**자료구조 05**

****

**중앙대학교**

**자료구조 05 분반**

**20165079 김영빈**

**목차**

과제

[**1.** **수식 트리 만들기** 3](#_Toc72813580)

[**2.** **사전 탐색 트리 만들기** 14](#_Toc72813581)

# **수식 트리 만들기**

* 1. **클래스와 메서드 정의**

*#Queue는 Queue의 역할을 할 수 있도록 간단하게만 구현함.*

class Queue():

**def** \_\_init\_\_(*self*):

*self*.items **=** []

**def** \_\_len\_\_(*self*):

**return** len(*self*.items)

**def** isEmpty(*self*):

**return** *self*.\_\_len\_\_() **==** 0

**def** enqueue(*self*, **item**):

*self*.items.append(item)

**def** dequeue(*self*):

**if** *self*.isEmpty():

            print("Nothing in Queue")

**else**:

            target **=** *self*.items[0]

*self*.items **=** *self*.items[1:]

**return** target

class Node :

**def** \_\_init\_\_(*self*, **item**, **left=**None, **right=**None) :

*self*.item **=** item

*self*.left **=** left

*self*.right **=** right

**def** \_\_str\_\_(*self*):

**return** *self*.item

class Tree :

**def** \_\_init\_\_(*self*):

*#tree의 root 노드를 pointing 할 변수*

*self*.root **=** None

*#preorder를 위한 class method*

    @classmethod

**def** preorder(*cls*, **n**):

**if** n **!=** None :

            print(n.item,**end=**" ")

*cls*.preorder(n.left)

*cls*.preorder(n.right)

*#inorder를 위한 class method*

    @classmethod

**def** inorder(*cls*, **n**):

**if** n **!=** None :

*cls*.inorder(n.left)

            print(n.item, **end=**" ")

*cls*.inorder(n.right)

*#postorder를 위한 class method*

    @classmethod

**def** postorder(*cls*, **n**):

**if** n **!=** None :

*cls*.postorder(n.left)

*cls*.postorder(n.right)

            print(n.item, **end=**" ")

*#levelorder를 위한 class method*

    @classmethod

**def** levelorder(*cls*, **n**):

**if** n **!=** None:

            queue **=** Queue()

            queue.enqueue(n)

**while** **not** queue.isEmpty():

                n **=** queue.dequeue()

**if** n **is** **not** None:

                    print(n.item, **end=**" ")

                    queue.enqueue(n.left)

                    queue.enqueue(n.right)

*#위에서 구현한 tree를 이용하여 만든 계산기 class*

class Calculator :

**def** \_\_init\_\_(*self*):

*#사용할 tree 인스턴스 변수*

*self*.tree **=** Tree()

*#사용자에게 입력받은 식을 저장할 인스턴스 변수*

*self*.equation **=** None

*#operator weight를 계산할 토큰*

*self*.operator **=** "+-\*/"

*#식의 값을 저장할 인스턴스 변수*

*self*.value **=** 0

*#사용자에게 식을 입력받는 메서드*

**def** get\_equation(*self*):

*self*.equation **=** input("수식을 입력하세요 : ").strip()

*#사용자에게 입력받은 식으로 트리를 구성하는 메서드*

**def** get\_tree(*self*):

        i **=** 1

**while** i **<** len(*self*.equation) **-** 1:

**if** *self*.tree.root **==** None:

                sub\_tree **=** *self*.get\_sub\_tree(Node(*self*.equation[i **-** 1]), Node(*self*.equation[i]), Node(*self*.equation[i **+** 1]))

*self*.tree.root **=** sub\_tree

**elif** *self*.operator.index(*self*.tree.root.item) **//** 2 **<** *self*.operator.index(*self*.equation[i])**//** 2:

*self*.tree.root.right **=** *self*.get\_sub\_tree(*self*.tree.root.right, Node(*self*.equation[i]), Node(*self*.equation[i **+** 1]))

**else**:

*self*.tree.root **=** *self*.get\_sub\_tree(*self*.tree.root, Node(*self*.equation[i]), Node(*self*.equation[i **+** 1]))

            i **+=** 2

*#2개의 피연산자와 1개의 연산자를 이용하여 서브 트리를 만드는 메서드*

**def** get\_sub\_tree(*self*, **left**, **root**, **right**):

        root.left **=** left

        root.right **=** right

**return** root

*#결과를 프린트해주는 메서드*

**def** print\_result(*self*):

        print("전위 순회 : ", **end=**"")

*self*.tree.preorder(*self*.tree.root)

        print("\n중위 순회 : ", **end=**"")

*self*.tree.inorder(*self*.tree.root)

        print("\n후위 순회 : ", **end=**"")

*self*.tree.postorder(*self*.tree.root)

        print("\n레벨 순회 : ", **end=**"")

*self*.tree.levelorder(*self*.tree.root)

*self*.get\_value()

*#결과값을 출력하고 초기화 하는 메서드*

**def** get\_value(*self*):

*self*.value **=** *self*.postorder\_calculate(*self*.tree.root)

        print("\n계산 결과 :", *self*.value)

*#Calculator의 run 메서드를 무한반복으로 실행했기 때문에 다음 run이 실행되기 전에 3개의 인스턴스 변수를 초기화 시켜줌*

*self*.value **=** 0

*self*.equation **=** None

*self*.tree.root **=** None

*#후위 계산으로 결과 값을 구하는 메서드*

**def** postorder\_calculate(*self*, **n**):

**if** n **!=** None :

            left **=** *self*.postorder\_calculate(n.left)

            op **=** (n.item)

            right **=** *self*.postorder\_calculate(n.right)

