Assignment #6

20190741 김 지수

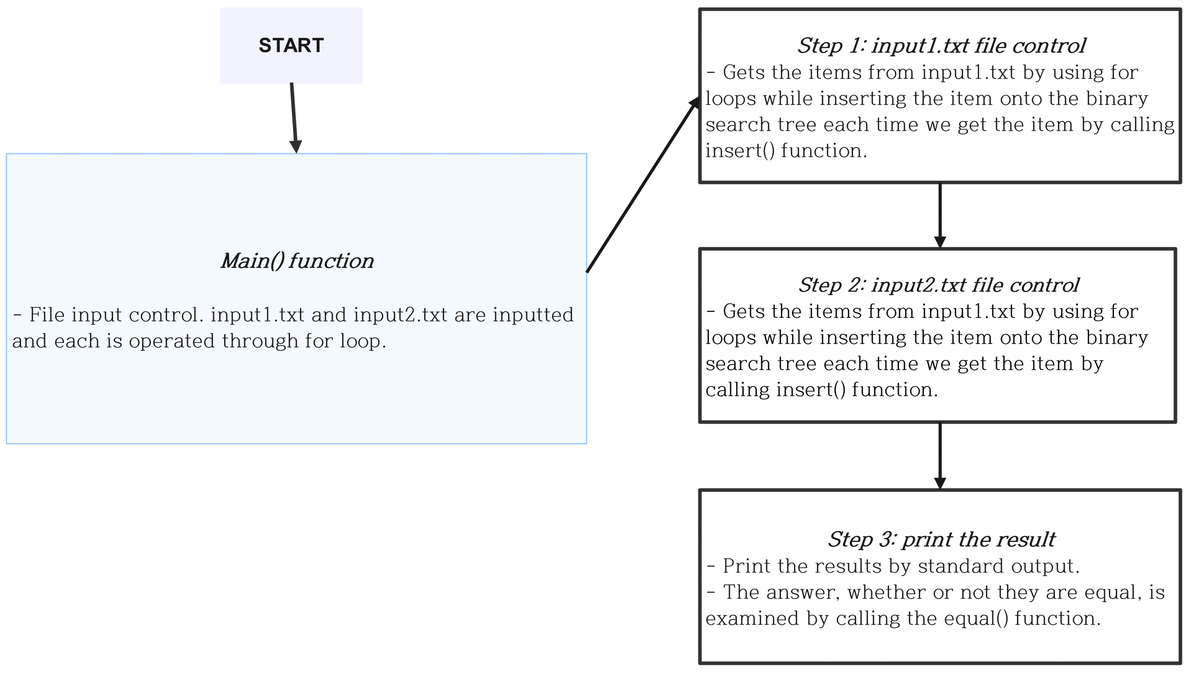
Q1) Comparing two binary search tree

note)

1. It is assumed in the HW6 pdf file that the input node's value is 1<= node <= 50 and it is given through the file input. It is assumed that this rule is followed by the text file maker, not the program. So, the program does not control about this problem.

Q1-1. Code Description with Flow chart

As the program starts, pre process begins. The program opens input1.txt and input2.txt files. Next, it gets the each of the input1 and input2 tree values from each file about the number of integers and their corresponding values. Each process for the input1 and input2, for loop is used to get the values while inserting its value lively by calling insert() function. Detailed explanations will be given in the Q1-2 part. Finally the program prints the result about whether the two trees are equal or not by calling equal() function. Note that, same node values for the same position means that they are equal. If the number of nodes are different, it has different position for the extra node(s). The flow chart of Q1 is as follows.


START
Main() function
- File input control. input1.txt and input2.txt are inputted and each is operated through for loop.
Step 1: input1.txt file control
- Gets the items from input1.txt by using for loops while inserting the item onto the binary search tree each time we get the item by calling insert() function.
Step 2: input2.txt file control
- Gets the items from input1.txt by using for loops while inserting the item onto the binary search tree each time we get the item by calling insert() function.
Step 3: print the result
- Print the results by standard output.
- The answer, whether or not they are equal, is examined by calling the equal() function.

