# 이진 탐색 트리 높이에 대한 분석

B911061 노현근

# 목차

# I. 이진 탐색 트리의 평균 높이에 대한 의문

이진 탐색 트리는 이진 트리와 같이 두 개의 자식 노드를 갖지만, 데이터를 넣는 과정에서 다른 면모를 보인다. 이진 트리의 경우 데이터를 넣는 방식에 정해진 규칙이 없지만, 이진 탐색 트리의 경우 부모 노드보다 작은 값들은 모두 왼쪽 노드에 들어가고, 큰 값들은 모두 오른쪽 노드에 들어간다. (작은 값이 오른쪽에 들어갈 수 도 있다. 왼쪽과 오른쪽에 어떤 값이 들어갈 지는 정하기 나름이지만, 통상적으로 왼쪽에 더 작은 값을 넣는다.)

값이 들어가는 규칙이 정해졌지만 이 규칙대로 트리에 값을 넣게 되면 값들이 순서대로 들어간다는 보장이 사라진다. 한 층을 다 채우기 전에 다음 층으로 들어갈 수도 있게 되어 항상 최소 높이가 나오지 않게 된다. 정말 극단적인 경우, 각 층에 하나의 노드만 채워질 수 도 있게 된다. 값이 들어오는 순서에 따라 높이가 항상 같지 않기에 이진 탐색 트리의 높이를 특정해서 기술하지 않고, 높이의 평균값을 기술한다.

이진 탐색 트리의 높이의 평균적인 시간복잡도를 나타낼 때, 노드의 개수를 n이라고 하면 평균적으로 O(lg(n))의 시간복잡도를 갖는다고 나타낸다. 하지만 이진 탐색 트리의 최소 높이를 구해보면 똑같이 log(n)이 나온다. 평균 값이 최소 값이라는 결과가 나오는 건 앞의 전제에 따르면 말이 안되는 상황이다. 이진 탐색 트리는 언제나 최소 높이가 나오지 않아서 평균을 구했던 것인데, 평균 값이 최소 높이라는 것은 수학적으로 말이 되지 않는 상황이다.

사실 위에서의 계산에는 약간의 트릭이 있다. 바로 둘의 값은 같다고 할 수 없기 때문이다. 시간복잡도로 나타낸 이진 탐색 트리의 높이의 경우, O(lg(n)) 이라고 기술했지만 실제 값이 log(n)이라고 할 수 없다. Big – O notation은 주로 최고 차 항 만을 다루기 때문에 O notation안에 기술된 값이 실제 값이라고 하기 힘들다.

그렇다면 이진 탐색 트리의 평균 높이는 실제로 어느 정도 될까? 그 해답을 얻기 위해 무작위 N개의 값이 들어간 트리 높이의 평균을 구하는 프로그램을 만들었다.

# II. 실험 프로그램 구성

이진 탐색 트리의 높이가 평균적으로 어느정도 값이 나오는 지를 확인해보기 위해, 무작위로 생성된 50, 100, 200 그리고 500개의 노드로 구성된 이진 탐색 트리 100만개의 평균 높이를 구하는 프로그램을 만들어 확인하기로 했다.
트리를 100만개 만들고 만들어진 각 트리의 높이를 기록하는 프로그램을 만들기 위해 아래와 같이 라이브러리를 사용하고, 코드를 작성했다. 여러 개의 파일로 나누어 작성을 하여 파일 단위로 왜 이렇게 구성했는지 코드 설명을 첨부했다.