**if** op **==** "+":

**return** left **+** right

**elif** op **==** "-":

**return** left **-** right

**elif** op **==** "\*":

**return** left **\*** right

**elif** op **==** "/":

**return** left **/** right

**else**:

**return** int(op)

*#계산기를 실행하는 메서드*

**def** run(*self*):

*self*.get\_equation()

*self*.get\_tree()

*self*.print\_result()

과제 1번을 수행하기 위해서 선언한 클래스는 다음과 같다.

|  |  |
| --- | --- |
| **클래스** | **내용** |
| Queue | Tree의 levelorder 메서드에서 사용할 큐 |
| Node | 식의 피연산자 혹은 연산자를 저장할 노드 |
| Tree | 노드를 연결해서 트리 모양으로 저장하는 자료 구조 |
| Calculator | 트리를 이용해서 수식을 출력하고 계산하는 클래스 |

Queue는 Tree클래스에서 클래스 메서드 levelorder를 구현하는데 사용하기 위해서 선언했다. 해당 과제에서는 Queue의 구현보다는 Tree의 구현에 좀 더 목적성을 두기 위해서 Queue의 역할을 할 수 있을 정도의 메서드와 변수만을 구현했다.

|  |  |
| --- | --- |
| **변수 or 메서드** | **내용** |
| Items | 데이터들이 저장되는 장소 |
| isEmpty() | 큐의 items 변수에 데이터의 존재 여부를 bool 값으로 반환하는 메서드 |
| Enqueue() | 데이터를 items의 맨 뒤에 넣는 메서드 |
| Dequeue() | Items의 맨 앞에서 데이터를 뽑아내는 메서드 |

Node 클래스는 수식을 구성하는 피 연산자와 연산자를 저장하기 위해서 구현했다. 각 노드는 정보를 저장할 item 변수와 왼쪽 child를 가리키는 left, 오른쪽 child를 가리키는 right 변수를 가지고 있다. 만약 Node 클래스를 이용해서 인스턴스를 생성할 때 파라미터로 아무것도 전달되지 않는다면 left와 right값은 None 값을 가리키게 했다.

|  |  |
| --- | --- |
| **변수 or 메서드** | **내용** |
| Item | 식의 피 연산자 혹은 연산자를 저장하는 변수 |
| Lefrt | 해당 Node의 left child를 가리키는 변수 |
| Right | 해당 Node의 right child를 가리키는 변수 |

Tree와 Calculator 클래스에 대해서는 아래에 자세하게 설명하겠다.

* 1. **실행**

c **=** Calculator()

**while** True:

    c.run()

구현한 계산기 클래스의 인스턴스를 하나 생성하여 c에 할당하고 이를 계속 동작하게 했다.

* 1. **Tree 클래스**

class Tree :

**def** \_\_init\_\_(*self*):

*#tree의 root 노드를 pointing 할 변수*

*self*.root **=** None

*#preorder를 위한 class method*

    @classmethod

**def** preorder(*cls*, **n**):

**if** n **!=** None :

            print(n.item,**end=**" ")

*cls*.preorder(n.left)

*cls*.preorder(n.right)

*#inorder를 위한 class method*

    @classmethod

**def** inorder(*cls*, **n**):

**if** n **!=** None :

*cls*.inorder(n.left)

            print(n.item, **end=**" ")

*cls*.inorder(n.right)

*#postorder를 위한 class method*

    @classmethod

**def** postorder(*cls*, **n**):

**if** n **!=** None :

*cls*.postorder(n.left)

*cls*.postorder(n.right)

            print(n.item, **end=**" ")

*#levelorder를 위한 class method*

    @classmethod

**def** levelorder(*cls*, **n**):

**if** n **!=** None:

            queue **=** Queue()

            queue.enqueue(n)

**while** **not** queue.isEmpty():

                n **=** queue.dequeue()

**if** n **is** **not** None:

                    print(n.item, **end=**" ")

                    queue.enqueue(n.left)

                    queue.enqueue(n.right)

Calculator 클래스에서 사용할 Tree 자료구조를 하나의 클래스로 구현했다. 해당 클래스는 트리의 최 상단에 위치한 root node를 root라는 변수로 기억하고 있다. 이후에 Tree 인스턴스를 이용하여 연산을 진행할 때 해당 root를 통해 root node에 접근해 필요한 기능을 수행할 수 있다.

Tree를 순회하는 방법은 4가지가 있다. 이 4가지 순회 방법들 중 전위, 중위, 후위 방법은 교수님께서 강의에서 언급하셨듯이 각 sub tree의 root node에 대한 처리를 어느 시점에 해주는지에 따라 동작의 차이를 보여준다. Preorder(전위)는 root node에 대한 처리를 먼저하고 root node의 왼쪽 child, root node의 오른쪽 child 순서로 처리를 진행한다. Inorder(중위)는 root node의 왼쪽 child에 대한 처리를 먼저 한 뒤, root node에 대한 처리를 하고, 마지막에 root node의 오른쪽 child에 대한 처리를 한다. Postorder(후위)는 root node의 왼쪽, 오른쪽 child에 대한 처리를 각각 한 뒤에, root node에 대한 처리를 가장 마지막에 한다. Levelorder은 Queue 자료 구조의 도움을 받아 level 0에 위치한 root node 부터 자식들을 차례대로 enqueue하여 가장 아래의 level에 있는 leaf 노드까지 각 level 별로 처리를 하는 것이다. 이번 과제에서는 각 방법들이 트리로 저장되어 있는 수식을 출력하는 용도로 사용되기 때문에 각 Node가 저장하고 있는 item을 출력하는 것을 하나의 처리로 보았다.

|  |  |
| --- | --- |
| **변수 or 메서드** | **내용** |
| Root | Tree의 root 노드를 pointing 할 변수 |
| Preorder() | Preorder를 위한 class 메서드 |
| Inorder() | Inorder를 위한 class 메서드 |
| Postorder() | Postorder를 위한 class 메서드 |
| Levelorder() | Levelorder를 위한 class 메서드 |

