HW6 REPORT

20200901 이효주

<HW6 - 1>

1) Problem Statement

The problem of determining whether two given binary search trees have the same node value in the same location. The input is given two text files, each of which lists the integers that make up the binary navigation tree, one per line. The output should output "YES" if the node values of the two binary navigation trees have the same value in the same location, or "NO" if not.

Input:
input1.txt: The number of nodes and positive integers in the first tree are given one per line.
input2.txt: The number of nodes and positive integers in the second tree are given one per line.

Output:
Outputs "YES" if both trees have the same node value in the same location.
Outputs "NO" if the two trees do not have the same node value in the same location.

Constraints:
Input is received as a file, and output is output as standard output (stdout).
The node has a value greater than or equal to 1 and less than or equal to 50.
Problem Solving Approach

1) Read the input1.txt and input2.txt files to create their respective binary navigation trees.
2) Make sure that the two generated binary navigation trees are traversed simultaneously and have the same node's values in the same location.
3) Outputs "YES" if it has the same node value in the same location, or "NO" if it does not.

2) Defining a Binary Search Tree Structure

```
typedef struct Node {
    int value; // Integer value stored by each node in the binary navigation tree.
```

```
    struct Node* left; // Pointer pointing to the left child node of the current node. Indicates the subtree on the left.
    struct Node* right; // Pointer pointing to the right child node of the current node. Indicates the subtree to the right.
} Node;
```

The code above is the part that defines the structure Node that represents the node in the binary navigation tree. Each node consists of a value member that stores the value, a left pointer to the left child node, and a right pointer to the right child node.

3) New node creation function (create)

```
/* create new node */
Node* create(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));

    newNode->value = value;
    newNode->left = NULL;
    newNode->right = NULL;

    return newNode;
}
```

The create function is responsible for creating a new binary discovery tree node initialized with a given value. The function dynamically allocates memory to create a new node, initializes the member variable, and returns a pointer to that node.

4) Binary search tree generation function (bst)

```
    /* create binary search tree from file */
    Node* bst(const char* file_name) {

        FILE* file = fopen(file_name, "r");
        if (file == NULL) {
            printf("Failed to open the file: %s₩n", file_name);
            return NULL;
        }

        int n;
```

```
        fscanf(file, "%d", &n);
        Node* root = NULL;

        for (int i = 0; i < n; i++) {
            int value;
            fscanf(file, "%d", &value);
            root = insertNode(root, value);
        }

        fclose(file);
        return root;
    }
```

The bst function uses integers read from a given file to create a binary search tree. The function accepts the file name file_name as a factor.

Open the file in read mode. If the file fails to open, it outputs an error message and returns NULL.

Reads the integer n from the file. n is the number of integer values to insert into the binary navigation tree.

Initialize the root. Set to NULL because there are no nodes in the beginning.

Repeat n times to read integer values from the file and insert nodes into the binary navigation tree.

int value; fscanf(file, "%d", &value);: Read the following integer values from the file.
root = insertNode(root, value);: insertNode function is called to insert a new node into the binary navigation tree. The insertNode function inserts the node in the appropriate location and returns the new root node.
Closes the file and returns root, the root node of the generated binary navigation tree.

5) Insert node function in binary search tree (insertNode)

```
/* insert node in binary search tree */
Node* insertNode(Node* root, int value) {
    if (root == NULL) {
        return create(value);
    }
```

```
    if (value < root->value) {
        root->left = insertNode(root->left, value);
    }
    else if (value > root->value) {
        root->right = insertNode(root->right, value);
    }

    return root;
}
```

The insertNode function is responsible for inserting a new node with a given value into the binary navigation tree root. The function performs the insertion process in a recursive manner.

If root is NULL, create a new node to set the value and return it. This is when you insert a new node as root when the tree is empty.

If the value is less than the value of the current node, perform a recursive call to the left child subtree. Root->left is assigned a return value of insertNode (root->left, value).

If the value is greater than the value of the current node, perform a recursive call to the right child subtree. Root->right is assigned a return value of insertNode (root->right, value).

