**연결 리스트[Linked Lists]**

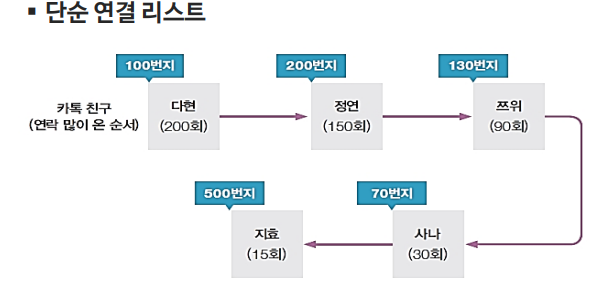
데이터를 저장하는 노드(Node)들이 **포인터**를 사용해 선형으로 연결된 자료 구조입니다.

컴퓨터 과학에서 사용하는 기본적인 선형 자료 구조 중 하나다. 링크드 리스트는 각 요소가 데이터와 다음 요소를 참조(링크)하는 정보를 포함하는 노드로 구성된다. 이러한 특징 덕분에 링크드 리스트는 데이터의 동적 추가, 삭제를 상대적으로 쉽게 할 수 있는 장점이 있다.

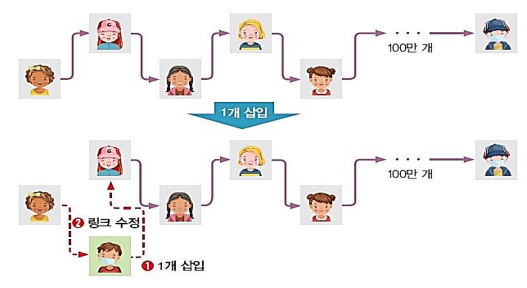
#노드들이 물리적으로 떨어진 곳에 위치

#각 노드의 번지도 순차적이지 않음

#화살표로 표시된 연결(링크, Link)을 따라가면 선형 리스트 순서와 같음 ex)

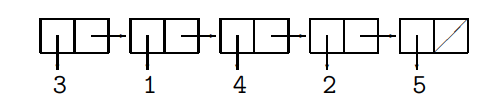


#데이터를 삽입/삭제할 경우



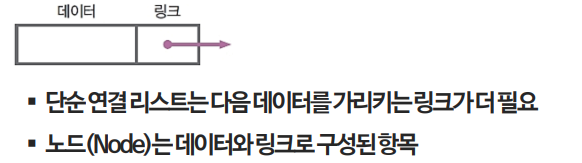
리스트는 **두 개의 셀**로 표현될 수 있습니다. 첫 번째 셀은 리스트 요소를 가리키는 포인터를 포함하고, 두 번째 셀은 비어 있는 리스트 또는 다른 두 셀을 가리키는 포인터를 포함합니다. 비어 있는 리스트를 가리키는 포인터는 대각선 막대 또는 교차 기호로 나타낼 수 있습니다.

<그래픽 표현: list [3, 1, 4, 2, 5]>



링크드 리스트의 핵심 요소

1. **노드(Node)**: 링크드 리스트의 기본 단위로서, 데이터를 저장하는 **데이터 필드** 와 다음 노드를 가리키는 **링크 필드** 로 구성된다.
2. **포인터(pointer)**:각 노드 안에서, 다음이나 이전의 노드와의 연결 정보를 가지고 있는 공간
3. **헤드(Head)**: 링크드 리스트에서 가장 처음 위치하는 노드를 가리킨다. 리스트 전체를 참조하는데 사용된다.
4. **테일(Tail)**: 링크드 리스트에서 가장 마지막 위치하는 노드를 가리킨다. 이 노드의 링크 필드는 Null을 가리킨다.



**추상 데이터 타입(Abstract Data Type) “List”**

추상적인 수준에서 리스트는 두 가지 생성자를 통해 구성할 수 있습니다

**EmptyList**, which gives you the empty list 비어 있는 리스트를 생성

**MakeList(element, list)**, which puts an element at the top of an existing list. 기존 리스트 위에 요소를 추가

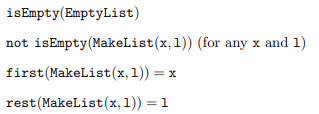
 list [3, 1, 4, 2, 5]

이러한 귀납적 데이터 구조 생성 접근법은 매우 강력하며, 이 모듈에서 여러 번 사용될 것입니다. 이 방법은 "기본 사례"인 EmptyList로 시작하여, MakeList(element, list) 연산자를 반복적으로 적용하여 점점 더 복잡한 리스트를 구성합니다.

