자료구조(Data Structure)
일련의 동일한 타입의 데이터를 정돈하여 저장한 구성체를 의미한다.

데이터를 정돈하는 목적은 프로그램에서 저장하는 데이터에 대해 탐색, 삽입, 삭제 등의 연산을 효율적으로 수행하기 위해서이다.

자료구조를 프로그램으로 구현할 때에는 데이터를 저장할 구조를 생성한 이후, 실제 저장되는 데이터를 처리하기 위한 연산을 정의해야 하는데,
추상데이터타입은 이러한 관계를 정형화시킨 개념이다.

추상데이터타입(Abstract Data Type)
데이터와 그 데이터에 대한 추상적인 연산들로써 구성된다.
여기서 '추상'의 의미는 연산을 구체적으로 어떻게 구현하여야 한다는 세부 명세를 포함하고 있지 않다는 의미이다.

자료구조는 추상데이터타입을 구체적(실제 프로그램)으로 구현한 것을 의미

<수행시간의 분석>
자료구조의 효율성은 자료구조에 대해 수행되는 연산의 수행시간으로 측정 가능

시간복잡도(Time Complexity) - 알고리즘의 수행시간
공간복잡도(Space Complexity) - 알고리즘이 수행되는 동안 사용되는 메모리 공간의 크기를 나타냄

알고리즘의 시간복잡도는 알고리즘이 실행되는 동안에 사용된 기본적인 연산 횟수를 입력 크기의 함수로 나타낸다.

기본 연산(Elementary Operation)이란, 데이터간 크기 비교, 데이터 읽기 및 갱신, 숫자 계산 등과 같은 단순한 연산을 의미한다.

1.4
파이썬 언어는 객체지향 프로그래밍 언어로서 클래스(Class)를 선언하여 객체에 데

이터를 저장하고 메소드(method)를 선언하여 객체들에 대한 연산을 구현한다.

인스턴스 변수(Instance Variable)는 객체에 정보를 저장하기 위해 선언한다.
객체를 생성하기 위한 객체 생성자(Constructor)는 클래스 내부에 선언하며, 객체 혹은 객체 내부 인스턴스 변수에 대한 연산을 수행하는 메소드도 클래스 내부에 정의한다.

*파이썬에서는 클래스 밖에서 def를 선언하면 이를 함수라 하고, 클래스 안에 def를 선언하면 메소드라고 일컫는다.

리스트(List)는 C나 자바 언어의 배열과 유사하다.
배열은 동일한 타입의 항목들을 저장하는 반면에 리스트는 서로 다른 타입의 항목들을 저장할 수 있다.
파이썬 리스트는 동적 배열(Dynamic Array) 개념으로 구현되어 있다.
리스트의 크기는 파이썬이 알아서 조절(C와 차이점)

*내장함수
ord('문자') #문자의 unicode 값을 리턴: 예)ord('A')는 65를 리턴한다.
list(reversed(리스트)) #역순으로 된 리스트를 리턴한다.
리스트.reverse() #리스트를 역순으로 만든다.

*lambda 함수는 함수의 이름도 return 도 없이 수행되는 함수이다.

lambda 인자(arguments) : 식(expression)
1-4

순환(Recursion) : 함수의 실행 과정 중 스스로를 호출하는 것
일반적으로 순환은 프로그램(알고리즘)의 가독성을 높일 수 있다는 장점을 갖지만 시스템 스택을 사용하기 때문에 메모리 사용 측면에서 비효율적이다.
그러나 반복문으로 변환하기 어려운 순환도 존재하며, 억지로 반복문으로 변환하는 경우 프로그래머가 시스템 스택의 수행을 처리해야 하므로 프로그램이 매우 복잡해질 수 있다.
또한 엄밀하게 따져 보았을 때 반복문으로 변환된 프로그램의 수행 속도가 순환으로 구현된 프로그램보다 항상 빠르다는 보장도 없다.

해시테이블
해싱 : 키를 간단한 함수를 사용해 변환한 값을 리스트의 인덱스로 이용하여 항목을 저장하는 것
해시함수 : 해싱에 사용되는 함수

해시값 or 해시주소 : 해시함수가 계산한 값
충돌 : 서로 다른 키들이 동일한 해시값을 가질 때 충돌이 발생했다고 한다.
-충돌 해결 방법
1. 개방주소방식
-충돌된 키들을 해시테이블 전체를 열린 공간으로 여기어 비어 있는 곳을 찾아 항목을 저장하는 방식
-즉, 충돌이 발생한 키를 원래의 해시값과 다른 곳에 저장한다는 의미
2. 폐쇄주소방식
-해시값에 대응되는 해시테이블 원소에 반드시 키를 저장한다. 따라서 충돌이 발생한 키들을 동일한 해시주소에 저장한다.

