Algorithm Report

-이진 트리_HW2-

제 출 일	2024.09.10	전 공	컴퓨터소프트웨어공학과
과 목	알고리즘	학 번	20233523
담당교수	홍 민	이 름	최윤호

목 차

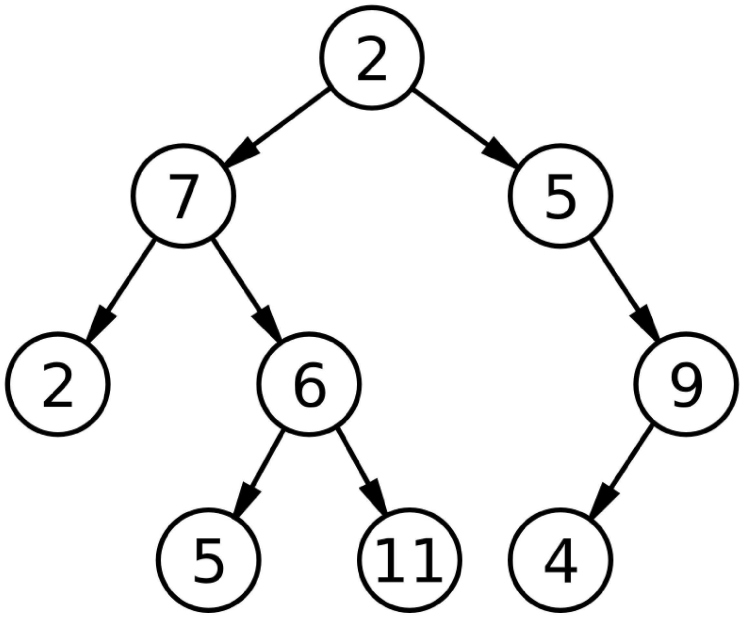
1. 문제 제시

순환과 반복으로 이루어진 이진 탐색 트리를 구현하고
모든 데이터를 순회하는데 걸리는 시간을 비교 하시오.
모든 데이터를 삽입하는데 걸리는 시간을 비교 하시오.
현재 구성된 트리의 전체 노드 개수가 몇 개인지를 구하는 코드를 구현하여 출력 하시오.
현재 구성된 트리의 높이를 구하는 코드를 구현하여 출력 하시오.
현재 구성된 트리의 단말 노드가 몇 개인지를 구하는 코드를 구현하여 출력 하시오.

2. 문제 분석

2.1 이진 트리

이진 트리에 대해 알아보자 이진 트리는 노드의 배정 방식에 따라 붙여진 이름으로 하나의 노드에는 두 개의 노드를 연결시킬 수 있다. 그렇기 때문에 이름이 이진(binary)트리인 것이다. 노드를 삽입하는 기준으로는 원래의 노드(부모 노드)의 값보다 작은 값을 가자는 노드는 부모 노드의 왼쪽, 큰 값을 가진 값을 가진 노드는 오른쪽에 배치한다. 이러한 특성으로 인해 데이터의 중복이 없어야만 생성이 가능하다. 아래의 사진은 이진 트리를 나타낸다.



2.2 이진 트리의 속성

이진 트리의 속성을 구하는 법을 알아보자.

2.2.1 노드의 개수

이진 트리에서 노드를 구하는 방법은 먼저 가장 위의 루트(1) + 루트의 왼쪽, 오른쪽 서브 트리의 노드 개수를 구해 더하는 방식으로 재귀 형식으로 구현할 수 있다.

2.2.2 단말 노드의 개수

단말 노드란 자식 노드가 없는 노르를 뜻한다 한마디로 가장 아래의 노드를 말한다. 단말 노드를 구하는 방법으로는 루트의 왼쪽, 오른쪽으로 연결된 노드가 없는 경우 1을 더하고 연결된 노드가 있는 경우 그 노드를 기준으로 해당 연산을 계속 반복해 나간다. 노드의 개수와 비슷하게 재귀 형식으로 구현한다.

2.2.3 트리의 높이

트리의 높이는 가장 깊은 높이를 말한다. 그렇기 때문에 가장 위의 루트(1)+ 왼쪽, 오른쪽 서브노드 중 가장 깊은 서브 노드의 높이를 계산한다. 이러한 연산도 재귀 형식으로 구현 가능하다.

2.3 노드 삽입

노드 삽입 (구현)을 하는 방법으로는 먼저 가장 위의 루트 노드의 데이터 값과 삽힙할 노드의 데이터값을 비교해 왼쪽, 오른쪽 서브루트를 선택한다. 또한 그 서브루트의 루트와 비교해 서브루트의 서브 루트로 내려가며. 이어진 노드가 없는 경우 그 위치에 삽입한다. 이러한 원리로 인해 노드 삽입은 순환(재귀) 형식과 반복적인 방법으로 구현할 수 있다.

