

자료구조2 이론 레포트



과목명		자료구조2 실습
담당교수		홍 민 교수님
학과		컴퓨터소프트웨어공학과
학년		2학년
학번		20204059
이름		이에빈

목 차

1. 순환과 반복으로 삽입하여 구현하는 이진 탐색 트리

1.1 문제 분석

1.2 소스 코드

1.3 소스 코드 분석

1.4 실행창

1.5 느낀점

1. 순환과 반복으로 삽입하여 구현하는 이진 탐색 트리

HW 2

- 순환과 반복으로 이루어진 이진 탐색 트리를 구현하고 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.

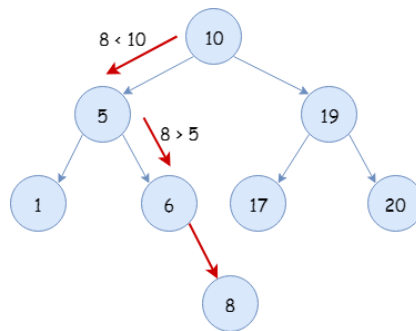
1.1 문제분석

이 문제는, 이진 트리를 구현하는 방법에서, 순환적인 방법으로 트리에 노드를 삽입하는데 걸린 시간과 반복적인 방법으로 트리에 노드를 삽입하는데 걸린 시간을 비교하고 구성된 트리 노드의 전체 노드 개수, 단말 노드의 개수, 그리고 트리의 높이를 구하는 프로그램을 작성하는 문제이다.

삽입

우선, 이진 탐색 트리는 루트 노드의 왼쪽 서브 트리에는 루트 노드의 키 값보다 작은 값을 가진 노드들을, 오른쪽 서브 트리에는 루트 노드의 키 값보다 큰 값을 가진 노드들을 갖는 트리이다. 따라서 삽입의 과정에서 삽입할 노드의 위치를 탐색한 후에, 새로운 노드를 부모 노드와 연결하는 과정이 필요하다. 탐색을 위해서는, 루트 노드부터 시작하여 노드의 키 값과, 삽입하려는 노드의 키 값을 비교하여 왼쪽 또는 오른쪽 자식 노드로 이동하기를 반복하여 마지막 NULL 값까지 도달하게 된다. 결국 마지막 위치가 새로 노드가 삽입될 위치가 되는 것인데, 이러한 삽입의 방법에는 순환적인 방법 또는 반복적인 방법이 있다.

다음의 예시를 살펴보면 각 함수의 동작을 간단히 살펴보자.



1. 순환적인 삽입

순환적인 방법은, 새로운 노드가 삽입될 위치의 주소를 부모 노드의 left 또는 right 과 연결하기 위해, 노드의 키 값과 삽입할 노드의 키 값을 비교하여 삽입될 위치를 탐색하기 위해 NULL 값에 도달할 때까지 오른쪽 자식 노드 또는 왼쪽 자식 노드를 따라가며 삽입 함수를 순환적으로 호출하기를 반복한다. 위의 그림에서, 8 이라는 수는 루트 노드의 10 의 값보다 작으므로 순환호출을 통해 10 의 왼쪽 서브 트리를 따라 내려가게 되고, 이후 5 와 비교한 후 순환 호출을 통해 5 의 오른쪽 서브 트리, 그리고 다시 6 의 오른쪽 서브 트리를 따라 내려가다 마지막 NULL 을 만나, 6 의 오른쪽 자식 노드와 연결이 된다.

2. 반복적인 삽입

반복적인 방법은, 노드에 대한 포인터, 그리고 부모 노드에 대한 포인터가 함께 오른쪽 또는 왼쪽 자식 노드를 따라 내려가며, NULL 값을 만날 때 까지의 과정이 반복문을 통해 구현된다. 순환적인 삽입의 과정과 동일하지만, 함수가 순환 호출되는 것이 아닌 반복문을 통해 포인터가 이동하여 삽입의 위치를 탐색한다는 부분에서 다르다.

트리의 전체 노드 개수

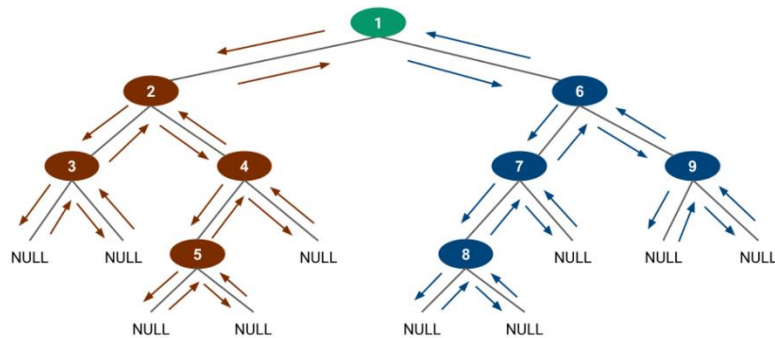
트리의 전체 노드 개수는 순환의 방법을 사용하여 구한다. 루트 노드의 왼쪽 서브 트리, 그리고 오른쪽 서브 트리를 따라 내려가 NULL 값이 나올 때까지 순환을 반복하여, 루트 노드는 자기 자신 노드, 그리고 자식 노드의 개수에 따라 그 값을 누적하여 더한다.

트리의 단말 노드 개수

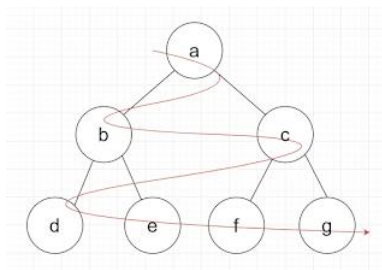
트리의 단말 노드 개수 또한 순환의 방법으로 구한다. 루트 노드에서부터, 왼쪽 서브 트리와 오른쪽 서브 트리를 순환적으로 호출하기를 반복하여, 왼쪽 자식 노드 그리고 오른쪽 자식 노드 모두 NULL 값인 경우 1 를 반환하여 단말 노드의 개수를 누적하여 더한다.

트리의 높이

기존 트리의 높이를 구하는 방법은 순환을 이용하여 루트 노드인 자기 자신의 노드에 루트 노드의 왼쪽 서브 트리, 그리고 오른쪽 서브 트리를 각각 전달한 순환 함수를 NULL 값에 도달할 때까지 호출하기를 반복하고 그 두 값 중 더 큰 값을 더하여, 가장 깊은 곳에 존재한 NULL 값까지의 최대 높이를 구할 수 있도록 구현된 방법이었다.