<Flow chart, Q1>

Q1-2. Main Algorithms with Pseudo code

The explanation below is about the functions that are used in this program. These functions are used also in the Q3 far below this document.

Function Explanations
int equal(tree_pointer first, tree_pointer second); int equal() examines whether or not two given binary trees are equal. The mechanism of this function is the same as that of the textbook.
void insert(tree_pointer* node, int num); void insert() inserts the given num value in the tree. It calls the modified_search() function to check whether or not the given num value is already in the tree. Precise mechanism will be explained in below. void insert() gets the double pointer and it changes the tree directly by using this.
tree_pointer modified_search(tree_pointer node, int num); modified_search() is implemented in line with the order that is given in the course material. It searches the binary search tree for the given value num. If the tree is empty or if num is presented, it returns NULL. Otherwise, it returns the pointer to the last node of the tree that was encountered during the operation. It is important to avoid segmentation fault error by correctly considering the boundary cases. Cleverly, insert function gets tree_pointer *node, which is double pointer and puts *node when calling this function. It is not the address, but the tree itself. Thus the tree is copied and operated in this function and the result pointer to temp.

In this program, there is no such specific algorithm to explain. The equal(), insert(), modified_search() functions are already explained in the upper chart. Extra explanations about operations are given on the upper <Pseudocode, Q1> by using /*explanation*/.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
typedef struct node *tree_pointer;
typedef struct node {
            int data;
            tree_pointer left_child;
            tree_pointer right_child;
};

int equal(tree_pointer first, tree_pointer second);
void insert(tree_pointer *node, int num);
tree_pointer modified_search(tree_pointer node, int num);

int main() {
            FILE* input1, * input2;
            tree_pointer tree1, tree2;
            int i, num, item, ans;

            tree1 = NULL;
            tree2 = NULL;
            /*file input and error handling*/
            input1 = fopen("input1.txt", "r");
            if (input1 == NULL) {
                        printf("File open error. Program terminates.\n");
                        exit(1);
            }
```

```
        /*receive input from the file and insert onto the tree1*/
        fscanf(input1, "%d", &num);
        for (i = 0; i < num; i++) {
                fscanf(input1, "%d", &item);
                insert(&tree1, item);
        }
        /*file input and error handling*/
        input2 = fopen("input2.txt", "r");
        if (input2 == NULL) {
                printf("File open error. Program terminates.\n");
                exit(1);
        }
         /*receive input from the file and insert onto the tree2*/
        fscanf(input2, "%d", &num);
        for (i = 0; i < num; i++) {
                fscanf(input2, "%d", &item);
                insert(&tree2, item);
        }

        ans = equal(tree1, tree2);
        if (ans) printf("YES\n");
        else printf("NO\n");

        fclose(input1);
        fclose(input2);
        return 0;
}

int equal(tree_pointer first, tree_pointer second) {
        return ((!first && !second) || (first && second && (first->data == second->data) &&
                equal(first->left_child, second->left_child) && equal(first->right_child, second->right_child)));
}
void insert(tree_pointer *node, int num) {
        /* if num is in the tree pointed at by nodet do nothing
        otherwise add a new node with data = num */

        tree_pointer ptr, temp = modified_search(*node, num);
        if (temp || !(*node)) {
                ptr = (tree_pointer)malloc(sizeof(struct node));
                if (!ptr) {
                        fprintf(stderr, "The memory is full. Program terminates.\n");
                        exit(1);
                }
                ptr->data = num;
                ptr->left_child = NULL;
                ptr->right_child = NULL;
                if (*node) {
                        if (num < temp->data)
                                temp->left_child = ptr;
                        else temp->right_child = ptr;
                }
                else *node = ptr;
        }
}
```

