

# 2-3Tree

학번 : 201002513

이름 : 최 혁수

## ☐ 실습 10. 2-3Tree

- 개요 & 알고리즘
- 주요 소스코드
  - Insert, Delete, Search, Print
- 실행결과 분석 & 구현상의 오류 및 한계

## ☐ 소감문 & 한계성

## ☐ All Source Code

# 실습 10. 2-3Tree

- 개요 & 알고리즘
- 주요 소스코드
  - Insert, Delete, Search, Print
- 실행결과 분석 & 구현상의 오류 및 한계

## 2-3Tree

### □ 개요

이원탐색트리(Binary Search Tree)의 기초적 알고리즘을 이해하고  
노드(Node)의 추가적 연결에 따른 무분별한 높이(Height)의 증가로 나타나는  
취약점을 말단노드(leaf node=external node와 연결되는 노드)를 밸런스(Balance)하게  
만드는 2-3Tree를 구현한다.

### □ 이원탐색트리 취약점

이원탐색트리의 취약점은  $n$ 개의 노드를 포함하는 트리의 높이가  $n$ 을 만족할  
때 일어난다. 즉 이원탐색트리의 기초 알고리즘을 만족하면서 높이의 증가로써  
노드가 추가가 되면 이를 시각적으로 구현할 때 선형구조를 띄게 된다.

이는 높이가  $n$ 으로 만족하고, 트리탐색의 시간 또한  $O(n)$ 시간이 걸리게 되며  
이원탐색트리에 기대하는  $O(\log n)$ 의 효과를 거둘수 없다.

취약점 보완 균형탐색트리 알고리즘으로 AVL, Red-Black, 2-3, 2-3-4, B-Tree,  
B+-Tree등이 제시되었다.

### □ Binary Search Tree Algorithm

1. 모든 원소는 키를 가지며 중복을 허용하지 않는다.
2. 왼쪽 서브트리의 키값은 부모의 키값보다 작다.
3. 오른쪽 서브트리의 키값은 부모의 키값보다 크다.
4. 모든 서브트리 또한 이원탐색트리이다.

### □ 2-3Tree Algorithm

1. 삽입단계 : 유효 Data의 개수는 2개이며, 추가적인 Insert함수가 호출될시  
중간값을 부모에게 보내고 이외 2개의 노드를 부모의 자식(혹은 자식의 추가  
데이터)으로 연결하여 구현
2. 삭제단계 : 삭제하고자 하는 Key를 가진 노드 혹은 대체할만한 Key를 가진  
Node를 탐색하고 부모와 형제노드의 데이터값 비교를 통해 적절한 위치에  
adjust 시킨다.
3. 탐색단계 : BST의 알고리즘을 응용하여 Key값에 따른 재귀적 Search를  
구현하여 말단노드까지 탐색한다.
4. 출력단계 : Queue를 설정하여 Tree의 Root부터 각 레벨별 노드를 Queue에  
넣고 출력하는 방법을 이용

# Header(1/2)

## □ 23tree.h

```
#ifndef _23tree_h
#define _23tree_h

typedef struct node23 {
    int small; // 2-3Tree의 small값
    int large;
    struct node23* left; // Left, Middle, Right Link
    struct node23* middle;
    struct node23* right;
    struct node23* parent; // Split을 위해 각 노드는 부모노드의 주소를 저장해둔다.
} node23;

typedef struct tree23{ // 2-3Tree 설정
    node23* root;
} tree23;

node23* makeNode23();
tree23* makeTree23();
int isLeaf(node23* node);

/* Search 연관 함수 */
node23* target23(node23* node, int key);
node23* search23(node23* node, int key);

/* Insert 연관 함수 */
int* rearrange(int a, int b, int c);
void insertIntoNode23(tree23* tree, node23* node, int key);
void splitLeaf23(tree23* tree, node23* node, int key);
void splitNode23(tree23* tree, node23* node, node23* newChild, int key);
void insert23(tree23* tree, int key);

/* Delete 연관 함수 */
node23* successor23(node23* node, int key);
void swapWithSuccessorRecursive(node23* node, int key);
int keyCount23(node23* node);
node23* deleteNode23(tree23* tree, node23* node, int key);
void delete23(tree23* tree, int key);
tree23* fixNode23(tree23* tree, node23* node);
void destroy23(node23* node);

/* Print 연관 함수 */
void print23(tree23* tree);
void depthFirstPrint23(node23* node);
void printNode23(node23* node);

#endif
```