그러나, 기존의 이 방법은 순환 호출된 각 함수에서 왼쪽과 오른쪽 서브 트리를 전달한 두 함수 중 반환하는 값이 더 큰 값을 구해내는 max 함수의 연산을 추가적으로 매번 실행해야 했기 때문에 실행 시간이 길어진다. 따라서, max 함수의 연산 대신 0으로 초기화된 왼쪽 서브 트리를 순환 호출하여 얻은 결과 값과 오른쪽 서브 트리를 순환 호출하여 얻은 값을 비교하여 더 큰 값을 루트 노드인 자기 자신 노드에 더하는 코드로 변경하였다.



순환적인 방법 외에, 원형 큐 구조를 사용하여 레벨 순회를 하는 반복적인 방법으로도 트리의 높이를 구할 수 있다. 충분히 큰 크기의 배열을 가지는 원형 큐에 루트 노드에서부터 시작하여, 삽입한 부모 노드를 삭제함과 동시에, 자식 노드들을 다시 삽입하기를 반복하여 FIFO(First In First Out의 순차적인 입출력)구조를 이용하여 각 레벨을 왼쪽 노드부터 순차적으로 순회하여 높이를 구하는 방법이다. 한 번 레벨 순회를 할 때마다 큐에 삽입되는 요소의 개수는 다음 레벨에 존재하는 노드의 개수를 의미하므로, 다음 레벨을 다시 순회하는 과정에서 노드의 개수만큼 다시 부모 노드를 삭제하고 자식 노드를 삽입하는 과정을 반복한 후에 높이의 변수를 1씩 증가시키면 마지막 레벨까지 순회할 때까지, 즉 큐에 더 이상 삽입된 요소가 존재하지 않을 때까지, 전체 트리의 높이를 구할 수 있게 된다.

2021/10/06

자세한 내용은 소스 코드 분석 단계에서의 설명을 통해 더 세밀하게 살펴보자.

1.2 소스 코드

```

1  /*
2  → 작성자 : .이예빈 (20204059)
3  → 작성일 : .2021.10.05
4  → 프로그램명 : .파일의 정수 데이터들을 바탕으로, .반복과 순환을 이용하여, 이진 탐색 트리의 삽입 시간을 비교하고,
5  → → → → 완성된 트리 노드의 노드 개수, .단말 노드 개수, .높이를 구하는 프로그램
6  */
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <time.h>
11
12 #define TRUE 1
13 #define FALSE 0
14 #define MAX_QUEUE_SIZE 10000
15
16
17 typedef struct TreeNode {
18     int key;
19     struct TreeNode* left, *right;
20 }TreeNode;
21
22 typedef TreeNode* element; → // 트리 노드의 포인터
23 typedef struct QueueType {
24     element data[MAX_QUEUE_SIZE];
25     int front, rear; → // 전단과 후단
26     int size; → → → // 현재 큐에 삽입되어 있는 요소의 개수
27 }QueueType;
28
29 void error(char* message) {
30     fprintf(stderr, "%s\n", message);
31     exit(1);
32 }
33
34 void init_queue(QueueType* q) {
35     q->front = q->rear = 0;
36     q->size = 0;
37 }
38
39 int is_empty(QueueType* q) {
40     return (q->front == q->rear);
41 }
42
43 int is_full(QueueType* q) {
44     return ((q->rear + 1) % MAX_QUEUE_SIZE == q->front);
45 }
46
47 void enqueue(QueueType* q, element item) {
48     if (is_full(q))
49         error("큐가 포화상태입니다.");
50     q->rear = (q->rear + 1) % MAX_QUEUE_SIZE;
51     q->data[q->rear] = item;
52 }
53
54 element dequeue(QueueType* q) {
55     if (is_empty(q))
56         error("큐가 공백상태입니다.");
57     q->front = (q->front + 1) % MAX_QUEUE_SIZE;
58     return q->data[q->front];
59 }
60
61 // 노드 개수 (순환)
62 int get_node_count(TreeNode* node) {

```

```

63     → int.count += 0;
64     → if (node != NULL)
65     →     → count += 1 + get_node_count(node->left) + get_node_count(node->right);
66     → return count;
67     }
68
69     // .단말·노드·개수 (순환)
70     int get_leaf_count(TreeNode* node) {
71     →     int.count += 0;
72     →     if (node != NULL)
73     →     →     if (node->left == NULL && node->right == NULL) return 1;
74     →     →     else count += get_leaf_count(node->left) + get_leaf_count(node->right);
75     →     return count;
76     }
77
78     // .트리의·높이· (순환)
79     int get_height_recur(TreeNode* node) {
80     →     int.height = 0, left_count = 0, right_count = 0;
81     →     if (node != NULL) {
82     →     →     left_count += get_height_recur(node->left);
83     →     →     right_count += get_height_recur(node->right);
84     →     →     height = 1 + (left_count > right_count ? left_count : right_count);
85     →     }
86     →     return height;
87     }
88
89     // .트리의·높이· (레벨·순회·반복)
90     int get_height_iter(TreeNode* ptr) {
91     →     QueueType q;
92     →     int i, height = 0;
93     →     int count;
94
95     →     init_queue(&q);
96     →     if (ptr == NULL) return 0;
97
98     →     enqueue(&q, ptr); → // 루트·노드·enqueue
99     →     q.size++; → → → // 루트·노드·enqueue·결과·size·1·증가
100    →     while (!is_empty(&q)) {
101    →     →     count = q.size; → → → // 현재·큐에·삽입되어·있는·노드의·개수
102    →     →     for (i = 0; i < count; i++) { → // count만큼·반복
103    →     →     →     ptr = dequeue(&q); → → // 큐의·dequeue·반환·값을·ptr에·저장
104    →     →     →     q.size--; → → → // dequeue·결과·size·1·감소
105    →     →     →
106    →     →     →     // ptr의·자식·노드·큐에·삽입
107    →     →     →     if (ptr->left) {
108    →     →     →     →     enqueue(&q, ptr->left);
109    →     →     →     →     q.size++;
110    →     →     →     →     }
111    →     →     →     if (ptr->right) {
112    →     →     →     →     enqueue(&q, ptr->right);
113    →     →     →     →     q.size++;
114    →     →     →     →     }
115    →     →     →     }
116    →     →     height++; → → // count만큼·반복한·후, height·증가· (한·레벨의·순회·완료)
117    →     }
118    →     return height;
119    }

