자료구조 개론 PA2 보고서

2022310315 이용하

```c
#define MAX_NUM 300


typedef struct position{
    int row;
    int col;
};

typedef struct node{
    char inorder_leftChild[MAX_NUM];
    char inorder_rightChild[MAX_NUM];
    char postorder_leftChild[MAX_NUM];
    char postorder_rightChild[MAX_NUM];
    char data;
    int level;
    struct position pos;
}node;

typedef struct{
    node key;
}element;
```

구조체 선언 부분입니다.

Position : 나중에 트리를 출력할 때 노드의 위치를 결정할 수 있도록 만들었습니다.

Node : 가장 핵심이 되는 구조체로 왼쪽 자식의 inorder, postorder, 오른쪽 자식의 inorder, postorder를 char배열로 가지고 있습니다.

Data : 자신의 문자를 의미합니다

Level: 해당 노드의 레벨을 의미합니다.

element : 노드를 저장하는 큐를 만들기 위해 만든 구조체입니다.

```
void inqueue(node item, element queue[]);
node dequeue(element queue[]);

int node_num(node nodes[]);
node find_root(node postorder[]);
void assign_children(node* parent, node subtree[], node postorder_subtree[]);
void chars_to_nodes(node node, struct node inorder_leftChild[], struct node inorder_rightChild[], struct node
    postorder_leftChild[], struct node postorder_rightChild[]);
int iteration_count(int level);
int root_position_col(int height_of_tree);

int front = -1;
int rear = -1;
int height_of_tree = 0;

element queue[MAX_NUM] = {0};

node inorder_leftChild[MAX_NUM] = {};
node inorder_rightChild[MAX_NUM] = {};
node postorder_leftChild[MAX_NUM] = {};
node postorder_rightChild[MAX_NUM] = {};
```

함수 선언 부분입니다.

Inqueue : 큐 안에 노드를 넣기 위한 함수입니다.

Dequeue : 큐 안에 노드를 빼기 위한 함수입니다.

Node_num : node배열의 길이를 구하는 함수입니다. (배열의 크기는 300으로 고정돼 있지만, 실제 값이 들어 있는 배열의 길이를 구합니다.

Find_root : postorder로 나열된 subtree를 계산해서 root를 구하는 함수입니다.

Assign_children : parent의 노드 값인 왼쪽 자식의 postorder, inorder와 오른쪽 자식의 postorder, inorder를 할당해주는 함수입니다.

Chars_to_nodes : 왼쪽 자식의 postorder, inorder와 오른쪽 자식의 postorder, inorder가 char 배열이기 때문에 node의 배열로 바꿔주는 함수입니다.

Iteration_count : 트리를 출력할 때 노드의 레벨에 따라 반복 횟수를 결정해주는 함수입니다. (간선 출력 횟수 결정)

root_position_col : 트리의 높이를 넣으면 전체 트리의 뿌리를 열을 결정해주는 함수입니다.

아래는 전역 변수 선언 부분입니다.

Fron t: dequeue를 할 때 사용하는 변수

Rear : inqueue를 할 때 사용하는 변수

Height_of_tree : 전체 트리의 높이

Queue : element 배열로 queue를 만들어 준다

Inorder_leftChild, ... : node 변수인 char 배열 Inorder_leftChild,... 를 node 배열로 할당해주기 위한 변수

```
void inqueue(node item, element queue[]){
    if(rear >= MAX_NUM-1)
        return;
    queue[++rear].key = item;
}
```

Inqueue 함수 구현 내용입니다.

Rear값을 먼저 증가시키고 증가시킨 인덱스에 해당하는 요소에 item을 넣습니다.

Rear가 queue의 범위를 넘어 가는 경우(queue의 요소가 다 찬 경우), 입력 받은 요소를 넣지 않습니다.

```
node dequeue(element queue[]){
    if(front == rear){
        node item = {};
        return item;
    }
    return queue[++front].key;
}
```

Dequeue 함수 구현 내용입니다.

Front값을 먼저 증가시키고 증가시킨 인덱스에 해당하는 요소를 반환합니다.

