성균관대학교

Assignment 2 Report

자료구조개론_SWE2015_42 (염익준 교수님)

2019313611 김지훈

2022-11-10

목차

1. 문제 설명 및 사전 이론

Rebuilding a binary tree from traversal results

Input : inorder and postorder traversal results

- First input is inorder traversal results
- Second input is postorder traversal results
- Each alphabet is separated by ", "

Output: a corresponding binary tree

- Only print the tree
- Please refer to the example execution screen

*** Child nodes are visited from the left.**

*** The output format must be the same as goorm example**

*** Assume that each node in the tree contains a alphabet letter**

*** Data of each node is unique alphabet (No duplicates)**

*** Uppercase and lowercase letters are distinguished**

*** Max depth of tree is 6**

① Binary Tree (이진 트리)

이진 트리는 child node를 최대 2개까지만 가지는 tree data structure입니다.

여기서 child node를 각각 left child, right child라고 부릅니다.

예시 코드

```
typedef struct node* treePointer;

typedef struct node {
    char data;
    treePointer leftChild, rightChild;
};
```

② Inorder traversal (LVR)

Inorder traversal은 다음과 같이 재귀적으로 진행됩니다.

Left Subtree Inorder traversal -> Root node -> Right Subtree Inorder traversal

③ Postorder traversal (LRV)

Postorder traversal은 다음과 같이 재귀적으로 진행됩니다.

Left Subtree Postorder traversal -> Right Subtree Postorder traversal -> Root node

2. Code 및 code 설명

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_STRING_LEN 200
#define MAX_TREE_SIZE 53
```

Uppercase and lowercase letter을 모두 합쳐도 52개 이므로 string의 max size와 tree의 max size를 위와 같이 정하였습니다.

```
typedef struct Node* treePointer;
typedef struct Node {
    char data;
    int level; // 해당 node 의 level variable
    treePointer leftChild, rightChild;
}Node;
```

Binary tree structure

특이점은 level이라는 int형 variable이 추가된 것인데, 이후 output format에서 level별 필요한 '/' or '\'의 개수가 다르기에 이후 계산을 위하여 structure에 element로 추가하였습니다.

```
char *input; // char pointer array of inputed string for user
char *inorder; // char pointer array of inorder traversal
char *postorder; // char pointer array of postorder traversal
treePointer binary_tree; // char pointer array of corresponding binary tree
char **display_binary_tree; // two-dimensional char pointer array to store
                               the binary_tree in the given format
int tree_size; // number of alphabet in binary_tree
int tree_depth; // depth of binary_tree
```

Global Variables (각 변수들의 설명은 주석으로 추가하였습니다.)

```
/* 새로운 node를 만들고 return하는 함수 */
treePointer newNode(char data, int level) {
    treePointer node = (treePointer)malloc(sizeof(Node));
    node->data = data;
    node->level = level;
    node->leftChild = node->rightChild = NULL;
    return node;
} // end function newNode
```

data와 level이 주어졌을 때, 새로운 node를 만들고 return하는 함수입니다.

```
/* inorder에서 해당 data의 index를 찾고 return하는 함수 */
int find_idx(char data) {
    int i, idx;
    // inorder의 앞에서부터 data가 나올 때 까지 idx를 증가시키며 해당 idx를
      찾는다.
    for(idx = 0 ; inorder[idx] != data ; idx++);

    // 해당 data를 inorder array에서 뺀다.
    for(i = idx ; inorder[i] != '\0' ; i++)
        inorder[i] = inorder[i+1];

