숙제 2 : n-ary tree

<Due : 2020-05-16 (토) 23:59>

강의시간에 배운 ordered tree 의 가장 간단한 형태였던 binary tree를 확장하여 일반적인 ordered n-ary tree를 구현하고 몇가지 기능을 구현해 보도록 하자. n-ary tree는 최대 degree가 n인 rooted tree를 말하며, non-leaf node의 child node 들은 **0번째 왼쪽 (가장 왼쪽) 에서 n-1 번째 왼쪽 (= 가장 오른쪽)** 에 있는 child node 순으로 정렬(ordered) 되어 있다. 이러한 tree를 구현하기 위해 숙제에서는 node + link 구조를 사용하며, n-ary tree의 각각의 node는 다음과 같이 정의되어 있다.

```
typedef struct nary_node {
    int key;
    struct nary_node* children[n]
}nary_node;
```

정의에 의해 children array 는 최대 n개의 child node들을 저장할 수 있으며 array index에 따라 **0번째 왼쪽 – (n-1)번째 왼쪽**으로 child node들의 순서가 결정된다 (예를 들어 children[0] 에 해당하는 child node는 children[3]에 해당하는 child node보다 왼쪽 (전)에 있는 node이다). 이 때 **children array의 중간 요소들이 NULL이 될 수 있으며, 해당 요소는 존재하지 않는 child node로 생각하는 것에 주의하자** (예를 들어 children[0] = node1, children[1] = NULL, children[2] = NULL , children[3] = node2 면 children[3]이 왼쪽에서 1번째 child node가 된다). n은 헤더 파일 (...)에 미리 정의가 되어 있으며 (추후 채점 시 변경 예정), 특별히 n = 2인 경우 (binary tree) 에 대해서는 강의 슬라이드와 마찬가지로 다음과 같이 따로 정의되어 있다.

```
typedef struct binarynode {
    int key;
    struct binarynode* leftchild;
    struct binarynode* rightchild;
}binarynode;
```

또한 다음과 같은 함수가 미리 정의 및 구현되어 있다.

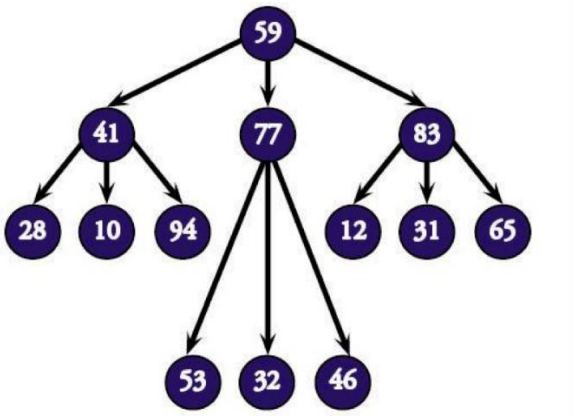
1) void binary_preorder (binarynode *root) : 해당 binary tree 를 preorder 순서대로 출력해주는 함수.

2) void binary_inorder (binarynode *root) : 해당 binary tree 를 inorder 순서대로 출력해주는 함수.

2) void print_error () : 에러 메시지를 출력하는 함수.

<구현해야 하는 함수들>

주의 : Tree 상의 각 node의 key 값은 모두 단일 int 형 값이며, 같은 key 값을 가지는 서로 다른 두 node는 존재하지 않는다 가정한다 (즉 모든 node의 key 값은 서로 다르다). 또한 **tree는 언제나 해당 tree의 root node를 인자로 받고 반환**한다.



1. nary_node * createtree (nary_node * root, int key) : root node 하나로 이루어진 n-ary tree를 생성 후 반환하는 함수.

2. nary_node * key_node (nary_node *root, int key) : n-ary tree 에서 key 값을 가진 node를 반환한다. 해당 node가 없으면 NULL을 반환한다.