```



```

120
121 //새로운.트리.노드.동적.할당
122 □TreeNode*new_node(int·item)·{
123     →   TreeNode*temp·=·(TreeNode*)malloc(sizeof(TreeNode));
124     →   temp->key·=·item;
125     →   temp->left·=·temp->right·=·NULL;
126     →   return·temp;
127 }
128
129 //노드.삽입.(순환)
130 □TreeNode*insert_node_recur(TreeNode*node,·int·key)·{
131     →   if·(node·==·NULL)·return·new_node(key);
132
133     →   if·(key·<·node->key)
134     →       node->left·=·insert_node_recur(node->left,·key);
135     →   else·if·(key·>·node->key)
136     →       node->right·=·insert_node_recur(node->right,·key);
137     →   return·node;
138 }
139
140 //노드.삽입.(반복)
141 □void·insert_node_iter(TreeNode**root,·int·key)·{
142     →   TreeNode*·p,·*·t; → //p:·부모노드,·t:·현재노드
143     →   TreeNode*·n; → //n:·새로운.노드
144
145     →   t·=·*root;
146     →   p·=·NULL;
147     →   while·(t·!=·NULL)·{
148     →       →   if·(key·==·t->key)·return;
149     →       →   p·=·t;
150     →       →   if·(key·<·t->key)·t·=·t->left;
151     →       →   else·t·=·t->right;
152     →   }
153     →   //key가.트리.안에.없으므로.삽입
154     →   n·=·(TreeNode*)malloc(sizeof(TreeNode));
155     →   if·(n·==·NULL)·return;
156     →   n->key·=·key;
157     →   n->left·=·n->right·=·NULL;
158
159     →   //부모.노드와.링크.연결
160     →   n->key·=·key;
161     →   n->left·=·n->right·=·NULL;
162     →   if·(p·!=·NULL)
163     →       →   if·(key·<·p->key)·p->left·=·n;
164     →       →   else·p->right·=·n;
165     →   else·*root·=·n;
166 }

```

```

166 // 후위 순회를 이용해 전체 트리 메모리 해제
167 void clear(TreeNode* root) {
168     if (root == NULL) return;
169     clear(root->left);
170     clear(root->right);
171     free(root);
172 }
173
174 double insert_recur(FILE* fp) {
175     TreeNode* root = NULL;
176     int temp;
177     clock_t start, end;
178
179     start = clock();
180     while (!feof(fp)) {
181         fscanf(fp, "%d", &temp);
182         root = insert_node_recur(root, temp);
183     }
184     end = clock();
185
186     clear(root);
187     return ((double)end - (double)start);
188 }
189
190 double insert_iter(FILE* fp) {
191     TreeNode* root = NULL;
192     int temp;
193     clock_t start, end;
194
195     start = clock();
196     while (!feof(fp)) {
197         fscanf(fp, "%d", &temp);
198         insert_node_iter(&root, temp);
199     }
200     end = clock();
201     clear(root);
202     return ((double)end - (double)start);
203 }
204
205 void print_bar() {
206     printf("\n=====
207 }

```

```

208
209 int main() {
210     FILE* fp;
211     TreeNode* root = NULL;
212     int i = 0, temp;
213     double recur[10], iter[10]; // 순환, 반복을 이용한 삽입 시간 (10번 반복)
214     double sum_recur = 0, sum_iter = 0; // 10번 전체 시간의 합
215
216     fp = fopen("data.txt", "rt");
217     if (fp == NULL) {
218         fprintf(stderr, "파일 열기 실패\n");
219         exit(1);
220     }
221
222     // 순환, 반복적인 트리 삽입의 시간을 각각 10번씩 저장
223     for (i = 0; i < 10; i++) {
224         recur[i] = insert_recur(fp);
225         sum_recur += recur[i];
226         rewind(fp);
227         iter[i] = insert_iter(fp);
228         sum_iter += iter[i];
229         rewind(fp);
230     }
231
232     printf("횟수\n");
233     for (i = 1; i <= 10; i++) printf("%5d\n", i);
234     printf("평균\n");
235     print_bar();
236
237     printf("순환\n"); // 순환을 이용한 삽입
238     for (i = 0; i < 10; i++) {
239         printf("%7.2f\n", recur[i]);
240     }
241     printf("%7.2f\n", sum_recur / 10.0); // 순환 삽입 시간 평균
242     print_bar();
243
244     printf("반복\n"); // 반복을 이용한 삽입
245     for (i = 0; i < 10; i++) {
246         printf("%7.2f\n", iter[i]);
247     }
248     printf("%7.2f\n", sum_iter / 10.0); // 반복 삽입 시간 평균
249     print_bar();
250
251
252     // 반복을 이용한 삽입을 통해 얻은 트리의 노드 개수 / 단말 노드 개수 / 높이 구하기
253     printf("\n\n<반복을 이용한 삽입>\n\n");
254     while (!feof(fp)) {
255         fscanf(fp, "%d", &temp);
256         insert_node_iter(&root, temp);
257     }
258
259     printf("트리의 노드 개수: %d\n", get_node_count(root));
260     printf("트리의 단말 노드 개수: %d\n", get_leaf_count(root));
261     printf("트리의 높이 (반복): %d\n", get_height_iter(root));
262     printf("트리의 높이 (순환): %d\n", get_height_recur(root));
263
264     clear(root); // 트리 메모리 해제
265     fclose(fp); // 파일 닫기
266     return 0;
267 }

```

1.3 소스 코드 분석

```

/*
→  작성자::이예빈 (20204059)
→  작성일::2021.10.05
→  프로그램명::파일의 정수 데이터들을 바탕으로, 반복과 순환을 이용하여 이진 탐색 트리의 삽입 시간을 비교하고,
→  →  →  완성된 트리 노드의 노드 개수, 단말 노드 개수, 높이를 구하는 프로그램
*/

#include<stdio.h>
#include<stdlib.h>
#include<time.h>

#define TRUE 1
#define FALSE 0
#define MAX_QUEUE_SIZE 10000