* 1. **Calculator 클래스**

class Calculator :

**def** \_\_init\_\_(*self*):

*#사용할 tree 인스턴스 변수*

*self*.tree **=** Tree()

*#사용자에게 입력받은 식을 저장할 인스턴스 변수*

*self*.equation **=** None

*#operator weight를 계산할 토큰*

*self*.operator **=** "+-\*/"

*#식의 값을 저장할 인스턴스 변수*

*self*.value **=** 0

*#사용자에게 식을 입력받는 메서드*

**def** get\_equation(*self*):

*self*.equation **=** input("수식을 입력하세요 : ").strip()

*#사용자에게 입력받은 식으로 트리를 구성하는 메서드*

**def** get\_tree(*self*):

        i **=** 1

**while** i **<** len(*self*.equation) **-** 1:

**if** *self*.tree.root **==** None:

                sub\_tree **=** *self*.get\_sub\_tree(Node(*self*.equation[i **-** 1]), Node(*self*.equation[i]), Node(*self*.equation[i **+** 1]))

*self*.tree.root **=** sub\_tree

**elif** *self*.operator.index(*self*.tree.root.item) **//** 2 **<** *self*.operator.index(*self*.equation[i])**//** 2:

*self*.tree.root.right **=** *self*.get\_sub\_tree(*self*.tree.root.right, Node(*self*.equation[i]), Node(*self*.equation[i **+** 1]))

**else**:

*self*.tree.root **=** *self*.get\_sub\_tree(*self*.tree.root, Node(*self*.equation[i]), Node(*self*.equation[i **+** 1]))

            i **+=** 2

*#2개의 피연산자와 1개의 연산자를 이용하여 서브 트리를 만드는 메서드*

**def** get\_sub\_tree(*self*, **left**, **root**, **right**):

        root.left **=** left

        root.right **=** right

**return** root

*#결과를 프린트해주는 메서드*

**def** print\_result(*self*):

        print("전위 순회 : ", **end=**"")

*self*.tree.preorder(*self*.tree.root)

        print("\n중위 순회 : ", **end=**"")

*self*.tree.inorder(*self*.tree.root)

        print("\n후위 순회 : ", **end=**"")

*self*.tree.postorder(*self*.tree.root)

        print("\n레벨 순회 : ", **end=**"")

*self*.tree.levelorder(*self*.tree.root)

*self*.get\_value()

*#결과값을 출력하고 초기화 하는 메서드*

**def** get\_value(*self*):

*self*.value **=** *self*.postorder\_calculate(*self*.tree.root)

        print("\n계산 결과 :", *self*.value)

*#Calculator의 run 메서드를 무한반복으로 실행했기 때문에 다음 run이 실행되기 전에 3개의 인스턴스 변수를 초기화 시켜줌*

*self*.value **=** 0

*self*.equation **=** None

*self*.tree.root **=** None

*#후위 계산으로 결과 값을 구하는 메서드*

**def** postorder\_calculate(*self*, **n**):

**if** n **!=** None :

            left **=** *self*.postorder\_calculate(n.left)

            op **=** (n.item)

            right **=** *self*.postorder\_calculate(n.right)

**if** op **==** "+":

**return** left **+** right

**elif** op **==** "-":

**return** left **-** right

**elif** op **==** "\*":

**return** left **\*** right

**elif** op **==** "/":

**return** left **/** right

**else**:

**return** int(op)

*#계산기를 실행하는 메서드*

**def** run(*self*):

*self*.get\_equation()

*self*.get\_tree()

*self*.print\_result()

Calculator클래스는 4개의 변수를 가진다. tree변수를 통해서 위에서 구현한 Tree 클래스를 사용하고 사용자가 입력한 식은 equation변수에 저장된다. Operator에 계산에 사용되는 연산자를 문자열로 저장하여 weight를 구하는데 사용하도록 했고, 식의 결과값을 저장할 value 변수도 선언했다.

Run 메서드를 통해서 Calculator 클래스가 동작하게 된다. 해당 메서드의 동작 순서를 설명하겠다. 우선 get\_equation 메서드를 통해서 사용자로부터 식을 입력 받아 equation 변수에 저장하게 된다. 그 후에 get\_tree 메서드를 통해 equation 변수에 저장되어 있는 식을 트리 형태로 저장하게 된다. 이때 중위식을 그대로 트리 형태로 바꾸기 위해서 equation을 3개의 element를 받아서 Node로 만들고 get\_sub\_tree 메서드를 호출한다. Get\_sub\_tree 메서드에 의해서 전달받은 3개의 Node 중 가운데에 있는 노드를 root 노드에 저장하고 양 옆의 노드들을 각각 root node의 left와 right child로 만든 뒤에 root를 반환한다. 이러한 과정은 equation에 있는 모든 연산자들이 다 Node에 저장되어 트리에 배치될 때 까지 진행된다.

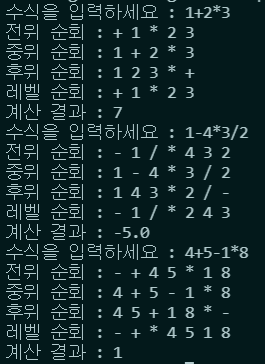
Get\_sub\_tree로 구성된 서브 트리 들이 전체 트리를 구성하는 방법은 다음과 같다. 우선 root가 None이라면 새로 만들어진 서브 트리를 root에 할당한다. 만약 root가 None이 아니라면, 서브 트리의 root에 존재하는 연산자와 새로 get\_sub\_tree 메서드로 들어가게 되는 연산자의 weight를 비교한다. 이때 weight는 클래스의 변수로 선언했던 operator 문자열과 index 메서드를 이용한다. index연산에 따라서 +는 0, -는1, \*는2, /는3의 값을 얻게 되는데 해당 값을 // 2 연산을 통해서 각각 0, 0, 1, 1으로 바꾸어 준다. 따라서 \*와 /의 weight가 +와 -의 weight보다 높아지게 된다. 이렇게 이미 만들어져 있는 서브 트리의 root node에 있는 연산자와 새로 추가되어야 하는 연산자의 우선순위를 각각 비교해서 알맞은 조치를 취하게 된다.