3. nary_node * add_i_child (nary_node * root, int parent, int i, int key) : n-ary tree 에서 **parent 값을 가지는 node의 (왼쪽에서) i번째 child로 key값을 가지는 새로운 node를 추가**해 준 다음 해당 tree 를 반환한다. 구현 세부 사항은 다음과 같다.

1) children array 중간에 NULL이 들어가는 것은 상관 없지만, children array는 1페이지에서 언급한 특징을 만족해야 한다.

2) 기존에 parent node의 i번째 child node가 이미 존재할 경우 i번째 node 부터 맨 오른쪽 node 까지 모두 한 position씩 오른쪽으로 이동하게 된다 (위 예제에서 83의 0번째 child에 새로운 node를 추가할 경우, 기존의 12는 83의 1번째 child가 됨)**.**

3) i가 현재 parent의 degree 이상 n-1 이하일 때는 새로운 추가하는 node는 parent의 맨 오른쪽 child node가 된다.

4) (i) 해당 parent node가 없거나, (ii) 이미 n개의 child를 가지고 있거나, (iii) i가 n 이상일 경우 print_error 함수를 호출 뒤 원래 인자로 받았던 tree를 반환한다.

4. nary_node * delete_i_child (nary_node * root, int parent, int i) : n-ary tree 에서 **parent 값을 가지는 node의 (왼쪽에서) i번째 child node를 NULL로 바꾼 후 tree 를 반환** 한다 (즉 반환 받은 tree 의 root node 로부터는 NULL 로 변경한 node를 root로 가진 (기존) subtree 상의 모든 node 들을 접근할 수 없게 됨). Parent나 해당하는 child node가 존재하지 않으면 print_error 함수를 호출 뒤 원래 인자로 받았던 tree를 반환한다. **i**번째 child node는 구현 방법에 따라 children[i]와 같지 않음에 유념하라.

예를 들어 root가 위 그림 tree의 root node라 하면 delete_i_child (root, 77, 0) 을 호출하면 node 53을 제거한 후 해당 tree를 반환한다.

5. int height_node (nary_node *root, int node_value) : n-ary tree 에서 node_value 값을 가지는 node의 height 를 반환하는 함수. 해당 node가 없을시 print_error 함수를 호출 뒤 -1을 반환한다.

6. int height_tree (nary_node *root) : n-ary tree root 의 height 를 반환하는 함수.

7. int depth_node (nary_node *root, int node_value) : n-ary tree root 에서 node_value 값을 가지는 node의 depth 를 반환하는 함수. 해당 node가 없을시 print_error 함수를 호출 뒤 -1을 반환한다.

8. int depth_tree (nary_node *root) : n-ary tree root 의 depth 를 반환하는 함수.

9. int child_rank (nary_node *root, int node_value) : n-ary tree 에서 node_value 값을 가지는 node가 해당 node의 parent의 **왼쪽부터 몇 번째 child인지** (즉 node보다 왼쪽에 있는 sibling들이 몇 개인지) 답하는 함수. 정의에 의해 root node를 제외한 node들의 child_rank는 0부터 n-1까지 존

재할 수 있다. 예를 들어 위 예제에서 32의 child_rank 는 1이고, 28의 child_rank 는 0이다. Root node의 경우 child_rank 를 -1로 정의하자. 해당 node_value 를 가진 node가 없을 시 print_error 함수를 호출한 뒤 아무 정수값이나 반환한다.

10. int child_select (nary_node *root, int parent, int i) : n-ary tree 에서 parent 값을 가지는 node의 **왼쪽에서 i번째에 위치한 child node**에 저장된 값을 출력한다 (NULL node는 node가 존재하지 않는 node로 생각하는 것에 다시 한번 주의하자) 위 예제에서 root가 위 그림 tree의 root node라 하면 child_select (root , 77, 0) = 53, child_select(root, 59, 2) = 83 이 된다. 해당 parent나 child node가 없을 시 print_error 함수를 호출한 뒤 아무 정수값이나 반환한다.