```

1. 작성자, 작성일, 프로그램명을 주석에 작성하고, 필요한 헤더 파일들을 추가하고, 기호 상수들을 정의한다. 동적 할당에 필요한 표준 라이브러리 헤더 파일, 그리고 시각을 측정하기 위한 타임 헤더 파일을 추가한다. 기호 상수로는, 코드의 가독성을 위해 TRUE 와 FALSE 를 각각 1 과 0 으로, 그리고 큐의 요소의 최대 크기를 임의로 10000 으로 정의한다.

```

typedef struct TreeNode {
→  int key;
→  struct TreeNode* left, **right;
}TreeNode;

typedef struct QueueType {
→  element.data[MAX_QUEUE_SIZE];
→  int front, rear; →  //전단과 후단
→  int size; →  →  //현재 큐에 삽입되어 있는 요소의 개수
}QueueType;

```

2. 프로그램에서 사용되는 구조체들을 정의한다. 정수형 키 key 과 왼쪽, 오른쪽 자식 노드를 가리키는 포인터 left 와 right 멤버로 이루어진 이진 트리의 노드 TreeNode 를 정의하고, 그리고 이 TreeNode 의 포인터를 element 로 정의한다. 원형 큐의 구조를 나타내는 QueueType 는 MAX_QUEUE_SIZE(500)의 크기의 TreeNode 의 포인터인 element 형 배열, 전단과 후단을 의미하는 front 와 rear, 그리고 큐에 삽입되어 있는 요소의 개수를 저장하는 size 를 멤버로 가진다.

main 함수의 동작을 살펴보기 전, 먼저 프로그램에서 사용되는 각 함수들을 자세히 살펴보자.

```

void error(char* message) {
    fprintf(stderr, "%s\n", message);
    exit(1);
}

```

3. 파일 입출력 또는 동적 할당의 과정에서 오류가 난 경우, message 를 출력하고 프로그램을 종료하는 error 함수이다.

```

void init_queue(QueueType* q) {
    q->front = q->rear = 0;
    q->size = 0;
}

int is_empty(QueueType* q) {
    return (q->front == q->rear);
}

int is_full(QueueType* q) {
    return ((q->rear + 1) % MAX_QUEUE_SIZE == q->front);
}

void enqueue(QueueType* q, element item) {
    if (is_full(q))
        error("큐가 포화상태입니다.");
    q->rear = (q->rear + 1) % MAX_QUEUE_SIZE;
    q->data[q->rear] = item;
}

element dequeue(QueueType* q) {
    if (is_empty(q))
        error("큐가 공백상태입니다.");
    q->front = (q->front + 1) % MAX_QUEUE_SIZE;
    return q->data[q->front];
}

```

4. 위는 큐의 연산에 있어서 기본이 되는 함수들이다. 큐의 front 와 rear, size 를 초기화하는 init_queue 함수. 공백 또는 포화를 검출하는 is_empty 와 is_full 함수, 큐에 요소를 삽입하는 enqueue 함수, 큐로부터 요소를 삭제하여 반환하는 dequeue 함수이다. 그동안 원형 큐의 삽입은 많이 다루었기 때문에, 각 연산 동작에 대한 자세한 설명은 생략한다.

```

// 노드 개수 (순환)
int get_node_count(TreeNode* node) {
    int count = 0;
    if (node != NULL)
        count = 1 + get_node_count(node->left) + get_node_count(node->right);
    return count;
}

```

4. 순환의 방법을 이용해 노드의 개수를 구하는 `get_node_count` 함수이다. 트리 노드의 포인터를 매개 변수로 전달받고, 노드가 NULL 값을 가지지 않을 때까지, 각각 왼쪽, 오른쪽 자식 노드의 주소를 다시 `get_node_count` 함수의 인자로 전달하여 호출하기를 반복한다. 자기 자신(부모 노드)을 포함하여야 하기 때문에 1 을 더하도록 한다.

```
//.단말.노드.개수(순환)
int get_leaf_count(TreeNode* node) {
    int count = 0;
    if (node != NULL)
        if (node->left == NULL && node->right == NULL) return 1;
        else count = get_leaf_count(node->left) + get_leaf_count(node->right);
    return count;
}
```

5. 순환적인 방법을 이용해 단말 노드의 개수를 구하는 `get_leaf_count` 함수이다. 왼쪽, 오른쪽 자식 노드 모두 NULL 값인 경우 1 을 반환하여 count 값을 증가시킨다. 이를 노드가 NULL 이 아닐 때까지, 각 자식 노드를 인수로 전달하여 순환적으로 함수를 호출하기를 반복한다.

```
//.트리의.높이.(순환)
int get_height_recur(TreeNode* node) {
    int height = 0, left_count = 0, right_count = 0;
    if (node != NULL) {
        left_count = get_height_recur(node->left);
        right_count = get_height_recur(node->right);
        height = 1 + (left_count > right_count ? left_count : right_count);
    }
    return height;
}
```

6. 순환적인 방법으로 트리의 높이를 구하는 `get_height_recur` 함수이다. 트리 전체의 height, 그리고 왼쪽 서브 트리의 높이 `left_count` 와 오른쪽 서브 트리의 높이 `right_count` 을 각각 0 으로 초기화한다. 기존에는 `max` 함수를 이용하여 왼쪽 자식 노드를 인수로 전달한 함수와 오른쪽 자식 노드를 인수로 전달한 함수 중 얻게 되는 값이 더 큰 것을 height 에 더하는 함수였지만, 추가되는 연산을 피하기 위해 순환 호출하여 얻은 값들을 `left_count` 와 `right_count` 에 더한 후, 두 값 중 더 큰 것을 최종 height 에 저장하도록 한다.