    return idx;
} // end function find_idx
```

위 함수에서는 해당 alphabet이 inorder array에서 몇 번째 index에 있는지 for문을 이용하여 탐색합니다. 이후 inorder array에서 해당 alphabet을 pop하고, index를 return하는 함수입니다.

inorder array에서 alphabet을 pop하는 이유는 우선 inorder traversal과 postorder traversal을 알았을 때, binary tree가 unique하게 만들어지는 과정을 알아야 합니다. 과정은 아래와 같습니다.

- postorder의 가장 마지막 alphabet을 root node의 data로 지정합니다.

- inorder에서 위 alphabet의 위치를 찾고 그 위치를 기준으로 left subtree, right subtree로 나눕니다.

- 위 과정을 각 subtree에서 recursive하게 반복합니다.

이 때, 두 번째 과정에서 inorder array에서 data alphabet은 array의 끝이 아닌 안쪽에 있고, postorder array에서 data alphabet은 array의 가장 오른쪽 끝에 있습니다. 따라서 buildTree 함수를 진행하며 inorder array와 postorder array의 index가 1개씩 어긋나게 되기 때문에, inorder array에서 data alphabet을 pop하여 해결하였습니다.

또한 right subtree보다 left subtree를 먼저 build하며 진행한다면 index가 어긋나기에 아래 buildTree function에서 rightChild를 먼저 recusion을 돌립니다.

```
/* inorder와 postorder를 이용하여 binary tree를 만드는 함수 */
treePointer buildTree(int front_idx, int last_idx, int level) {
    if(front_idx > last_idx)
        /* Case of NULL */
        return NULL;

    char data = postorder[last_idx]; // postorder의 마지막 data가 현재 tree의
                                        head node
    treePointer node = newNode(data, level); // 현재 data와 level로 새로운
                                                node를 만든다.
    /* Update tree_depth. */
    if(node->level > tree_depth)
        tree_depth = node->level;

    /* Case of this node has no child */
    if(front_idx == last_idx)
        return node;

    // 현재 node의 data가 inorder에서 몇 번째 index에 있는지 찾는다.
    int inorder_idx = find_idx(node->data);
    level++; // child의 level은 현재 level보다 1 높게 setting한다.
    // 앞에서 구한 inorder_idx를 기준으로 왼쪽은 leftChild, 오른쪽은
      rightChild로 subtree를 만든다.
    node->rightChild = buildTree(inorder_idx, last_idx - 1, level);
    node->leftChild = buildTree(front_idx, inorder_idx - 1, level);
    return node;
} // end function buildTree
```

위에서 설명한 것과 같이 buildTree 재귀함수를 구현하였습니다. 이 때, level은 현재 node의 level이고, 가장 높은 level을 tree의 depth로 설정합니다. 각 line의 설명은 주석으로 추가하였습니다.

```
/* 2^n을 return하는 함수 */
int pow2(int n) {
    int i, value = 1;
    for(i = 0 ; i < n ; i++)
        value *= 2;
    return value;
} // end function pow2
```

n을 argument로 받았을 때, 2^n값을 return하는 함수입니다. 이후, number_of_slash function과 display_binary_tree의 row와 col을 계산할 때 사용됩니다.

```
/* 해당 node 에서 child node 로 갈 때 필요한 '/' or '\'의 개수를 return 하는 함수
*/
int number_of_slash(treePointer node) {
    if(tree_depth - node->level == 1)
        return 1;
    return 3 * pow2(tree_depth - node->level - 2) - 1;
} // end function number_of_slash
```

number of slash를 계산할 때는 밑에서부터 몇 번째 높이인지가 중요하기에 x를 (tree_depth – node->level) 라고 잡고, $f(x)$를 number of slash의 개수 함수라고 잡으면 점화식은 다음과 같이 세워집니다.

$$f(1) = 1 \qquad f(2) = 2 \qquad f(x+1) = 2f(x) + 1 \ (x \geq 2)$$

위 점화식을 풀면 $f(1) = 1$, $f(x) = 3 \times 2^{x-2} - 1 \ (x \geq 2)$ 이라는 일반항을 구할 수 있습니다.