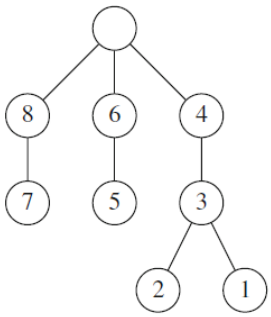
11. void preorder (nary_node * root) : n-ary tree 를 preorder traversal을 하면서 tree의 모든 node 들 방문한 순서대로 해당 node의 key 값들을 출력하는 함수. n-ary tree에서의 preorder traversal 은 binary tree와 마찬가지로 root node 방문 -> **왼쪽부터** 모든 subtree들을 차례대로 방문으로 정의된다. 위 예제에서는 59 41 28 10 94 77 53 32 46 83 12 31 65 순서대로 방문하고 출력한다. 출력시 각 node들의 key값 들은 **공백문자 하나로 구분**해주고 마지막에 **new line character (₩n) 은 출력하지 않는다**.

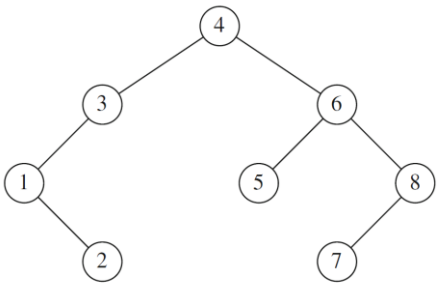
12. (BONUS 문제) binarynode * lcps(nary_node *root) : n-ary tree 를 lcps (last child previous sibling) 방법을 이용하여 binary tree로 변환해 준 후, 변환한 binary tree를 반환한다. n-ary tree T를 binary tree B로 바꾸는 방법은 lcps 방법은 다음과 같다.

(1) T의 root node는 무시 (n-ary tree가 root node 하나로 구성되어 있을 시 NULL을 반환한다).

(2) T의 root node의 맨 오른쪽 child가 B의 root node.

(3) T상의 node n에 대해 **n의 left sibling** (n '바로' 왼쪽 에 위치한 node) 은 B에서 node **n의 right child** 가 되며, T 상의 node n에 대해 **n의 제일 오른쪽 (child node가 하나일 시 제일 왼쪽 node = 제일 오른쪽 node) child** 는 B에서 node **n의 left child**가 된다. 예를 들어 아래 그림 예시에서 왼쪽 3-ary tree를 lcps로 변환한 후 binary tree는 오른쪽 tree와 같다.





변환한 binary tree를 숙제에서 주어진 binary_preorder 및 binary_inorder 를 이용해 제대로 변환이 되었나 확인해 보자.

3. 주의 **(이 중 하나라도 어기면 0점)**

- 주어진 파일은 다음과 같은 형식으로 되어 있음

1. nary_tree.h : nary_tree.c 에서 정의한 함수들을 모아 둔 헤더파일.

2. **nary_tree.c : nary_tree.h 에서 정의한 함수들의 세부 구현 (숙제에서 제출해야 할 파일).**

3. main.c : 테스트용 main file (결과값이 주석에서 언급한 답과 일치하는지 확인할 것.

4. Makefile : 리눅스, 맥 환경에서 작업하는 학생들을 위한 Makefile.

- 숙제 제출은 due 전까지 e-campus의 과제탭의 **과제물 제출 – 파일 첨부를 이용**하여, 오직 **nary_tree.c 파일만을 학번_nary_tree.c 파일로 업로드** 할 것. (예 : 2019000000_nary_tree.c). **그 외 어떠한 파일도 받지 않음**.

- 숙제의 delay는 받지 않음.

- 문제의 nary_tree.h 와 nary_tree.c 두 파일의 skeleton code 에서 주어진 헤더 파일 외의 **어떠한 헤더 파일도 include 하지 말 것**. 또한 **nary_tree에서 미리 구현된 함수 3개 (print_preorder, print_inorder 및 print_error) 도 변경하지 말 것**.

- 제출하기 전에 예제로 주어진 main.c 파일을 컴파일 및 실행해서 실제로 올바르게 출력되는지 확인할 것.