Front가 rear가 같아지는 경우(빈 queue인 경우), 0과 널문자로 초기화된 node를 반환합니다.

```
int node_num(node nodes[]){
    int count = 0;
    for(int i = 0; i < MAX_NUM; ++i){
        if(nodes[i].data == '\n')
            break;
        else if(nodes[i].data == '\0')
            break;
        count++;
    }
    return count;
}
```

Node_num 함수 구현 내용입니다.

Node 배열을 입력 받으면 실제 데이터 값이 들어 있는 배열의 길이를 구합니다. 먼저 getchar에서 받아온 \n나 널 문자가 노드 배열의 마지막 문자이기 때문에 반복문을 break 해줍니다.

위의 경우가 아닌 경우마다 count를 1씩 더해주고 count값을 반환하면 전체 노드의 개수가 나옵니다.

```
node find_root(node postorder_subtree[])
{
    int nodenum = node_num(postorder_subtree);
    node item = postorder_subtree[nodenum-1];

    return item;
}
```

다음은 find_root의 구현 내용입니다.

Postorder의 node 배열을 넣으면 해당 subtree의 root node를 결정하는 함수입니다.

postorder로 나열된 subtree의 마지막 노드가 subtree의 뿌리가 되는 원리를 이용합니다.

```
void assign_children(node* parent, node inorder_subtree[], node postorder_subtree[]){
    int index = 0;
    int nodenum = node_num(inorder_subtree);

    for(int i = 0; i < nodenum; ++i){
        if(inorder_subtree[i].data == (*parent).data)
            break;
        ++index;
    }

    //왼쪽 자녀 할당(inorder)
    for(int i = 0; i < index; ++i){
        (*parent).inorder_leftChild[i] = inorder_subtree[i].data;
    }
    //왼쪽 자녀 할당(postorder)
    int leftchildren_num = strlen((*parent).inorder_leftChild);      ⚠ Implicit conversion loses in
    for(int i = 0; i < leftchildren_num; ++i){
        (*parent).postorder_leftChild[i] = postorder_subtree[i].data;
    }

    //오른쪽 자녀 할당(inorder)
    for(int i = index + 1; i < nodenum; ++i){
        (*parent).inorder_rightChild[i - (index + 1)] = inorder_subtree[i].data;
    }
    //오른쪽 자녀 할당(postorder)
    for(int i = leftchildren_num; i < nodenum-1; ++i){
        (*parent).postorder_rightChild[i -(leftchildren_num)] = postorder_subtree[ i].data;
    }


}
```

Assign_children의 함수 구현 내용입니다.

이 함수의 역할은 parent 노드의 inorder_leftChildren, inorder_rightChildren, postorder_leftChildren, postorder_rightChildren 배열을 할당하는 역할을 합니다.

Main 함수에 있는 parent를 직접 수정하기 위해 포인터로 입력 받고, parent를 포함한 자식들의 inorder_subtree와 postorder_subtree를 받습니다. Inorder_subtree에서 parent를 기준으로 왼쪽이 parent의 왼쪽 자식들, 오른쪽이 parent의 오른쪽 자식들이므로, parent의 인덱스를 받아오는 것이 필요합니다. 그 과정을 맨 위 for문에서 진행합니다. Parent의 노드

의 문자가 inorder 노드 배열에서 일치할 때까지 인덱스를 증가하면서 parent의 인덱스를 계산합니다.
Inorder_leftChildren 배열 할당 방법: 계산한 parent의 인덱스까지 inorder의 요소를 가져옵니다(inorder_subtree에서 parent의 왼쪽이 inorder 순서의 왼쪽 자식들이므로).

Inorder_rightChildren 배열 할당 방법: 계산한 parent의 인덱스 + 1 부터 inorder의 요소를 가져옵니다(inorder_subtree에서 parent의 오른쪽이 inorder 순서의 오른쪽 자식들이므로).

