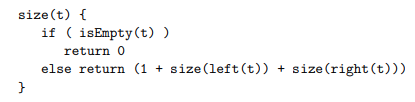
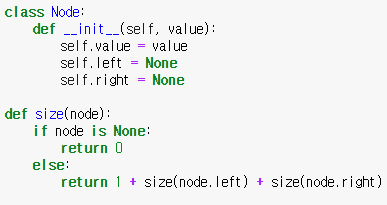
**이진 트리의 크기[The size of a binary tree]**

일반적으로 이진 트리는 완벽하게 균형 잡혀 있지 않기 때문에, 트리에 포함된 노드의 개수인 크기를 결정하기 위한 알고리즘이 필요합니다.

재귀 함수를 사용하면 쉽게 구현할 수 있습니다. 재귀의 종료 조건은 매우 간단합니다. 빈 트리의 크기는 0입니다. 그렇지 않은 경우, 모든 이진 트리는 루트 노드, 왼쪽 서브트리 l, 오른쪽 서브트리 r로 구성되며, 크기는 구성 요소의 크기의 합, 즉 루트의 크기 1, l의 크기, r의 크기의 합입니다

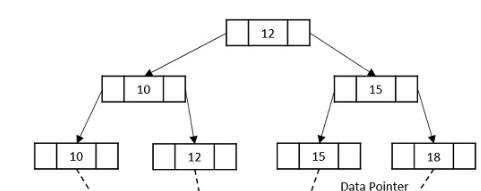


이 함수는 전체 트리를 재귀적으로 처리하며, 처리되는 트리가 각 호출마다 작아지고 결국 0을 반환하는 빈 트리에 도달하기 때문에 종료된다는 것을 알 수 있습니다.



**트리의 구현[Implementation of trees]**

**트리**는 계층적인 구조를 가진 데이터 구조입니다. 각 노드는 하나 이상의 자식 노드를 가질 수 있으며, 이러한 관계가 트리의 형태를 이룹니다.



이진 트리는 각 노드에 대해 단순히 **노드 값**과 두 개의 **자식 노드 포인터**로 구성된 데이터 레코드로 구현할 수 있습니다.

**노드의 구성:**

* 각 노드는 데이터 값과 자식 노드를 가리키는 포인터들로 구성됩니다.
* 자식 노드의 개수는 트리의 종류에 따라 다릅니다. 예를 들어, 이진 트리는 각 노드가 최대 두 개의 자식 노드를 가질 수 있습니다.

**루트 노드:**

* 트리의 최상위 노드를 루트 노드라고 합니다.
* 루트 노드는 부모 노드가 없습니다.

**자식 노드:**

* 각 노드는 0개 이상의 자식 노드를 가질 수 있습니다.
* 자식 노드는 부모 노드의 하위 레벨에 위치합니다.

**효율적인 검색과 삽입**: 특정 노드를 찾거나 삽입하는 데 효율적인 알고리즘을 사용할 수 있습니다.

**계층적 데이터 표현**: 계층적인 데이터를 직관적으로 표현할 수 있습니다.

**재귀적 알고리즘의 적용**: 나무 구조는 재귀적인 알고리즘을 효과적으로 적용할 수 있습니다.

**재귀 알고리즘 [Recursive Algorithms]**

재귀(recursion)란 "알고리즘이 스스로를 호출한다"는 표현 입니다. 하지만, 실제로는 알고리즘 자체는 단순히 수동적인 존재일 뿐이며, 스스로 어떤 작업을 수행하거나 자신을 호출할 수 없습니다. 실제로 일어나는 일은 **프로세서(processor)**(사람이나 기계)가 알고리즘을 실행하는 것입니다.

Ex.

1. **트리 t가 비어있는지 확인**:

* 비어 있다면 0을 반환하고 계산을 종료합니다.

2. **트리가 비어 있지 않은 경우**:

* 트리 t는 왼쪽 서브트리 l과 오른쪽 서브트리 r를 가지며, 이는 left(t)와 right(t) 셀렉터를 사용해 추출할 수 있습니다.
* John은 그의 학생인 Steve와 Mary에게 각각 트리 l과 r에 대해 동일한 알고리즘을 실행하도록 요청합니다.

3. **학생들이 결과를 반환**:

* 예를 들어, Steve가 m을 반환하고, Mary가 n을 반환했다고 합시다. 그러면 John은 1 + m + n을 계산해 자신의 트리 크기를 구합니다. 여기서 1은 루트 노드를 의미합니다.

4. **재귀적 위임**:

* 만약 Steve와 Mary가 빈 트리가 아닌 서브트리를 받았다면, 그들 역시 자신의 서브트리에 대해 동일한 알고리즘을 실행하도록 다른 사람들에게 작업을 위임합니다.

알고리즘이 스스로를 호출하는 것이 아니라, 여러 사람이 동일한 알고리즘 복사본을 사용해 각각 다른 트리를 처리하는 방식으로 작동합니다.

##**트리의 재귀적 정의** 덕분에, 서브트리 역시 트리입니다.

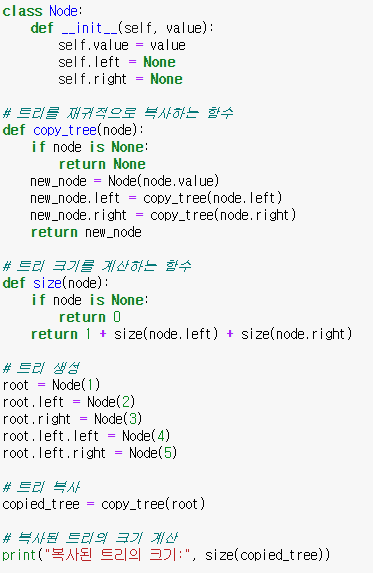
루트 노드에서 왼쪽 자식 노드(왼쪽 서브트리)를 가리키는 포인터만 알고 있으면, 전체 왼쪽 서브트리를 처리할 수 있습니다. 오른쪽 서브트리도 동일한 방식으로 처리됩니다.

