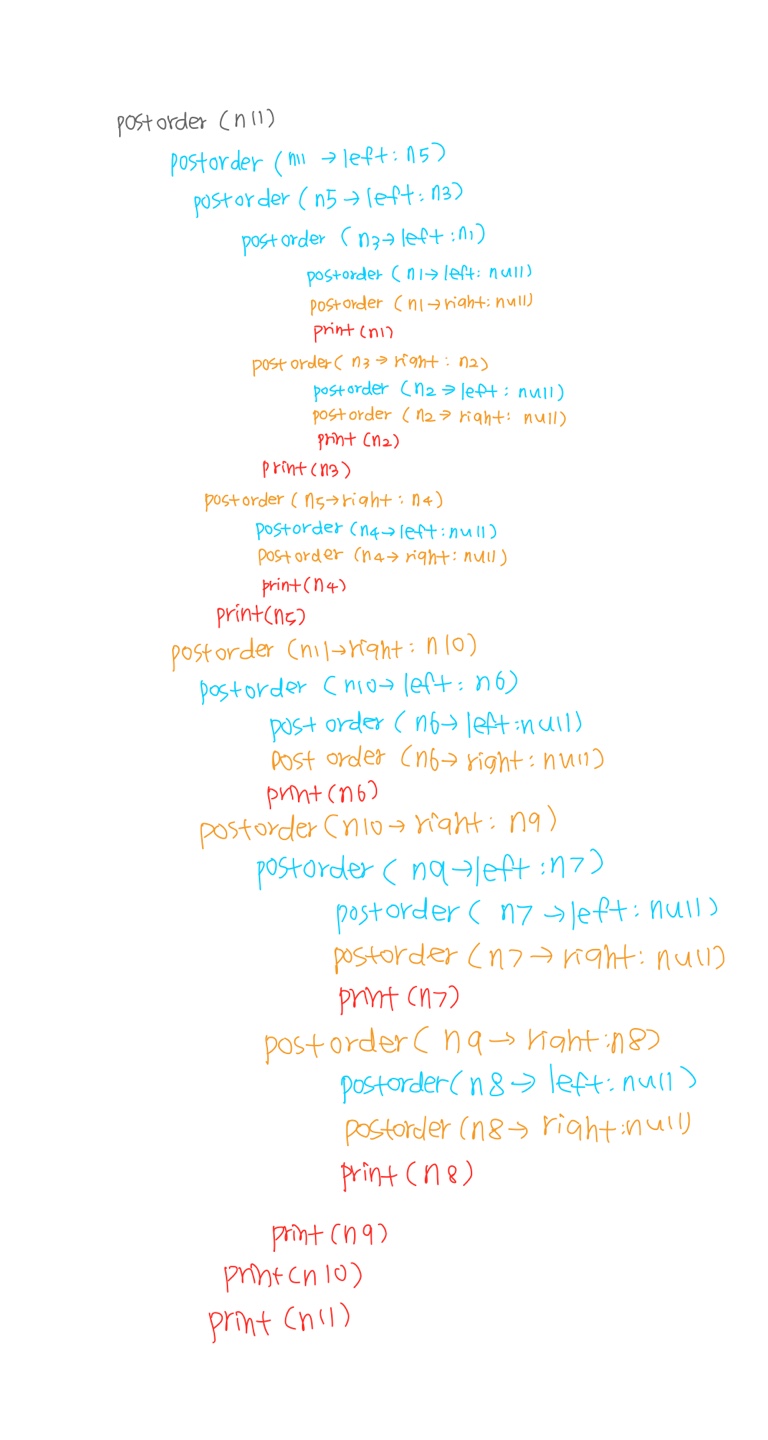
**Homework 5\_1 [order of postorder traversal]**

****

**Homework 5\_2 [Tree\_successor.cpp]**

**Variable analysis**

|  |  |
| --- | --- |
| **TreeNode** | |
| **type** | **name** |
| int | data |
| TreeNode\* | left |
| TreeNode\* | right |
| TreeNode\* | parent |

- data : a value that node has

- left : address of left child node

- right : address of right child node

- parent : address of parent node

**Function analysis**

1. Tree\_successor

: in order to find successor in inorder traversal, the existence of right subtree is important. Because the sequence of inorder traversal is left ->vertex ->right, which means the next node of current(vertex) is one of the nodes in right subtree. This Algorithm is divided into two cases on the basis of the right subtree of the input node being empty or not