아래는 반복적인 방법, 그 중에서도 원형 큐를 이용한 레벨 순회를 이용하여 트리의 높이를 구하는 `get_height_iter` 함수이다. 코드를 자세히 살펴보자.

```

//.트리의.높이.(레벨.순회-반복)
int get_height_iter(TreeNode* ptr){
    QueueType q;
    int i, height=0; //.반복문.제어.변수.i,.트리의.높이.height
    int count; //.큐에.삽입되어.있는.요소의.개수.임시.저장

    init_queue(&q);
    if(ptr==NULL) return 0;

```

8. 높이를 구할 트리 노드의 루트 노드 포인터 ptr 을 매개 변수로 전달받는다. QueueType 원형 큐의 q 를 선언한다. 아래에서 사용할 반복문의 제어 변수 i 를 선언하고, 트리의 높이 height 을 0 으로 초기화한다. 큐에 삽입되어 있는 요소의 개수를 임시 저장하여 반복문의 횟수를 제어할 count 변수를 선언한다.

```

    enqueue(&q, ptr); //루트.노드.enqueue
    q.size++; //루트.노드.enqueue.결과.size.1.증가

```

9. 우선, 원형 큐에 ptr 을 삽입한 후, q 의 size 멤버를 1 증가시킨다.

```

while(!is_empty(&q)){
    count=q.size; //현재.큐에.삽입되어.있는.노드의.개수
    for(i=0;i<count;i++){ //count만큼.반복
        ptr=dequeue(&q); //큐의.dequeue.반환.값을.ptr에.저장
        q.size--; //dequeue.결과.size.1.감소.
    }
}

```

10. 큐가 공백이 아닌 상태일동안 트리 노드의 height 를 구하는 while 반복문을 실행시킨다. while 반복문 내에서, 우선 count 변수에 q 의 size 멤버 변수의 값을 대입한다. 이후 for 반복문을 count 값만큼 실행시키도록 한다. (여기서, count 값은 이전 레벨의 자식 노드의 개수, 즉 현재 순회하는 레벨의 노드의 개수만큼 반복문이 실행된다는 의미이다.) 큐에 삽입되어 있는 요소를 삭제하고 반환하여 ptr 에 저장하도록 하고, q 의 size 멤버는 1 감소시킨다.

```

    //ptr의.자식.노드.큐에.삽입
    if(ptr->left){
        enqueue(&q, ptr->left);
        q.size++;
    }
    if(ptr->right){
        enqueue(&q, ptr->right);
        q.size++;
    }
    height++; //count만큼.반복한.후,.height.증가.(한.레벨의.순회.완료)
}
return height;
}

```

11. 이후 ptr 의 왼쪽, 그리고 오른쪽 자식 노드가 있는 경우 큐에 삽입하고, size 를 각각 증가시킨다.

12. 결국, for 문에서 count 만큼 반복을 한다는 것은 한 레벨을 순회하며 각 노드들을 큐에서 삭제하여 다시 그 노드의 자식 노드들을 큐에 삽입하는 것과 같다. 이러한 과정이 끝나면, 레벨의 모든 노드들을 순회한 것이고, 마지막으로 height 값을 1 증가시킨다.

```
//새로운.트리.노드.동적.할당
TreeNode* new_node(int item) {
    →  TreeNode* temp = (TreeNode*) malloc(sizeof(TreeNode));
    →  temp->key = item;
    →  temp->left = temp->right = NULL;
    →  return temp;
}

//노드.삽입. (순환)
TreeNode* insert_node_recur(TreeNode* node, int key) {
    →  if (node == NULL) return new_node(key);

    →  if (key < node->key)
    →  →  node->left = insert_node_recur(node->left, key);
    →  else if (key > node->key)
    →  →  node->right = insert_node_recur(node->right, key);
    →  return node;
}
```

13. 순환을 이용하여 노드를 삽입하는 insert_node_recur 함수, 그리고 새로운 트리 노드를 동적 할당하여 반환하는 new_node 함수이다. 루트 노드에서부터, key 값과 node 의 key 값을 비교하여 삽입할 위치를 순환 호출을 통해 오른쪽 또는 왼쪽 자식 노드를 따라가며 탐색하여 NULL 값에 도달하는 경우 동적 할당하여 얻은 새로운 노드의 주소 값을 해당 위치에 삽입한다. 마지막으로 변경된 루트 node 포인터를 반환한다.

다음은 반복을 이용하여 노드를 삽입하는 insert_node_iter 함수이다. 코드를 자세히 살펴보자.

```
//노드.삽입. (반복)
void insert_node_iter(TreeNode** root, int key) {
    →  TreeNode* p, *t; →  //p: 부모노드, t: 현재노드
    →  TreeNode* n; →  →  //n: 새로운.노드

    →  t = *root;
    →  p = NULL;
```

14. 트리 노드의 루트 포인터를 가리키는 포인터(이중 포인터) root 와 key 값을 매개변수로 전달받는다. 트리 노드 포인터 p, t, n 을 선언한다. t 는 현재 노드, p 는 부모 노드, 그리고 n 은

새로 삽입할 노드를 각각 가리킨다. t 에는 우선적으로, root 포인터 값을 대입하고, p 는 NULL 값으로 초기화한다.

```

→ while (t != NULL) {
→   → if (key == t->key) return;
→   → p = t;
→   → if (key < t->key) t = t->left;
→   → else t = t->right;
→ }

```

15. while 반복문은 t 가 NULL 에 도달할 때까지 실행된다. 삽입하고자 하는 key 값이 노드의 key 값과 동일하다면, 함수는 종료된다. 그런 경우가 아니라면, 부모 노드를 가리키던 p 는 t 를 가리키게 되고, t 는 key 값의 비교를 통해 왼쪽 또는 오른쪽 자식 노드로 이동한다. (결국 NULL 값에 도달한 경우는, key 가 트리 안에 없었으며, 삽입 위치를 탐색 완료한 경우이다.)

```

→ // key가 트리 안에 없으므로 삽입
→ n = (TreeNode*) malloc (sizeof (TreeNode));
→ if (n == NULL) return;
→ n->key = key;
→ n->left = n->right = NULL;

```

16. key 값이 트리 안에 없으므로 동적 할당을 하고, 삽입을 하도록 한다. 포인터 n 에 노드를 동적 할당하고, key 값을 대입하고 left 와 right 은 모두 NULL 로 초기화한다.

```

→ // 부모 노드와 링크 연결
→ if (p != NULL)
→   → if (key < p->key) p->left = n;
→   → else p->right = n;
→ else *root = n;
→ }

```

17. 이제, 부모 노드와 링크를 연결하도록 한다. t 의 부모 노드인 p 가 NULL 이 아니라면, key 값을 비교하여 p 의 왼쪽 또는 오른쪽 자식 노드 포인터에 n 의 주소를 대입하도록 한다. p 가 NULL 값인 경우라면 아직 트리에 어떠한 노드도 존재하지 않은 상태이므로, 루트 포인터에 새로운 노드 n 의 값을 대입한다.