```
tree_pointer modified_search(tree_pointer node, int num) {
        if (node == NULL || node->data == num) {
                return NULL;
        }
        tree_pointer current = node;
        tree_pointer last_encountered = NULL;

        while (current != NULL) {
                last_encountered = current;
                if (num < current->data) {
                        current = current->left_child;
                }
                else if (num > current->data) {
                        current = current->right_child;
                }
                else {
                        /* input value is already in the tree.*/
                        return NULL;
                }
        }
        return last_encountered;
}
```

<Pseudocode, Q1>

Q1-3. Test cases simulated

input1.txt	input2.txt	output
3 10 12 9	3 10 12 9	YES
4 8 9 10 4	4 8 10 9 4	NO
8 1 2 9 8 49 50 30	9 1 2 9 8 49 50 30 100	NO
0	0	YES
1 20	1 20	YES

As simulating the many test cases like upper table, the program could be more reliable by checking the corner cases. As a result, there is no program error or incorrect result found for several test cases. **input1.txt 0 and input2.txt 0 is a corner case.** Two empty trees could be defined 'equal' because of the a binary search tree is a binary tree.

Q2) Insertion, Deletion of Max Heap

Note:

1) The program uses std input/output with some requirements. It is assumed that there is no exceptional inputs that do not follow the given input form as HW 6 PDF file said.

2) Note that input key value should be in the boundary of integer variable type in C. It is ass umed that the input number is given in this manner. +, 0, - do not affect the program result.

Program requirements:

* Commands that are required for the program is as follows.

i k – insert node in the heap whose key value is k.

d – delete the node that has the biggest key value. (Found at root, constant time deletion)

q – program termination command

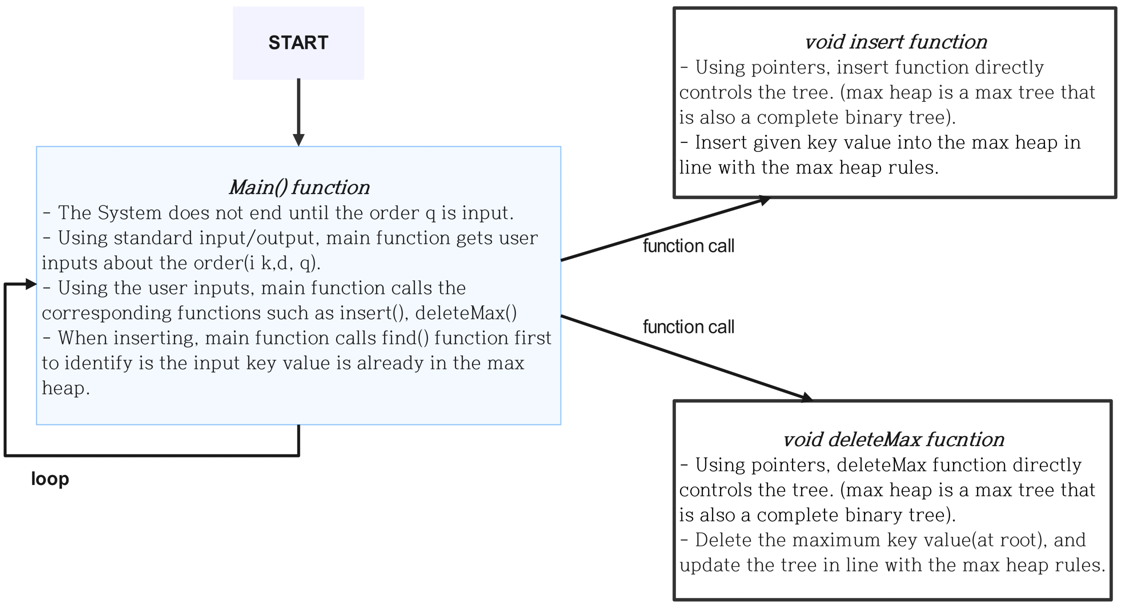
* Output results that are required for the inputs are as follows.

If i k successfully operated, output Insert k, otherwise(the case that the key value is already e xists) output Exist number.

If d successfully operated, output Delete K, otherwise(the case that heap is empty) output The heap is empty.

Q2-1. Code Description with Flow chart

The flow chart of the Q2 is as follows. When the program starts, the main() function gets fil e input about each element. The main function itself is loop that does not end until the order q is input. For each line of command main function calls the needed function(s). Each detaile d information about the program will be explained at Q2-2 with pseudo code.


START
Main() function
- The System does not end until the order q is input.
- Using standard input/output, main function gets user inputs about the order(i k,d, q).
- Using the user inputs, main function calls the corresponding functions such as insert(), deleteMax()
- When inserting, main function calls find() function first to identify is the input key value is already in the max heap.
loop
function call
function call
void insert function
- Using pointers, insert function directly controls the tree. (max heap is a max tree that is also a complete binary tree).
- Insert given key value into the max heap in line with the max heap rules.
void deleteMax fucntion
- Using pointers, deleteMax function directly controls the tree. (max heap is a max tree that is also a complete binary tree).
- Delete the maximum key value(at root), and update the tree in line with the max heap rules.

<Flow chart, Q2>

Q2-2. Main Algorithms with Pseudo code

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

typedef struct node* treePointer;
typedef struct node {
    int key;
    treePointer parent;
    treePointer leftChild, rightChild;
};

void insert(treePointer* root, int key);
void deleteMax(treePointer* root);
void find(treePointer root, int num, int* identify);

int main() {
    char order;
    int num;
    treePointer root = NULL;
    /*Standard input, output representation of the program. find function searches if the num value
is already in the root. If there is, the identifier will be change to 0 to identify whether or not to
insert the input number in the max heap. Searching time complexity of the find is number of