## II – I. 헤더 파일(tree_height.h)

여러 파일로 나누어 프로그램을 만들었기에 이들을 이어주기 위해 헤더 파일을 생성했다. 가장 먼저 아래와 같이 스탠다드 라이브러리들과 추가로 사용할 그래픽 라이브러리 헤더들을 추가해주었다.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "mlx/mlx.h"
```

stdio.h 헤더 파일은 입력과 출력을 위해 넣었고, stdlib.h 헤더 파일은 rand()함수를 위해 넣었다. 이번 과제에서는 동적 할당 없이 POOL 배열을 만들어서 이를 동적 할당한 것처럼 사용하긴 하지만, 트리 높이 결과를 저장하는 배열을 메모리 효율적이게 구성하고 싶어 calloc() 함수를 사용했다. 따라서 stdlib.h 헤더 파일은 rand()와 calloc()을 사용하기 위해 넣었다.

다음으로 time.h 헤더 파일은 rand()함수의 seed를 매번 무작위로 바꿔주기 위해 넣었다. rand()함수를 srand()없이 그냥 사용하게 되면 프로그램을 몇번을 돌려도 랜덤 함수에서 매번 고정된 무작위 값을 출력하게 된다. 매 실행 마다 다른 결과를 보고 싶었기에, srand()를 사용하여 seed를 변경해주었다.

math.h 헤더 파일의 경우 log2함수를 사용하기 위해 넣었다. 사실 이 함수는 리눅스에서는 정의 되어 있지 않은 함수이기에 윈도우나 맥 환경에서만 동작한다. 하지만 다른 곳에 배포하기 위해 만든 프로그램이 아니라 그저 최소 높이의 값이 맞는지 개인 PC에서 확인하기 위해 만든 프로그램이기에 다른 플랫폼은 고려하지 않고 사용했다.

mlx/mlx.h 헤더 파일의 경우 맥에서 간단하게 그래픽을 송출할 수 있게 해주는 라이브러리이다. openGL이나, OpenCV, SDL2 같은 다른 그래픽 라이브러리가 많이 있지만, 다른 그래픽 라이브러리의 경우 C++을 사용해야하기도 하고, 간단히 막대그래프를 만들기 위해 사용하기에는 과도한 것 같아 간단한 라이브러리를 사용했다. 이 라이브러리는 C 언어와 Swift, 그리고 Objective-C로 작성되어 있지만, 실제 사용에서는 C언어를 사용해야하므로 적절하다고 판단하여 사용하였다. 하지만 C 언어를 사용한다고는 하지만 맥에서만 사용할 수 있다.

위와 같이 헤더 파일을 추가한 후, 500개 까지 사용할 수 있게 헤더파일에서 POOL_SIZE를 500개로 늘렸다. 그리고 구조체를 숫자와 왼쪽, 오른쪽 구조체 포인터가 들어간 num 구조체를 정의했다. num 구조체의 경우 자주 사용될 거 같아서 num 타입으로 정의했다.

```c
#define POOL_SIZE 500

struct num {
    int num;
    struct num *left;
    struct num *right;
}typedef num;
```

헤더 파일에는 다른 함수들과 또 다른 구조체인 all 구조체가 있다. all 구조체의 경우, 뒤에 나올 다른 함수들 에게 인자를 간단하게 건네주기 위해 만들었다.

이렇게 많은 변수들을 한 구조체에 넣고 다른 함수들의 매개변수로 사용을 한 행위는 사실 굉장히 비효율적인 구성이라고 할 수 있다. 구조체의 경우 가장 큰 변수의 크기를 기준으로 패딩을 하여 메모리를 기존보다 더 쓰는 현상이 있다. 이런 현상을 안다면 사실 더더욱 이렇게 프로그래밍을 하면 안되지만, 앞에서 언급했듯이 배포할 프로그램도 아니고, 확인을 위해 작성한 프로그램이기에 빠르게 만들기 위해 이렇게 작성했다.

```c
struct all {
    void *mlx;
    void *win;
    void *img;
    char *addr;
    int bpp;
    int line_length;
    int endian;
    num pool[POOL_SIZE];
    num *top;
    num *data;
    int *result;
    int loop_time;
    int N;
    int min_height;
    int average;
    int max_draw;
    int draw_flag;
    int limit_flag;
    int max_value;
}typedef all;
```

mlx 라이브러리를 쓰기 위해, mlx 포인터부터 endian 변수까지 만들었다. 그리고 아래의 변수들은 전역 변수로 프로그래밍하고 싶지 않아서 all 구조체에 pool[]배열과 트리에 관련된 정보들을 넣었다.

각 종류별로 구조체를 분할하면 더 보기 좋았겠지만, 역시나 더 빠르게 작성하려고 구분없이 한 구조체에 다 넣어서 프로그래밍했다. all 구조체도 많이 사용되어서 하나의 타입으로 정의했다.

이 아래에는 프로그램에 사용된 다른 함수들을 파일 이름에 맞춰 구분해 놓았다. 프로그램 작성 중에 다른 오류가 많이 발생했는데, 어떤 파일에 어떤 함수가 있는 지 적어 놓음으로써 더 빠르게 필요한 파일을 고칠 수 있게 되었다.