Postorder_leftChildren 배열 할당 방법: 먼저 계산한 inorder_leftChildren을 이용해 leftChildren의 개수를 먼저 구합니다(strlen을 쓴 이유는 char배열이기 때문). 왼쪽 자식들의 개수만큼 차례대로 postorder의 요소를 가져옵니다. 예를 들어, postorder가 ABC이고 parent가 C인 경우, 왼쪽 자식들은 2명이기 때문에 postorder를 앞에서부터 2번 받아오면 post_order 순으로 나열된 왼쪽 자식들이 받아와 집니다. [i – (index+1)]을 한 이유는 인덱스가 0부터 시작하도록 한 것입니다.

Postorder_rightChildren 배열 할당 방법: 왼쪽 자식들의 개수로 인덱스를 시작하고 parent는 받지 않기 위해 nodenum-2까지 받아옵니다. 예를 들어, postorder가 ABCDF이고 parent가 F인 경우이고 왼쪽 자식들이 AB인 경우, 왼쪽 자식들의 개수만큼 건너뛰고 CD를 읽어 post_order 순으로 나열된 오른쪽 자식들이 받아와 집니다. [i-(leftchild_num)]을 한 이유는 인덱스가 0부터 시작하도록 한 것입니다.

```
void chars_to_nodes(node node, struct node inorder_leftChild[], struct node inorder_rightChild[], struct node
    postorder_leftChild[], struct node postorder_rightChild[]){
    for(int i = 0; i < strlen(node.inorder_leftChild); ++i){
        inorder_leftChild[i].data = node.inorder_leftChild[i];
    }
    for(int i = 0; i < strlen(node.postorder_leftChild); ++i){
        postorder_leftChild[i].data = node.postorder_leftChild[i];
    }
    for(int i = 0; i < strlen(node.inorder_rightChild); ++i){
        inorder_rightChild[i].data = node.inorder_rightChild[i];
    }
    for(int i = 0; i < strlen(node.postorder_rightChild); ++i){
        postorder_rightChild[i].data = node.postorder_rightChild[i];
    }
}
```

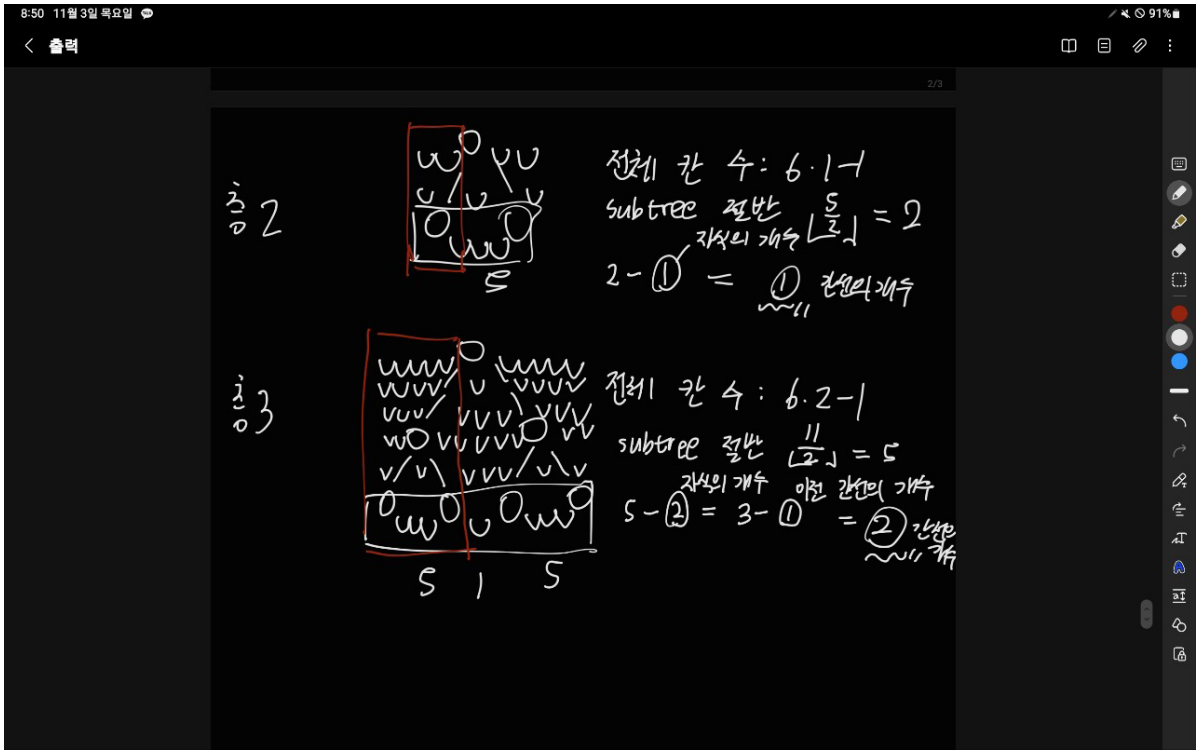
Char 배열을 node 배열에 할당하는 함수입니다.
Node 구조체 내부에 inorder_leftChildren,..,postorder_rightChildren 은 char 배열이기 때문에 node 배열을 입력 받는 함수들에 사용하지 못하므로 char 배열을 node 배열로 변환하는 작업이 필요합니다. Node 배열로 이루어진 inorder_leftChildren,..,postorder_rightChildren을 argument로 받고 argument로 받아온 node의 inorder_leftChildren,..,postorder_rightChildren를 각각 데이터 값에 할당합니다.

```
int iteration_count(int level){
    int floor = height_of_tree - level;
    switch(floor){
        case 0:
            return 0;
        case 1:
            return 1;
        case 2:
            return 2;
        case 3:
            return 5;
        case 4:
            return 11;
        case 5:
            return 23;
        case 6:
            return 47;
        default:
            return -1;
    }

}
```