elements in the max heap.
    */

    while (1) {
        scanf(" %c", &order);
        if (order == 'q') {
            break;
        }
        else if (order == 'i') {
            scanf("%d", &num);
            int identifier = 1;
            find(root, num, &identifier);
            if (identifier) {
                insert(&root, num);
                printf("Insert %d\n", num);
            }
            else {
                printf("Exist number\n");
            }
        }
        else if (order == 'd') {
            if (root != NULL) {
                deleteMax(&root);
            }
            else {
                printf("The heap is empty\n");
            }
        }
    }
    return 0;
}

void insert(treePointer* root, int key) {
    treePointer newNode = (treePointer)malloc(sizeof(struct node));
    /* error handling */
    if (!newNode) {
        fprintf(stderr, "Memory allocation error. Program terminates.\n");
        exit(1);
    }
    newNode->key = key;
    newNode->parent = NULL;
    newNode->leftChild = NULL;
    newNode->rightChild = NULL;
    if (*root == NULL) {
        *root = newNode;
        return;
    }

    treePointer currentNode = *root;
    while (1) {
    /*max heap is a complete binary tree. Thus, leftChild is searched first. */
        if (currentNode->leftChild == NULL) {
            currentNode->leftChild = newNode;
            newNode->parent = currentNode;
            break;
        }
        else if (currentNode->rightChild == NULL) {
            currentNode->rightChild = newNode;
            newNode->parent = currentNode;
            break;
        }
        else {
 /*each node must have a value greater than or equal to its children. To maintain the Max Heap
property, the smaller right child should have a larger value than the current node.*/
            if (currentNode->leftChild->key >= currentNode->rightChild->key) {
                currentNode = currentNode->rightChild;
            }
            else {
                currentNode = currentNode->leftChild;
            }
        }
    }
    /*Max Heap reordering*/
    while (newNode->parent != NULL && newNode->key > newNode->parent->key) {
        int tempKey = newNode->key;
        newNode->key = newNode->parent->key;
        newNode->parent->key = tempKey;
        newNode = newNode->parent;
    }
}

void deleteMax(treePointer* root) {
    if (*root == NULL) {
        printf("The heap is empty\n");
        return;
    }
    int deletedValue = (*root)->key;
    treePointer lastNode = *root;

    /*exceptinal error handling. if key value of the root only exists.*/
    if (lastNode->leftChild == NULL && lastNode->rightChild == NULL) {
        free(lastNode);
        *root = NULL;
        printf("Delete %d\n", deletedValue);
        return;
    }

    /*search and update following max heap rule until the lastNode is leaf node. */
    while (lastNode->leftChild != NULL || lastNode->rightChild != NULL) {
        if (lastNode->rightChild != NULL) {
            if (lastNode->leftChild != NULL && lastNode->leftChild->key >= lastNode->rightChild->key)
{
                lastNode->key = lastNode->leftChild->key;
                lastNode = lastNode->leftChild;
            }
            else {
                lastNode->key = lastNode->rightChild->key;
                lastNode = lastNode->rightChild;
            }
        }
        else { /* lastNode->rightChild is NULL */
            lastNode->key = lastNode->leftChild->key;
            lastNode = lastNode->leftChild;
        }
    }
    /*update parent and child relationship and finally delete lastNode*/
    if (lastNode->parent != NULL) {
        if (lastNode->parent->leftChild == lastNode) {
            lastNode->parent->leftChild = NULL;
        }
        else {
            lastNode->parent->rightChild = NULL;
        }
        free(lastNode);
    }
    else {
        free(lastNode);
        *root = NULL;
    }
    printf("Delete %d\n", deletedValue);
}

void find(treePointer root, int num, int *identify) {
    /*if the input key value is already in the heap, change identify to 0*/
    /*find function just searches with recursive mechanism*/
    if (root == NULL) {
        return;
    }
    if (root->key == num) {
        *identify = 0;
    }
    find(root->leftChild, num, identify);
    find(root->rightChild, num, identify);
}
```