2.4 순회

노드의 값을 살펴보는 과정을 일컫는다. 전위, 중위, 후위 순회로 3가지의 방법이 대표적이며 루트의 값을 본 뒤, 왼쪽, 오른쪽 서브 루트를 살펴보면 전위 순회, 왼쪽 서브 루트의 값을 본 뒤 루트의 값 그리고 오른쪽 서브 루트의 값을 확인하면 중위 순회, 후위 순회는 왼쪽, 오른쪽 서브 루트의 데이터를 확인한 후 가장 마지막에 루트 노드의 데이터값을 본다. 이해를 돕기위해 아래의 사진을 보자

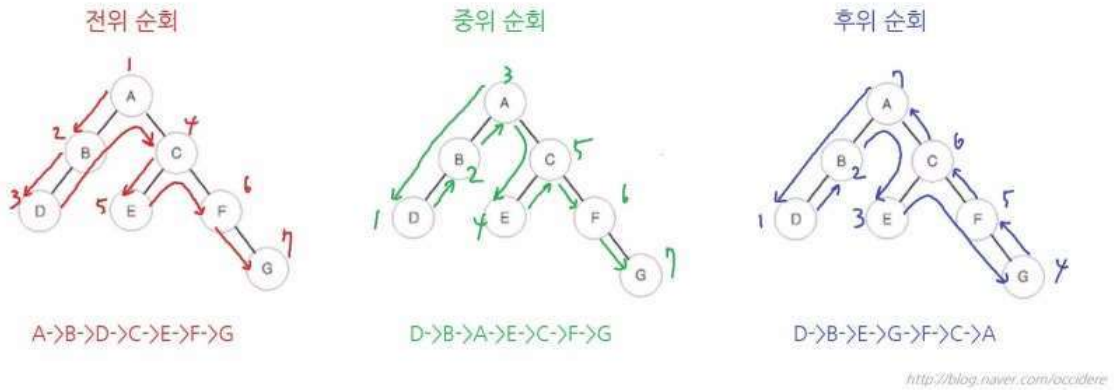


그림 5 https://blog.naver.com/occidere/220899936160

3. 전체 코드

뒷장에 첨부. (파일이 여러개임으로 상단 파일명 주의)

```
1  //이진  트리    노드    구조체  정의
2  #pragma once
3  typedef int element;
4
5  typedef struct node {
6      element data;
7      struct node* right, * left;
8  }node;
9
10
11
12
```

```c
/*
노드를 삽입하는 코드를 구현한 소스 파일
*/
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include<stdlib.h>
#include<time.h>.
#include "tree_data.h"

//노드 생성
node* create_node(element data)
{
    node* new_node = (node*)malloc(sizeof(node));//노드 생성
    new_node->data = data;
    new_node->left = new_node->right = NULL;

    return new_node;//생성된 노드 반환
}

//순환(재귀)를 이용한 이진트리 삽입
node* insert_node_by_circulation(node* root, element data)
{
    if (root == NULL)//루트노드가 비어있을 경우
    {
        root = create_node(data);//새로운 노드 생성
        return root;//생성된 노드 반환
    }

    if (root->data < data)//새로운 노드의 데이터가 루트노드의 데이터 보다 큰(오른쪽)일 경우
    {
        root->right = insert_node_by_circulation(root->right, data);//오른쪽 서브트리에 삽입
    }
    else if (root->data > data)//새로운 노드의 데이터가 루트노드의 데이터 보다 작은(왼쪽)일 경우
    {
        root->left = insert_node_by_circulation(root->left, data);//왼쪽 서브트리에 삽입
    }

    return root;//삽입된 루트노드 반환
}

//반복을 이용한 이진트리 삽입
void insert_node_by_repetition(node** root, element data)//이중포인터를 사용하는 이유는 루트노드가 변경될 수 있기 때문
{

    node* p, * t;//p는 부모, t는 자식

    t = *root;//루트노드를 가리키는 포인터
    p = NULL;//부모노드를 가리키는 포인터

    //삽입해야할 위치 탐색
    while (t != NULL)//t가 NULL이 아닐때까지 반복
```

```
52      {
53          p = t;//부모노드를 가리키는 포인터에 자식노드를 가리키는 포인터를 대입
54          if (t->data > data)//새로운 노드의 데이터가 루트노드의 데이터 보다 작은 (왼쪽)일 경우
55          {
56              t = t->left;//왼쪽 서브트리로 이동
57          }
58          else if (t->data < data)//새로운 노드의 데이터가 루트노드의 데이터 보다 큰 (오른쪽)일 경우
59          {
60              t = t->right;//오른쪽 서브트리로 이동
61          }
62          else//중복된 값이 있을 경우
63          {
64              return;//함수 종료
65          }
66      }
67
68      //노드 추가
69      node* new_node = create_node(data);//새로운 노드 생성
70      if (p == NULL)//부모노드가 비어있을 경우
71      {
72          *root = new_node;//루트노드에 새로운 노드 대입
73      }
74      else//부모노드가 비어있지 않을 경우
75      {
76          if (p->data > data)//삽입할 위치의 부모노드보다 작은(왼쪽)의 경우
77          {
78              p->left = new_node;//왼쪽 서브트리에 삽입
79          }
80          else {
81              p->right = new_node;//오른쪽 서브트리에 삽입
82          }
83      }
84  }
85
86
```

```c
/*
트리의 노드 개수, 단말 노드의 개수, 트리의 높이를 구하는 코드를 구현한 소스 파일
*/
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "tree_data.h"

//전체 노드 개수 구하기
int count_node(node* root)//노드 개수를 구하는 함수
{
    int re = 0;//리턴값을 저장할 변수
    if (root != NULL)//루트노드가 비어있지 않을 경우
    {
        re = 1 + count_node(root->left) + count_node(root->right);//노드 개수를 구하는 재귀함수
    }
    return re;
}


//단말 노드 개수 구하기
int count_leaf(node* root)//단말 노드 개수를 구하는 함수
{
    int re = 0;//리턴값을 저장할 변수
    if (root != NULL)//루트노드가 비어있지 않을 경우
    {
        if (root->left == NULL && root->right == NULL)//왼쪽과 오른쪽 자식노드가 모두 비어있을 경우
        {
            return 1;//단말노드이므로 1을 반환
        }
        else//왼쪽과 오른쪽 자식노드가 모두 비어있지 않을 경우
        {
            re = count_leaf(root->left) + count_leaf(root->right);//단말 노드 개수를 구하는 재귀함수
        }
    }
    return re;//단말 노드 개수 반환
}

//트리 높이 구하기
int tree_height(node* root)
{

    if (root == NULL) { // 루트 노드가 비어있을 경우
        return 0;
    }

    int l_height = tree_height(root->left);//왼쪽 서브트리의 높이
    int r_height = tree_height(root->right);//오른쪽 서브트리의 높이

    // 더 큰 높이에 1을 더하여 현재 노드의 높이를 반환
    return 1 + (l_height > r_height ? l_height : r_height);
}
```

```c
/*
트리 순회를 구현한 소스 파일
*/
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include<stdlib.h>
#include<time.h>
#include "tree_data.h"


//순환을 이용한 순회

//전위 순회
void preorder(node* root, FILE* output)
{
    if (root == NULL)
    {
        return;
    }

    fprintf(output, "%d ", root->data);//루트노드 출력
    preorder(root->left, output);//왼쪽 서브트리 순회
    preorder(root->right, output);//오른쪽 서브트리 순회

}

//중위 순회
void inorder(node* root, FILE* output)
{
    if (root == NULL)
    {
        return;
    }

    inorder(root->left,output);//왼쪽 서브트리 순회
    fprintf(output, "%d ", root->data);//루트노드 출력
    inorder(root->right, output);//오른쪽 서브트리 순회

}

//후위 순회
void postorder(node* root, FILE* output)
{
    if (root == NULL)
    {
        return;
    }

    postorder(root->left, output);//왼쪽 서브트리 순회
    postorder(root->right, output);//오른쪽 서브트리 순회
    fprintf(output, "%d ", root->data);//루트노드 출력
}

//반복을 이용한 순회
int top = -1;//스택의 top을 -1로 초기화