## Header(2/2)

□ **queue.h**     \* Print함수를 구현할 때 트리를 깊이순으로 출력하기 위해 Queue 사용

```
#ifndef _queue_h
#define _queue_h
#include "23tree.h"

/* queue및 double linked list의 알고리즘을 결합하여 node를 결합 */
typedef struct queueNode {
    struct queueNode* next;
    struct queueNode* prev;
    node23* object;
} queueNode;

typedef struct queue {
    queueNode* head; // queue의 시작
    queueNode* tail; // queue의 끝
} queue;

queueNode* createQueueNode(node23* object);
queue* createQueue();
queue* enqueue(queue* self, node23* object);
node23* dequeue(queue* self);

#endif
```

# INSERT

## ❑ void insert23(tree23\* tree, int key)

```
void insert23(tree23* tree, int key){ // Key값을 Insert할때 처음 호출되는 함수

    node23* newNode;
    // Leaf Node가 아닌경우 탐색 & 삽입 과정을 재수행
    if (tree->root) insertIntoNode23(tree, tree->root, key);
    if (!tree->root) { // Leaf Node인 경우
        newNode = makeNode23();
        tree->root = newNode;
        // Leaf node일 경우, Root를 할당한 새로운 노드를 가리키게 만든다.
        // 이는 Tree내의 Node가 Empty상태일때 일어나며 Insert Key를 Tree에 저장하는
        // insert23()이 호출된다.
        insert23(tree, key);
        return;
    }
}
```

## ❑ void insertIntoNode23(tree23\* tree, node23\* node, int key)

```
// 실질적인 Insert 구현 함수이며, 데이터의 갯수에 따라 Split를 발생시켜
// 2-3Tree 알고리즘에 따라 위치시킨다.
void insertIntoNode23(tree23* tree, node23* node, int key){
    int* newNode;
    if(isLeaf(node)){ // Leaf node경우
        if (!keyCount23(node)){ // node에 데이터가 존재하지 않을때
            node->small = key; // small값에 저장
        }
        else if(keyCount23(node) == 1) { // 데이터가 1개 존재(=int small)
            node->large = key; // large에 저장
        }
        else{ // 2개의 데이터가 존재하는 노드에 Key를 추가로 넣을때(3개 데이터 = Split)
            // 정렬된 데이터를 newNode가 가리킨다.
            newNode = rearrange(node->small, node->large, key);
            // 중간값을 제외한 두 데이터를 각각 저장
            node->small = newNode[0]; node->large = newNode[2];
            splitLeaf23(tree, node, newNode[1]); // promoted된 node & data를 가지고 Split를 발생
        }
        return;
    }

    else{ // Leaf node가 아닌경우 key값으로 node를 targeting 한다.
        node23* tempLeaf = target23(node, key);
        // target23() 호출전 임시로 저장된 tempLeaf가 가리키는 노드 & key로 재귀탐색 구현
        if(tempLeaf) insertIntoNode23(tree, tempLeaf, key);
        return;
    }
}
```