해시함수
-가장 이상적인 해시함수는 키들을 균등하게 해시테이블의 인덱스로 변환하는 함수
해시함수의 대표적인 예
1. 중간제곱 함수 : 키를 제곱한 후, 적절한 크기의 중간부분을 해시값으로 사용
2. 접기 함수 :큰 자릿수를 갖는 십진수를 키로 사용하는 경우, 몇 자리씩 일정하게 끊어서 만든 숫자들의 합을 이용해 해시값을 만든다.
ex) 123456789012에 대해서 1234 + 5678 + 9012 = 15924를 계산한 후에 해시테이블 크기가 1000이라면 15924에서 3자리 수만을 해시값으로 사용한다.
3, 곱셈 함수 : 1보다 작은 실수 델타를 키에 곱하여 얻은 숫자의 소수 부분을 테이블 크기 M과 곱한다.

하지만 실세계에서 가장 널리 사용되는 해시함수는 나눗셈 함수이다.

개방주소방식(open adressing)
- 해시테이블 전체를 열린 공간으로 가정하고 충돌된 키를 일정한 방식에 따라서 찾아낸 empty 원소에 저장한다.
-대표적인 개방주소방식에는 선형조사(Linear Probing), 이차조사(Quadratic Probing), 랜덤조사(Random Probing), 이중해싱(Double Hashing) 있다.

선형조사(Linear Probing)
충돌이 일어난 원소에서부터 순차적으로 검색하여 처음 발견한 empty 원소에 충돌이 일어난 키를 저장한다. (충돌이 일어나면 바로 다음 원소를 검사한다)
-선형조사는 순차탐색으로 empty 원소를 찾아 충돌된 키를 저장하므로 해시테이블의 키들이 빈틈없이 뭉쳐지는 현상이 발생한다. 이를 1차 군집화(primary Clustering)라고 하는데, 이러한 군집화는 탐색, 삽입, 삭제 연산을 수행할 때 군집된 키들을 순차적으로 방문해야 하는 문제점을 일으킨다.
-군집화는 해시테이블에 empty 원소 수가 적을수록 더 심화되며 해시 성능을 극단적으로 저하시킨다.

이차조사(Quadratic Probing)
선형조사와 근본적으로 동일한 충돌 해결 방법이다.
다만, 충돌 후 1차원 리스트 a에서 선형조사보다 더 멀리 떨어진 곳에서 empty 원소를 찾는다. (충돌이 나면 갈수록 더 멀리 떨어진 원소를 검사한다)
-이차조사는 이웃하는 빈 곳이 채워져 만들어지는 1차 군집화 문제를 해결하지만, 같은 해시값을 갖는 서로 다른 키들인 동의어들이 똑같은 점프 시퀀스를 따라 empty원소를 찾아 저장하므로 결국 또 다른 형태의 군집화인 2차 군집화를 야기한다는 문제점을 갖는다.
-또한 점프 크기가 제곱만큼씩 커지므로 1차원 리스트에 empty원소가 있는데도 empty원소를 건너뛰어 저장에 실패하는 경우도 피할 수 없다.

랜덤조사(Random Probing)
선형조사와 이차조사의 규칙적인 점프 시퀀스와는 달리 점프 시퀀스를 무작위화하여 empty 원소를 찾는 충돌 해결 방법이다.
-파이썬 언어의 dictionary 는 랜덤조사를 기반으로 구현되어 있다.
(충돌이 나면 일정한 규칙 없이 비어 있는 원소를 검사하자)
-랜덤조사 방식도 동의어들이 똑같은 점프 시퀀스에 따라 empty 원소를 찾아 키를 저장하게 되고 이 때문에 2차 군집화와 유사한 형태의 3차 군집화(Tertiary Clustering)가 발생한다.

이중해싱(Double Hashing)
2개의 해시함수를 사용하는 충돌 해결 방법이다.
(충돌이 나면 다른 해시함수의 해시값을 이용하여 원소를 검사하자)
-두 해시함수 중 하나는 기본적인 해시함수 h(key)로 키를 해시테이블의 인덱스로 변환하고, 제 2의 함수 d(key)는 충돌 발생 시 다음 위치를 위한 점프 크기를 다음의 규칙에 따라 정한다.
-선형조사는 1차 군집화 때문에 성능이 저하될 수 있고, 이차조사와 랜덤조사는 각각 2차와 3차 군집화를 야기할 수 있다. 그러나 **이중해싱은 동의어들이 저마다 제2의 해시함수를 갖기 때문에 점프 시퀀스가 일정하지 않다.**
-따라서 이중해싱은 모든 군집화 문제를 해결하는 충돌 해결 방법이다.
-또한 해시 성능을 저하시키지 않는 동시에 해시테이블에 많은 키들을 저장할 수 있다는 장점을 갖고 있다.