```

```c
void push(node* stack[], node* p, int n)//스택에 노드를 푸시하는 함수
{
    if (top < n - 1)
    {
        stack[++top] = p;
    }
}


node* pop(node* stack[])
{
    if (top >= 0) { // top이 -1보다 크거나 같을 때
        return stack[top--];
    }
    else
    {
        return NULL; // 스택이 비어있을 경우 NULL 반환
    }
}


//반복을 이용한 순회
void preorder_by_repetition(node* stack[], node* root, int n, FILE* output)
{
    while (1)
    {
        // 왼쪽 자식 노드로 내려가면서 노드를 스택에 푸시
        for (; root; root = root->left) {
            fprintf(output, "%d ", root->data); // 현재 노드 출력
            push(stack, root, n); // 스택에 현재 노드 푸시
        }

        // 스택에서 노드를 팝
        root = pop(stack);
        if (!root) {
            break; // 스택이 비어있으면 종료
        }

        // 오른쪽 자식 노드로 이동
        root = root->right;
    }
}

//중위 순회
void inorder_by_repetition(node* stack[], node* root, int n, FILE* output)
{
    while (1)
    {
        for (; root; root = root->left)//왼쪽서브트리로 이동하면서 스택에 푸시
        {
            push(stack, root, n);
        }

        root = pop(stack);//스택에서 팝
        if (!root) {
            break;
```

```
113         }
114
115         fprintf(output, "%d ", root->data);//현재 노드 출력
116         root = root->right;//오른쪽 서브트리로 이동
117     }
118 }
119
120
121 void postorder_by_repetition(node* stack[], node* root, int n, FILE* output)//후위 순회
122 {
123     node* temp = NULL;//임시 노드
124
125     while (1) {
126
127         // 왼쪽 자식으로 내려가면서 스택에 푸시
128         for (; root; root = root->left) {
129             push(stack, root, n);
130         }
131
132         while (1)//스택에서 팝하면서 후위 순회
133         {
134             root = pop(stack);//스택에서 팝
135             if (!root)
136             {
137                 return;
138             }
139
140             if (!root->right || temp == root->right)//오른쪽 자식이 없거나 오른쪽 자식이 방문한 노드일 경우
141             {
142                 fprintf(output, "%d ", root->data);//현재 노드 출력
143                 temp = root;//임시 노드에 현재 노드 대입
144             }
145             else//오른쪽 자식이 있을 경우
146             {
147                 push(stack, root, n);//스택에 현재 노드 푸시
148                 root = root->right;//오른쪽 서브트리로 이동
149                 break;//반복문 종료
150             }
151         }
152
153     }
154 }
155
```

```c
/*
작성일 : 2024_10_10
작성자 : 최윤호
프로그램 명 : 이진트리 구현 프로그램
*/
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include<stdlib.h>
#include<time.h>
#include "tree_data.h"

//tree에 할당한 메모리 해제
void delete_tree(node* root)
{
    if (root == NULL)
    {
        return;
    }

    delete_tree(root->left);//왼쪽 서브트리 삭제
    delete_tree(root->right);//오른쪽 서브트리 삭제
    free(root);
}

int main()
{

    FILE* fp;
    fp = fopen("data.txt", "r");
    //루트 노드 생성
    node* root_in_circulation_Tree = (node*)malloc(sizeof(node));
    element data;

    //파일 출력
    FILE* output;
    output = fopen("output.txt", "w");

    ////////////////////////////////////////노드 삽입////////////////////////////////////////
    printf("================================================================================₩n");
    printf("노드 삽입의 소요시간₩n");
    printf("================================================================================₩n₩n");
    //순환을 이용한 삽입연산
    clock_t start = clock();
    fscanf(fp, "%d", &data);//루트 노드 생성
    root_in_circulation_Tree->data = data;//루트 노드 생성
    root_in_circulation_Tree->left = root_in_circulation_Tree->right = NULL;//루트 노드 생성
    while (!feof(fp))//파일의 끝까지 읽어들임
    {
        fscanf(fp, "%d", &data);//파일에서 데이터를 읽어들임
        insert_node_by_circulation(root_in_circulation_Tree, data);//삽입연산
    }
    clock_t end = clock();
```

```
53
54      printf("순환을 이용한 삽입의 소요 시간: %lf₩n", (double)(end - start) /
          CLOCKS_PER_SEC);
55
56      //파일포인터를 맨 처음으로
57      rewind(fp);
58
59      node* root_in_repetition_Tree = (node*)malloc(sizeof(node));//루트 노드 생성
60
61      //반복을 이용한 삽입연산
62      start = clock();