1. If right subtree of *node*is not *NULL*, then succlies in right subtree. So, starting from top of right subtree, return leftmost node.
2. If right subtree of *node*is NULL, then *succ*is one of the ancestors.   
   Travel up using the parent pointer until you see a node which is left child of its parent. The parent of such a node is the *succ*. The reason why we allow only left child relation is that the parent node has already been passed if it has right child relation

TreeNode\* tree\_successor(TreeNode\* x)

{ //case 1: x’s right subtree is not null

if (x->right != NULL)

{

x = x->right; //set right sub tree

while (x->left != NULL) { //leftmost of right subtree

x = x->left; //move to left child

}

return x;

}

//case 2: x’s right subtree is null

TreeNode\* y = x->parent;

while (y != NULL and x == y->right) { // continue searching until the relation between child and parent is left.

x = y;

y = y->parent;

}

return y; //return parent

}

1. main

void main()

{

TreeNode\* exp = &n7;

TreeNode \*q = exp;

n1.parent = &n3;

n2.parent = &n3;

n3.parent = &n7;

n4.parent = &n6;

n5.parent = &n6;

n6.parent = &n7;

n7.parent = NULL;

while (q->left) q = q->left; // Go to the leftmost node

do

{

printf("%c\n", q->data); // Output data

// Call the successor

q = tree\_successor(q);

} while (q); // If not null

}

: To start from first node in inorder traversal, move to leftmost node. And ouput data and move to successor until q is null. By doing this, we can traverse all nodes in tree.

**[result]**



**Homework 5\_3 [Tree\_predecessor.cpp]**

**Variable analysis**

Same as Homework 5\_2 [Tree\_successor.cpp]

**Function analysis**

1. Tree\_predecessor

**Pseudo code**

텍스트이(가) 표시된 사진

자동 생성된 설명

: in order to find predecessor in inorder traversal, the existence of left subtree is important. Because the sequence of inorder traversal is left ->vertex ->right, which means the previous node of current(vertex) is one of the nodes in left subtree. Algorithm implemented is divided into two cases on the basis of the left subtree of the input node being empty or not

1. If left subtree of *node*is not *NULL*, then predlies in left subtree. So, starting from top of left subtree, return rightmost node.
2. If left subtree of *node*is NULL, then predis one of the ancestors.   
   Travel up using the parent pointer until you see a node which is right child of its parent. The parent of such a node is the pred. The reason why we allow only right child relation is that the parent node is one of the next nodes if it has left child relation

TreeNode\* tree\_predecessor(TreeNode\* x) {

if (x->left != NULL) //x's left subtree is not null

{

x = x->left; //set left sub tree

while (x->right != NULL) { //rightmost of left subtree

x = x->right; //move to right child

}

return x;

}

//x's left subtree is null

TreeNode\* y = x->parent;

while (y != NULL && x == y->left) {// continue searching until the relation between child and parent is right.

x = y;

y = y->parent;

}

return y;

}

1. main

void main() {

TreeNode\* q = exp;

n1.parent = &n3;

n2.parent = &n3;

n3.parent = &n7;

n4.parent = &n6;

n5.parent = &n6;

n6.parent = &n7;

n7.parent = NULL;

while (q->right) q = q->right; // Go to the rightmost node

do {

printf("%c\n", q->data); // Output data

// Call the predecessor

q = tree\_predecessor(q);

} while (q); // If not null

}

: To start from last node in inorder traversal, move to rightmost node. And ouput data and move to predecessor until q is null. By doing this, we can traverse all node in tree reversely.

**[result]**



**Homework 5\_4 [bst\_insertion\_deletion.cpp]**

**Variable analysis**

|  |  |
| --- | --- |
| **TreeNode** | |
| **type** | **name** |
| int | data |
| TreeNode\* | left |
| TreeNode\* | right |

- data : a value that node has

- left : address of left child node

- right : address of right child node

**Function analysis**

1. delete\_node

When we delete a node, three possibilities arise.

**Case 1) Node to be deleted is the** **leaf:** Simply remove from the tree.

텍스트, 장치, 측정기, 게이지이(가) 표시된 사진

자동 생성된 설명

Case 2) **Node to be deleted has only one child:** Copy the child to the node and delete the child

텍스트, 장치이(가) 표시된 사진

자동 생성된 설명

Case **3) Node to be deleted has two children:**Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used. *We can use predecessor in this time.*

텍스트, 장치, 게이지이(가) 표시된 사진

자동 생성된 설명

However, Unlike above two examples, we has no parent information in struct TreeNode this time. But, we need information about parent of given key.

void delete\_node(TreeNode \*\*root, int key)

: So to find parent of key, we set variable root as parameter. The reason why we use double pointer(\*\*) is the root can be changed while implementing this function.