트리는 재귀적으로 정의되는 데이터 구조이기 때문에, **포인터만 사용**하여 서브트리를 가리키면 **전체 트리를 복사하거나 새롭게 저장하지 않아도 됩니다**. 이렇게 하면 **효율적으로 메모리와 처리 시간을 절약**하면서 작업을 수행할 수 있습니다.

Ex.

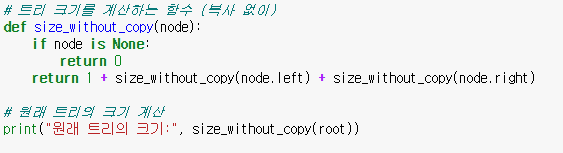
**트리 전체 복사 방식**

트리 전체를 복사한 후 복사된 트리를 이용하여 크기를 계산하는 방식입니다.



**포인터만 전달 방식**

트리를 복사하지 않고 원래 트리의 포인터만 사용하여 크기를 계산합니다.



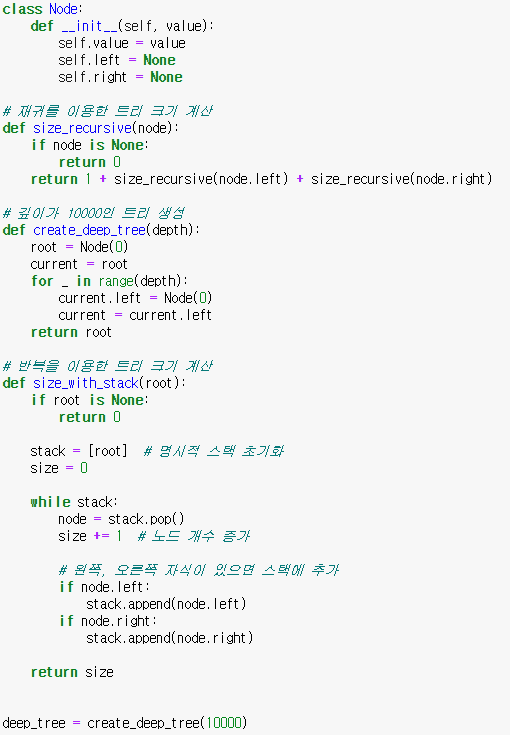
|  |  |  |
| --- | --- | --- |
| 항목 | 전체 복사 방식 | 포인터만 전달 방식 |
| 메모리 사용량 | 트리를 복사하므로 노드 수만큼 추가 메모리 사용 | 추가 메모리 사용 없음 |
| 시간 복잡도 | O(n) 복사 시간 + O(n) 크기 계산 → 총 O(2n) | O(n) 크기 계산 |
| 효율성 | 비효율적 (큰 트리에서는 과도한 메모리 소모) | 매우 효율적 |
| 코드의 간단함 | 복사 함수 작성 필요 | 간단한 재귀 호출로 해결 가능 |

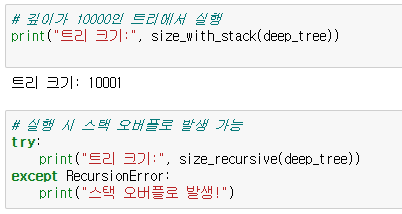
**하나의 프로세서로 처리하는 경우: 스택을 활용한 재귀 알고리즘 구현**

재귀 알고리즘은 여러 "프로세서"[사람이나 기계가 동시에 각각의 작은 처리하는 것처럼 동작합니다.]가 각각 자신의 작업을 수행하는 방식으로 이해할 수 있지만, 실제로는 하나의 프로세서가 **스택(stack)**을 사용하여 동일한 작업을 흉내 낼 수 있습니다. 스택은 함수 호출과 해당 실행 위치를 추적하여 처리하는 데 사용됩니다.

재귀 호출은 함수가 자기 자신을 호출하는 방식으로 동작하며, 호출마다 호출 스택(Call Stack)에 새로운 프레임이 추가됩니다. 하지만 트리가 너무 크거나 재귀 호출 깊이가 제한을 초과하면 **스택 오버플로(재귀 호출은 함수가 자기 자신을 호출하는 방식으로 동작하며, 호출마다 호출 스택(Call Stack)에 새로운 프레임이 추가됩니다. 하지만 트리가 너무 크거나 재귀 호출 깊이가 제한을 초과하면 스택 오버플로(Stack Overflow) 문제가 발생할 수 있습니다.)** 문제가 발생할 수 있습니다.

스택 오버플로 ez.데이터를 담아두는 더미를 넘어섰다





재귀 호출 대신 명시적 스택을 사용하는 반복 방식은 **스택 오버플로 문제를 방지**하며, **큰 트리에서도 안정적으로 작동**할 수 있습니다. 이는 호출 스택의 크기 제한을 우회하여 더 많은 데이터를 처리할 수 있도록 합니다.

|  |  |  |
| --- | --- | --- |
| 항목 | 재귀 방식 | 반복 방식 (명시적 스택) |
| 스택 오버플로 | 깊은 트리에서는 발생 가능 | 발생하지 않음 |
| 메모리 사용량 | 호출 스택 깊이와 비례하여 증가 | 명시적 스택 크기에 따라 제어 가능 |
| 코드의 간결함 | 간결하고 직관적 | 약간 더 복잡함 |
| 안정성 | 트리 깊이에 따라 불안정 | 큰 트리에서도 안정적 |