Homework1

CSI3108-1 Programming

27167033 남다예

1. 알고리즘

자료구조

sum_stack	`public static Stack<Integer> sum_stack = new Stack<Integer>();`
node_stack	`public static Stack<nodeClass> node_stack = new Stack<nodeClass>();`

주어진 합을 만족시키기 위해 남은 수를 저장하는 sum_stack과 현재 노드와 부모 노드들의 정보를 저장하고 있는 node_stack을 사용하였다. Stack을 사용한 이유는 모든 인풋을 저장하지 않아도 되기 때문에 메모리 효율성을 높일 수 있기 때문이다.

sum_stack	node_stack

`int remain_sum;`	`public static class nodeClass {`
	`  int val;`
	`  int side;`
	`}`

sum_stack에는 남은 수들을 Integer로, node_stack에는 그 노드의 값을 저장해주는 val라는 Integer 변수와 현재 왼쪽 자식을 보고있는지 오른쪽 자식을 보고 있는지를 기록해주는 side라는 Integer변수를 nodeClass로 만들어서 저장하였다.

PARSING

모든 parsing은 ParseLine이라는 class안에 있는 getNextLine이라는 함수 안에서 진행된다. getNextLine은 메인 함수 안에서 파일에서 읽은 한 줄을 인풋으로 받아 한번 부를 때마다 다음에 위치 해 있는 lexeme을 리턴하며, 줄의 마지막에서는 -5를 리턴해 한 줄의 파싱이 끝났음을 알려준다. 리턴하는 값을 모두 Integer로 통일하기 위해(정수 값을 Character로 리턴하게 되면 메인 함수에서 정수로의 변환 작업을 해 주어야 하기 때문에) '('는 -3으로, ')'는 -4로 리턴하였다. (모든 노드의 값들은 양의정수이므로 음의 정수를 사용할 수 있다.) 하나를 리턴할 때마다 index를 1씩 증가하여 다음에 같은 함수를 부르면 index가 가르키고 있는 곳부터 시작해 lexeme을 리턴하였다.

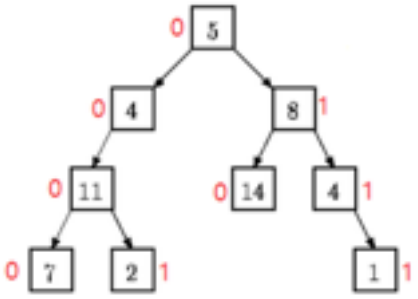
'('	-3을 리턴한다.
')'	-4를 리턴한다.

숫자	숫자를 읽으면 다음 Character를 미리 보고, 다음 Character 또한 숫자라면 여러자리의 한 정수이므로 임시 변수에 값을 저장한 후 다음 숫자를 받을 때 그 임시 변수에 10을 곱하고 다음 숫자를 더하는 방법으로 하나의 정수를 만들어 준 후 숫자가 끝이 났을 때 그 수를 리턴한다.
줄의 마지막	다음 Character가 없다면 (현재의 index와 전체 character의 갯수가 같다면) -5를 리턴해 줄의 파싱이 끝났음을 알린다.

MAIN ALGORITHM

주어진 인풋을 왼쪽에서부터 차례로 읽으면서 스택에 정보를 저장한다. sum_stack 에는 현재 노드에서 주어진 summation 을 만족시키려면 얼마를 더 더해야하는지를 push 하고, 자식 노드의 탐색을 마치고 위로 올라갈 때 pop한다. node_stack에는 현재 노드의 값과 어떤 자식 노드를 탐색하고 있는지를 기록하고, 마찬가지로 모든 자식 노드의 탐색을 마치면 pop한다. 자식 노드가 없을 때 sum_stack의 가장 위에 있는 값이 0이라면 지금까지 더한 수가 summation 과 정확히 일치하므로 원하는 output이 된다. 이 때 node_stack 에 있는 side 정보만을 아래서 부터 위로 출력하면 현재 노드까지의 path를 출력할 수 있다.