```
/*about node*/
void        init_pool(all *t);
struct num  *new_node(all *t);

/*about tree*/
void        add(int n, all *t);
int         height(num *t);

/*about random loop*/
int         add_and_height(all *t);
void        loop(all *t);

/*about draw*/
void        check(all *t);
void        draw_line(all *t, int n, int x, int y, int width, int color);
void        draw(all *t);

/*about itoa*/
char        *itoa(int value, char *buffer, int radix);
```

## II - II. 노드 코드(node.c)

node.c 파일에는 init_pool(all *t) 함수와 new_node(all *t) 함수를 정의했다. 프로그램 전체 부분에서 노드를 할당하고 생성하는 부분을 모아놓았다.

init_pool(all *t) 함수의 경우, 이전에 교수님께서 작성하신 함수를 가져와 작성했다. 작동 방식은 같으나, all 구조체에 맞게 변수 이름을 조금 수정해주었다. 기존에는 pool[] 배열이 전역 변수로 선언 되어있어 매개변수로 넣어주지 않아도 사용할 수 있었지만, 지금 이 프로그램에서는 전역 변수가 아닌 구조체의 변수로 들어가 있기 때문에 구조체 안에 있는 것을 빼서 사용하는 식으로 코드를 구성했다.

```
void init_pool(all *t) {
    int i;
    struct num *r = t->pool;
    struct num *s;

    t->pool[POOL_SIZE - 1].right = NULL;
    for (i = 1; i < POOL_SIZE; i++) {
        s = r++;
        s->right = r;
    }
    t->top = t->pool;
}
```

기존의 POOL_SIZE는 10으로 설정 되어있어 프로그램에 큰 부담이 없었지만, 이번 실험에 맞게 500으로 고치면서 프로그램에 부담이 크게 가해졌다. 트리를 초기화하는 코드를 짜지 않고 init_pool(all *t)함수를 실행시키면서 트리를 초기화 시키게 했는데, 단순히 100만번을 돌리게 된다고만 해도 단순 초기화 만을 위해 50억번 연산하게 된다. 하지만 따로 초기화 하는 코드를 작성하여 초기화를 관리한다고 해도, init_pool(all *t) 함수와 비슷하게 동작하기 때문에 별도의 초기화 코드는 만들지 않았다.

다음으로는 new_node(all *t) 함수를 넣었는데, 이 역시 교수님께서 작성하신 함수를 가져와서 일정 부분을 수정하여 사용했다. 그냥 사용하게 되면 매개변수 타입이 맞지 않을 뿐더러, 현재는 pool[]의 가장 윗부분을 가리키는 top변수가 전역 변수가 아니므로 현 상황에 맞게 변수 이름을 수정해주었다.

```
num *new_node(all *t) {
    num *r;

    if (t->top == NULL)
        return NULL;

    r = t->top;
    t->top = r->right;
    return r;
}
```

노드 관련 코드는 간단하게 해결되었다.

## II – III. 트리 코드(tree.c)

tree.c 파일에는 add(all *t) 함수와 height(all *t) 함수를 배치했다. 둘 다 트리와 관련된 함수로 add(int n, all *t) 함수는 트리의 구성을, height(all *t) 함수는 트리의 높이 정보 반환을 맡았다.

add(int n, all *t)함수는 이진 탐색 트리의 특징대로 루트 노드보다 작으면 왼쪽, 크면 오른쪽 자식으로 가게끔 설계했다. 이전에 만들었던 함수를 거의 그대로 사용했는데, 이 역시 위의 노드 함수들과 같이 all 구조체에 맞게 변수 이름과 타입을 바꿔줘야 했다.

기존의 add()함수와 다른 점은 노드에 들어가는 정보의 종류가 달라 compare() 함수 없이 단순 숫자를 비교해도 된다는 것이다. 전화번호부 프로그램에서는 노드에 char name[3] 변수와 char number[4] 변수가 있어, name[3] 변수를 비교할 때 별도로 생성한 compare() 함수를 이용해야 했다. 하지만 지금 이 실험은 데이터의 저장이 목적이 아니라 트리의 높이가 목적이라 내부 데이터로 int 변수를 넣어 더 빠르게 구동될 수 있도록 했다.

