적재율이 낮거나, 데이터의 개수가 정해져 있거나, 공간을 효율적으로 사용해야 하는(포인터 사용x) 경우.

13.5. STL

13.5.1. set STL

Definition 75 *<set> 라이브러리를 사용함.*

set은 key만을 저장할 수 있고, map은 key/value를 쌍으로 저장할 수 있음.

set/map은 내부적으로 self-balancing BST(red-black tree)로 구현되어 있고, 이에 따라 기본적으로 key에 대해 오름차순 정렬되어 있음.

정렬되어 있지 않은 set/map을 사용하려면 unordered_set/unordered_map STL을 사용해야 하는데, 이는 내부적으로 red-black tree가 아닌 hash map으로 구현되어 있음. c++에서 이는 murmur hash를 hash function으로 하고 separate hashing을 사용함.

key는 중복이 불가능함. 중복을 허용하는 set/map을 사용하려면 multiset/multimap STL을 사용해야 함.

아래와 같이 선언하여 사용함.

```
// 선언
map<int, int> m;

// 사용 예시
m.insert({ 5, 10 });
```

여기서 설명한 각 STL들은 사용법이 대체로 동일하지만, 내부적으로 다르게 구현되어 있어 다른 성능을 가질 수 있기 때문에 유의해야 함.

Part IV

알고리즘

1. recursion

1.1. recursion

1.1.1. recursion

문제가 자기 자신에 대해 반복적으로 정의되어 있으면 해당 문제는 재귀적(recursive) 속성을 가지고 있는 것임. 재귀적인 속성을 가지고 있는 문제에 재귀적인 접근(recursion)을 사용하면 쉬운 해결이 가능함.

재귀 함수는 1. base case와 2. 재귀 호출로 구성되어 있음. base case가 존재하지 않는 경우 함수가 무한히 호출되며 stack overflow가 날 수 있음.

물론 모든 문제를 재귀적으로 접근할 수 있는 것은 아니고, 재귀적 속성을 가진 문제더라도 재귀적으로 접근했을 때 항상 효율적인 것은 아님.

재귀에 대해서는 끝나는 경우에 반환값이 어떻게 돌아 오는지를 잘 확인해야 코드를 깔끔하게 짤 수 있음.

1.1.2. recursion design

문제에 대한 분할 정복(Divide & Conquer)을 통해 재귀적 알고리즘을 설계할 수 있음. 문제를 작게 쪼개고(divide) 각각을 해결하는(conquer) 것. 또한 필요 시 각 결과를 종합함(Aggregate).

1.1.3. correctness analysis

수학적 귀납법(mathematical induction)을 통해 재귀적 접근이 유효한지를 판단할 수 있음.

1.1.4. 시간복잡도 분석

재귀함수에 대한 시간복잡도는 재귀적으로 표현됨. 점화식을 풀어서 시간복잡도를 구할 수 있음.

T(n) = T(n-1) + C 꼴의 점화식은 반복 대치(repeated substitution)를 통해 그 일반항을 구할 수 있음. T(n-1) = T(n-2) + C를 반복적으로 대입하는 방법.

n이 짝수/홀수인지에 따라 시간복잡도가 나뉜 경우에는 $n = 2^k$로 가정하여 rough하게 시간복잡도를 구할 수 있음. 더 정확하게 구하려면 다양한 수학적 기법을 사용해야 함.

이런 식의 계산 시에는 상수에 해당하는 연산을 rough하게 C로 뭉뚱그려 처리할 수 있음.

1.2. 구현

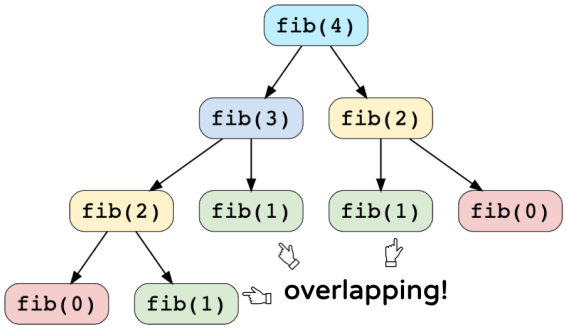
1.2.1. 피보나치 수열

n번째 항이 n-1, n-2번째 항을 더한 것으로 정의되는 수열을 피보나치 수열이라고 함. 0, 1번째 항은 0임.

1. 단순 구현
이 경우 시간복잡도의 점화식은 T(n)=T(n-1)+T(n-2)+C 이므로 정리하면 $O(2^n)$에 속함.

우선 T(n)은 n이 커지면 T(n)도 커지는 단조증가함수임. 그러므로 이 점화식은 $T(n) = T(n-1) + T(n-2) + C \leq 2T(n-2) + C \leq ... \leq 2^n T(0) + (2^n - 1)C$으로 정리할 수 있음. 즉, $T(n) \in O(2^n)$ rough하게 상한만을 본 것이므로 big-O로 표기함.

2. memoization
재귀호출이 일어나는 형태를 트리로 나타낸 것을 recursion tree라고 하고, 이때 root가 아닌 것들을 subproblem이라고 함. 또한 각 경우에 대해 가지가 2개로 뻗어 나가는 tree를 binary recursion이라고 함.



이때 서로 겹치는 subproblem들의 계산을 생략하여 더 효율적으로 개선할 수 있는데, 이를 memoization이라고 함.

각 subproblem의 계산 결과를 따로 저장하여 동일한 subproblem의 계산을 생략할 수 있음. 또는 미리 한 번씩 계산을 해 둘 수도 있음. 이 경우 시간복잡도는 T(n)=T(n-1)+T(n-2)+...+T(0)+Cn이므로 $\Theta(n)$에 속함.

1.2.2. Hanoi tower

n개를 옮기는 경우 아래의 과정을 거쳐 재귀적으로 해결할 수 있음.

base case. n이 1인 경우 1번 옮기면 됨.
1. n-1개를 경유 막대로 옮김.
2. 1개를 목적지 막대로 옮김.
3. n-1를 목적지 막대로 옮김.

횟수뿐만 아니라 각 원판이 어떻게 이동하는지를 알려면 아래와 같이 함수를 작성하면 됨.
hanoi(n, peg_a, peg_b, peg_c). 재귀적으로 호출 시에 시작 막대, 경유 막대, 목적지 막대를 지정해 주면 됨.

횟수만 구하는 경우에는 T(n)=T(n-1)+C의 시간복잡도를 가지고, $\Theta(O)$에 속함. 각 원판의 이동까지 고려해야 하는 경우 T(n)=2T(n-1)+C의 시간복잡도를 가지는데, 이는 $\Theta(2^n)$에 속함. 1번 과정과 3번 과정을 다른 경우로 다뤄야 하므로 더 오래 걸리는 것.