	sum_stack	node_stack
'('	remain_sum(summation을 위해 더 필요한 수)를 push	
')'	1. 제일 위의 값 == 0 1) node_stack을 체크하고, node_stack의 side가 1이라면 자식노드가 없는 것이므로 현재까지의 path를 출력 2) node_stack의 side가 0이라면 기다림 2. 제일 위의 값 ≠ 0 1) node_stack을 체크하고, node_stack의 side가 1이라면 현재 노드는 summation을 만족시킬 수 없으므로 pop 2) node_stack의 side가 0이라면 기다림	1. 제일 위의 side가 0이면 아직 오른쪽 노드가 남은 것이므로 side를 1로 업데이트 2. 제일 위의 side가 1이라면 노드의 자식들이 모두 탐색된 것 1) sum_stack의 가장 위의 값이 0이었다면 summation을 만족하므로 현재까지의 path 출력 2) 0이 아니었다면 만족하지 않는 노드이므로 node정보 pop 하고 다시 부모 노드로 이동. 이 때 remain_sum에 다시 pop한 노드의 값을 더해줘야함.
숫자	remain_sum에서 현재 노드 만큼을 -	노드 정보를 push



(5 (4 (11 (7 () ()) (2 () ())) ())
(8 (13 () ()) (4 () (1 () ()))))

lex	(
RS	22

S	N
22	

lex	5
RS	17

S	N
22	5,0

lex	(
RS	17

S	N
17	
22	5,0

lex	4
RS	13

S	N
17	4,0
22	5,0

lex	(
RS	13

S	N
13	
17	4,0
22	5,0

lex	11
RS	2

S	N
13	11,0
17	4,0
22	5,0

lex	(
RS	2

S	N
2	
13	11,0
17	4,0
22	5,0

lex	7
RS	-5

S	N
2	7,0
13	11,0
17	4,0
22	5,0

lex	(
RS	-5

S	N
-5	
2	7,0
13	11,0
17	4,0
22	5,0

lex	)
RS	-5

S	N
2	7,1
13	11,0
17	4,0
22	5,0

lex	(
RS	-5

S	N
-5	
2	7,1
13	11,0
17	4,0
22	5,0

lex	)
RS	2

S	N
2	
13	11,0
17	4,0
22	5,0

lex	(
RS	2

S	N
2	
13	11,0
17	4,0
22	5,0

lex	2
RS	0

S	N
2	2,0
13	11,0
17	4,0
22	5,0

lex	(
RS	0

S	N
0	
2	2,0
13	11,0
17	4,0
22	5,0

lex	)
RS	0

S	N
2	2,1
13	11,0
17	4,0
22	5,0

lex	(
RS	0

S	N
0	
2	2,1
13	11,0
17	4,0
22	5,0

lex	)
RS	0

S	N
0	
2	**2,1**
13	**11,0**
17	**4,0**
22	**5,0**

2. 예시 결과 및 실행시간

1. 4 (1(2()())(3()()))

➜ 01

2. 18 (5(4(11(7()())(2()()))()) (8(13()())(4()(1()()))))

➜ 0111

3. 57 (1(2(4(8(16()())(17()()))(9(18()())(19()()))))(5(10(20()())(21()()))(11(22()())(23()()))))(3(6(12(24()())(25()()))(13(26()())(27()()))))(7(14(28()())(29()()))(15(30()())(31()()))))

➜ 01111

Test Case	1	2	3	4	5	6	7	8	9	10	평균
1	2	1	2	1	2	1	1	2	1	2	1.5
2	0	0	0	0	1	0	0	1	0	0	0.2
3	1	0	1	1	0	1	0	0	0	2	0.6

실행시간을 분석 해 본 결과, 테스트케이스 1의 경우가 평균 1.5 ms, 2의 경우 0.2ms, 3의 경우가 0.6ms이 걸리는 것을 확인할 수 있었다. 가장 짧은 테스트 케이스의 실행시간이 가장 높은 것이 논리적지 않게 느껴져 더 많은 depth를 가진 트리들로 테스트 해 본 결과 일정 depth를 지나면 트리의 크기와 실행시간이 비례하는 것을 확인할 수 있었고, 이를 통해 위의 실행시간들은 테스트케이스의 크기가 작아 비교적 비슷하게 나타난 것이라고 결론내릴 수 있다.