```
void add(int n, all *t) {
    struct num *new;
    struct num *current;
    struct num *parent;
    int i;

    parent = NULL;
    current = t->data;
    while (current)
    {
        parent = current;
        if (n <= parent->num)
            current = parent->left;
        else
            current = parent->right;
    }
    new = new_node(t);
    if (!new) {
        return ;
    } else {
        new->num = n;
        new->left = NULL;
        new->right = NULL;
        if (parent) {
            if (n <= parent->num)
                parent->left = new;
            else
                parent->right = new;
        }
        else
            t->data = new;
    }
}
```

다음에는 height(num *t) 함수를 구성했다. height(num *t) 함수의 경우 이전 함수들과는 달리 매개변수로 num 구조체가 들어간다. 루트 노드만 넣어도 그 외의 것들을 구할 수 있기 때문에 num 구조체

만 넣었다.

height(num *t) 함수는 재귀 함수로 구성되어있고, 왼쪽과 오른쪽에 재귀적으로 트리의 단말 노드까지 방문하여 더 큰 값을 더하여 다시 올라오는 식으로 구성되어있다. 재귀 방식을 이용하면 함수 호출 시간이 있어 너무 많이 호출이 되면 시간이 더 오래 걸리는 단점이 있지만, 이 실험에서는 재귀가 호출되는 횟수가 500번으로 제한이 되어있기 때문에(노드가 500개라 그 이상으로 호출되지 않는다.) 시간적으로는 크게 지장이 없다. 하지만 100만번 돌리게 되면 시간이 누적되어 오래 걸리게 된다.

이 부분을 최적화 하기 위해 다른 방식을 생각해보았지만, 다른 방법은 재귀 함수처럼 함수 구성이 깔끔하지 않기도 하고, 시간적으로 크게 차이가 없을 거라고 판단되어 이 함수로 최종 결정하였다.

```
int height(num *t) {
    int left;
    int right;

    if (!t)
        return -1;
    else {
        left = height(t->left);
        right = height(t->right);
        return 1 + (left > right ? left : right);
    }
}
```

## II – IV. 랜덤 반복 코드(random_loop.c)

random_loop.c 파일이 이번 실험의 핵심이라고 할 수 있다. 이번 실험은 100만개의 표본을 구해 공식과 실제가 같은 지 확인하는 실험이므로 100만번 반복하여 표본을 구해야 한다. 따라서 무작위 값 반복 함수가 굉장히 중요한 역할을 한다. 그래서 이 파일에는 무작위 값들로 트리를 만들고 그 높이를 반환하는 add_and_height(all *t) 함수와 이를 반복해주는 loop(all *t) 함수를 배치했다.

add_and_height(all *t) 함수는 단순히 노드 개수(N)만큼 add(int n, all *t) 함수를 반복하여 트리를 만들고, 그 트리의 높이를 반환하는 함수다. 앞의 tree.c 파일의 함수들을 응용한 함수라고 할 수 있다.

이 함수를 한 번 실행 시키면 무작위 노드 N개로 구성된 트리의 높이가 하나 나오게 된다. 우리는 이 결과물이 100만개 필요하기 때문에 이를 반복시킬 또 다른 함수를 만들었다.

```
int add_and_height(all *t) {
    int i;

    for (i = 0; i < t->N; i++) {
        add(rand(), t);
    }
    return height(t->data);
}
```

그 함수가 loop(all *t) 함수이다. loop(all *t) 함수는 main()에서 입력 받은 반복 횟수 만큼 add_and_height(all *t) 함수를 반복시켜준다. 매 실행 마다 나온 결과를 배열에 저장을 해준다. 100만번 실행이 끝이 나면, 마지막으로 배열에 저장된 값을 토대로 평균 높이를 구한다. 정수 값으로 저장하기 때문에 자동으로 floor()함수가 적용된다고 할 수 있다.

```
void loop(all *t) {
    int i;
    int sum;