이렇게 tree의 구성이 끝나면 print\_result 메서드를 통해서 각 순회 방법에 맞도록 tree를 순회하며 Node의 item을 출력하고 get\_value 메서드를 부르게 된다.

Get\_value 메서드는 postorder\_calculator 메서드를 통해서 계산의 결과 값을 얻게 되는데, 해당 메서드는 트리를 후위 순회 하면서 결과 값을 구하게 된다. 마지막으로 계산 결과를 출력하고 다음 계산식을 입력 받기 위해서 value, equation, tree.root를 초기화 해준다.

|  |  |
| --- | --- |
| **변수 or 메서드** | **내용** |
| Tree | 계산기 클래스에서 사용할 tree 변수 |
| Equation | 사용자에게 입력 받은 식을 저장할 변수 |
| Operator | Operator weight를 계산할 때 참고할 변수 |
| Value | 식의 값을 저장할 인스턴스 변수 |
| Get\_equation() | 사용자에게 식을 입력 받는 메서드 |
| Get\_tree() | 사용자에게 입력을 받은 식으로 트리를 구성하는 메서드 |
| Get\_sub\_tree() | 2개의 피 연산자와 1개의 연산자를 이용해서 서브트리를 만드는 메서드 |
| Print\_result() | 결과를 출력 해주는 메서드 |
| Get\_value() | 결과값을 출력하고 초기화 하는 메서드 |
| Psotorder\_calculate | 후위 계산으로 결과 값을 구하는 메서드 |
| Run | 계산기를 실행하는 메서드 |

* 1. **결과**



# **사전 탐색 트리 만들기**

* 1. **클래스와 메서드 정의**

**import** random

class Node :

**def** \_\_init\_\_(*self*, **item**, **left=**None, **right=**None) :

*self*.item **=** item

*self*.left **=** left

*self*.right **=** right

**def** \_\_str\_\_(*self*):

**return** *self*.item

*#이진 탐색 트리*

class TreeA :

**def** \_\_init\_\_(*self*):

*self*.root **=** None

*#중위 순위를 하면서 트리의 모든 단어를 정렬된 상태로 출력해주는 메서드*

*#디버깅을 위해서 구현*

    @classmethod

**def** print(*cls*, **n**):

**if** n **!=** None :

*cls*.print(n.left)

            print(n.item)

*cls*.print(n.right)

*#입력받은 item을 트리에서 찾아서 돌려주는 탐색 메서드*

**def** search(*self*, **item**):

**if** *self*.root **is** None:

**return** None

**else**:

**return** *self*.\_\_search\_node(*self*.root, item, **level** **=** 0)

*#search메서드에서 단어를 탐색하는데 사용하는 메서드*

**def** \_\_search\_node(*self*, **cur**, **item**, **level**):

**if** cur **is** None:

**return** None, None

**if** cur.item[0] **==** item:

**return** cur, level

**else**:

**if** cur.item[0] **>=** item:

                level **+=** 1

**return** *self*.\_\_search\_node(cur.left, item, level)

**else**:

                level **+=** 1

**return** *self*.\_\_search\_node(cur.right, item, level)

*#item을 트리에 이진 탐색 트리에 특성에 맞도록 추가하는 메서드*

**def** insert(*self*, **item**):

**if** *self*.root **is** None:

*self*.root **=** Node(item)

**else**:

*self*.\_\_insert\_node(*self*.root, item)

*#insert메서드에서 단어를 추가하는데 사용하는 메서드*

**def** \_\_insert\_node(*self*, **cur**, **item**):

*#root에 있는 단어의 값이 크면 왼쪽으로*

**if** cur.item[0] **>=** item[0]:

**if** cur.left **is** **not** None:

*self*.\_\_insert\_node(cur.left, item)

**else**:

                cur.left **=** Node(item)

*#root에 있는 단어의 값이 작으면 오른쪽으로*

**else**:

**if** cur.right **is** **not** None:

*self*.\_\_insert\_node(cur.right, item)

**else**:

                cur.right **=** Node(item)

*#트리의 높이를 구하는 메서드*

**def** calc\_height(*self*, **n=**None):

**if** n **is** None: **return** 0

**return** 1 **+** max(*self*.calc\_height(n.left), *self*.calc\_height(n.right))

*# 총 몇개의 노드가 있는지를 구하는 메서드*

**def** get\_total\_count(*self*, **n**):

**if** n **is** None: **return** 0

**return** 1 **+** *self*.get\_total\_count(n.left) **+** *self*.get\_total\_count(n.right)

*#스스로 균형을 잡는 이진 탐색 트리*

class TreeB :

**def** \_\_init\_\_(*self*):

*self*.root **=** None

*#중위 순위를 하면서 트리의 모든 단어를 정렬된 상태로 출력해주는 메서드*

*#디버깅을 위해서 구현*

    @classmethod

**def** print(*cls*, **n**):

**if** n **!=** None :

*cls*.print(n.left)

            print(n.item, **end=**" ")

*cls*.print(n.right)