# SPLIT

\* void splitLeaf23() 프로젝트 파일 첨부

```
void splitNode23(tree23* tree, node23* node, node23* newChild, int key)
/* Leaf node가 아닌 Tree의 중간노드에서 Split가 일어난 경우
   데이터는 Leaf node를 기준으로 insert 되므로, 해당 Split은 insert로 생긴 splitLeaf23()로부터 발생되었다. */
void splitNode23(tree23* tree, node23* node, node23* newChild, int key){
    int newKey=0;
    if (!node->parent) { // parent node가 없는 Tree내의 단일 노드인 경우(위 splitLeaf23()의 한 조건과 동일)
        node23* newRoot = makeNode23();
        newRoot->small = key;
        newRoot->left = node; newRoot->middle = newChild;
        node->parent = newChild->parent = newRoot;
        tree->root = newRoot;
        return; }
    // parent의 데이터가 1개인 경우(=1-2 Node), (위 splitLeaf23() 과 동일)
    if (keyCount23(node->parent) == 1) {
        if (key < node->parent->small) { // small값보다 작을때
            node->parent->right = node->parent->middle;
            node->parent->large = node->parent->small;
            node->parent->middle = newChild;
            newChild->parent = node->parent;
            node->parent->small = key;
            return; }
        if (key > node->parent->small) { // small값보다 클때
            node->parent->right = newChild;
            newChild->parent = node->parent;
            node->parent->large = key;
            return; }
    }
    if (keyCount23(node->parent) == 2) { // parent의 데이터가 2개 존재하는 경우(=2-3Node)
        node23* newNode = makeNode23();
        if (key < node->parent->small) {
            // 기본알고리즘은 splitLeaf23과 동일하게 key값을 저장하기 위해 data와 child link를 해당 위치로 한단계 이동하며,
            // parent의 데이터 혹은 key값을 newKey에 저장한 후 다시 Split가 발생할 parent node에서 Promoted된 이 값으로
            // 반복적 Split를 발생시킨다.
            newNode->left = node->parent->middle; newNode->small = node->parent->large;
            newNode->middle = node->parent->right;
            node->parent->middle = newChild; newChild->parent = node->parent;
            node->parent->large = 0; node->parent->right = NULL;
            newKey = node->parent->small; node->parent->small = key;
            splitNode23(tree, node->parent, newNode, newKey);
            return;
        }
        // small < key < large 경우
        if (key > node->parent->small && key < node->parent->large) {
            newNode->left = newChild; newChild->parent = newNode;
            newNode->small = node->parent->large;
            newNode->middle = node->parent->right;
            node->parent->large = 0; node->parent->right = NULL;
            // promoted key = key(중간값)
            newKey = key;
            splitNode23(tree, node->parent, newNode, newKey);
            return;
        }
        if (key > node->parent->large) { // small < large < key 경우
            newNode->left = node->parent->right; newNode->small = key;
            newNode->middle = newChild; newChild->parent = newNode;
            node->parent->right = NULL;
            // promoted key = parent->large(중간값)
            newKey = node->parent->large; node->parent->large = 0;
            splitNode23(tree, node->parent, newNode, newKey);
            return;
        }
    }
}
```



# Delete(1/2)

## □ node23\* deleteNode23(tree23\* tree, node23\* node, int key)

node23\* deleteNode23(tree23\* tree, node23\* node, int key){ // key값을 찾은 후 delete 구현

```
    if (isLeaf(node)){
        node23* match = search23(node, key); // key값을 가진 node를 match가 가리킨다.
        if (match) { // key값이 존재하면 계승할 자식노드를 Recursive하게 탐색하여 해당 node의 데이터값을 변경한다.
            swapWithSuccessorRecursive(match, key);
            deleteNode23(tree, match, key); // 단말노드까지 위 두 함수를 재귀호출
            return node; // node는 key
        } else { return NULL; } // 삭제할 key가 존재하지 않으면 NULL 리턴
    }
    else { // 단말노드일 경우 key값을 삭제하고 삭제된 위치에 다른 데이터로 대체한다.
        if (key == node->large) {
            node->large = 0;
            return node; }
        if (key == node->small && node->large) {
            node->small = node->large;
            node->large = 0;
            return node; }
        if (key == node->small && !node->large){
            if (!node->parent){ // parent가 NULL일때, key를 삭제하면 Tree는 empty가 되며 null을 반환
                tree->root = NULL;
                return NULL; }
            // parent가 존재할 경우 알고리즘은 key값을 삭제하면 삭제될 node를 잡고 있으면서 parent와 parent의 child의 데이터 갯수를
            // 산출하여 node를 유지하면서 key, child link를 이동시킨다.
            if (keyCount23(node->parent) == 1) { // 데이터를 1개 가지고 있는 경우 : redistribute 수행
                if (key < node->parent->small) { // key < small일때
                    // 부모의 small값을 node로 가져온 후 부모중간자식의 small데이터로 대체
                    node->small = node->parent->small;
                    node->parent->small = node->parent->middle->small;
                    // 부모의 중간자식의 유효데이터가 2개 존재할 경우
                    if (keyCount23(node->parent->middle) == 2){
                        // 부모의 small로 가져온 중간자식의 small의 자리를 large의 데이터로 대체한 후 large 초기화(0)
                        node->parent->middle->small = node->parent->middle->large;
                        node->parent->middle->large = 0;

                        // 벨런싱되었으므로 해당 노드 return
                        return node;
                    }
                }
                // 중간자식의 데이터가 1개인 경우, 그 데이터를 node로 가져온 후 중간자식을 없앴 후 parent부터 fixing 한다.
                node->parent->small = 0;
                node->large = node->parent->middle->small;
                destroy23(node->parent->middle);
                fixNode23(tree, node->parent);
                return node;
            }
        } else {
            node->small = node->parent->small;
            node->parent->small = 0;
            if (keyCount23(node->parent->left) == 2) {
                node->parent->small = node->parent->left->large;
                node->parent->left->large = 0;
                return node;
            }
            node->parent->left->large = node->small;
            destroy23(node->parent->middle);
            fixNode23(tree, node->parent);
            return node;
        }
    }
}
```