    sum = 0;
    t->result = calloc(t->N, sizeof(int));
    for (i = 0; i < t->loop_time; i++) {
        init_pool(t);
        t->data = NULL;
        t->result[add_and_height(t)]++;
    }
    for (i = 0; i < t->N; i++) {
        sum += t->result[i] * i;
    }
    t->average = sum / t->loop_time;
}
```

이 두 함수를 통해 트리의 높이 분포와 평균 높이를 구할 수 있다. 사실 여기서 배열의 값과 평균 값을 출력하면 높이 분포와 평균 높이를 구할 수 있다. 하지만 이를 숫자로만 보게 되면 데이터를 한 눈에 보기 힘들고, 흥미가 떨어질 거 같아 이를 시각적으로 볼 수 있게 그래픽 코드를 추가하였다.

## II - V. 그래픽 코드(draw.c)

draw.c 파일은 시각적으로 결과를 보고 싶다는 생각으로 넣었다. 1080 x 720 크기의 화면을 바탕으로 결과를 시각적으로 그리게 했다. 내부에는 check(all *t), my_mlx_pixel_put(all *t, int x, int y, int color), draw_line(all *t, int n, int x, int width, int color) 그리고 draw(all *t)함수들이 들어있다. 결과들을 픽셀 단위로 화면에 출력해주는 역할을 맡았다.

check(all *t) 함수는 결과가 너무 크게 나올 때, 그래프 막대를 축약 시키는 단위를 얻기 위해 만들었다. 이 함수는 어느 정도로 축약 시킬지를 결정하는 함수이고, 실제로 축약 시켜 출력하는 것은 draw(all *t)에서 한다.

그래프를 출력하는 영역은 세로로 600픽셀인데, 크기 1당 픽셀 1로 출력하게 설정을 해놓아서 100만번 실행한 결과를 출력하게 되면 화면을 벗어나게 된다. 그냥 화면만 벗어나면 다행이지만, 이는 결국 배열 인덱스 범위를 초과하여 값을 다룬 것이 되므로 segfualt 에러가 나게 된다. 그래서 600단위로 결과를 축약 시키기 위해 check(all *t)함수를 구성했다.

```
void check(all *t) {
    int i;

    for (i = 0; i < t->N; i++) {
        if (t->result[i] != 0)
            t->max_draw = i;
        if (t->result[i] > t->max_value) {
            t->max_value = t->result[i];
            t->limit_flag = t->result[i] / 600;
            t->limit_flag++;
        }

    }
}
```

다음은 my_mlx_pixel_put(all *t, int x, int y, int color) 함수로, 이 함수가 화면에 출력할 이미지를 만들어준다.

all *t 구조체 안에는 이미지를 구성하는 void 포인터인 img 포인터와 addr 포인터가 있다. 이 함수를 실행하기 전에 main()함수에서 img 포인터를 mlx_new_image(t.mlx, 1080, 720)으로 초기화 시키는 작업을 한다. 그리고 addr 포인터를 mlx_get_data_addr(t.img, &t.bpp, &t.line_length, &t.endian)를 통해 초기화를 해준다.

다 초기화를 했다면, t->addr에서 아래의 공식대로 이동한 포인터의 위치에 hex코드로 정의된 색상 값을 넣는 방식으로 픽셀을 저장한다.

따라서 이 my_mlx_pixel_put(all *t, int x, int y, int color)함수는 x, y위치에 매개변수로 넣은 색상을 한 픽셀 찍어준다. 이 함수를 연속적으로 사용하면 그래프 모양의 이미지를 만들 수 있게 된다.

```
void my_mlx_pixel_put(all *t, int x, int y, int color)
{
    char    *dst;

    dst = t->addr + (y * t->line_length + x * (t->bpp / 8));
    *(unsigned int*)dst = color;
}
```

위의 my_mlx_pixel_put(all *t, int x, int y, int color) 함수를 연속적으로 사용한 것이 draw_line(all *t, int n, int x, int y, int width, int color) 함수이다. 이 함수는 100만번 실행하여 나온 트리의 높이 분포 중 하나를 막대로 출력하는 함수이다.

픽셀 출력 함수를 세로 n, 가로 width 만큼 반복하여 길이에 맞게 막대 모양을 이미지에 막대 모양 정보를 저장해준다.

이제 이 함수를 높이에 따라 실행해주면 된다.

```
void draw_line(all *t, int n, int x, int y, int width, int color) {
    int i;
    int j;
    char *dst;

    for (i = 0; i < n; i++) {
        for (j = 0; j < width; j++) {
            my_mlx_pixel_put(t, 100 + x + j, y - i, color);
        }
    }
}
```

draw.c 파일의 마지막 함수는 모든 작업을 수행하는 함수인 draw(all *t) 함수다. 이 함수에서는 트리 높이의 수에 따라 막대 그래프의 가로 길이를 정해주고, 그래프를 구분하기 쉽게 색상이 번갈아 가며 나오게 색상을 정해준 후, 막대를 순서대로 출력한다. 막대를 다 출력한 다음에는 해당 막대의 높이와 수치 정보를 출력하고, 가장 아래에는 높이의 평균을 정수 값으로 출력을 해준다.