```
/* binary tree 를 출력할 display_binary_tree 2nd array 를 setting 하는 함수 */
void set_display(treePointer node, int row, int col) {
    if(!node)
        /* Case of NULL */
        return;
    // 현재 좌표에 data 를 넣는다.
    display_binary_tree[row][col] = node->data;

    int i, now_row, now_col; // for 문에 이용할 variables

    if(node->leftChild) {
        /* Case of leftChild of the current node exists */
        // 현재 좌표에서 왼쪽 아래로 number_of_slash(node)만큼 내려가며 '/'를
          넣는다.
        for(i = 0, now_row = row, now_col = col ;
                                    i < number_of_slash(node) ; i++)
            display_binary_tree[++now_row][--now_col] = '/';
        // leftChild 를 기준으로 다시 setting 한다. (좌표도 함께 이동)
        set_display(node->leftChild, ++now_row, --now_col);
    }

    if(node->rightChild) {
        /* Case of rightChild of the current node exists */
        // 현재 좌표에서 오른쪽 아래로 number_of_slash(node)만큼 내려가며 '\'를
          넣는다.
        for(i = 0, now_row = row, now_col = col ;
                                    i < number_of_slash(node) ; i++)
            display_binary_tree[++now_row][++now_col] = '\\';
        // rightChild 를 기준으로 다시 setting 한다. (좌표도 함께 이동)
        set_display(node->rightChild, ++now_row, ++now_col);
    }
} // end function set_display
```

display binary tree를 setting하는 함수입니다. 각 code의 설명은 주석으로 추가하였습니다.

```
int main(void) {
    int i, j, idx; // for 문에 사용할 variables
    input = (char *)malloc(sizeof(char)*MAX_STRING_LEN);
    int input_len; // 입력된 input 의 길이

    /* inorder traversal results 를 input 에 받고 inorder array 에 alphabet 만
      저장한다. */
    gets(input);
    input_len = strlen(input);
    inorder = (char *)malloc(sizeof(char)*MAX_TREE_SIZE);
    for(i = 0, idx = 0 ; i < input_len ; i += 3)
        inorder[idx++] = input[i];
    inorder[idx] = '\0';

    /* postorder traversal results 를 input 에 받고 postorder array 에
      alphabet 만 저장한다. */
    gets(input);
    postorder = (char *)malloc(sizeof(char)*MAX_TREE_SIZE);
    for(i = 0, idx = 0 ; i < input_len ; i += 3)
        postorder[idx++] = input[i];
    postorder[idx] = '\0';
```

gets function을 이용하여 ", "를 포함해서 input에 받습니다. 이후 3개마다 alphabet이 들어오므로 for문을 이용하여 inorder array와 postorder array에 alphabet들을 차례로 받습니다.

```
    /* binary_tree 를 입력받은 inorder, postorder 에 맞게 만든다. */
    tree_size = idx; // tree 의 node 개수
    binary_tree = buildTree(0, tree_size-1, 1);
```

binary_tree를 buildTree function을 이용하여 만듭니다. (treePointer 구조)

```
    /* binary_tree 를 주어진 형식에 맞게 display 한다. */
    int row, col; // display_binary_tree 배열의 row, col 개수
    if(tree_depth == 1)
        row = col = 1;
    else {
        row = 3 * pow2(tree_depth - 2);
        col = 3 * pow2(tree_depth - 1) - 1;
    }

    // display_binary_tree 를 위에서 계산한 row, col 에 맞게 memory
allocation 한다.
    display_binary_tree = (char **)malloc(sizeof(char *)*row);
    for(i = 0 ; i < row ; i++)
        display_binary_tree[i] = (char *)malloc(sizeof(char)*col);
```

x를 tree_depth 라고 잡고, $f(x)$를 x에 따라 필요한 행렬의 row, $g(x)$를 x에 따라 필요한 행렬의 col라고 잡으면 점화식은 다음과 같이 세워집니다.

$$f(1) = 1 \qquad f(2) = 3 \qquad f(x+1) = 2f(x)\ (x \geq 2)$$

$$g(1) = 1 \qquad g(x) = 2f(x) - 1\ (x \geq 2)$$

위 점화식을 풀면 $f(1) = g(1) = 1$, $f(x) = 3 \times 2^{x-2}\ (x \geq 2)$, $g(x) = 3 \times 2^{x-1} - 1\ (x \geq 2)$ 이라는 일반항을 구할 수 있습니다.