Returns root finally to return the inserted tree.

6) Binary search tree comparison function (compare)

```
/* compare two node in position & value */
int compare(Node* node1, Node* node2) {
    if (node1 == NULL && node2 == NULL) {
        return 1;
    }

    if (node1 == NULL || node2 == NULL) {
        return 0;
    }

    if (node1->value != node2->value) {
```

```
        return 0;
    }

    int leftResult = compare(node1->left, node2->left);
    int rightResult = compare(node1->right, node2->right);

    return (leftResult && rightResult);
}
```

The compare function compares the two binary search trees to determine whether they have the same node value in the same location. The function takes two node pointers node1 and node2 as factors.

If both nodes are NULL, determine that they have the same value in the same location and return 1.

If only one of the two nodes is NULL, it determines that the values are different in the same location and returns 0.

If two nodes have different values, determine that they have different values in the same location and return 0.

Recursively call the compare function for the left and right subtrees of node1 and node2 to see if they have the same node's value in the same location.

If the results of both the left and right subtrees are true, determine that they have the same value in the same position and return 1. Otherwise, returns 0.

7) Check function (checkSameNodes)

```
/* compare and check */
void checkSameNodes(const char* input_file1, const char* input_file2) {

    Node* root1 = bst(input_file1);
    Node* root2 = bst(input_file2);

    if (compare(root1, root2)) {
        printf("YES₩n");
```

```
    }
    else {
        printf("NO₩n");
    }
}
```

The checkSameNodes function creates a binary navigation tree from two input files and compares the two generated trees to output whether they have the same node value in the same location. The function takes two string pointers input_file1 and input_file2 as factors.

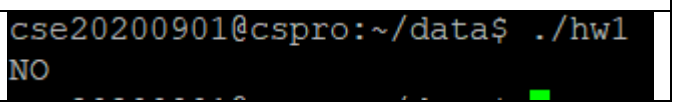
Use input_file1 and input_file2 to call the bst function to create binary search trees root1 and root2 from each file.

Call the compare function to compare root1 and root2.

Outputs the result according to the return value of the compare function. Outputs "YES" if both trees have the same node value in the same location, and "NO" if not.

This function allows you to receive two binary navigation trees and compare them to see if they have the same node's values in the same location.

8) Example

Input1.txt	Input2.txt
4 8 9 10 4	4 8 10 9 4
output	

<HW6 - 2>

1) Problem Statement

The problem given is to write insertion and deletion functions for Max Heap. Assume that each node has a data value and parent node, left child node, and right child node. The input is given commands in the form "ik", "d", and "q", and appropriate actions must be performed according to each command. The insertion command inserts a node with a key value of k into the heap, and the deletion command deletes the node with the largest key value in the heap. The "q" command exits the program. Programs should be implemented using Linked Presentation.

The following approaches were used to resolve the problem.

Implement Max Heap using Linked Representation. Each node is defined as a structure with a data value and parent node, left child node, and right child node.

Creates an insertion function. Create a new node, locate the location to be inserted into the current heap, and attach the new node to that location. Outputs "Insert k" if the insertion is successful, or "Exist number" if the key value already exists.

Create a deletion function. Locate and delete the node with the largest key value in the current heap. Outputs the deleted key value, and "The heap is empty" if heap is empty.

The input is received according to the given input format, and functions that fit each command are called and operated. If you receive input repeatedly and the "q" command is entered, exit the program.

2) Data structure

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

treePointer root = NULL;
```

The Node structure represents each node in the Max Heap.
The key represents the value of the node, and parent, leftChild, and rightChild point to the parent, left child, and right child nodes of that node.
A treePointer is a pointer type to a Node structure that points to each node.

3) Insertion function