TreeNode \*p, \*child, \*pred, \*pred\_p, \*t;

// search node t with key, p: t's parent

p = NULL;

t = \*root;

while (t != NULL && t->key != key) {

p = t;

t = (key < t->key) ? t->left : t->right;

}

: after this routine, t points to node with key and p points to t’s parent.

// Case 1: deletion of leaf node

if ((t->left == NULL) && (t->right == NULL)) { // no both subtree = leaf node

// If the parent node is not NULL,

if (p != NULL) {

if (p->left == t)

p->left = NULL;

else p->right = NULL;

}

// If the parent node is NULL, the node to be deleted is the root

else

\*root = NULL;

: There are two possibilities in case 1.

1. the parent node is not NULL : depending on relation between t and t’s parent (left child or right child), disconnect that connection.
2. the parent node is NULL : the node to be deleted is the root, which is only one left node in tree. So delete root and make root pointer point to nothing.

// Case 2 : deletion of node with single sub tree

else if ((t->left == NULL) || (t->right == NULL)) {

child = (t->left != NULL) ? t->left : t->right;

if (p != NULL) {

if (p->left == t)

p->left = child;

else p->right = child;

}

// If the parent node is NULL, the node to be deleted is the root

else

\*root = child;

}

: firstly, find child depending on which place it has child at. Then there are two possibilities.

1. the parent node is not NULL : connect parent with child depending on what relation parent and t(deleted node) has.
2. the parent node is NULL : the node to be deleted is the root. So just change root to t’s child.

// Case 3 : deletion of node with both sub tree

else {

// Find the predecessor at left subtree

pred\_p = t;

pred = t->left;

// Keep moving to the right and find the predecessor

while (pred->right != NULL) {

pred\_p = pred;

pred = pred->right;

}

// new connection between pred\_p and pred 'child

if (pred\_p->left == pred)

pred\_p->left = pred->left;

else

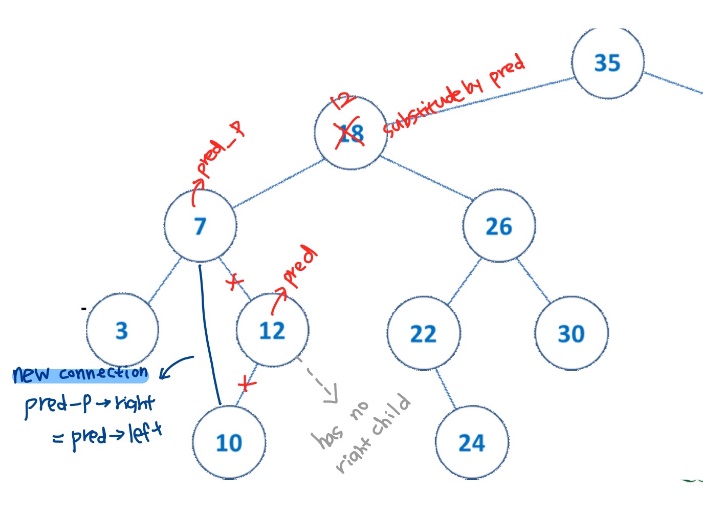
pred\_p->right = pred->left;

t->key = pred->key; //copy contents of pred

t = pred; //set node that have to be free().

}

: If the deleted node has both subtree, the problem becomes more complex. We have to find substitution of the deleted in this tree that doesn’t break the rules. But we can simply solve this problem by finding predecessor of the deleted. Because Predecessor is the largest value among nodes smaller than the deleted in subtree. A predecessor in inorder traversal can be found as the rightmost node of left subtree. (same logic as Homework 5\_3 [Tree\_predecessor.cpp] ).



After find predecessor, we just copy contents of the inorder predecessor to the deleted position and make new connection between parent of pred and pred’s child. The reason why using not `child` but `pred ->right` is pred has no right subtree since it is rightmost node.

[result]

1. key =18

텍스트, 모니터, 화면, 실외이(가) 표시된 사진

자동 생성된 설명

1. key =35

텍스트, 모니터, 화면, 텔레비전이(가) 표시된 사진

자동 생성된 설명

1. key =7

텍스트, 모니터, 텔레비전, 화면이(가) 표시된 사진

자동 생성된 설명