폐쇄주소방식(Closed Addressing)
-폐쇄주소방식의 충돌 해결 방법은 키에 대한 해시값에 대응되는 곳에만 키를 저장한다. 다라서 충돌이 발생한 키들은 한 위치에 모아 저장된다.
-대표적인 방법이 체이닝(Chaining)이다.
-M개의 연결 리스트가 독립적으로 관리되므로 체이닝을 분리 체이닝이라고 일컫기

도 한다.
(충돌이 난 키들을 같은 곳에 모아 놓자)

기타 해싱
*2-방향 체이닝(Two-way Chaining)
: 체이닝과 동일하나 2개의 해시함수를 이용하여 연결리스트의 길이가 짧은 족에 새 키(항목)를 저장한다.
(2개의 해시함수로 계산된 두 체인을 검사해서 짧은 체인에 새 항목을 삽입)
-2-방향 체이닝은 2개의 해시함수를 계산해야 하고, 연결리스트의 길이를 비교해야 하며, 추후에 탐색을 위해선 두 연결리스트를 탐색해야 하는 경우도 발생한다.
-총 N개의 키를 2-방향 체이닝으로 저장하엿을 때, 연결리스트의 평균 길이는 O(loglongN)으로 매우 잛아서 실제로 매우 좋은 성능을 보인다.

*뻐꾸기해싱(Cuckoo Hashing)
-뻐꾸기 해싱은 뻐꾸기가 다른 새의 둥지에 알을 낳고, 부화된 뻐꾸기 새끼가 다른 새의 알이나 새끼들을 둥지에서 밀어내는 습성을 모방한 해싱 방법
(2개의 해시함수와 각 함수에 대응되는 해시테이블을 이용해 충돌이 발생하면 그곳에 있는 키를 쫓아낸다)
-삽입 도중에 싸이클이 발생하면 삽입 과정이 종료되지 않는다.
-장점은 탐색과 삭제를 O(1) 시간에 보장한다는 것/삽입은 높은 확률로 O(1) 시간에 수행이 가능하다.

*재해시(Rehash)
해시테이블에 비어있는 원소가 적으면, 삽입에 실패하거나 해시 성능이 급격히 저하되는 현상을 피할 수 없다.
-이러한 경우, 해시테이블을 확장시키고 새로운 해시함수를 사용하여 모든 키들을 새로운 해시테이블에 다시 저장하는 재해시가 필요하다.
-재해시는 오프라인에서 이루어지고 모든 키들을 다시 저장해야 하므로 O(N) 시간이 소요된다.
재해시 수행 여부는 적재율에 따라 결정된다.

*동적 해싱
대용량의 데이터베이스를 위한 해시방법으로 재해시를 수행하지 않고 동적으로 해시테이블의 크기를 조절한다.
(확장 해싱과 선형 해싱이 있다.)
확장 해싱 : 디렉터리를 주기억장치에 저장하고, 데이터는 디스크 블록 크기의 버킷 단위로 저장한다. 여기서 버킷이란 키를 저장하는 곳이다.
-확장 해싱에서는 버킷에 overflow가 발생하면 새 버킷을 만들어 나누어 저장하며, 이 때 이 버킷들을 가리키던 디렉터리는 2배로 확장된다.

선형 해싱 : 디렉터리 없이 삽입할 때 버킷을 순서대로 추가하는 방식
추가되는 버킷은 삽입되는 키가 저장되는 버킷과 무관하게 순차적으로 추가된다.
-디렉터리를 사용하지 않는다는 장점을 가지며, 인터렉티브 응용에 적합하다.

해시방법의 성능 비교 및 응용
-해시방법의 성능은 탐색이나 삽입 연산을 수행할 대 성공과 실패한 경우를 각각 분석하여 측정한다.

7. 정렬
: 일반적으로 주어진 데이터를 오름차순으로 나열하는 것
정렬 알고리즘
1. 내부 정렬(Internal Sort) : 입력의 크기가 메인 메모리의 용량을 초과하지 않는 경우에 수행된다.
2. 외부 정렬(External Sort) : 보조기억장치에 있는 대용량의 데이터를 정렬함

[1] 선택정렬(Selection Sort)
리스트에서 아직 정렬 안 된 부분의 원소들 중에서 최솟값을 '선택'하여 정렬 안 된 부분의 가장 왼쪽의 원소와 교환하는 정렬 알고리즘이다.