출력을 할 때, 반복 횟수를 결정해주는 함수입니다(간선 개수). Parent의 level을 입력 받으면, 자식들을 출력할 때까지 몇 개의 간선을 출력해줘야 할지 결정합니다. Floor는 층의 의미로 (트리의 높이-level)로 결정됩니다. 수학적 원리가 있지만, 최대 트리의 깊이가 6점을 가만해 직접 미리 계산해서 switch문으로 각각의 경우를 반환해줬습니다.

수학적 원리:



층이 2인 경우- floor((6*1-1)/2) – 1 = 1

층이 3인 경우- floor((6*2-1/2)) – 2 -1 = 2

층이 4인 경우 – floot((6*3-1)/2) – 3 - 3 =5

…

```
int root_position_col(int height_of_tree){
    switch(height_of_tree){
        case 0:
            return 0;
        case 1:
            return 1;
        case 2:
            return 2;
        case 3:
            return 5;
        case 4:
            return 11;
        case 5:
            return 23;
        case 6:
            return 47;
        case 7:
            return 97;
        default:
            return -1;
    }
}
```

이 함수는 트리의 높이에 따라 전체 트리의 뿌리의 col위치를 결정하는 함수입니다. 앞에서 구한 수학적 원리에 의해 미리 계산해 switch 구문을 이용해 반환했습니다.

트리의 높이가 0 인 경우(노드 0개) – 0

트리의 높이가 1인 경우(노드 1개) – 1

트리의 높이가 2인 경우 – floor((6*1-1)/2) = 2

트리의 높이가 3인 경우 – floor((6*2-1)/2) = 5

…

Main 함수 내용)

```
int main(){

    node inorder[MAX_NUM] = {};
    node postorder[MAX_NUM] = {};

    for(int i = 0; (inorder[i].data = getchar()) != '\n';){
        if(inorder[i].data != ' ' && inorder[i].data != ',')
            ++i;
    }
    for(int i = 0; (postorder[i].data = getchar()) != '\n';){
        if(postorder[i].data != ' ' && postorder[i].data != ',')
            ++i;
    }

    int nodenum = node_num(inorder);

    node first = postorder[nodenum-1];
    first.level = 1;
    assign_children(&first, inorder, postorder);
    inqueue(first, queue);
```

Inorder : 전체 트리의 inorder 노드 배열을 만듭니다.

Postorder : 전체 트리의 Postorder 노드 배열을 만듭니다.