*#입력받은 item을 트리에서 찾아서 돌려주는 탐색 메서드*

**def** search(*self*, **item**):

**if** *self*.root **is** None:

**return** None

**else**:

**return** *self*.\_\_search\_node(*self*.root, item, **level** **=** 0)

*#search메서드에서 단어를 탐색하는데 사용하는 메서드*

**def** \_\_search\_node(*self*, **cur**, **item**, **level**):

**if** cur **is** None:

**return** None, None

**if** cur.item[0] **==** item:

**return** cur, level

**else**:

**if** cur.item[0] **>=** item:

                level **+=** 1

**return** *self*.\_\_search\_node(cur.left, item, level)

**else**:

                level **+=** 1

**return** *self*.\_\_search\_node(cur.right, item, level)

*#item을 트리에 이진 탐색 트리에 특성에 맞도록 추가하는 메서드*

**def** insert(*self*, **item**):

**if** *self*.root **is** None:

*self*.root **=** Node(item)

**else**:

*self*.root **=** *self*.\_\_insert\_node(*self*.root, item)

*#insert메서드에서 단어를 추가하는데 사용하는 메서드*

**def** \_\_insert\_node(*self*, **cur**, **item**):

*#root에 있는 단어의 값이 크면 왼쪽으로*

**if** cur.item[0] **>=** item[0]:

**if** cur.left **is** **not** None:

                cur.left **=** *self*.\_\_insert\_node(cur.left, item)

**else**:

                cur.left **=** Node(item)

*#root에 있는 단어의 값이 작으면 오른쪽으로*

**else**:

**if** cur.right **is** **not** None:

                cur.right **=** *self*.\_\_insert\_node(cur.right, item)

**else**:

                cur.right **=** Node(item)

*#만약의 트리가 편향 트리가 되었다면 다시 균형을 맞추어주기 위해 실행한다.*

**return** *self*.rebalance(cur)

*#트리의 균형을 맞추기 위한 메서드*

**def** rebalance(*self*, **cur**):

        left\_height **=** *self*.calc\_height(cur.left)

        right\_height **=** *self*.calc\_height(cur.right)

**if** left\_height **-** right\_height **<=** **-**2:

**if** *self*.calc\_height(cur.right.left) **-** *self*.calc\_height(cur.right.right) **<** 0:

                cur **=** *self*.rrrotate(cur)

**else**:

                cur **=** *self*.rlrotate(cur)

**elif** left\_height **-** right\_height **>=** 2:

**if** *self*.calc\_height(cur.left.left) **-** *self*.calc\_height(cur.left.right) **<** 0:

                cur **=** *self*.lrrotate(cur)

**else**:

                cur **=** *self*.llrotate(cur)

**return** cur

**def** llrotate(*self*, **cur**):

        B **=** cur.left

        cur.left **=** B.right

        B.right **=** cur

**return** B

**def** rrrotate(*self*, **cur**):

        B **=** cur.right

        cur.right **=** B.left

        B.left **=** cur

**return** B

**def** rlrotate(*self*, **cur**):

        B **=** cur.right

        cur.right **=** *self*.llrotate(B)

**return** *self*.rrrotate(cur)

**def** lrrotate(*self*,**cur**):

        B **=** cur.left

        cur.left **=** *self*.rrrotate(B)

**return** *self*.llrotate(cur)

*#트리의 높이를 구하는 메서드*

**def** calc\_height(*self*, **n=**None):

**if** n **is** None: **return** 0

**return** 1 **+** max(*self*.calc\_height(n.left), *self*.calc\_height(n.right))

*# 총 몇개의 노드가 있는지를 구하는 메서드*

**def** get\_total\_count(*self*, **n**):

**if** n **is** None: **return** 0

**return** 1 **+** *self*.get\_total\_count(n.left) **+** *self*.get\_total\_count(n.right)

과제 2번을 수행하기 위해서 선언한 클래스는 다음과 같다.

|  |  |
| --- | --- |
| **클래스** | **내용** |
| Node | 사전의 내용을 담기 위한 노드 |
| TreeA | 사전을 구현한 이진 탐색 트리 |
| TreeB | TreeA에서 성능을 개선한 이진 탐색 트리 |

Node 클래스는 영어 단어와 뜻을 저장하기 위해서 구현했다. 각 노드는 정보를 저장할 item 변수와 왼쪽 child를 가리키는 left, 오른쪽 child를 가리키는 right 변수를 가지고 있다. 만약 Node 클래스를 이용해서 인스턴스를 생성할 때 파라미터로 아무것도 전달되지 않는다면 left와 right값은 None 값을 가리키게 했다.

|  |  |
| --- | --- |
| **변수 or 메서드** | **내용** |
| Item | 식의 피 연산자 혹은 연산자를 저장하는 변수 |
| Lefrt | 해당 Node의 left child를 가리키는 변수 |
| Right | 해당 Node의 right child를 가리키는 변수 |

TreeA와 TreeB 클래스에 대해서는 아래에 보다 자세하게 설명하겠다.

* 1. **실행**

*#결과를 출력하는 함수*

**def** print\_result(**tree**, **search\_list**):

    print("사전 파일을 모두 읽었습니다. %d개의 단어가 있습니다." **%** tree.get\_total\_count(tree.root))

    print("트리의 전체 높이는 %d 입니다." **%** tree.calc\_height(tree.root))

    print("랜덤하게 선택된 단어 %d개 : " **%** len(search\_list), **end=**"")

**for** word **in** search\_list:

        print(word, **end=**" ")

    print()

**for** word **in** search\_list:

        result, level **=** tree.search(word)

        print("%s %s (레벨 %d)" **%** (result.item[0], result.item[1], level))

*#몇 개의 단어를 랜덤하게 뽑을지 지정하는 변수*

DATA\_SIZE **=** 10

*#파일을 열어서 단어를 읽어온다.*

**with** open('./randdict\_utf8.TXT', 'r', **encoding=**"utf-8") **as** f:

*# i 는 디버깅을 위한 변수*

*#i = 0*

    treeA **=** TreeA()

    treeB **=** TreeB()