```
void insertion(int key) {
    treePointer newNode = (treePointer)malloc(sizeof(Node));
    newNode->key = key;
    newNode->parent = NULL;
    newNode->leftChild = NULL;
    newNode->rightChild = NULL;

    if (root == NULL) {
        root = newNode;
        printf("Insert %d₩n", key);
        return;
    }

    treePointer currentNode = root;
    while (currentNode != NULL) {
        if (currentNode->key == key) {
            printf("Exist number₩n");
            free(newNode);
            return;
        }

        if (key > currentNode->key) {
            if (currentNode->rightChild != NULL) {
                currentNode = currentNode->rightChild;
            }
            else {
                currentNode->rightChild = newNode;
                newNode->parent = currentNode;
                printf("Insert %d₩n", key);
                return;
            }
```

```
            }
            else {
                if (currentNode->leftChild != NULL) {
                    currentNode = currentNode->leftChild;
                }
                else {
                    currentNode->leftChild = newNode;
                    newNode->parent = currentNode;
                    printf("Insert %d₩n", key);
                    return;
                }
            }
        }
    }
```

create a node with a new key value. This node will be inserted into the Max Heap.

First, make sure that heap is empty. If empty, set the new node to root, output "Insertk" and exit the function.

If Heap is not empty, create a variable called currentNode and initialize it to root.
While Move the currentNode through the loop to find the position to insert.
Compare the key value of the currentNode with the key value you want to insert.
If a node already exists with the same key value as the current key value, output "Exist number", release the new node from memory, and exit the function.

If the key value you want to insert is greater than the current key value, go to the right child node.
If the key value you want to insert is less than the current key value, go to the left child node.
When the appropriate location is found, connect the new node in the direction it connects to the currentNode.

If it is greater than the current key value, connect to the right child.
If it is less than the current key value, connect with the left child.
Output "Insertk" and exit the function.

In this way, the insertion function inserts a given key value into the Max Heap.

4) Deletion function

```
void deletion() {
    if (root == NULL) {
        printf("The heap is empty₩n");
        return;
    }

    treePointer currentNode = root;

    // Find the rightmost leaf node
    while (currentNode->rightChild != NULL) {
        currentNode = currentNode->rightChild;
    }

    int key = currentNode->key;

    if (currentNode->parent != NULL) {
        treePointer parent = currentNode->parent;

        // Remove the node
        if (parent->rightChild == currentNode) {
            parent->rightChild = NULL;
        }
        else {
            parent->leftChild = NULL;
        }

        free(currentNode);
    }
    else {
        // currentNode is the root
        free(currentNode);
        root = NULL;
    }

    printf("Delete %d₩n", key);
}
```

First, make sure that the Heap is empty. If empty, output "The heap is empty" and exit the function.

If Heap is not empty, create a variable called currentNode and initialize it to root.

find the rightmost leaf node through the while loop. (Leaf nodes are nodes with rightChild as NULL)
Save the key value of the node that currentNode points to in the variable key.
If the parent node (parent) of the currentNode is not NULL, verify that the currentNode is the right or left child of the parent.
If you are a right child, delete the currentNode by setting the parent's rightChild to NULL.
If it is a left child, delete the currentNode by setting the parent's leftChild to NULL.
Release currentNode from memory.
If currentNode is root, that is, if the deleted node is root, set root to NULL.
Output "Delete" and exit the function.

In this way, the deletion function deletes the node with the largest key value in the Max Heap. Outputs the deleted key value and removes it from the Max Heap.

5) Main function

```
int main() {
	char command;
	int key;

	// while loop
	while (1) {
		scanf("%c", &command);

		// if input is 'q', the program will be quit
		if (command == 'q') {
			break;
		}

		if (command == 'i') {
			scanf("%d", &key);
			insertion(key);
		}
		else if (command == 'd') {
			deletion();
		}
```