```
void draw(all *t) {
    int i;
    int width;
    char a[10];
    char b[20] = "index";
    char c[20] = "count";
    char d[20] = "average";
    int color;
    int n;

    check(t);
    width = 800 / (t->max_draw - t->min_height);
    for (i = 0; i < t->max_draw - t->min_height; i++) {
        color = (i % 2) ? 0xEEEEEE : 0xDDDDDD;
        n = t->result[i + t->min_height] / t->limit_flag;
        if (t->result[i + t->min_height] != 0 && n == 0)
            n = 1;
        draw_line(t, n, i * width, 600, width, color);
    }
    mlx_put_image_to_window(t->mlx, t->win, t->img, 0, 0);
    for (i = 0; i < t->max_draw - t->min_height; i++) {
        itoa(i + t->min_height, a, 10);
        mlx_string_put(t->mlx, t->win, i * width + 100 + width / 2, 630, 0xD
DDDDD, a);
    }
    for (i = 0; i < t->max_draw - t->min_height; i++) {
        itoa(t->result[i + t->min_height], a, 10);
        mlx_string_put(t->mlx, t->win, i * width + 100 + width / 2, 660, 0xD
DDDDD, a);
    }
    itoa(t->average, a, 10);
    mlx_string_put(t->mlx, t->win, 50, 630, 0xDDDDDD, b);
    mlx_string_put(t->mlx, t->win, 50, 660, 0xDDDDDD, c);
    mlx_string_put(t->mlx, t->win, 50, 690, 0xDDDDDD, d);
    mlx_string_put(t->mlx, t->win, 150, 690, 0xDDDDDD, a);
    t->draw_flag = 0;
}
```

## II – VI. 메인 코드(main.c)

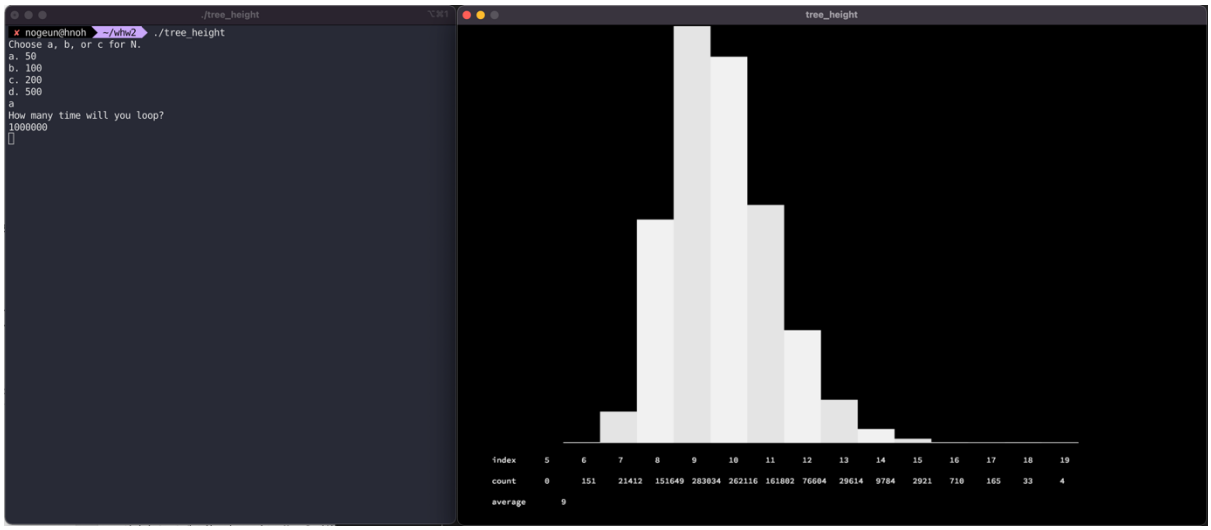
이제 앞에서 만든 모든 함수들을 연결시킬 main()함수를 작성하였다. 구조체에서 필요한 값들을 초기화 시키고, 프로그램을 돌릴 때 필요한 지시 문을 출력하도록 짰다. main()에서는 보통 초기화나 앞에서 만든 함수들을 실행시키는 것이 전부이기 때문에, 작동 방식이나 구성 방식 면에서 크게 중요한 부분은 아니다.