*#word\_list는 단어를 랜덤하게 선택하기 위해 저장해두는 리스트*

    word\_list **=** list()

**for** line **in** f:

        line **=** line.strip().split(":")

        line **=** list(map(lambda **x**: x.strip(), line))

*#형식에 맞지 않으면 tree와 전체 word\_list에 추가하지 않는다.*

**if** line[1] **==** "":

**continue**

**else**:

            treeA.insert(line)

            treeB.insert(line)

            word\_list.append(line[0])

*#아래 2줄은 디버깅을 위한 코드*

*#print(i, line)*

*#i+=1*

*#랜덤으로 DATA\_SIZE만큼의 단어를 선택함*

search\_list **=** random.sample(word\_list, DATA\_SIZE)

*#treeA와 treeB의 결과 출력*

print\_result(treeA, search\_list)

print()

print\_result(treeB, search\_list)

우선 연산의 결과를 지정된 format으로 출력해주는 print\_result 함수를 정의했다. 해당 함수는 tree와 랜덤으로 선택된 단어 리스트를 받는다. 이후에 tree에 저장된 단어 수, 트리의 높이, 랜덤으로 선택된 단어의 수와 내용을 출력해준다. 이후에 tree 클래스의 인스턴스 메서드 search를 이용하여 해당 단어가 저장된 노드와 해당 노드가 몇 level에 존재하는지를 돌려받은 뒤 단어와 단어의 뜻, 해당 단어가 저장된 노드의 위치를 출력하도록 했다.

과제 설명에는 랜덤으로 10개의 단어를 선택하라고 했지만 이후에 보다 유연하게 선택하는 단어의 수를 변경할 수 있도록 DATA\_SIZE에 10을 할당한 뒤에 해당 변수를 이용해서 단어를 랜덤하게 선택하도록 했다.

제공된 TXT파일의 정보를 사용하기 위해서 randdict\_utf8.TXT 파일을 읽기 모드로 open한 뒤에, 각 line에서 단어와 의미를 분리했다. 이때 뜻이 형식에 맞지 않는 단어에 대해서는 tree에 추가하지 않았다. treeA는 TreeA 클래스의 인스턴스를 저장하고 treeB는 TreeB 클래스의 인스턴스를 저장했다. 마지막으로 word\_list는 해당 과제의 요구 조건에 따라 랜덤하게 단어를 선택하기 위해 따로 선언한 변수이다. Word\_list 변수에서 random.sample 메서드를 이용하여 DATA\_SIZE만큼의 단어를 임의로 선택한 뒤에 search\_list에 할당한 뒤, 각 treeA와 treeB를 search\_list와 함께 위에서 정의한 print\_result 함수에 전달하면서 과제에서 요구하는 결과를 출력하도록 했다.

* 1. **TreeA 클래스**

*#이진 탐색 트리*

class TreeA :

**def** \_\_init\_\_(*self*):

*self*.root **=** None

*#중위 순위를 하면서 트리의 모든 단어를 정렬된 상태로 출력해주는 메서드*

*#디버깅을 위해서 구현*

    @classmethod

**def** print(*cls*, **n**):

**if** n **!=** None :

*cls*.print(n.left)

            print(n.item)

*cls*.print(n.right)

*#입력받은 item을 트리에서 찾아서 돌려주는 탐색 메서드*

**def** search(*self*, **item**):

**if** *self*.root **is** None:

**return** None

**else**:

**return** *self*.\_\_search\_node(*self*.root, item, **level** **=** 0)

*#search메서드에서 단어를 탐색하는데 사용하는 메서드*

**def** \_\_search\_node(*self*, **cur**, **item**, **level**):

**if** cur **is** None:

**return** None, None

**if** cur.item[0] **==** item:

**return** cur, level

**else**:

**if** cur.item[0] **>=** item:

                level **+=** 1

**return** *self*.\_\_search\_node(cur.left, item, level)

**else**:

                level **+=** 1

**return** *self*.\_\_search\_node(cur.right, item, level)

*#item을 트리에 이진 탐색 트리에 특성에 맞도록 추가하는 메서드*

**def** insert(*self*, **item**):

**if** *self*.root **is** None:

*self*.root **=** Node(item)

**else**:

*self*.\_\_insert\_node(*self*.root, item)

*#insert메서드에서 단어를 추가하는데 사용하는 메서드*

**def** \_\_insert\_node(*self*, **cur**, **item**):

*#root에 있는 단어의 값이 크면 왼쪽으로*

**if** cur.item[0] **>=** item[0]:

**if** cur.left **is** **not** None:

*self*.\_\_insert\_node(cur.left, item)

**else**:

                cur.left **=** Node(item)

*#root에 있는 단어의 값이 작으면 오른쪽으로*

**else**:

**if** cur.right **is** **not** None:

*self*.\_\_insert\_node(cur.right, item)

**else**:

                cur.right **=** Node(item)

*#트리의 높이를 구하는 메서드*

**def** calc\_height(*self*, **n=**None):

**if** n **is** None: **return** 0

**return** 1 **+** max(*self*.calc\_height(n.left), *self*.calc\_height(n.right))

*# 총 몇개의 노드가 있는지를 구하는 메서드*

**def** get\_total\_count(*self*, **n**):

**if** n **is** None: **return** 0

**return** 1 **+** *self*.get\_total\_count(n.left) **+** *self*.get\_total\_count(n.right)

해당 클래스는 트리의 최 상단에 위치한 root node를 root라는 변수로 기억하고 있다. 이후에 TreeA 인스턴스를 이용하여 연산을 진행할 때 해당 root를 통해 root node에 접근해 필요한 기능을 수행할 수 있다. 또한 과제를 구현하면서 디버깅을 위해 Tree를 중위 순회 하면서 item을 출력하는 print 메서드를 구현했다. 트리를 중위 순회 하면서 item을 출력하면 자동으로 정렬된 리스트를 얻을 수 있었다.