이로써 key 값을 가진 새로운 하나의 노드를 트리에 삽입하는 과정이 끝난다.

```
// 후위 순회를 이용해 전체 트리 메모리 해제
void clear(TreeNode* root) {
    if (root == NULL) return;
    clear(root->left);
    clear(root->right);
    free(root);
}
```

18. 후위 순회를 하며 전체 트리의 노드들을 메모리 해제하는 clear 함수이다.

```
double insert_recur(FILE* fp) {
    TreeNode* root = NULL;
    int temp;
    clock_t start, end;

    start = clock();
    while (!feof(fp)) {
        fscanf(fp, "%d", &temp);
        root = insert_node_recur(root, temp);
    }
    end = clock();

    clear(root);
    return ((double)end - (double)start);
}
```

19. 파일 포인터 fp를 매개변수로 전달받아, 포인터가 가리키는 곳에서부터 정수 데이터를 읽어 순환을 이용하여 트리에 노드를 삽입한 뒤, 전체 삽입에 걸린 시간을 반환하는 insert_recur 함수이다. 트리의 루트 노드 포인터 root를 NULL 값으로 초기화하여 선언하고, 파일에서 읽는 정수를 임시로 저장할 temp 변수, 그리고 clock 함수가 호출된 시각을 저장하는 clock_t 형 변수 start와 end를 선언한다.

파일의 끝에 도달할 때까지 정수를 읽고, insert_node_recur 함수를 호출하여 이진 트리에 삽입한다. 이 과정이 시작하기 전 시각을 start에, 그리고 과정이 끝난 시각을 end에 저장한다. 완성된 트리는 clear 함수에 전달하여 메모리를 해제하도록 하고, 끝 시각 end에서 처음 시각 start를 뺀 값을 반환한다.

```

double insert_iter(FILE* fp) {
    →  TreeNode* root := NULL;
    →  int temp;
    →  clock_t start, end;

    →  start = clock();
    →  while (!feof(fp)) {
    →      →  fscanf(fp, "%d", &temp);
    →      →  insert_node_iter(&root, temp);
    →      →  }
    →  end = clock();
    →  clear(root);
    →  return ((double)end - (double)start);
}

```

20. 반복을 이용한 트리 삽입에 걸린 시간을 구하는 insert_iter 함수이다. 트리에 삽입하기 위해 호출하는 함수가 insert_node_iter 로 바뀐 것을 제외하고는, 삽입에 소요되는 시간을 구하는 방법은 위의 insert_recur 함수의 동작 방식과 동일하다.

```

void print_bar() {
    →  printf("\n=====
}

```

21. 매우 긴 구분선을 출력하는 print_bar 함수이다.

이제 마지막으로, main 함수의 동작을 살펴보자.

```

int main() {
    →  FILE* fp;
    →  TreeNode* root := NULL;
    →  int i = 0, temp;
    →  double recur[10], iter[10]; → // 순환, 반복을 이용한 삽입 시간 (10번 반복)
    →  double sum_recur = 0, sum_iter = 0; → // 10번 전체 시간의 합

    →  fp = fopen("data.txt", "rt");
    →  if (fp == NULL) {
    →      →  fprintf(stderr, "파일 열기 실패\n");
    →      →  exit(1);
    →  }
}

```

22. 파일 포인터 fp, NULL 값으로 초기화한 트리 노드 포인터 root, 반복문 제어 변수 i와 임시로 정수 데이터를 저장할 temp 를 선언한다. 그리고 순환과 반복을 이용한 트리 삽입에 걸리는 시간을 double 형(시간의 단위: ms 밀리초)으로 저장하기 위해, 10 의 크기의 recur 과 iter 배열을 선언한다. 평균 값을 구하기 위해, 10 번의 double 형 시간을 모두 더하여 저장할 sum_recur 과 sum_iter 을 0 으로 초기화하여 선언한다.

23. 32,500 개의 정수 데이터가 저장되어 있는 data.txt 파일을 읽기 모드로 연다.

```
→ //·순환,·반복적인·트리·삽입의·시간을·각각·10번씩·저장
→ for·(i·=·0;i·<·10;i++)·{
→   recur[i]·=·insert_recur(fp);
→   sum_recur·+=·recur[i];
→   rewind(fp);
→   iter[i]·=·insert_iter(fp);
→   sum_iter·+=·iter[i];
→   rewind(fp);
→ }

→ printf("·횟수·|");
→ for·(i·=·1;i·<=·10;i++)·printf("%5d···|",·i);
→ printf("··평균··|");
→ print_bar();
```

24. 순환과 반복의 방법을 사용한 트리 삽입에 걸리는 시간을 총 10 번 구하여 그 값들을 각각 recur, iter 배열에 저장하도록 한다. sum_recur 과 sum_iter 에 그 값들을 누적하여 합한다. 각 삽입이 끝난 후에는 rewind 함수를 이용해 파일 포인터를 다시 파일의 처음으로 이동시킨다.

25. 한눈에 결과를 볼 수 있도록, 양식에 맞추어 "횟수"와 "평균", 그리고 1 부터 10 까지의 정수를 출력한다.

```
→ printf("·순환·|"); → //·순환을·이용한·삽입
→ for·(i·=·0;i·<·10;i++)·{
→   printf("%7.2f·|",·recur[i]);
→ }
→ printf("%7.2f·|",·sum_recur·/·10.0); → //·순환·삽입·시간·평균
→ print_bar();

→ printf("·반복·|"); → //·반복을·이용한·삽입
→ for·(i·=·0;i·<·10;i++)·{
→   printf("%7.2f·|",·iter[i]);
→ }
→ printf("%7.2f·|",·sum_iter·/·10.0); → //·반복·삽입·시간·평균
→ print_bar();
```