<Pseudocode, Q2>

```
typedef struct node* treePointer;
typedef struct node {
    int key;
    treePointer parent;
    treePointer leftChild, rightChild;
};
```

The upper code is the defined data structures used in this program. As the question required to use the given data structure to implement, the program uses the exact same data structure that is given.

As the program has been made slightly difficult since the student wanted to design by using only three functions except the main function, the detailed explanation about the code is given at the upper Pseudocode, Q2.

Specifically note about the deleteMax function. Because it is implemented slightly differently from the array using max heap deletion algorithm in our textbook because of the linked lists data structure. The deleteMax function in the code performs the task of deleting the maximum value from the root of a max heap. With each invocation of the function, the maximum value at the root is deleted. Also at the next invocation, the next largest value should be deleted.

In order to implement this function, within the loop, the lastNode is moved to the last node of the heap by traversing to the child node with the larger value. This process maintains the max heap property by comparing the parent and child nodes based on their values.

Consequently, after the loop finishes executing, lastNode becomes the last node of the heap, and it is then deleted to adjust the heap's size. Thus, the deleteMax function correctly performs the task of deleting the maximum value from the heap's root while maintaining the max heap rules.

<Test results, Q2>

input	output
i 10	Insert 10
i 5	Insert 5
i 7	Insert 7
i 3	Insert 3
i 8	Insert 8
i 12	Insert 12
i 6	Insert 6
i 6	Exist number
i 5	Exist number
i 10	Exist number
i 10	Exist number
d	Delete 12
i 9	Insert 9
d	Delete 10
d	Delete 9
d	Delete 8
d	Delete 7
d	Delete 6
d	Delete 5
d	Delete 3
d	The heap is empty
q	
i 12431	Insert 12431
i 12431	Exist number
i 0	Insert 0
i -123	Insert -123
d	Delete 12431
i 12	Insert 12
d	Delete 12
d	Delete 0
d	Delete -123
d	The heap is empty
q	
i 4	Insert 4
i 4	Exist number
i 5	Insert 5
d	Delete 5
d	Delete 4
d	The heap is empty
i 3	Insert 3
q	

As simulating the many test cases like upper table, the program could be more reliable by checking the corner cases. As a result, there is no program error or incorrect result found for several test cases.