# Delete(2/2)

## ❑ node23\* deleteNode23(tree23\* tree, node23\* node, int key)

} else if (keyCount23(node->parent) == 2) { // 부모의 데이터가 2개인 경우 : merge를 수행

if (key < node->parent->small) {

if (keyCount23(node->parent->middle) == 2) {

node->small = node->parent->small;

node->parent->small = node->parent->middle->small;

node->parent->middle->small = node->parent->middle->large;

node->parent->middle->large = 0;

return node;

} else if (keyCount23(node->parent->right) == 2) {

node->small = node->parent->small;

node->parent->small = node->parent->middle->small;

node->parent->middle->small = node->parent->right->small;

node->parent->right->small = node->parent->right->large;

node->parent->right->large = 0;

return node;

} else {

node->small = node->parent->small;

node->large = node->parent->middle->small;

node->parent->small = node->parent->large;

node->parent->middle->small = node->parent->right->small;

node->parent->large = 0;

destroy23(node->parent->right);

return node;

}

} else if (key < node->parent->small && key > node->parent->large) {

if (keyCount23(node->parent->left) == 2) {

node->small = node->parent->small;

node->parent->small = node->parent->left->large;

node->parent->left->large = 0;

return node;

} else if (keyCount23(node->parent->right) == 2) {

node->small = node->parent->large;

node->parent->large = node->parent->right->small;

node->parent->right->small = node->parent->right->large;

node->parent->right->large = 0;

return node;

} else {

node->small = node->parent->large;

node->large = node->parent->right->small;

node->parent->large = 0;

destroy23(node->parent->right);

return node;

}

} else {  
if (keyCount23(node->parent->middle) == 2) {

node->small = node->parent->large;

node->parent->large = node->parent->middle->large;

node->parent->middle->large = 0;

return node;

} else if (keyCount23(node->parent->left) == 2) {

node->small = node->parent->large;

node->parent->large = node->parent->middle->small;

node->parent->middle->small = node->parent->small;

node->parent->small = node->parent->left->large;

node->parent->left->large = 0;

return node;

} else {

node->parent->middle->large = node->parent->large;

node->parent->large = 0;

destroy23(node->parent->right);

return node;

}}}}

return NULL;

}

# Search

## □ node23\* search23(node23\* node, int key)

```
node23* search23(node23* node, int key){ // target23() 함수와 구조는 비슷하지만 반환값 & if문의 정의는 다르다.
    node23* newNode = NULL;
    // key를 찾으면 해당 node를 반환한다.
    if(key == node->small || key == node->large) return node;
    else if(isLeaf(node)) return NULL; // leaf노드면 key가 존재하지 않으므로 NULL 반환
    if(!node->large){ // node->large값이 존재하면
        if (key < node->small && node->left) newNode = search23(node->left, key);
        if (key > node->small && node->middle) newNode = search23(node->middle, key);
    }

    else{ // node->large값이 존재하지 않으면
        if (key < node->small && node->left) newNode = search23(node->left, key);
        if (key > node->small && key < node->large && node->middle) newNode =
search23(node->middle, key);
        if (key > node->large && node->right) newNode = search23(node->right, key);
    }
    return newNode; // target node를 반환
}
```