For문은 사용자로부터 inorder, postorder 배열의 요소를 입력 받는 역할을 합니다.
Getchar를 사용해서 문자 하나씩을 '\n'이 입력될 때까지 계속 받습니다. 이때 if문을 이용해 콤마와 띄어쓰기는 배열에 저장하지 않도록 합니다(i를 증가시키지 않음).

Node_num 함수를 통해 inorder 노드 배열의 길이를 구하고 postorder의 마지막 요소를 뽑아 뿌리 노드로 저장합니다(postorder의 마지막은 무조건 전체 트리의 뿌리).
뿌리의 레벨을 1로 설정합니다.
뿌리의 inorder_leftChildren, … , postorder_rightChildren을 assign_children함수를 통해 할당합니다.
그리고 큐에 뿌리 노드를 넣습니다.

첫번째 while문 – 트리를 구현(뿌리 노드부터 아래쪽으로 왼쪽에서 오른쪽 순서로 나열함)하고 트리의 높이를 구함

```
while(1){
    memset(inorder_leftChild, 0, sizeof(inorder_leftChild));
    memset(inorder_rightChild, 0, sizeof(inorder_rightChild));
    memset(postorder_leftChild, 0, sizeof(postorder_leftChild));
    memset(postorder_rightChild, 0, sizeof(postorder_rightChild));

    node parent = dequeue(queue);

    if(!parent.data)
        break;

    if((!strlen(parent.postorder_leftChild) && !strlen(parent.postorder_rightChild)))
        continue;

    chars_to_nodes(parent, inorder_leftChild, inorder_rightChild, postorder_leftChild, postorder_rightChild);

    node leftChild = find_root(postorder_leftChild);
    leftChild.level = parent.level + 1;
    node rightChild = find_root(postorder_rightChild);
    rightChild.level = parent.level + 1;

    height_of_tree = leftChild.level;


    assign_children(&leftChild, inorder_leftChild, postorder_leftChild);
    assign_children(&rightChild, inorder_rightChild, postorder_rightChild);




    if(leftChild.data != '\0')
        inqueue(leftChild, queue);
    if(rightChild.data != '\0')
        inqueue(rightChild, queue);


}
```

1.Memset 함수를 통해 Inorder_leftChild, … , postorder_rightChild를 초기화 해줍니다.

2.Parent를 큐에서 뽑아 옵니다. (첫번째인 경우 뿌리 노드)

3.Parent의 문자가 널문자인 경우 while문을 빠져나옵니다(큐가 모두 빔)

4.parent의 왼쪽 자식과 오른쪽 자식이 모두 0개인 경우 continue를 합니다. Leaf node가 뒤의 연산을 하지 않기 위한 의도임(특히, 트리의 높이를 구할 때 원래 높이보다 1큰 것이 문제가 됨)

5. chars_to_nodes 함수를 통해 parent의 멤버 변수이자 char 배열인 Inorder_leftChild, … , postorder_rightChild를 노드 배열인 Inorder_leftChild, … , postorder_rightChild에 할당해줌

6. parent의 왼쪽 자식 노드를 왼쪽 자식들의 postorder를 find_root에 입력해 가져옵니다.

7. parent의 오른쪽 자식 노드를 오른쪽 자식들의 postorder를 find_root에 입력해 가져옵니다.

8. 왼쪽 자식과 오른쪽 자식의 레벨에 부모 레벨 + 1을 할당합니다.

9. 트리의 높이를 자식의 레벨로 갱신합니다.

10. 왼쪽 자식과 오른쪽 자식 각각에 assign_children을 이용해 멤버 변수이자 char 배열인 Inorder_leftChild, … , postorder_rightChild를 할당해줍니다.

11. 왼쪽 자식의 문자가 널문자가 아닌 경우 큐에 넣어줍니다.

12. 오른쪽 자식의 문자가 널문자가 아닌 경우 큐에 넣어줍니다.