-선택정렬의 특징은 입력이 이미 정렬되어 있거나 역으로 정렬되어 있는 경우일지라도 항상 O(N^2) 수행시간이 소요된다. (따라서 입력에 민감하지 않은 알고리즘)
-원소를 교환하는 횟수가 N-1 밖에 안 된다는 특징을 갖고 있다. (최소 교환)
-효율성 측면에서 다른 정렬 알고리즘에 비해 매우 뒤떨어지므로 활용 X

[2]삽입정렬(Insert Sort)
리스트가 정렬된 부분과 정렬 안 된 부분으로 나뉘며, 정렬 안 된 부분의 가장 왼쪽 원소를 정렬된 부분에 '삽입'하는 방식의 정렬 알고리즘이다.

-입력에 민감한 알고리즘으로, 입력이 거의 정렬되어 있을 때 정렬이 매우 빠르고, 입력이 역으로 정렬되어 있을 때에는 최악의 경우로 매우 느리다.
-입력이 이미 정렬된 경우에는 N-1번만 비교하면 되지만, 최악의 경우 O(N^2)
-

Sorting	장점	단점
버블 정렬	- 구현이 쉽다	- 시간이 오래 걸리고 비효율적이다.
선택 정렬	- 구현이 쉽다 - 비교하는 횟수에 비해 교환이 적게 일어난다.	- 시간이 오래 걸려서 비효율적이다.
퀵 정렬	- 실행시간 O(NlogN)으로 빠른 편이다	- Pivot에 의해서 성능의 차이가 심하다. - 최악의 경우 $O(N^2)$이 걸리게 된다.
힙 정렬	- 항상 O(NlogN)으로 빠른 편이다.	- 실제 시간이 다른 O(NlogN)이 정렬법들에 비해서 오래걸린다.
병합 정렬	- 항상 O(NlogN)으로 빠른 편이다.	- 추가적인 메모리 공간을 필요로 한다.
삽입 정렬	- 최선의 경우 O(N)으로 굉장히 빠른 편이다. - 성능이 좋아서 다른 정렬법에 일부로 사용됨.	- 최악의 경우$O(N^2)$ 으로, 데이터의 상태에 따라서 성능 차이가 심하다.
쉘 정렬	- 삽입정렬보다 더 빠른 정렬법이다.	- 설정하는 '간격'에 따라서 성능 차이가 심하다.
기수 정렬	- O(N)이라는 말도 안 되는 속도를 갖는다.	- 추가적인 메모리가 '많이' 필요하다. - 데이터 타입이 일정해야 한다. - 구현을 위한 조건이 많이 붙는다.
카운팅 정렬	- O(N)이라는 말도 안 되는 속도를 갖는다.	- 추가적인 메모리 공간이 필요하다. - 일부 값 때문에 메모리의 낭비가 심해질 수 있다.

Sorting	최악의 경우(Worst)	일반(평균)적인 경우(Average)	최선의 경우(Best)
버블 정렬	$O(N^2)$	$O(N^2)$	$O(N^2)$
선택 정렬	$O(N^2)$	$O(N^2)$	$O(N^2)$
퀵 정렬	$O(N^2)$	$O(NlogN)$	$O(NlogN)$
힙 정렬	$O(NlogN)$	$O(NlogN)$	$O(NlogN)$
병합 정렬	$O(NlogN)$	$O(NlogN)$	$O(NlogN)$
삽입 정렬	$O(N^2)$	$O(N^2)$	$O(N)$
쉘 정렬	$O(N^2)$	$O(N^{1.3,1.5})$	$O(N)$
기수 정렬	$O(N)$	$O(N)$	$O(N)$
카운팅 정렬	$O(N)$	$O(N)$	$O(N)$

O(1) > O(logN) > O(N) > O(NlogN) > O(N^2) > O(2^N) > O(N!)

[3]쉘정렬(Shell)
삽입정렬을 이용하여 작은 값을 가진 원소들을 리스트의 앞부분으로 옮기며 큰 값을 가진 원소들이 리스트의 뒷부분에 자리 잡도록 만드는 과정을 반복하여 정렬하는 알고리즘이다.

-쉘정렬은 삽입정렬에 전처리 과정을 추가한 것이다.
-전처리 과정이란 작은 값을 가진 원소들을 리스트의 앞부분으로 옮기며 큰 값을 가진 원소들이 리스트의 뒷부분에 자리 잡도록 만드는 과정을 의미한다.
-h-정렬 : 일반적으로 간격이 h인 원소들끼리 정렬하는 것

안정성
안정 O : 삽입정렬, 합병정렬, Tim Sort
안정 X : 선택정렬, 쉘정렬, 힙정렬, 퀵정렬