Q3) Binary search tree insertion/deletion/output

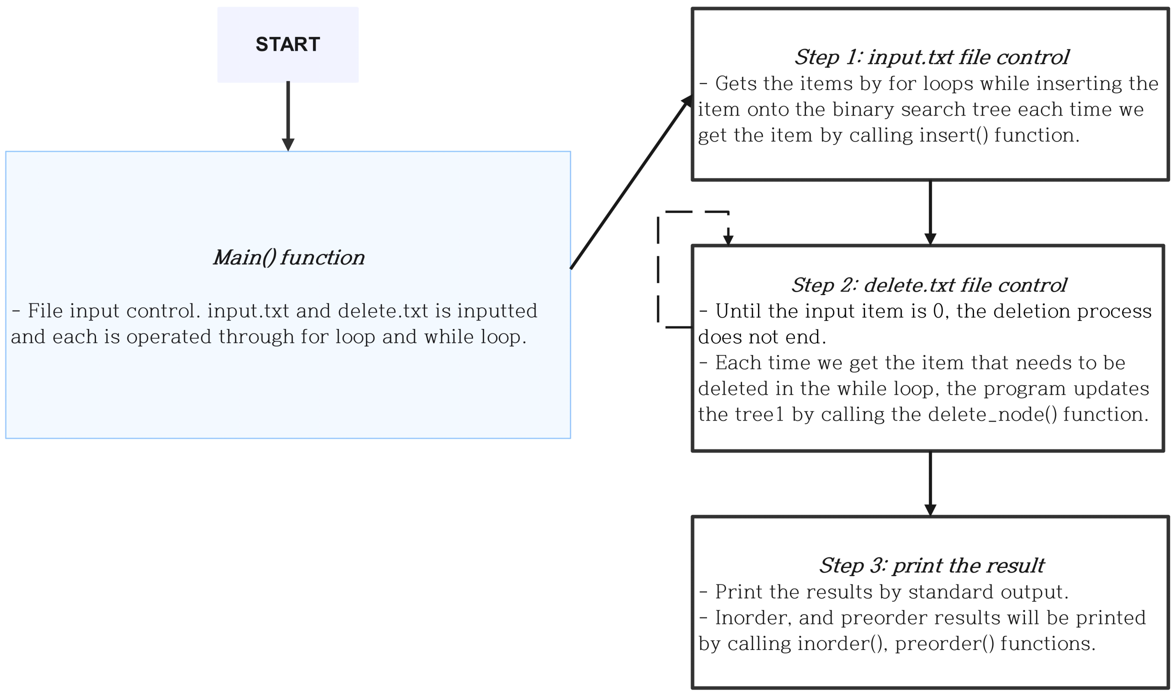
notes)

1. It is assumed in the HW6 pdf file that the input node's value is 1<= node <= 50 and it is gi ven in this order. So, the program does not control about this problem.
2. The program follows the delete rule that is given by HW6 pdf file.
3. The key value of the delete node in the delete.txt file should be exist in the input.txt file. It is assumed that this rule is followed by the text file maker, not the program.

Q3-1. Code Description with Flow chart

The flow chart of the Q3 is as follows. When the program starts, main function gets inputs by file input. When the input.txt file is read, the program simultaneously inserts the value int o the tree by using for loop. Next, when the delete.txt file is read, the program simultaneousl y deletes the value at the tree by using while loop until the 0 is given in the file. Finally, the program prints the inorder and preorder status or the final binary search tree.

Specific information about the called functions in main will be introduced at the below Q3-2 part and the flow chart is as follows.


START
Main() function
- File input control. input.txt and delete.txt is inputted and each is operated through for loop and while loop.
Step 1: input.txt file control
- Gets the items by for loops while inserting the item onto the binary search tree each time we get the item by calling insert() function.
Step 2: delete.txt file control
- Until the input item is 0, the deletion process does not end.
- Each time we get the item that needs to be deleted in the while loop, the program updates the tree1 by calling the delete_node() function.
Step 3: print the result
- Print the results by standard output.
- Inorder, and preorder results will be printed by calling inorder(), preorder() functions.