다음과 같은 코드로 인해, 트리의 뿌리를 먼저 큐에서 뽑아내고, 왼쪽 자식부터 오른쪽 자식을 각각 큐에 넣게 된다. 다음으로, 왼쪽 자식을 큐에서 뽑아내고 왼쪽 자식의 자식이 있는 경우 또 큐에 넣게 된다. 그 다음 뿌리의 오른쪽 자식을 큐에서 뽑아내고 같은 원리로 작동한다. 결국, 트리의 뿌리부터 시작해 아래쪽으로 그리고 왼쪽에서 오른쪽 순으로 큐에 저장하게 된다(널문자는 저장 안 됨). 그렇게 모든 노드를 순회하게 되고 큐가 빈 경우 while문을 빠져나가는 것이다.
이런 과정을 하면서, Inorder_leftChild, … , postorder_rightChild를 초기화하는 과정은 반드시 필요하다. 왜냐하면, 전에 계산했던 노드의 값들과 혼선될 수 있기 때문이다.

Tree를 이차원 배열로 동적 할당하는 코드

```
front = -1;
rear = -1;
first.pos.col = root_position_col(height_of_tree);
first.pos.row = 0;
int width = first.pos.col*2 + 1;
int height = height_of_tree;
for(int i = 0; i < height_of_tree; ++i){
    height += iteration_count(height_of_tree-i);
}

if(!height)
    height = 1;

char** tree = (char**)malloc(height*sizeof(char*));
for(int i = 0; i < height; ++i){
    tree[i] = (char*)malloc(width*sizeof(char));
}
for(int i = 0; i < height; ++i){
    for(int j = 0; j < width; ++j){
        tree[i][j] = ' ';
    }
}

tree[first.pos.row][first.pos.col] = first.data;
inqueue(first, queue);
```

1.큐에서 노드를 다시 가져오고 위치를 할당한 노드를 새로 넣기 위해 front와 rear를 -1로 할당해준다.
2.전체 노드의 뿌리 노드의 열 위치(root_position_col 함수를 이용해)와 행 위치(0)를 할당해준다.
3.tree의 전체 너비는 뿌리 노드의 열 위치 * 2 + 1 이므로 설정해준다
4.height는 트리의 높이 + (각각의 레벨에 따른 간선 개수의 합)과 같으므로 설정해준다.

5.height가 0인 경우, 동적 할당이 정상적으로 되지 않으므로, 1로 설정해줌 [이를 통해, 빈 트리는 공백을 출력하게 함]

6.tree를 이차원 포인터를 이용해 동적 할당하는 공식을 사용함.

7.tree의 모든 요소를 공백으로 채움.

8.tree의 뿌리 노드의 위치에 문자를 넣어줌.

9.뿌리 노드를 큐에 넣어줌

두 번째 while문 1 - 이차원 배열 tree에 간선과 자식 노드를 넣어줌

```
while(1){
    memset(inorder_leftChild, 0, sizeof(inorder_leftChild));
    memset(inorder_rightChild, 0, sizeof(inorder_rightChild));
    memset(postorder_leftChild, 0, sizeof(postorder_leftChild));
    memset(postorder_rightChild, 0, sizeof(postorder_rightChild));
    int i,j;

    node parent = dequeue(queue);
    int count = iteration_count(parent.level);
    chars_to_nodes(parent, inorder_leftChild, inorder_rightChild, postorder_leftChild, postorder_rightChild)

    node leftChild = find_root(postorder_leftChild);
    leftChild.level = parent.level + 1;
    node rightChild = find_root(postorder_rightChild);
    rightChild.level = parent.level + 1;


    assign_children(&leftChild, inorder_leftChild, postorder_leftChild);
    assign_children(&rightChild, inorder_rightChild, postorder_rightChild);

    if(!parent.data)
        break;
```

1.parent노드를 큐에서 뽑아줌(첫번째인 경우, 뿌리 노드)

2.parent의 레벨에 해당하는 간선 출력 반복 횟수를 count에 저장함.

3. chars_to_nodes 함수를 통해 parent의 멤버 변수이자 char 배열인 Inorder_leftChild, … , postorder_rightChild를 노드 배열인 Inorder_leftChild, … , postorder_rightChild에 할당해줌 (find_root와 assign_children의 알규먼트로 입력하기 위해)

4. parent의 왼쪽 자식과 오른쪽 자식을 받아오고 레벨도 할당함.