```
            // Clear the input buffer
            while (getchar() != '₩n');
        }

        return 0;
    }
```

While receives input from the user through the loop. If the input is 'q', exit the loop and exit the program.

If the input is 'i', the key value is entered and the key value is inserted into the Max Heap by calling the insertion function.

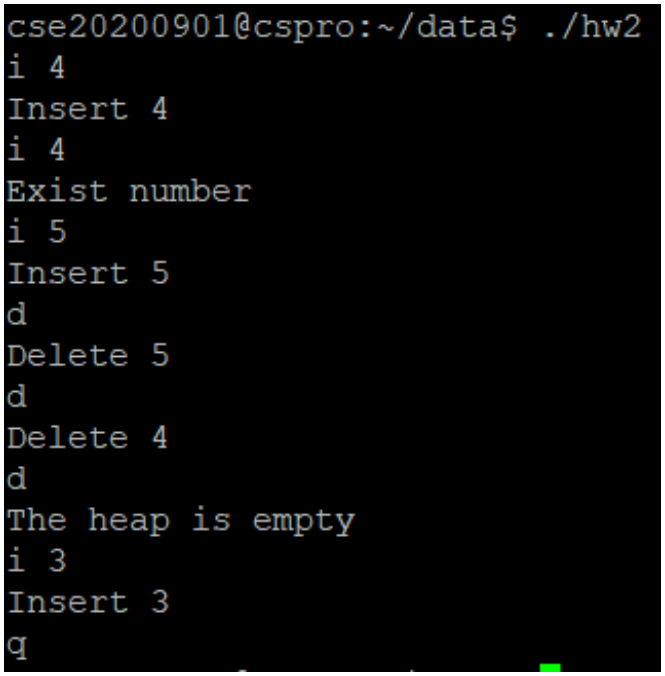
If the input is 'd', call the deletion function to delete the node with the largest key value in Max Heap.

Empty the input buffer after processing the input.

When the while loop ends, the main function ends and the program ends.

The main function receives the input and calls the insertion and deletion functions according to the input to insert or delete the node into the Max Heap. Repeat the loop until the user enters 'q' to process the input.

6) Example



<HW6 - 3>

1) Problem Statement

You must create a binary search tree by reading the n positive integers given to the input file input.txt, and write a program that deletes the integers given to the input file delete.txt from the configured binary search tree. The deleted and reconstructed binary search tree must be outputted as inorder and preorder to verify that the tree is configured correctly.

To construct a binary search tree, insertNode() functions are used to sequentially insert a given amount of integers.
Use the deleteNode() function to delete the integers given in delete.txt from the binary search tree. When you find a node to delete, find the largest value in the subtree on the left and replace it.
Outputs a reconstructed binary search tree to inorder and preorder using the inorderTraversal() and preorderTraversal() functions.

2) Struct TreeNode

```
struct TreeNode {
    int val;
    struct TreeNode* left;
    struct TreeNode* right;
};
```

Structure representing the nodes in the binary search tree. Each node has a val value, a left pointer to the left child node, and a right pointer to the right child node.

3) Create Funtion

```
struct TreeNode* createNode(int val) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->val = val;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

The createNode() function is a function created by dynamically assigning new nodes with a given value of val.

When the function is invoked, it allocates memory of sizeof(structTreeNode) to create a new node. Use the malloc() function to dynamically allocate memory and store the address of the allocated memory in the newNode pointer.

The newNode->val stores the value of the val passed as a parameter. This is the value that the new node has.

The newNode->left and newNode->right pointers initialize the new node's left and right child as pointers to point to. Set to NULL because there are no child nodes yet at this time.

Returns the newNode, which is the created node. This completes the creation of a new node and the return of its address.

The createNode() function is used to create a new node in the binary search tree and returns the node's value and child node pointer initialized.

4) Insert function

```
struct TreeNode* insertNode(struct TreeNode* root, int val) {
    if (root == NULL) {
        return createNode(val);
    }
    if (val < root->val) {
        root->left = insertNode(root->left, val);
    }
    else {
        root->right = insertNode(root->right, val);
    }
    return root;
}
```

The insertNode() function is a function that inserts a new node into a binary search tree.

When a function is called, it receives the root, which is the root node of the current subtree, and the value, val, which you want to insert.

First, check if the current subtree is empty by verifying that the root is NULL. If empty, call the createNode(val) function to create and return a new node with a value.

If the root is not NULL, compare the value val you want to insert with the value root->val of the current node to find the appropriate location.