## □ node23\* target2323(node23\* node, int key)

```
node23* target23(node23* node, int key){ // Key를 넣을 Target Node를 찾는다
    node23* newNode = NULL;
    if (isLeaf(node)) return node; // Leaf node이면 해당노드를 리턴
    if (key == node->small || key == node->large) return NULL; // Key가 존재하면 NULL 반환
    if (keyCount23(node) == 1){ // node의 data가 1개이면(=Only exist small data)
        // 왼쪽 자식이 존재하고, small값보다 key가 작을때 = 왼쪽자식에서 target 재시작
        if (key < node->small && node->left) newNode = target23(node->left, key);
        // 중간 자식이 존재하고, small값보다 클때
        if (key > node->small && node->middle) newNode = target23(node->middle, key);
    }

    else{ // node의 데이터가 2개(=즉, small, large값 둘다 존재)
        if (key < node->small && node->left) newNode = target23(node->left, key);
        if (key > node->small && key < node->large && node->middle) newNode =
target23(node->middle, key);
        // 오른쪽 자식이 존재하고, large값보다 클때
        if (key > node->large && node->right) newNode = target23(node->right, key);
    }
    return newNode; // target node를 반환
}
```

# PRINT

## ❑ void print23(tree23\* tree)

```
/* 2-3tree의 깊이순으로 queue를 이용한 출력 */
void print23(tree23* tree){
    if(tree && tree->root) depthFirstPrint23(tree->root);
}
```

## ❑ void depthFirstPrint23(node23\* node)

```
void depthFirstPrint23(node23* node){

    node23* temp;
    queue* nodeQueue = createQueue(); // queue 생성
    enqueue(nodeQueue, node); // queue에 node를 삽입(Tree내의 최상위 노드부터 순차적으로 삽입)
    while (nodeQueue->tail) { // 삽입된 node가 저장된 queue가 empty가 될때까지 수행
        temp = dequeue(nodeQueue); // node를 꺼내 temp가 가리킨다.
        printNode23(temp); // temp내의 데이터를 출력
        // temp를 기준으로 왼쪽자식을 다시 queue에 삽입
        if (temp->left) enqueue(nodeQueue, temp->left);
        if (temp->middle) enqueue(nodeQueue, temp->middle);
        if (temp->right) enqueue(nodeQueue, temp->right);
    }
}
```

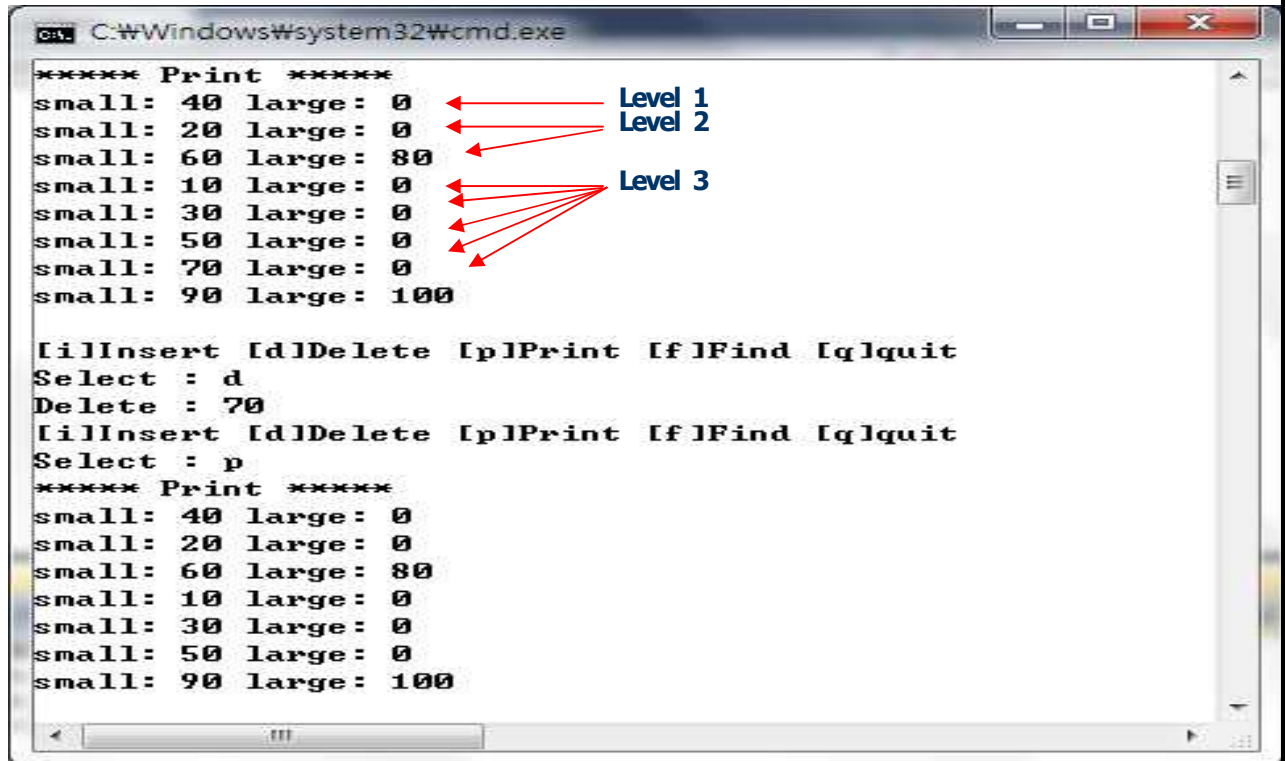
## ❑ void printNode23(node23\* node){

```
void printNode23(node23* node){

    int a, b;
    a = b = 0; // 유효 데이터를 가진 node의 데이터를 저장 및 출력
    if (node->small) a = node->small;
    if (node->large) b = node->large;
    printf("small: %d large: %d\n", a, b);
}
```