<Flow chart, Q3>

Q3-2. Main Algorithms with Pseudo code

In this program, there are many functions that is purposely made for structured programmi ng. So, in this part, each function is explained on the next page.

Function Explanations
int equal(tree_pointer first, tree_pointer second); int equal() examines whether or not two given binary trees are equal. The mechanism of this function is the same as that of the textbook.
void insert(tree_pointer* node, int num); void insert() inserts the given num value in the tree. It calls the modified_search() function to check whether or not the given num value is already in the tree. Precise mechanism will be explained in below. void insert() gets the double pointer and it changes the tree directly by using this.
tree_pointer modified_search(tree_pointer node, int num); modified_search() is implemented in line with the order that is given in the course material. It searches the binary search tree for the given value num. If the tree is empty or if num is presented, it returns NULL. Otherwise, it returns the pointer to the last node of the tree that was encountered during the operation. It is important to avoid segmentation fault error by correctly considering the boundary cases. Cleverly, insert function gets tree_pointer *node, which is double pointer and puts *node when calling this function. It is not the address, but the tree itself. Thus the tree is copied and operated in this function and the result pointer to temp.
tree_pointer delete_node(tree_pointer root, int num); delete_node() is implemented with the idea that is given in the course material. if the deletion node is found, there are 4 cases to consider. Deletion of a leaf node, of a non-leaf node with single child(left_child, right_child each), and of a non-leaf node with two children.
tree_pointer find_max(tree_pointer node); find_max() finds the node in position that has maximum data value. As it is a binary search tree its implementation is simple.
void inorder(tree_pointer root); void inorder() gets the root of the binary search tree and prints the data in accordance with the inorder rule.
void preorder(tree_pointer root); void preorder() gets the root of the binary search tree and prints the data in accordance with the preorder rule.

As a result, used functions are explained. Next, the explanations about the meaningful operations to utilize these functions are given with <Pseudocode, Q3> using /*explanation*/ form.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

typedef struct node* tree_pointer;
typedef struct node {
            int data;
            tree_pointer left_child;
            tree_pointer right_child;
};
int equal(tree_pointer first, tree_pointer second);
void insert(tree_pointer* node, int num);
tree_pointer modified_search(tree_pointer node, int num);
tree_pointer delete_node(tree_pointer root, int num);
tree_pointer find_max(tree_pointer node);
void inorder(tree_pointer root);
void preorder(tree_pointer root);

int main() {
	FILE* input, * delete;
	tree_pointer tree1;
	int i, num, item;

	tree1 = NULL;

	input = fopen("input.txt", "r");
	if (input == NULL) {
		printf("File open error. Program terminates.\n");
		exit(1);
	}

	fscanf(input, "%d", &num);
	for (i = 0; i < num; i++) {
		fscanf(input, "%d", &item);
		insert(&tree1, item);
	}

	delete = fopen("delete.txt", "r");
	if (delete == NULL) {
		printf("File open error. Program terminates.\n");
		exit(1);
	}

	fscanf(delete, "%d", &item);

	while (1) {
		/*get input until the input number is 0.*/
		if (item == 0) break;
		tree1 = delete_node(tree1, item);
		fscanf(delete, "%d", &item);
	}

	printf("inorder : "); inorder(tree1); printf("\n");
	printf("preorder : "); preorder(tree1); printf("\n");

	fclose(input);
	fclose(delete);
	return 0;
}

int equal(tree_pointer first, tree_pointer second) {
	return ((!first && !second) || (first && second && (first->data == second->data) &&
		equal(first->left_child, second->left_child) && equal(first->right_child, second->right_child)));
}
void insert(tree_pointer* node, int num) {
	/* if num is in the tree pointed at by nodet do nothing
	otherwise add a new node with data = num */

```