63      fscanf(fp, "%d", &data);
64      root_in_repetition_Tree->data = data;
65      root_in_repetition_Tree->left = root_in_repetition_Tree->right = NULL;
66      while (!feof(fp))
67      {
68          fscanf(fp, "%d", &data);
69          insert_node_by_repetition(&root_in_repetition_Tree, data);
70      }
71      end = clock();
72      printf("반복을 이용한 삽입의 소요 시간: %lf₩n₩n", (double)(end - start) /
          CLOCKS_PER_SEC);
73
74
75      printf("=============================================================================
          ₩n");
76      printf("트리 속성₩n");
77      printf("=============================================================================
          ₩n₩n");
78      //트리의 노드 대수, 단말 노드 개수 , 높이
79      int height = tree_height(root_in_circulation_Tree);//트리의 높이
80      int n = count_node(root_in_circulation_Tree);//트리의 전체 노드 개수
81      printf("트리의 전체 노드 개수 : %d₩n",n);
82      printf("트리의 높이 : %d₩n", height);
83      printf("트리의 단말 노드 개수 : %d₩n", count_leaf(root_in_circulation_Tree));
84
85
86
87      ////////////////////////////////////트리 순
          회//////////////////////////////////////
88      printf
          ("₩n=============================================================================
          ₩n");
89      printf("순회 시간 비교₩n");
90      printf("=============================================================================
          ₩n");
91      //순환을 이용한 트리 순회
92      printf("₩n순환을 이용한 순회의 소요시간₩n₩n");
93      fprintf(output, "순환을 이용한 순회₩n₩n");
94      //전위 순회
95      fprintf(output, "전위 순회₩n");
96      start = clock();
97      preorder(root_in_repetition_Tree, output);
98      end = clock();
99      fprintf(output, "₩n");
100     printf("전위 순회의 소요 시간: %lf₩n", (double)(end - start) / CLOCKS_PER_SEC);
```

```
101
102     //중위 순회
103     fprintf(output, "중위 순회₩n");
104     start = clock();
105     inorder(root_in_repetition_Tree, output);
106     end = clock();
107     fprintf(output, "₩n");
108     printf("중위 순회의 소요 시간: %lf₩n", (double)(end - start) / CLOCKS_PER_SEC);
109
110     //후위 순회
111     fprintf(output, "후위 순회₩n");
112     start = clock();
113     postorder(root_in_repetition_Tree, output);
114     end = clock();
115     printf("후위 순회의 소요 시간: %lf₩n", (double)(end - start) / CLOCKS_PER_SEC);
116
117     ////////////////////////////////////////////////////////////////////////////////
          //////////
118     ////////////////////////////////////////////////////////////////////////////////
          //////////
119
120     //반복(스택)을 이용한 트리 순회
121     node** stack = (node**)malloc(sizeof(node*) * n);//스택 생성
122     printf("₩n반복을 이용한 순회의 소요시간₩n₩n");
123     fprintf(output, "₩n₩n반복을 이용한 순회₩n₩n");
124     //전위 순회
125     fprintf(output, "전위 순회₩n");
126     start = clock();
127     preorder_by_repetition(stack,root_in_repetition_Tree, n,output);
128     end = clock();
129     fprintf(output, "₩n");
130     printf("전위 순회의 소요 시간: %lf₩n", (double)(end - start) / CLOCKS_PER_SEC);
131
132     //중위 순회
133     fprintf(output, "중위 순회₩n");
134     start = clock();
135     inorder_by_repetition(stack, root_in_repetition_Tree, n, output);
136     end = clock();
137     fprintf(output, "₩n");
138     printf("중위 순회의 소요 시간: %lf₩n", (double)(end - start) / CLOCKS_PER_SEC);
139
140     //후위 순회
141     fprintf(output, "후위 순회₩n");
142     start = clock();
143     postorder_by_repetition(stack, root_in_repetition_Tree, n, output);
144     end = clock();
145     printf("후위 순회의 소요 시간: %lf₩n", (double)(end - start) / CLOCKS_PER_SEC);
146     printf("========================================================================
          ₩n");
147
148
149
150     //생성된 이진트리 메모리 해제
151     delete_tree(root_in_circulation_Tree);
152     delete_tree(root_in_repetition_Tree);
153
```

```
154         //파일 닫기
155         fclose(fp);
156         fclose(output);
157 
158 
159         return 0;
160     }
```