26. 순환을 이용한 삽입을 10 번 반복하여 얻은 시간의 값들이 저장되어 있는 recur 배열의 값들을 모두 양식에 맞추어 출력하도록 한다. sum_recur 을 10 으로 나누어 얻는 평균 값 또한 출력한다. 반복을 이용한 삽입의 시간 값들 또한 동일하게 출력한다.

```

→ //반복을 이용한 삽입을 통해 얻은 트리의 노드 개수 / 단말 노드 개수 / 높이 구하기
→ printf("\n\n<반복을 이용한 삽입>\n\n");
→ while(!feof(fp)){
→     fscanf(fp, "%d", &temp);
→     insert_node_iter(&root, temp);
→ }

→ printf("트리의 노드 개수: %d\n", get_node_count(root));
→ printf("트리의 단말 노드 개수: %d\n", get_leaf_count(root));
→ printf("트리의 높이 (반복): %d\n", get_height_iter(root));
→ printf("트리의 높이 (순환): %d\n\n", get_height_recur(root));

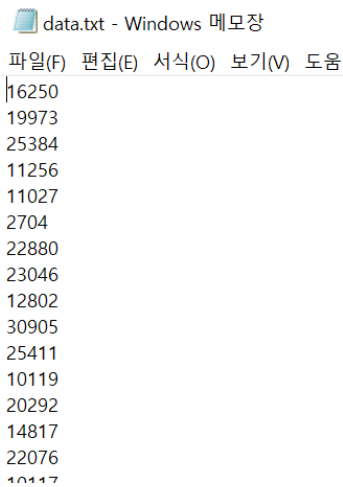
→ clear(root); → //트리 메모리 해제
→ fclose(fp); → //파일 닫기
→ return 0;
}

```

26. 위에서 반복과 순환적인 삽입에 걸리는 시간을 비교하였고, 결과 값에 의해 삽입의 시간이 더 적게 걸리는 '반복'의 방법을 선택하여 다시 파일의 데이터들을 바탕으로 이진 트리를 완성한다. 완성된 이진 트리를 바탕으로, 트리의 노드 개수, 단말 노드 개수, 그리고 반복/순환적인 방법으로 얻는 트리의 높이를 차례대로 출력한다.

27. 마지막으로 clear 함수로 트리의 모든 메모리를 해제하고, 파일을 닫고 프로그램 전체를 종료한다.

1.4 실행창



```

data.txt - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움
16250
19973
25384
11256
11027
2704
22880
23046
12802
30905
25411
10119
20292
14817
22076
10117
  
```

과제로 주어진 "data.txt" 파일에는 16,250의 정수부터 총 32,500 개의 정수 데이터가 저장되어 있다. 이 정수 데이터들을 바탕으로 얻은 실행 결과들은 다음과 같다.

순환과 반복을 각각 10번 실행하여 얻은 결과들(총 각각 40번), 그리고 각 10번에 대한 평균 값이 나타나 있다.

반복을 이용한 삽입으로 만들어진 이진 트리의 노드 개수, 단말 노드 개수, 높이가 나타나 있다.

결과 1

Microsoft Visual Studio 디버그 콘솔												
횟수	1	2	3	4	5	6	7	8	9	10	평균	
순환	132.00	271.00	174.00	222.00	116.00	102.00	93.00	89.00	85.00	135.00	141.90	
반복	86.00	177.00	191.00	98.00	86.00	74.00	84.00	94.00	78.00	103.00	107.10	

<반복을 이용한 삽입>												
트리의 노드 개수 :	32500											
트리의 단말 노드 개수 :	10843											
트리의 높이(반복) :	34											

결과 2

Microsoft Visual Studio 디버그 콘솔												
횟수	1	2	3	4	5	6	7	8	9	10	평균	
순환	186.00	164.00	123.00	121.00	109.00	105.00	127.00	405.00	136.00	126.00	160.20	
반복	82.00	101.00	120.00	85.00	87.00	107.00	132.00	194.00	69.00	127.00	110.40	

<반복을 이용한 삽입>

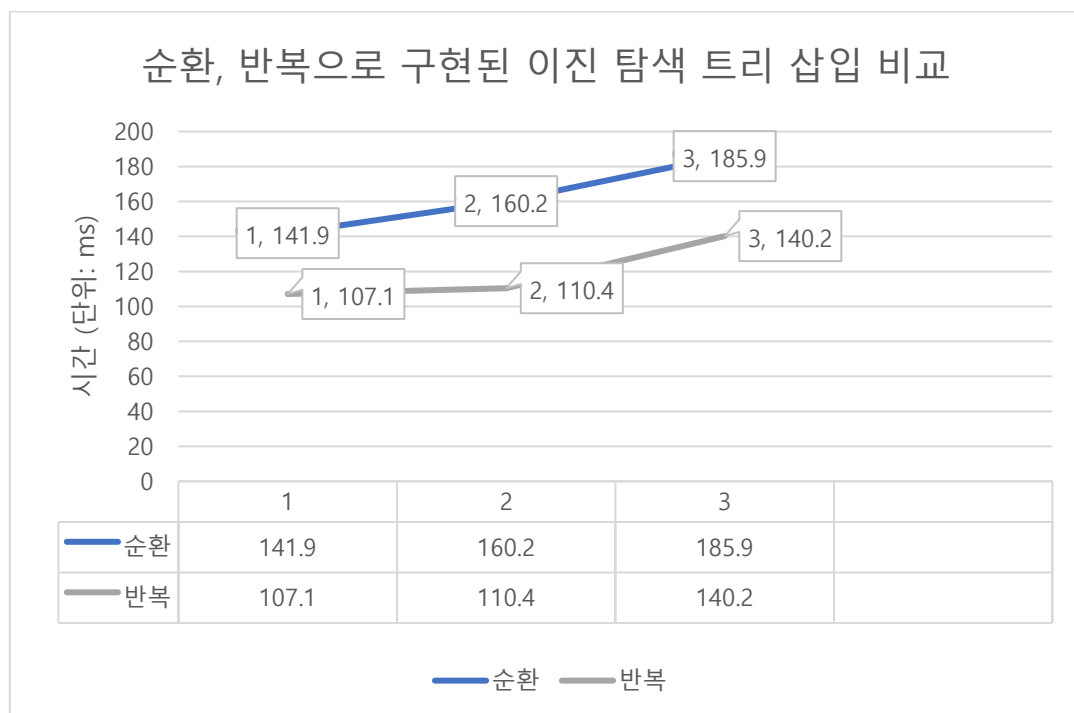
트리의 노드 개수 : 32500
 트리의 단말 노드 개수 : 10843
 트리의 높이(반복) : 34

결과 3

Microsoft Visual Studio 디버그 콘솔												
횟수	1	2	3	4	5	6	7	8	9	10	평균	
순환	113.00	173.00	130.00	426.00	294.00	192.00	108.00	181.00	117.00	125.00	185.90	
반복	87.00	157.00	117.00	239.00	181.00	159.00	144.00	152.00	86.00	80.00	140.20	

<반복을 이용한 삽입>

트리의 노드 개수 : 32500
 트리의 단말 노드 개수 : 10843
 트리의 높이(반복) : 34



위의 세 결과들을 바탕으로, 순환과 반복적인 삽입의 평균값들을 표로 나타냈다. 순환을 이용한 삽입의 시간이 모두 오래 걸렸음을 확인할 수 있다.