TreeA는 insert와 \_\_insert\_node 메서드를 통해서 단어 정보를 트리에 저장하게 된다. 이진 탐색 트리의 특성에 맞추어 단어를 저장하기 위해서 root node와 입력되는 단어를 비교해서 왼쪽 혹은 오른쪽으로 순회하면서 알맞은 위치를 찾은 다음 새로운 Node가 추가되도록 했다.

Search와 \_\_search\_node를 통해 파라미터로 전달 되는 단어를 트리에서 탐색하여 해당 단어가 저장되어 있는 node를 반환 하도록 했다. 이때 해당 노드의 level 역시 함께 반환한다.

트리의 높이를 구하는 메서드인 calc\_height 메서드와 총 몇 개의 노드가 트리에 연결되어 있는지를 구하는 get\_total\_count 메서드를 구현했다.

|  |  |
| --- | --- |
| **변수 or 메서드** | **내용** |
| Root | 사전의 root node를 pointing 하는 변수 |
| Print() | Tree로 구성된 사전을 중위 순회하며 출력하는 메서드 |
| Search() | 입력 받은 item을 트리에서 찾아서 돌려주는 탐색 메서드 |
| \_\_search\_node() | Search 메서드에서 단어를 탐색하는데 사용하는 메서드 |
| Insert() | Item을 트리에 이진 탐색 트리의 특성에 맞도록 추가하는 메서드 |
| \_\_insert\_node() | Insert 메서드에서 단어를 추가하는데 사용하는 메서드 |
| Calc\_height() | 트리의 높이를 구하는 메서드 |
| Get\_total\_count() | 총 몇 개의 노드가 있는지 구하는 메서드 |

* 1. **TreeB 클래스**

*#스스로 균형을 잡는 이진 탐색 트리*

class TreeB :

**def** \_\_init\_\_(*self*):

*self*.root **=** None

*#중위 순위를 하면서 트리의 모든 단어를 정렬된 상태로 출력해주는 메서드*

*#디버깅을 위해서 구현*

    @classmethod

**def** print(*cls*, **n**):

**if** n **!=** None :

*cls*.print(n.left)

            print(n.item, **end=**" ")

*cls*.print(n.right)

*#입력받은 item을 트리에서 찾아서 돌려주는 탐색 메서드*

**def** search(*self*, **item**):

**if** *self*.root **is** None:

**return** None

**else**:

**return** *self*.\_\_search\_node(*self*.root, item, **level** **=** 0)

*#search메서드에서 단어를 탐색하는데 사용하는 메서드*

**def** \_\_search\_node(*self*, **cur**, **item**, **level**):

**if** cur **is** None:

**return** None, None

**if** cur.item[0] **==** item:

**return** cur, level

**else**:

**if** cur.item[0] **>=** item:

                level **+=** 1

**return** *self*.\_\_search\_node(cur.left, item, level)

**else**:

                level **+=** 1

**return** *self*.\_\_search\_node(cur.right, item, level)

*#item을 트리에 이진 탐색 트리에 특성에 맞도록 추가하는 메서드*

**def** insert(*self*, **item**):

**if** *self*.root **is** None:

*self*.root **=** Node(item)

**else**:

*self*.root **=** *self*.\_\_insert\_node(*self*.root, item)

*#insert메서드에서 단어를 추가하는데 사용하는 메서드*

**def** \_\_insert\_node(*self*, **cur**, **item**):

*#root에 있는 단어의 값이 크면 왼쪽으로*

**if** cur.item[0] **>=** item[0]:

**if** cur.left **is** **not** None:

                cur.left **=** *self*.\_\_insert\_node(cur.left, item)

**else**:

                cur.left **=** Node(item)

*#root에 있는 단어의 값이 작으면 오른쪽으로*

**else**:

**if** cur.right **is** **not** None:

                cur.right **=** *self*.\_\_insert\_node(cur.right, item)

**else**:

                cur.right **=** Node(item)

*#만약의 트리가 편향 트리가 되었다면 다시 균형을 맞추어주기 위해 실행한다.*

**return** *self*.rebalance(cur)

*#트리의 균형을 맞추기 위한 메서드*

**def** rebalance(*self*, **cur**):

        left\_height **=** *self*.calc\_height(cur.left)

        right\_height **=** *self*.calc\_height(cur.right)

**if** left\_height **-** right\_height **<=** **-**2:

**if** *self*.calc\_height(cur.right.left) **-** *self*.calc\_height(cur.right.right) **<** 0:

                cur **=** *self*.rrrotate(cur)

**else**:

                cur **=** *self*.rlrotate(cur)

**elif** left\_height **-** right\_height **>=** 2:

**if** *self*.calc\_height(cur.left.left) **-** *self*.calc\_height(cur.left.right) **<** 0:

                cur **=** *self*.lrrotate(cur)

**else**:

                cur **=** *self*.llrotate(cur)

**return** cur

**def** llrotate(*self*, **cur**):

        B **=** cur.left

        cur.left **=** B.right

        B.right **=** cur

**return** B

**def** rrrotate(*self*, **cur**):

        B **=** cur.right

        cur.right **=** B.left

        B.left **=** cur

**return** B

**def** rlrotate(*self*, **cur**):

        B **=** cur.right

        cur.right **=** *self*.llrotate(B)

**return** *self*.rrrotate(cur)

**def** lrrotate(*self*,**cur**):

        B **=** cur.left

        cur.left **=** *self*.rrrotate(B)