**리스트 요소 접근**

리스트는 항상 첫 번째 요소와 나머지 리스트로 구성된다는 점에 주목해야 합니다. 따라서 비어 있지 않은 리스트에서 첫 번째 요소와 나머지 리스트를 가져올 수 있습니다. 비어 있지 않은 리스트에서만 작동하며, 비어 있는 리스트에 대해 에러를 발생시킵니다. 리스트가 비어 있는지 확인해야 합니다.

선택자 (Selector)는 first(list): *리스트의 첫 번째 요소를 반환*, rest(list): *나머지 리스트를 반환*, 조건은 isEmpty(list): *리스트가 비어 있는지 확인*



**변경자[mutator]**





**XML Representation**

XML(eXtensible Markup Language)은 사실상의 표준으로 자리 잡았습니다. 예를 들어, 앞서 언급한 리스트 [3, 1, 4, 2, 5]는 다음과 같이 XML로 표현할 수 있습니다:



XML은 매우 복잡한 구조를 균일한 방식으로 표현하고 통신할 수 있을 만큼 유연합니다.

간단한 리스트에 비해 복잡해 보이지만, 그렇지는 않습니다. 단지 길이가 조금 길 뿐입니다.

**리스트의 구현[Implementation of Lists]**

사용하는 프로그래밍 언어와 리스트의 특성에 따라 가장 적합한 방법이 달라집니다.

예를 들어, Lisp 같은 언어는 리스트를 기본 데이터 구조로 사용하기 때문에 포인터 기반 구현이 자연스럽습니다.

하지만, 리스트의 크기가 미리 정해져 있는 경우에는 배열 기반 구현이 더 효율적일 수 있습니다.

#리스트를 구현하는 방법은 다양하며, 각 방법마다 장단점이 있습니다. 따라서, 어떤 방법을 선택할지는 문제의 특성과 사용하는 프로그래밍 언어를 고려하여 결정해야 합니다.

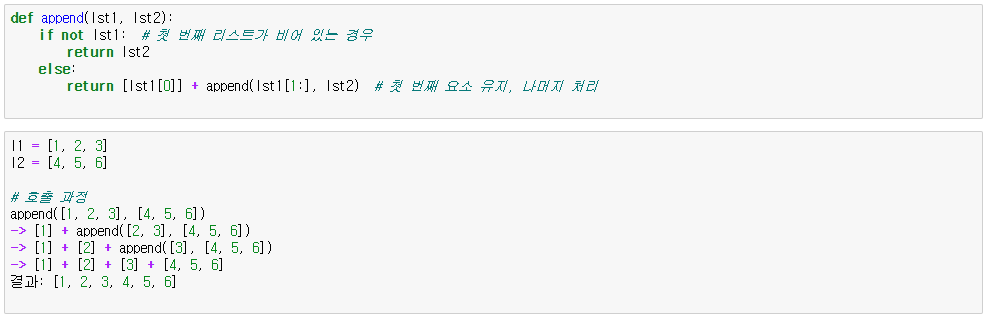
**재귀[Recursion]**

어떤 함수가 자신을 다시 호출하는 프로그래밍 기법입니다. 이 기법은 특정 작업을 여러 단계로 나누어야 할 때 매우 유용합니다. 배열(array)에서는 인덱스를 사용해 빠르게 원하는 위치의 요소에 접근할 수 있습니다. 그러나 연결 리스트는 한 요소에서 다음 요소로 이동해야 하기 때문에, 배열처럼 반복문으로 처리하기보다는 **재귀**를 사용하는 것이 더 직관적일 때가 많습니다. **인덱스가 없는 자료 구조를 처리할 때 자연스럽게 사용됩니다**.

**last**: 연결 리스트의 마지막 요소 찾기 #과정은 리스트의 길이만큼 반복됩니다.



**append**: 두 연결 리스트를 이어붙이기





**재귀함수를 사용하는 이유**

**1. 재귀적 표현이 자연스러운 알고리즘일 경우**

- 대표적인 예시로 **피보나치 수열 점화식**이 있다.

*\*피보나치 수열 점화식 ( f(n) = f(n-1) + f(n-2) )*

결과값을 구하기 위해 함수인자만 변경하고 자기 자신을 다시 호출해야 하는 경우이다.