4. 코드 분석

코드의 구조를 보다 명확하게 파악할 수 있도록 여러 개의 소스파일, 헤더 파일로 작성하였기 때문에 소스 파일별로 코드를 분석하도록 하겠다.

4.1 tree_data.h

```
1  //이진  트리   노드   구조체  정의
2  #pragma once
3  typedef int element;
4
5  typedef struct node {
6      element data;
7      struct node* right, * left;
8  }node;
```

여러 소스 파일에서 사용할 수 있도록 노드의 데이터와, 노드 구조체를 헤더 파일에 구현한다. 노드의 경우 왼쪽, 오른쪽 서브 트리를 가리키는 포인터를 변수로 갖는다.

4.2 insert_node

노드를 삽입하는 코드가 구현되어있는 소스 파일이다.

```
8  node* create_node(element data)
9  {
10     node* new_node = (node*)malloc(sizeof(node));//노드 생성
11     new_node->data = data;
12     new_node->left = new_node->right = NULL;
13
14     return new_node;//생성된 노드 반환
15 }
```

노드를 삽입하기 위해 새로운 노드를 생성하는 함수이다. 새로운 노드가 가질 데이터를 매개변수로 노드를 malloc()함수를 사용해 동적으로 할당한 뒤 새로운 노드를 반환한다.

```
20 //순환(재귀)를 이용한 이진트리 삽입
21 node* insert_node_by_circulation(node* root, element data)
22 {
23     if (root == NULL)//루트노드가 비어있을 경우
24     {
25         root = create_node(data);//새로운 노드 생성
26         return root;//생성된 노드 반환
27     }
28
29     if (root->data < data)//새로운 노드의 데이터가 루트노드의 데이터 보다 큰(오른쪽)일경우
30     {
31         root->right = insert_node_by_circulation(root->right, data);//오른쪽 서브트리에 삽입
32     }
33     else if (root->data > data)//새로운 노드의 데이터가 루트노드의 데이터 보다 작은(왼쪽)일 경우
34     {
35         root->left = insert_node_by_circulation(root->left, data);//왼쪽 서브트리에 삽입
36     }
37
38     return root;//삽입된 루트노드 반환
39 }
```

순환 형식으로 노드를 삽입하는 형식이다. 만약 루트가 NULL인 경우(삽입 해야할 위치) root노드를 새로 삽입할 노드로 변환한다. 그렇지 않을경우는 데이터의 값을 비교한다. 새로운 노드의 데이터가 루트 노드의 데이터 보다 작을(왼쪽)의 경우 루트 노드에 왼쪽 서브 트리의 루트 노드와 비교하기 위해 재귀 형식으로 함수를 다시 호출한다. 만약 데이터가 루트 노드의 데이터보다 큰 경우 오른쪽 서브 트리의 루트를 매개변수로 함수를 다시 호출한다. 만약 중복된 값인 경우 root를 반환해 삽입하지 않는다.

```
41 //반복을 이용한 이진트리 삽입
42 void insert_node_by_repetition(node** root, element data)//이중포인터를 사용하는 이유 ➲
     는 루트노드가 변경될 수 있기 때문
43 {
44
45     node* p, * t;//p는 부모, t는 자식
46
47     t = *root;//루트노드를 가리키는 포인터
48     p = NULL;//부모노드를 가리키는 포인터
49
50     //삽입해야할 위치 탐색
51     while (t != NULL)//t가 NULL이 아닐때까지 반복
52     {
53         p = t;//부모노드를 가리키는 포인터에 자식노드를 가리키는 포인터를 대입
54         if (t->data > data)//새로운 노드의 데이터가 루트노드의 데이터 보다 작은 (왼쪽)➲
             일 경우
55         {
56             t = t->left;//왼쪽 서브트리로 이동
57         }
58         else if (t->data < data)//새로운 노드의 데이터가 루트노드의 데이터 보다 큰 (오➲
             른쪽)일 경우
59         {
60             t = t->right;//오른쪽 서브트리로 이동
61         }
62         else//중복된 값이 있을 경우
63         {
64             return;//함수 종료
65         }
66     }
67
68     //노드 추가
69     node* new_node = create_node(data);//새로운 노드 생성
70     if (p == NULL)//부모노드가 비어있을 경우
71     {
72         *root = new_node;//루트노드에 새로운 노드 대입
73     }
74     else//부모노드가 비어있지 않을 경우
75     {
76         if (p->data > data)//삽입할 위치의 부모노드보다 작은(왼쪽)의 경우
77         {
78             p->left = new_node;//왼쪽 서브트리에 삽입
79         }
80         else {
81             p->right = new_node;//오른쪽 서브트리에 삽입
82         }
83     }
84 }
85
```