If val is less than root->val, the location where you want to insert the val is the left subtree of the

current node. Therefore, insertNode (root->left, val) is called to root->left to perform recursive insertion on the left subtree.
If val is greater than or equal to root->val, the location where you want to insert the val is the subtree on the right side of the current node. Therefore, insertNode (root->right, val) is called to root->right to recursively insert the right subtree.
When the recursive insertion operation is complete, returns the root node of the changed subtree. This configures a binary search tree with a new node inserted.

The insertNode() function is used to insert a new node into the binary search tree, and inserts the node in the appropriate location while preserving the properties of the binary search tree.

5) Find max node function

```
struct TreeNode* findMaxNode(struct TreeNode* node) {
    struct TreeNode* current = node;
    while (current->right != NULL) {
        current = current->right;
    }
    return current;
}
```

The findMaxNode() function is a function that finds and returns the node with the largest value in a given binary search tree. This function continues to move to the right child node starting with the given node to find the node with the largest value. This process repeats until you find a node with no right child, that is, a node with the largest value.

When a function is called, it receives the node, which is the root node of the current subtree, as a parameter.
Starting with the current node node, it continues to move while the right child exists. Use this to find the node with the largest value among the right child nodes.
If you find a node with no right child, that is, the node with the largest value, returns that node.

6) delete function

```
struct TreeNode* deleteNode(struct TreeNode* root, int val) {
    if (root == NULL) {
```

```
            return root;
        }
        if (val < root->val) {
            root->left = deleteNode(root->left, val);
        }
        else if (val > root->val) {
            root->right = deleteNode(root->right, val);
        }
        else {
            if (root->left == NULL) {
                struct TreeNode* temp = root->right;
                free(root);
                return temp;
            }
            else if (root->right == NULL) {
                struct TreeNode* temp = root->left;
                free(root);
                return temp;
            }
            struct TreeNode* temp = findMaxNode(root->left);
            root->val = temp->val;
            root->left = deleteNode(root->left, temp->val);
        }
        return root;
    }
```

The deleteNode() function is a function that deletes a node with a specific value from a given binary search tree. This function deletes the node using the following procedure:

When a function is called, it receives the root, which is the root node of the current subtree, and the val, which represents the value you want to delete, as parameters.

First, check if the current subtree is empty by verifying that the root is NULL. If empty, return root as is.

If root is not NULL, compare the value val you want to delete with the value root->val of the current node.

If val is less than root->val, the value you want to delete is likely to exist in the left subtree of the current node. Therefore, call deleteNode (root->left, val) for root->left to recursively perform deletion in the left subtree.

If val is greater than root->val, the value you want to delete is likely to exist in the subtree on the

right side of the current node. Therefore, call deleteNode(root->right, val) for root->right to recursively perform the delete operation in the right subtree.

If val and root->val are the same, the current node is the node to be deleted. At this point, depending on the node to be deleted, it can be divided into three cases:
If the left child of the current node is NULL: Saves the right child of the current node to a temporary node temp, deletes the current node and returns temp.
If the right child of the current node is NULL: Saves the left child of the current node to the temporary node temp, deletes the current node and returns temp.
If both children of the current node exist: Locate the node with the largest value in the left subtree of the current node, save it to temp, and copy the value from temp to the current node. Recursively deletes a node with a temp->val value from the left subtree of the current node.

Returns the root node of the changed subtree. This configures a binary search tree that does not contain deleted nodes.

7) Traverse function

```
void inorderTraversal(struct TreeNode* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->val);
        inorderTraversal(root->right);
    }
}

void preorderTraversal(struct TreeNode* root) {
    if (root != NULL) {
        printf("%d ", root->val);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}
```

The **inorderTraversal() function** is a function that outputs the values of nodes in ascending order by mid-circulating a given binary tree. This function uses recursive methods to navigate the tree.

When a function is called, it receives the root, which is the root node of the current subtree, as a parameter.
If the root is not NULL, visit the nodes in median order.
To visit in a median order, do the following:
Recursively calls inorderTraversal (root->left) for the left subtree of root. This makes a median tour of the left subtree.
Outputs root->val, which is the value of root.
Recursively calls inorderTraversal (root->right) for the right subtree of root. This makes a median tour of the right subtree.
In this way, the values for each node are printed in ascending order through the median rotation.