이럴 경우 알고리즘을 기술한 그대로 코드로 표현할 수 있다.

**2. 변수 사용을 줄여준다.**

- 변수가 차지하는 **메모리에 대한 부분이 아닌** **mutable state (변경 가능한 상태)를 제거**하여

프로그램 오**류가 발생할 수 있는 가능성**을 줄일 수 있다.

*함수형 프로그래밍에서 함수를 일급객체 취급하여 인자전달 하는것과 비슷한 맥락인듯 하다.*

**3. 가독성**

2번 특징에 더하여 재귀는 반복문에 비해 코드량이 적다. 때문에 **가독성을 향상**시키는 코드를 짤 수 있다.

**! 재귀 알고리즘의 필수 요소**

**기저 조건[Base case]:** 적어도 하나의 recursion에 빠지지 않는 경우가 존재해야 한다.

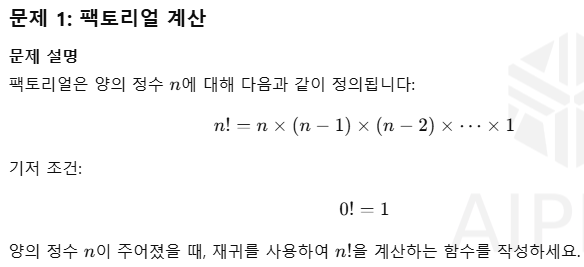
**재귀 조건[Recursive case]:** recursion을 반복하다보면 결국 base case로 수렴해야 한다.

입력값의 점진적인 감소 또는 수렴: 입력값이 재귀 호출을 통해 점점 줄어들거나 기저 조건에 가까워져야 합니다.

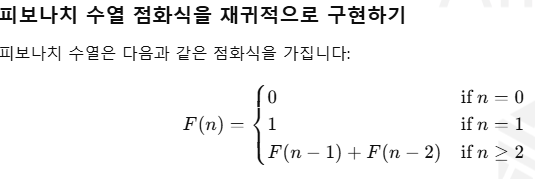
종료 가능성(Termination): 재귀 호출이 반드시 멈추도록 보장해야 합니다.

올바른 반환값(Return Value): 재귀 호출의 결과를 활용하거나 반환값을 계산해야 합니다.

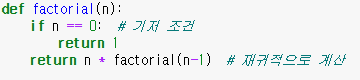
**문제 –**

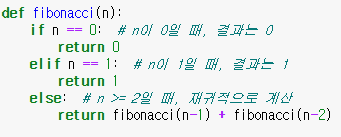
****

**문제 2:**

****

정답

****

****

**스택[Stacks]** 스택은 후입 선출(Last In First Out) 개념이다.

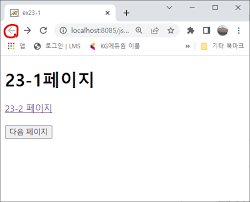
추상적인 수준에서 연결 리스트와 동등합니다. 이들은 검색에서 선입후출(FILO) 또는 후입선출(LIFO) 전략을 모델링하는 이상적인 데이터 구조입니다.

**# FILO (First-In-Last-Out)**

첫 번째로 들어간 데이터(First-In)가 마지막에 나온다(Last-Out)는 의미입니다.

*처음 저장된 데이터가 가장 마지막에 꺼내어짐.*

Ex. 쌓아둔 접시

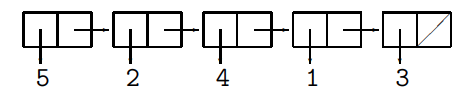
**# LIFO (Last-In-First-Out)**

마지막으로 들어간 데이터(Last-In)가 가장 먼저 나온다(First-Out)는 의미입니다.

*가장 최근에 저장된 데이터가 가장 먼저 꺼내어짐.*

Ex. 웹 브라우저의 뒤로 가기 버튼

<그래픽 표현: list [3, 1, 4, 2, 5]>



생산자(constructors)

**EmptyStack**: 빈 스택

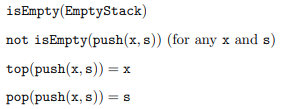
**push(element, stack)**: 요소를 받아 기존 스택의 맨 위에 푸시합니다.

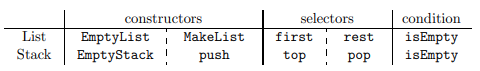
선택자(selectors)  
**top(stack):** 스택의 맨 위 요소를 반환합니다.