5. 왼쪽 자식과 오른쪽 자식의 Inorder_leftChild, … , postorder_rightChild를 할당함.

6. parent가 널문자인 경우 while문을 나옴(큐가 모두 빈 상태).

두 번째 while문 2)

```
    if(leftChild.data)
    {
        for( i= parent.pos.row+1, j = 1; j <= count; ++i,++j){
            tree[i][parent.pos.col-j] = '/';
        }
        tree[i][parent.pos.col-j] = leftChild.data;
        leftChild.pos.row = i;
        leftChild.pos.col = parent.pos.col-j;
        inqueue(leftChild, queue);
    }
    if(rightChild.data)
    {
        for( i= parent.pos.row+1, j = 1; j <= count; ++i,++j){
            tree[i][parent.pos.col+j] = '\\';
        }
        tree[i][parent.pos.col+j] = rightChild.data;
        rightChild.pos.row = i;
        rightChild.pos.col = parent.pos.col+j;
        inqueue(rightChild, queue);
    }

}
```

7.왼쪽 자식이 널 문자가 아닌 경우, 간선을 출력하는 for문을 돎.

8.부모의 행의 +1부터 행을 설정하고, 열은 부모의 열의 –j로 설정함. 이때, j는 1부터 시작해 반복횟수(간선의 개수)까지 반복한다. 즉, for문을 반복하면서 행은 점점 증가해 아래로 1칸씩 내려가고 부모의 열을 기준으로 왼쪽으로 한칸씩 '/'을 배열에 넣는다.

9.마지막으로 왼쪽 자식의 행의 위치를 i로 설정하고, 열의 위치를 부모의 열의 위치 –j로 설정해준다. 그 위치에 왼쪽 자식의 문자도 넣어준다.

10.왼쪽 자식을 큐에 넣어준다(기존의 왼쪽 자식의 정보[첫번째 while문에서 설정한]에 위치까지 추가하는 꼴)

11.오른쪽 자식이 널 문자가 아닌 경우, 간선을 출력하는 for문을 돎.

12.부모의 행의 +1부터 행을 설정하고, 열은 부모의 열의 +j로 설정함. 이때, j는 1부터 시작해 반복횟수(간선의 개수)까지 반복한다. 즉, for문을 반복하면서 행은 점점 증가해 아래로 1칸씩 내려가고 부모의 열을 기준으로 오른쪽으로 한칸씩 '\'을 배열에 넣는다.

13.마지막으로 오른쪽 자식의 행의 위치를 i로 설정하고, 열의 위치를 부모의 열의 위치 +j로 설정해준다. 그 위치에 오른쪽 자식의 문자도 넣어준다.

14.오른쪽 자식을 큐에 넣어준다(기존의 오른쪽 자식의 정보[첫번째 while문에서 설정한]에 위치까지 추가하는 꼴)

두번째 while문 추가 이해) 기존의 첫번째 while문과 비슷한 부분은 첫번째 while문에서 만들었던 노드의 정보를 모두 복사하는 과정이라고 볼 수 있다. 두번째 while문에서 추가

하는 것은 각각의 노드의 위치이다. Parent의 위치에 따라 상대적으로 자식들의 위치가 설정되기 때문에 노드의 위치를 할당해주고 큐에 집어넣는 과정이 필요하다. 기존의 첫 번째 while문과 비슷하게 뿌리 노드부터 시작해 아래쪽으로 왼쪽부터 오른쪽 순으로 노드를 각각 큐에 넣고 뽑아 오는 과정이다. 노드를 뽑아내고 자식들을 각각 출력하는 알고리즘인 것이다. 큐에서 모든 노드를 뽑아낸 경우, while문을 종료한다.

```
for(int i = 0; i < height; ++i){
    for(int j = 0; j < width; ++j){
        printf("%c", tree[i][j]);
    }
    printf("\n");
}
```

마지막으로 tree의 모든 요소를 출력해주면 끝이다.