먼저 root를 변환하며 삽입해야 할 위치를 찾아야 하기 때문에 본래 root의 값이 변경되지 않도록 이중 포인터를 통해 root를 받아온다. 그 뒤 부모 노드와 자식 노드를 저장할 root 포인터를 선언한다. 그 후 먼저 삽입해야 할 위치를 while문을 통해 탐색한다. t 노드의 데이터를 비교하며 t가 NULL 일 때까지 왼쪽이나 오른쪽 서브 트리로 이동한다. 만약 t가 NULL이 된다면 t의 부모노드 p의 자식 노드의 위

치로 삽입해야 한다는 뜻으로 새로운 노드를 생성 후 p 노드의 데이터와 비교해 왼쪽이나 오른쪽에 새로 생성한 노드를 삽입한다.

4.3 tree_attibute

트리의 노드 개수, 단말 노드의 개수, 트리의 높이를 구하는 코드를 구현한 소스파일이다. 먼저 전체 노드의 개수를 구하는 함수를 보자.

```
10 //전체 노드 개수 구하기
11 int count_node(node* root)//노드 개수를 구하는 함수
12 {
13     int re = 0;//리턴값을 저장할 변수
14     if (root != NULL)//루트노드가 비어있지 않을 경우
15     {
16         re = 1 + count_node(root->left) + count_node(root->right);//노드 개수를 구하는?
               재귀함수
17     }
18     return re;
19 }
```

루트 노드가 존재한다는 것은 최소 1의 높이라는 것이다. 그 후 왼쪽, 오른쪽 서브노트의 높이를 구해 더해주며 루트가 존재하지 않을 때까지(트리가 끝나면) 더해주며 재귀 함수로 구현한다.

```
22 //단말 노드 개수 구하기
23 int count_leaf(node* root)//단말 노드 개수를 구하는 함수
24 {
25     int re = 0;//리턴값을 저장할 변수
26     if (root != NULL)//루트노드가 비어있지 않을 경우
27     {
28         if (root->left == NULL && root->right == NULL)//왼쪽과 오른쪽 자식노드가 모두 ?
              비어있을 경우
29         {
30             return 1;//단말노드이므로 1을 반환
31         }
32         else//왼쪽과 오른쪽 자식노드가 모두 비어있지 않을 경우
33         {
34             re = count_leaf(root->left) + count_leaf(root->right);//단말 노드 개수를  ?
                  구하는 재귀함수
35         }
36     }
37     return re;//단말 노드 개수 반환
```

단말 노드의 개수를 구하는 함수이다. 만약 루트 노드의 왼쪽, 오른쪽에 연결된 노드가 없다면 단말 노드이므로 1을 반환하고 그렇지 않으면 왼쪽, 오른쪽 노드를 루트 노드로 하는 서브트리의 단말 노드를 다시 탐색해 더해주며 재귀적으로 동작한다.

```
40  //트리 높이 구하기
41  int tree_height(node* root)
42  {
43
44      if (root == NULL) { // 루트 노드가 비어있을 경우
45          return 0;
46      }
47
48      int l_height = tree_height(root->left);//왼쪽 서브트리의 높이
49      int r_height = tree_height(root->right);//오른쪽 서브트리의 높이
50
51      // 더 큰 높이에 1을 더하여 현재 노드의 높이를 반환
52      return 1 + (l_height > r_height ? l_height : r_height);
```

트리의 높이를 구하는 함수이다. root 노드가 NULL이 아니라는 것은 최소한의 높이 1이라는 것이기 때문에 1+ (왼쪽 서브 트리의 높이와 오른쪽 서브 트리의 높이 중 더 큰 높이)이며 왼쪽 서브 트리의 높이와 오른쪽 서브 트리의 높이는 재귀적으로 연산한다.