**pop(stack):** 맨 위 요소를 제거한 스택을 반환합니다.

조건(condition) 선택자는 비어 있지 않은 스택에 대해서만 작동하므로, 스택이 비어 있는지 여부를 알려주는 조건이 필요합니다

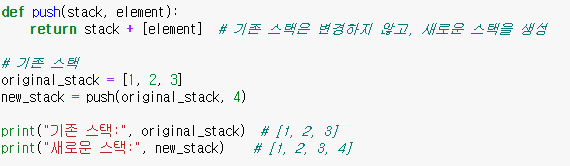
**isEmpty(stack):** 스택이 비어 있는지 여부를 알려줍니다

****

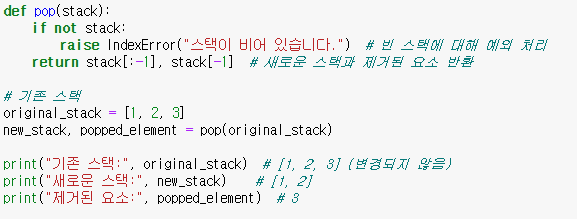
스택과 연결 리스트는 생성자와 선택자의 이름만 다를 뿐 본질적으로 동일한 구조입니다.****

**스택의 구현  
함수형 접근 (Functional Approach)**

* 특징: 기존 데이터를 변경하지 않고, 새로운 데이터를 생성합니다. 따라서 \*\*불변성(immutability)\*\*을 유지합니다.
* 장점: 동일한 입력에 대해 항상 동일한 출력을 보장하므로, 디버깅과 테스트가 용이합니다.
* 단점: 새로운 데이터를 생성하므로, 메모리 사용량이 증가할 수 있습니다.



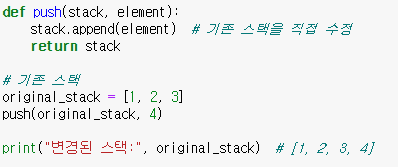
여기서 original\_stack은 그대로 유지되고, 새로운 스택 new\_stack이 생성되었습니다.



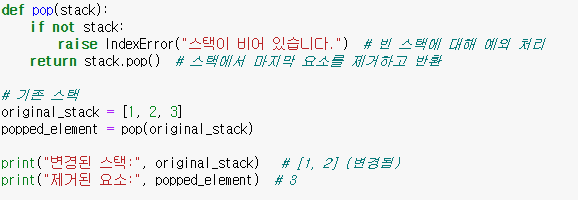
기존 스택은 변경되지 않고 그대로 유지됩니다.

**파괴적 접근 (Destructive Approach)**

* **특징**: 기존 데이터를 직접 변경합니다. 데이터가 가변적(mutable)이므로, 상태 변화가 발생합니다.
* **장점**: 메모리 사용량이 적고, 속도 면에서 더 효율적일 수 있습니다.
* **단점**: 데이터의 상태가 바뀌므로, 의도치 않은 동작이 발생할 수 있어 주의가 필요합니다.



여기서는 original\_stack 자체가 변경되었음을 알 수 있습니다.



스택의 상태가 변경되므로, original\_stack 자체가 수정됩니다.

**결론**

* 함수형 접근은 안전하게 상태를 유지하며 코드를 작성할 수 있습니다.
* 파괴적 접근은 간결하고 메모리 효율적인 작업에 유리하지만, 상태 관리에 주의해야 합니다.
* 어떤 접근을 사용할지는 **코드의 목적**과 **안정성** 또는 **성능** 요구 사항에 따라 선택하면 됩니다.

**두 접근 방식의 비교**

|  |  |  |
| --- | --- | --- |
| **구분** | **함수형 접근** | **파괴적 접근** |
| **데이터 변경** | 기존 데이터 변경 없음 | 기존 데이터 변경 |
| **메모리 사용** | 더 많은 메모리 필요 | 상대적으로 적은 메모리 사용 |
| **디버깅 용이성** | 디버깅과 테스트에 유리 | 상태 변화 추적이 어려울 수 있음 |
| **적용 예시** | 함수형 프로그래밍, 불변성이 필요한 경우 | 성능 최적화가 중요한 경우 |

두 방식은 사용 목적과 상황에 따라 선택하면 됩니다. 함수형 접근은 안정적이고 안전한 코드를 작성할 때 유용하며, 파괴적 접근은 성능과 메모리 효율성이 중요한 경우 적합합니다.