**return** *self*.llrotate(cur)

*#트리의 높이를 구하는 메서드*

**def** calc\_height(*self*, **n=**None):

**if** n **is** None: **return** 0

**return** 1 **+** max(*self*.calc\_height(n.left), *self*.calc\_height(n.right))

*# 총 몇개의 노드가 있는지를 구하는 메서드*

**def** get\_total\_count(*self*, **n**):

**if** n **is** None: **return** 0

**return** 1 **+** *self*.get\_total\_count(n.left) **+** *self*.get\_total\_count(n.right)

TreeB 클래스의 경우, TreeA의 구현에서 추가적으로 구현되었거나 수정된 메서드에 대해서만 설명하겠다. 우선 새로운 단어를 Tree에 insert할 때 사용되던 \_\_insert\_node 메서드의 마지막 부분에 rebalance 메서드를 추가로 실행하게 했다. 이 rebalance 메서드는 트리에 새로운 노드가 추가되면서 만약 편향 트리가 되었다면 균형을 맞추어 주는 역할을 한다.

Rebalance 메서드에서 Tree의 현 상태가 편향되었는지 균형 잡혀 있는지 판단하는 방법은 다음과 같다. 우선 전달받은 트리의 루트를 기준으로 왼쪽과 오른쪽의 height를 구한다. 이때 calc\_height 메서드를 사용한다. 두 height의 차(left height – right height)가 -1, 0, 1이라면 균형이 잡혀 있다고 판단하고 아무 조치를 취하지 않는다. 이때 차가 -1이라면 왼쪽보다 오른쪽 child의 level이 1만큼 더 크다는 의미이다. 0이라면 같은 level을 가지는 상태이고, 1이라면 왼쪽 child가 오른쪽보다 level이 1만큼 더 크다는 의미이다.

만약 두 height의 차가 2 이상이거나 -2 이하라면 해당 트리는 현재 편향 상태에 접어들었다고 판단하고 조치를 취하게 된다. 이때 두 height의 차가 -2 이하라면 오른쪽이 왼쪽보다 편향되어 있다는 것이고 2 이상이라면 왼쪽이 오른쪽보다 편향되어 있다는 것이다. 이렇게 편향된 방향을 판단하고 왼쪽과 오른쪽 중 어느 Node가 root node와 위치 교환이 이루어지는지를 판단한다.

이때 고려해야 할 사항이 한 가지 더 있다. 편향된 트리의 root node와 root node의 왼쪽 혹은 오른쪽 node를 교환하면서 균형을 맞추게 되는데 이때 원래 root node의 왼쪽 혹은 오른쪽 node에 연결되어 있던 child node에 대한 조치도 취해야 한다. 따라서 root node와 위치를 교환하게 되는 node의 왼쪽 child와 오른쪽 child의 height를 각각 구한 뒤 올바른 방향에 맞추어 rotate를 시켜야 한다. 만약 왼쪽보다 오른쪽 child의 height가 더 크다면 오른쪽으로 rotate를 시키고, 왼쪽 child의 height가 더 크다면 왼쪽 방향으로 rotate를 시키게 된다.

궁극적으로 편향 트리를 균형 잡힌 트리로 변환 하는 방법은 총 4가지가 존재하게 된다.

|  |  |
| --- | --- |
| Left Left | 루트 노드를 기준으로 왼쪽 child의 height가 더 크고, 루트 노드의 왼쪽 child의 왼쪽 child가 루트 노드의 왼쪽 child의 오른쪽 child 보다 더 큰 height를 가진 경우 |
| Left Right | 루트 노드를 기준으로 왼쪽 child의 height가 더 크고, 루트 노드의 왼쪽 child의 오른쪽 child가 루트 노드의 왼쪽 child의 왼쪽 child 보다 더 큰 height를 가진 경우 |
| Right Left | 루트 노드를 기준으로 오른쪽 child의 height가 더 크고, 루트 노드의 오른쪽 child의 왼쪽 child가 루트 노드의 오른쪽 child의 오른쪽 child 보다 더 큰 height를 가진 경우 |
| Right Right | 루트 노드를 기준으로 오른쪽 child의 height가 더 크고, 루트 노드의 오른쪽 child의 오른쪽 child가 루트 노드의 오른쪽 child의 왼쪽 child 보다 더 큰 height를 가진 경우 |

그렇게 상황에 맞는 메소드를 사용하여 트리에 새로운 노드가 추가될 때 최대한 균형을 맞추도록 하여, 트리가 편향되어 height가 지나치게 높아지는 것을 사전에 방지한다.

|  |  |
| --- | --- |
| **변수 or 메서드** | **내용** |
| Root | 사전의 root node를 pointing 하는 변수 |
| Print() | Tree로 구성된 사전을 중위 순회하며 출력하는 메서드 |
| Search() | 입력 받은 item을 트리에서 찾아서 돌려주는 탐색 메서드 |
| \_\_search\_node() | Search 메서드에서 단어를 탐색하는데 사용하는 메서드 |
| Insert() | Item을 트리에 이진 탐색 트리의 특성에 맞도록 추가하는 메서드 |
| Calc\_height() | 트리의 높이를 구하는 메서드 |
| Get\_total\_count() | 총 몇 개의 노드가 있는지 구하는 메서드 |
| **\_\_insert\_node()** | Insert 메서드에서 단어를 추가하는데 사용하는 메서드 |
| **Rebalance()** | 트리의 편향 여부를 판단하여 균형을 맞추는 메서드 |
| **Llrotate()** | Left Left 노드 교환 메서드 |
| **Rrrotate()** | Right Right 노드 교환 메서드 |
| **Rlrotate()** | Right Left 노드 교환 메서드 |
| **Lrrotate()** | Left Right 노드 교환 메서드 |

\*bold 된 메서드가 새로 구현되었거나 수정된 메서드이다.

* 1. **결과**