아래는 기존 교재에 나타나 있는 순환의 방법을 이용하여 얻은 트리의 높이에 대한 결과도 추가하였다. 반복, 순환의 방법을 통해 얻은 높이의 값이 동일함을 확인하였고, 이로써 트리의 높이를 반복을 이용해 구하는 방법을 검증하였다.

Microsoft Visual Studio 디버그 콘솔

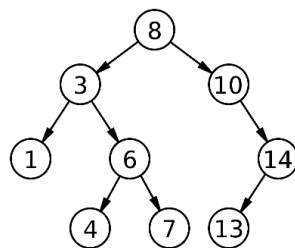
횟수	1	2	3	4	5	6	7	8	9	10	평균
순환	182.00	115.00	170.00	178.00	164.00	91.00	91.00	97.00	104.00	93.00	128.50
반복	70.00	129.00	198.00	115.00	71.00	70.00	73.00	79.00	78.00	95.00	97.80

<반복을 이용한 삽입>

트리의 노드 개수 : 32500
 트리의 단말 노드 개수 : 10843
 트리의 높이(반복) : 34
 트리의 높이(순환) : 34

30,000개가 넘는 대량의 정수 데이터들을 트리에 삽입하고, 트리에 대한 정보를 얻은 것이므로 한 눈에 트리의 정보들을 쉽게 파악할 수 있는 적은 데이터를 사용하여 확실히 검증해보고자 하였다.

매우 소량의 9개의 정수 데이터로 만들어진 다음 이진 트리의 단말 노드 개수는 4개이고 트리의 높이는 4이다.



다음 결과 실행창에서 트리의 정보들을 정확하게 구하여 출력한 것을 확인할 수 있다.

결과 1

Microsoft Visual Studio 디버그 콘솔

횟수	1	2	3	4	5	6	7	8	9	10	평균
순환	0.00	1.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.30
반복	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.10

<반복을 이용한 삽입>

트리의 노드 개수 : 9
 트리의 단말 노드 개수 : 4
 트리의 높이(반복) : 4
 트리의 높이(순환) : 4

결과 2

Microsoft Visual Studio 디버그 콘솔

횟수	1	2	3	4	5	6	7	8	9	10	평균
순환	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.10
반복	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.10

<반복을 이용한 삽입>

트리의 노드 개수 : 9
 트리의 단말 노드 개수 : 4
 트리의 높이(반복) : 4
 트리의 높이(순환) : 4

결과 3

Microsoft Visual Studio 디버그 콘솔

횟수	1	2	3	4	5	6	7	8	9	10	평균
순환	0.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.20
반복	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

<반복을 이용한 삽입>

트리의 노드 개수 : 9
 트리의 단말 노드 개수 : 4
 트리의 높이(반복) : 4
 트리의 높이(순환) : 4

위의 결과들은, 역시나 순환적인 방법을 이용한 삽입의 시간이 반복적인 방법보다 오래 걸린다는 것을 확인할 수 있다. 다만, 매우 적은 데이터들을 삽입했기 때문에, 결과 2와 같은 경우 반복과 순환의 방법에 차이가 나지 않았다. 따라서, 데이터의 수가 많아질수록 순환의 삽입과 반복의 삽입을 하는데 걸리는 시간의 차이가 더욱 커진다는 것을 확인할 수 있다.

트리의 높이를 구하는 과정에서 반복과 순환적인 방법의 시간 차이

```
start=clock();
printf("트리의 높이(반복) : \t%d\n", get_height_iter(root));
end=clock();
printf("시간 : %f\n\n", (double)end-(double)start);

start=clock();
printf("트리의 높이(순환) : \t%d\n", get_height_recur(root));
end=clock();
printf("시간 : %f\n", (double)end-(double)start);
```

추가적으로 반복과 순환을 이용하여 트리의 높이를 구하는 함수를 호출하기 전과 호출하기 이후의 시각을 구하여 각 방법에 소요된 시간을 구해 보았다.

결과 1

```
=====|

<반복을 이용한 삽입>

트리의 노드 개수 : 32500
트리의 단말 노드 개수 : 10843

트리의 높이(반복) : 34
시간 : 7.000000

트리의 높이(순환) : 34
시간 : 7.000000
```

결과 2

```
=====|

<반복을 이용한 삽입>

트리의 노드 개수 : 32500
트리의 단말 노드 개수 : 10843

트리의 높이(반복) : 34
시간 : 7.000000

트리의 높이(순환) : 34
시간 : 6.000000
```

결과 3

```
=====|

<반복을 이용한 삽입>

트리의 노드 개수 : 32500
트리의 단말 노드 개수 : 10843

트리의 높이(반복) : 34
시간 : 12.000000

트리의 높이(순환) : 34
시간 : 8.000000
```

2021/10/06

삽입 연산에 비해 높이를 구하는 연산은 소요된 시간이 비교적으로 매우 적었지만, 레벨 순회를 이용한 반복적인 방법이 순환적인 방법보다 시간이 더 오래 걸린다는 것을 확인할 수 있었다.

1.5 느낀점

이번 과제를 통해 이진 탐색 트리의 삽입 구현에 있어서 순환적인 방법의 구현이 단순하지만, 반복적인 방법보다 효율적이지 않은 방법이라는 것을 직접 시간 계산을 해봄으로써 확실히 알게 되었다. 특히 데이터의 개수가 많아지며 트리의 깊이가 매우 깊어질수록 시간 복잡도의 차이가 확연히 드러남을 것을 볼 수 있었다. 반면, 기존의 코드를 수정하여 순환을 이용해 트리의 높이를 구하는 방법이 레벨 순회를 통한 반복적인 방법으로 높이를 구하는 것보다 시간이 단축된다는 것을 확인할 수 있었다. 따라서 적절한 상황에 적절한 방법을 이용하여 시간의 효율을 고려한 프로그램을 만드는 것이 중요하다는 것을 깨달았다. 특히 데이터의 개수가 많아질수록 시간의 효율의 차이는 반복과 순환적인 방법에 있어서 차이가 커지므로, 더 좋은 방법을 택하는 것이 프로그램의 효율에 많은 기여를 할 것 같다.