```
        tree_pointer ptr, temp = modified_search(*node, num);
        if (temp || !(*node)) {
                ptr = (tree_pointer)malloc(sizeof(struct node));
                if (!ptr) {
                        fprintf(stderr, "The memory is full. Program terminates.\n");
                        exit(1);
                }
                ptr->data = num;
                ptr->left_child = NULL;
                ptr->right_child = NULL;
                if (*node) {
                        if (num < temp->data)
                                temp->left_child = ptr;
                        else temp->right_child = ptr;
                }
                else *node = ptr;
        }
}

tree_pointer modified_search(tree_pointer node, int num) {
        if (node == NULL || node->data == num) {
                return NULL;
        }
        tree_pointer current = node;
        tree_pointer last_encountered = NULL;

        while (current != NULL) {
                last_encountered = current;

                if (num < current->data) {
                        current = current->left_child;
                }
                else if (num > current->data) {
                        current = current->right_child;
                }
                else {
                        /* input value is already in the tree.*/
                        return NULL;
                }
        }
        return last_encountered;
}

tree_pointer delete_node(tree_pointer root, int num) {
        /*Assumed that input num is in the given tree*/
        if (root == NULL) {
                return root;
        }

        if (num < root->data) {
                root->left_child = delete_node(root->left_child, num);
        }
        else if (num > root->data) {
                root->right_child = delete_node(root->right_child, num);
        }
        else {
                /* Deletion node is found */
                if (root->left_child == NULL && root->right_child == NULL) {
                        return NULL;
                }
                else if (root->left_child == NULL) {
                        tree_pointer temp = root->right_child;
                        return temp;
                }
                else if (root->right_child == NULL) {
                        tree_pointer temp = root->left_child;
                        return temp;
                }
                else {
                        tree_pointer temp = find_max(root->left_child);
                        root->data = temp ->data;
                        root->left_child = delete_node(root->left_child, temp->data);
                }
        }
        return root;
}

tree_pointer find_max(tree_pointer node) {
        tree_pointer current = node;

        while (current && current->right_child != NULL) {
                current = current->right_child;
        }
        return current;
}

void inorder(tree_pointer root) {
        if (root != NULL) {
                inorder(root->left_child);
                printf("%d ", root->data);
                inorder(root->right_child);
        }
}

void preorder(tree_pointer root) {
        if (root != NULL) {
                printf("%d ", root->data);
                preorder(root->left_child);
                preorder(root->right_child);
        }
}
```

<Pseudocode, Q3>

Q3-3. Test cases simulated

As simulating the test cases could make the program more reliable by finding the corner cas es, cases are tested. Each cases, I made a binary search tree and erase by my hand using pe n and pencil. After that process, I checked the result of the program. As a result, there was no program error or incorrect result found for several test cases.

input.txt	delete.txt	output
13 8 4 12 2 6 10 14 1 3 5 7 9 11	10 4 2 9 14 12 0	inorder : 1 3 5 6 7 8 11 preorder : 8 3 1 6 5 7 11
	10 0	inorder : 1 2 3 4 5 6 7 8 9 11 12 14 preorder : 8 4 2 1 3 6 5 7 12 9 11 14
	10 4 0	inorder : 1 2 3 5 6 7 8 9 11 12 14 preorder : 8 3 2 1 6 5 7 12 9 11 14
	10 4 2 0	inorder : 1 3 5 6 7 8 9 11 12 14 preorder : 8 3 1 6 5 7 12 9 11 14
	10 4 2 9 0	inorder : 1 3 5 6 7 8 11 12 14 preorder : 8 3 1 6 5 7 12 11 14
	10 4 2 9 14 0	inorder : 1 3 5 6 7 8 11 12 preorder : 8 3 1 6 5 7 12 11
	10 4 2 9 14 12 13 8 6 1 3 5 7 0	inorder : 11 preorder : 11
10 3 9 8 2 5 10 7 1 4 6	1 7 9 3 0	inorder : 2 4 5 6 8 10 preorder : 2 8 5 4 6 10
10 3	10 0	inorder : 3 preorder : 3
10 3	10 3 0	inorder : preorder :