4.4 tree_traversal

트리의 순회를 구현한 소스 파일이다. 순환으로 구현한 전위, 중위, 후위 순회와 반복으로 구현한 전위, 중위, 후위 순회를 구현하였다. 먼저 순환으로 구현한 전위, 중위, 후위 순회를 보자. 또한 printf함수를 사용해 방문한 노드를 출력하면 데이터의 양이 너무 많아 출력 창의 가독성이 떨어져 printf 대신 fprint 를 사용해 output파일에 작성하였다.

```
13  //전위 순회
14  void preorder(node* root, FILE* output)
15  {
16      if (root == NULL)
17      {
18          return;
19      }
20
21      fprintf(output, "%d ", root->data);//루트노드 출력
22      preorder(root->left, output);//왼쪽 서브트리 순회
23      preorder(root->right, output);//오른쪽 서브트리 순회
24
25  }
26
27  //중위 순회
28  void inorder(node* root, FILE* output)
29  {
30      if (root == NULL)
31      {
32          return;
33      }
34
35      inorder(root->left,output);//왼쪽 서브트리 순회
36      fprintf(output, "%d ", root->data);//루트노드 출력
37      inorder(root->right, output);//오른쪽 서브트리 순회
38
39  }
40
41  //후위 순회
42  void postorder(node* root, FILE* output)
43  {
44      if (root == NULL)
45      {
46          return;
47      }
48
49      postorder(root->left, output);//왼쪽 서브트리 순회
50      postorder(root->right, output);//오른쪽 서브트리 순회
51      fprintf(output, "%d ", root->data);//루트노드 출력
52  }
```

순환(재귀)로 순환을 구현하며 엄청나게 간단하다. 전위, 중위, 후위에서 전-중-후에 해당하는 기준은 root 노드이며 root 노드를 먼저 방문할지 중간에 방문할지 제일 나중에 방문하는지에 따라 다른 것이다. 그렇기 때문에 세세한 설명은 전위 순회를 예로 하겠다. 먼저 전위 순회는 root 노드를 제일 먼저 방문하기 때문에 매개변수인 root의 데이터를 출력한다. 그 후 root 노드의 왼쪽 서브 트리를 다시 전위 순회한다. 왼쪽 root의 전위 순회가 끝나면 마지막으로 오른쪽 서브 트리를 전위 순회한다. 이렇게 재귀를 통해 노드를 순회하므로 코드에서 보듯이 함수의 호출 순서에 따라 전위, 중위, 후위가 결정된다.

```
54 //반복을 이용한 순회
55 int top = -1;//스택의 top을 -1로 초기화
56
57 void push(node* stack[], node* p, int n)//스택에 노드를 푸시하는 함수
58 {
59     if (top < n - 1)
60     {
61         stack[++top] = p;
62     }
63 }
64
65
66 node* pop(node* stack[])
67 {
68     if (top >= 0) { // top이 -1보다 크거나 같을 때
69         return stack[top--];
70     }
71     else
72     {
73         return NULL; // 스택이 비어있을 경우 NULL 반환
74     }
75 }
76
```

반복을 통해 순회를 하기 위해서는 스택이 필요하다 선입 후출의 원리인 스택을 이용해 가장 나중에 방문할 노드를 root에서 부터 스택에 push하는 것이다.

```
78 //반복을 이용한 순회
79 void preorder_by_repetition(node* stack[], node* root, int n, FILE* output)
80 {
81     while (1)
82     {
83         // 왼쪽 자식 노드로 내려가면서 노드를 스택에 푸시
84         for (; root; root = root->left) {
85             fprintf(output, "%d ", root->data); // 현재 노드 출력
86             push(stack, root, n); // 스택에 현재 노드 푸시
87         }
88
89         // 스택에서 노드를 팝
90         root = pop(stack);
91         if (!root) {
92             break; // 스택이 비어있으면 종료
93         }
94
95         // 오른쪽 자식 노드로 이동
96         root = root->right;
97     }
98 }
```

순회의 특성에 따라 가장 먼저 매개변수로 받은 root를 바로 출력(방문)하며 왼쪽 서브 트리의 root들을 모두 push한다. 이 과정에서 root를 출력(방문)하며 왼쪽 서브 트리의 root이자 root의 왼쪽 노드를 방문하게 되는 것이다. 그 후 가장 아래의 노드부터 그 노드의 오른쪽 노드를 방문한다. 스택이 비었다는 것은 남은 노드가 없다는 것으로 순회를 마친다.

```
100 //중위 순회
101 void inorder_by_repetition(node* stack[], node* root, int n, FILE* output)
102 {
103     while (1)
104     {
105         for (; root; root = root->left)//왼쪽서브트리로 이동하면서 스택에 푸시
106         {
107             push(stack, root, n);
108         }
109 
110         root = pop(stack);//스택에서 팝
111         if (!root) {
112             break;
113         }
114 
115         fprintf(output, "%d ", root->data);//현재 노드 출력
116         root = root->right;//오른쪽 서브트리로 이동
117     }
118 }
```