```
    // display_binary_tree의 모든 element를 ' '로 initialization한다.
    for(i = 0 ; i < row ; i++)
        for(j = 0 ; j < col ; j++)
            display_binary_tree[i][j] = ' ';

    set_display(binary_tree, 0, row-1); // Make display_binary_tree
```

display_binary_tree의 모든 element를 ' '로 초기화한 후, head node의 위치를 맨 윗줄의 가운데 (0, row-1)로 잡습니다. 이후 set_display function을 이용하여 binary_tree를 output format으로 변환한 2차원 배열을 만듭니다.

```
    /* Display the result */
    for(i = 0 ; i < row ; i++) {
        for(j = 0 ; j < col ; j++)
            printf("%c", display_binary_tree[i][j]);
        puts("");
    }

    // Frees the allocated memory.
    free(input);
    free(inorder);
    free(postorder);
    free(binary_tree);
    for(i = 0 ; i < row ; i++)
        free(display_binary_tree[i]);
    free(display_binary_tree);
} // end main function
```

결과를 출력한 후 할당 받은 동적 메모리를 모두 해제 시킵니다.

3. 코드 실행 결과

① Case 1 (Min size)

Input

A

A

Output

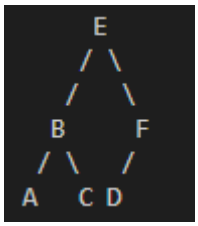
② Case 2

Input

A, B, C, E, D, F

A, C, B, D, F, E

Output



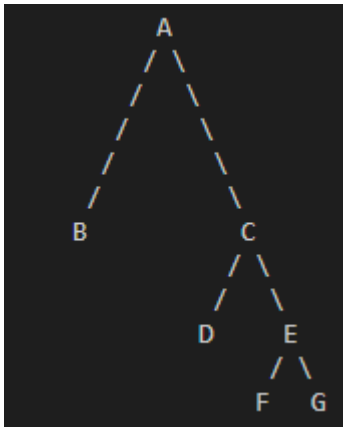
③ Case 3

Input

B, A, D, C, F, E, G

B, D, F, G, E, C, A

Output



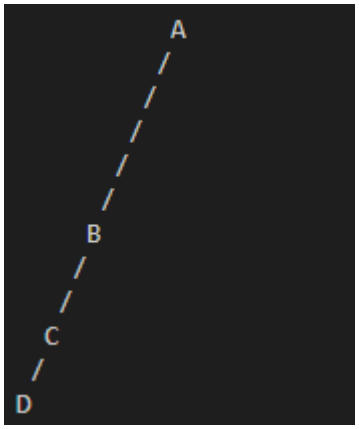
④ Case 4

Input

D, C, B, A

D, C, B, A

Output



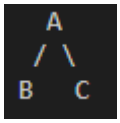
⑤ Case 5

Input

B, A, C

B, C, A

Output



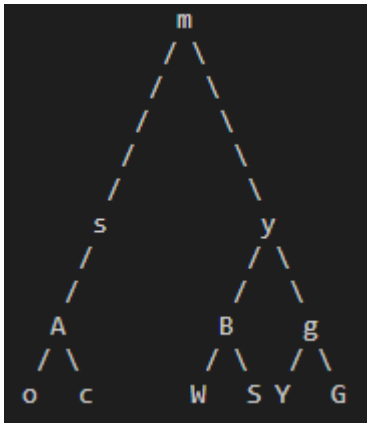
⑥ Case 6

Input

o, A, c, s, m, W, B, S, y, Y, g, G

o, c, A, s, W, S, B, Y, G, g, y, m

Output



⑦ Case 7 (Max size)

Input

f, P, g, H, h, Q, i, D, j, R, k, I, l, S, m, B, n, T, o, J, p, U, q, E, r, V, s, K, t, W, u, A, v, X, w, L, x, Y, y, F, z, Z, M, a, C, b, N, c, G, d, O, e

f, g, P, h, i ,Q, H, j, k, R, l, m, S, I, D, n, o, T, p, q, U, J, r, s, V, t, u, W, K, E, B, v, w, X, x, y, Y, L, z, Z, a, M, F, b, c, N, d, e, O, G, C, A

Output