The **preorderTraversal() function** is a function that outputs the value of a node by traversing the potential of a given binary tree. This function also uses recursive methods to navigate the tree.

When a function is called, it receives the root, which is the root node of the current subtree, as a parameter.
If the root is not NULL, visit the nodes in the order of potential.
To visit in the order of potential, do the following:
Outputs root->val, which is the value of root.
Recursively calls preorderTraversal(root->left) for the left subtree of root. This allows the left subtree to be electrically rotated.
Recursively calls preorderTraversal(root->right) for the right subtree of root. This will allow potential circulation through the right subtree.
This is how the potential is circulated and the values for each node are printed.
These two functions leverage recursive structures to explore all nodes in the tree and output values. The median loop outputs nodes in the binary search tree in ascending order, and the potential loop outputs nodes in a way that shows the structure of the tree.

8) Construct tree function

```
struct TreeNode* constructTree(const char* filename) {
    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        printf("Failed to open the file.₩n");
        exit(1);
    }
```

```
    int n;
    fscanf(file, "%d", &n);
    struct TreeNode* root = NULL;

    for (int i = 0; i < n; i++) {
        int val;
        fscanf(file, "%d", &val);
        root = insertNode(root, val);
    }

    fclose(file);
    return root;
}
```

The constructTree() function is a function that creates a binary tree based on the values read from the file.

When a function is called, it receives the filename as a parameter.
Use the fopen() function to open the filename file in read mode. If the file cannot be opened, output an error message and exit the program.
Reads integer values from the file and stores them in variable n. This value represents the number of nodes to insert into the binary tree.
Initializes the pointer root representing the root node to NULL.
Use the for loop to insert a node into the binary tree by reading the value from the file with n iterations.
Use the fscanf() function to read integer values from the file and store them in the variable val.
Invoke the insertNode() function to insert a new node into the root. At this point, insert the value of val.
The insertNode() function inserts the node in the appropriate location of the binary tree and returns the root of the inserted tree. Therefore, renew the root.
Close the file.
Returns the root node of the generated binary tree.

The constructTree() function performs the process of creating a binary tree based on the values read from the file. The first value of the file represents the number of nodes to be inserted, and subsequent values are inserted into the binary tree with respective node values. This configures a binary tree from the file and returns the root node of the finally created tree.

9) Delete node function

```
struct TreeNode* deleteNodes(struct TreeNode* root, const char* filename) {
    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        printf("Failed to open the file.₩n");
        exit(1);
    }

    int val;
    while (fscanf(file, "%d", &val) == 1 && val != 0) {
        root = deleteNode(root, val);
    }

    fclose(file);
    return root;
}
```

The deleteNodes() function is a function that deletes a node from a binary tree based on the value read from the file.

When the function is called, it receives a filename, which is the filename containing the node values to be deleted, and a pointer root indicating the root node of the binary tree as a parameter.
Use the fopen() function to open the filename file in read mode. If the file cannot be opened, output an error message and exit the program.
Use the while loop to read the value from the file and delete the node only if the value is nonzero.
Use the fscanf() function to read integer values from the file and store them in the variable val.
Call the deleteNode() function to delete a node with that value from the root only if the value is not zero and the value is read from the file normally.
The deleteNode() function finds and deletes nodes in the binary tree that correspond to a given value, adjusts the appropriate connection, and returns the root of the changed tree. Therefore, renew the root.
Close the file.
Returns the root node of the final binary tree where the delete operation was completed.

The deleteNodes() function performs the process of deleting nodes from the binary tree based on the values read from the file. Reads a nonzero value from the file and deletes the node corresponding to that value using the deleteNode() function. Returns the root node of the final tree

where the delete operation was completed.

10) Example

Input.txt	Delete.txt
10 3 9 8 2 5 10 7 1 4 6	1 7 9 3 0
Output:	cse20200901@cspro:~/data$ ./hw3 inorder: 2 4 5 6 8 10 preorder: 2 8 5 4 6 10