중위 순회이다 중 위순회는 왼쪽 서브 트리부터 모드 방문해야되기 때문에 root의 데이터를 출력(방문)하지 않고 왼쪽 서브 트리의 root를 모두 push한다. 그 후 가장 마지막에 들어간 노드는 가장 아래 왼쪽 서브 트리의 단말 노드 이기 때문에 가장 먼저 방문 한 후 다음 노드를 pop하면 방문한 노드의 root이기때문에 해당 노드를 방문 뒤 오른쪽 서브트리로 이동해 해당 과정을 통해 중위 순회를 진행한다. 만약 root가 NULL인 경우는 stack이 비었다는 뜻으로 모든 노드를 방문한 것이기 때문에 함수를 종료한다.

```
121 void postorder_by_repetition(node* stack[], node* root, int n, FILE* output)//후위 순회
122 {
123     node* temp = NULL;//임시 노드
124 
125     while (1) {
126 
127         // 왼쪽 자식으로 내려가면서 스택에 푸시
128         for (; root; root = root->left) {
129             push(stack, root, n);
130         }
131 
132         while (1)//스택에서 팝하면서 후위 순회
133         {
134             root = pop(stack);//스택에서 팝
135             if (!root)
136             {
137                 return;
138             }
139 
140             if (!root->right || temp == root->right)//오른쪽 자식이 없거나 오른쪽 자식이 방문한 노드일 경우
141             {
142                 fprintf(output, "%d ", root->data);//현재 노드 출력
143                 temp = root;//임시 노드에 현재 노드 대입
144             }
145             else//오른쪽 자식이 있을 경우
146             {
147                 push(stack, root, n);//스택에 현재 노드 푸시
148                 root = root->right;//오른쪽 서브트리로 이동
149                 break;//반복문 종료
150             }
151         }
152 
153     }
154 }
155 
```

반복문을 이용한 후위 순회의 경우 꽤나 복잡하다. 먼저 왼쪽 서브 트리의 root를 모두 스택이 push 한다. 그 후 가장 마지막 노드를 pop 해 저장한 후 그 노드의 오른쪽 노드를 모두 후위 순회로 순회하기 위해 스택에 push를 하며 방문을 확인하기 위해 temp변수에 저장한 후 방문하지 않은 오른쪽 서브 트리가 없을때까지 반복한 다. 그렇게 왼쪽 노드를 방문 후 오른쪽 노드를 방문한 뒤 다음 root를 pop 해 해당 과정을 반복한다. 스택이 비어 root가 NULL인 경우 모든 노드를 방문한 것이기때문에 함수를 종료한다.

4.5 main

main 소스 파일의 경우는 스택을 생성하거나 파일을 읽고 쓰거나 다른 위에 설명한 함수들을 호출하는 소스이기 때문에 상세 설명은 생략하나 메모리 해제에 대한 부분만 설명하겠다.

```
10  //tree에 할당한 메모리 해제
11  void delete_tree(node* root)
12  {
13      if (root == NULL)
14      {
15          return;
16      }
17
18      delete_tree(root->left);//왼쪽 서브트리 삭제
19      delete_tree(root->right);//오른쪽 서브트리 삭제
20      free(root);
21  }
```

트리의 가장 상위 root를 매개 변수로 해 재귀 형식으로 왼쪽, 오른쪽 서브 루트에 할당된 메모리를 해제하고 자기 자신을 해제한다.

4.6 그 외

트리의 구현 방식이 다르다고 트리의 값이 달라지지 않기 때문에 순회를 위해서 사용한 트리는 반복문으로 구현한 트리를 사용하였음.

5. 실행창

```
================================================================================
노드 삽입의 소요시간
================================================================================

순환을 이용한 삽입의 소요 시간: 0.039000
반복을 이용한 삽입의 소요 시간: 0.037000

================================================================================
트리 속성
================================================================================

트리의 전체 노드 개수 : 32500
트리의 높이 : 34
트리의 단말 노드 개수 : 10843

================================================================================
순회 시간 비교
================================================================================

순환을 이용한 순회의 소요시간

전위 순회의 소요 시간: 0.019000
중위 순회의 소요 시간: 0.019000
후위 순회의 소요 시간: 0.016000

반복을 이용한 순회의 소요시간

전위 순회의 소요 시간: 0.014000
중위 순회의 소요 시간: 0.016000
후위 순회의 소요 시간: 0.016000
================================================================================
```

output.txt

```
1   순환을 이용한 순회
2
3   전위 순회
4   16250 11256 11027 2704 146 127 11 10 3 1 2 4 6 5 7 9 8 112 41 28 12 25 13 21 16 15
5   중위 순회
6   1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
7   후위 순회
8   2 1 5 8 9 7 6 4 3 10 14 15 17 19 18 20 16 23 24 22 21 13 26 27 25 12 34 33 32 31 35
9
10  반복을 이용한 순회
11
12  전위 순회
13  16250 11256 11027 2704 146 127 11 10 3 1 2 4 6 5 7 9 8 112 41 28 12 25 13 21 16 15
14  중위 순회
15  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
16  후위 순회
17  2 1 5 8 9 7 6 4 3 10 14 15 17 19 18 20 16 23 24 22 21 13 26 27 25 12 34 33 32 31 35
```

그림 23 output.txt의 경우 가독성을 위해 IDE에서 캡처 하였음.

6. 느낀 점

재귀 함수에 원리에 대해 더욱 심도 있게 생각할 수 있었으며, 재귀와 반복의 장단점을 배울 수 있었음.