# 실행결과 분석 & 구현상의 오류 및 한계

## □ 실행 결과화면



```
***** Print *****
small: 40 large: 0
small: 20 large: 0
small: 60 large: 80
small: 10 large: 0
small: 30 large: 0
small: 50 large: 0
small: 70 large: 0
small: 90 large: 100

[i]Insert [d]Delete [p]Print [f]Find [q]quit
Select : d
Delete : 70
[i]Insert [d]Delete [p]Print [f]Find [q]quit
Select : p
***** Print *****
small: 40 large: 0
small: 20 large: 0
small: 60 large: 80
small: 10 large: 0
small: 30 large: 0
small: 50 large: 0
small: 90 large: 100
```

## □ 구현상의 오류 및 한계

본 2-3Tree 구현함에 있어 10단위로 10~100까지 10번 Insert하는 과정이 끝난 후 delete 70을 하였을 때 실습pdf 파일과 delete된 그림이 다르다.

이는 코드를 구현하면서 delete 실행단계 정책적 설정을 다르게 했기 때문에 나타난다. \* Insert는 본 코드와 똑같이 나타난다.

이는 Delete과정중 수많은 예외상황에 대한 처리를 할 수 있는 알고리즘이 여러개가 존재하며 이러한 과정중에 사용자가 설정한 한 알고리즘이 그대로 중복되어 적용되어 delete가 이뤄지기 때문이다.

또한 트리를 실습pdf처럼 찍는 모습은 구현방법을 찾아내지 못했으며 모색해볼 부분이다.

# 소감문 & 한계성

# 소감문 & 한계성

## □ 소감문

본 2-3Tree 알고리즘을 구현하면서 첫째로는 Insert를 Recursive로 구현하고자 할 때 Split이 일어나는 상황에 대한 함수를 별도로 정의하면서 복잡하고 중복되는 과정을 쉽게 나눌수 있었으며, 둘째로는 Delete를 구현함에 있어 각각 삭제되고자 하는 데이터가 존재하는 노드의 parent의 데이터 개수, parent child에 대한 데이터 개수)등을 if의 예외문 처리를 하여 각각 다른 동작을 만드는 과정에서 2-3Tree에 대한 이해가 심도있게 진행되었다.

## □ 한계성

Delete의 다양한 방식에 코드구현에 채택된 방식이 실습 pdf와 달라 Insert는 똑같지만 Delete의 방식은 항상 같지 않다는걸 알게되었으며, 일부 적용된 코드의 변화이기 때문에 Insert가 적은 값이 되었다면(=서로 알고리즘이 같은 방식의 제어문이 존재한다면) Delete되었을때도 같은 방식으로 나올수 있다는걸 알 수 있다. 실제로 일정값 이하일 때 Delete의 print가 똑같다.

또한 트리를 큐를 이용해서 깊이(level)별로 찍어내려 했으나 pdf처럼 깔끔하게 트리를 출력하지 못하여서 별도로 표기하였다.

# All Source Code

\* 코드가 길기 때문에 첨부된 프로젝트 파일 내 .c, .h 파일로 대체한다.