```
int main() {
    all t;
    char flag;

    t.top = t.pool;
    t.data = NULL;
    t.draw_flag = 1;
    t.max_value = 0;
    srand(time(NULL));
    init_pool(&t);
    printf("Choose a, b, or c for N.\n");
    printf("a. 50\nb. 100\nc. 200\nd. 500\n");
    flag = getchar();
    if (flag == 'a') {
        t.N = 50;
        t.min_height = (int)(log2(t.N));
    } else if (flag == 'b') {
        t.N = 100;
        t.min_height = (int)(log2(t.N));
    } else if (flag == 'c'){
        t.N = 200;
        t.min_height = (int)(log2(t.N));
    } else if (flag == 'd') {
        t.N = 500;
        t.min_height = (int)(log2(t.N));
    }
    printf("How many time will you loop?\n");
    scanf("%d", &t.loop_time);
    loop(&t);

    t.mlx = mlx_init();
    t.win = mlx_new_window(t.mlx, 1080, 720, "tree_height");
    t.img = mlx_new_image(t.mlx, 1080, 720);
    t.addr = mlx_get_data_addr(t.img, &t.bpp, &t.line_length, &t.endian);
    draw(&t);
    mlx_loop(t.mlx);
}
```

# III. 프로그램 실행 결과

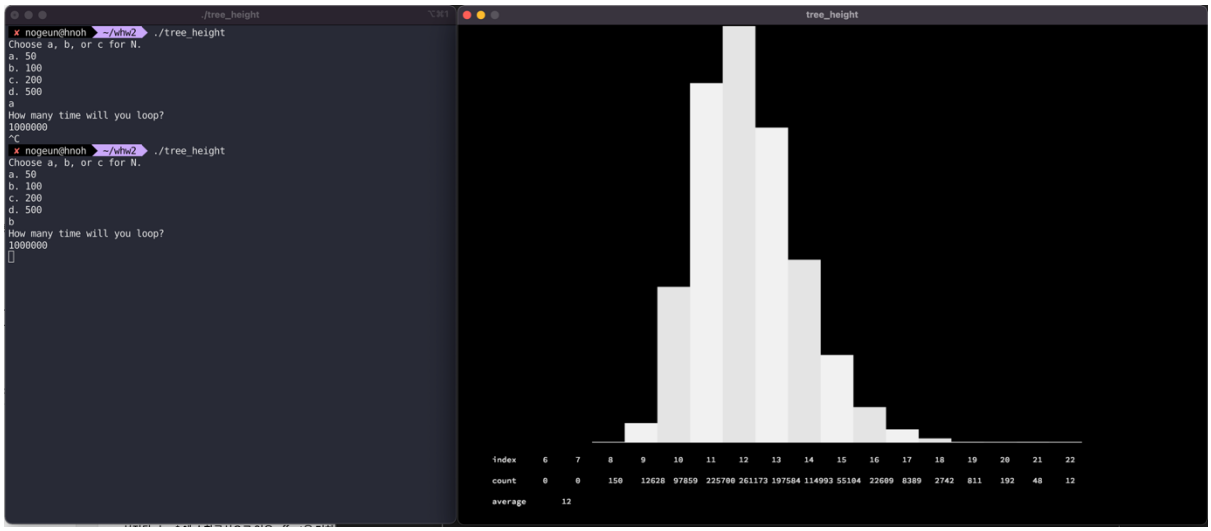
위의 프로그램을 실행하여 얻은 결과는 각각 아래와 같다.

1. N = 50



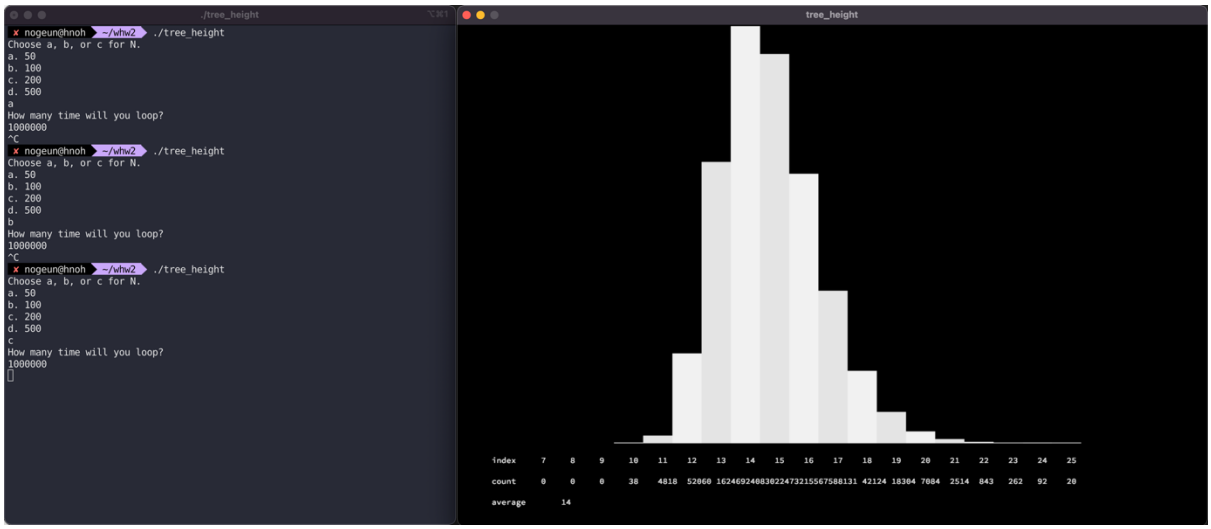

최소 높이는 $\log_2(50)$으로 5정도 되는데, 평균 값은 9정도 나왔고, 9가 가장 많은 것을 볼 수 있다.

2. N = 100



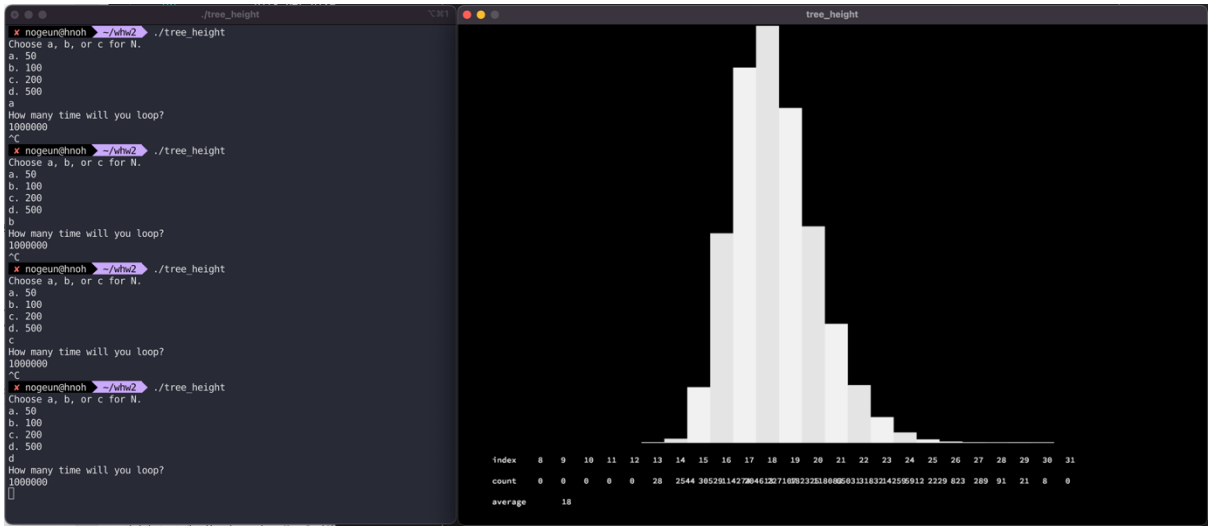

최소 높이는 $\log_2(100)$으로 6정도 되고, 평균 값은 12이며, 12가 가장 많다.

3. N = 200




최소 높이는 $\log_2(200)$으로 7정도 되고, 평균 값은 14이며, 14가 가장 많다. 15도 비슷하게 있는 것으로 봐서는 정확한 평균 수치는 15에 가까울 것 같다.

4. N = 500




최소 높이는 $\log_2(500)$으로 8정도 되고, 평균 값은 16이며, 16이 가장 많다. 15가 다음으로 많은 것을 봐서는 평균 수치는 16에 근접할 것이다.

4가지 경우로 나누어 평균 높이를 구해보았는데, 4가지 경우 모두 log(N)이 아닌 2log(N)에 근접한 것을 알 수 있다.

# IV. 실행 결과에 대한 고찰

이진 트리의 평균 높이를 수학적으로 구한 결과를 검색하니, 실험을 통해 얻은 결과 값과 굉장히 유사하게 나왔다. 하지만 이를 수학적으로 구하는 과정은 상당히 복잡하기도 하고, 이해하는 데에 어려움이 있었다. 수학적으로 접근하는 방법은 다른 책에서도 소개가 되어있기에 차근차근 읽어보았을 때, 어떤 맥락으로 이런 결과가 나왔는지 이해할 수 있었다.

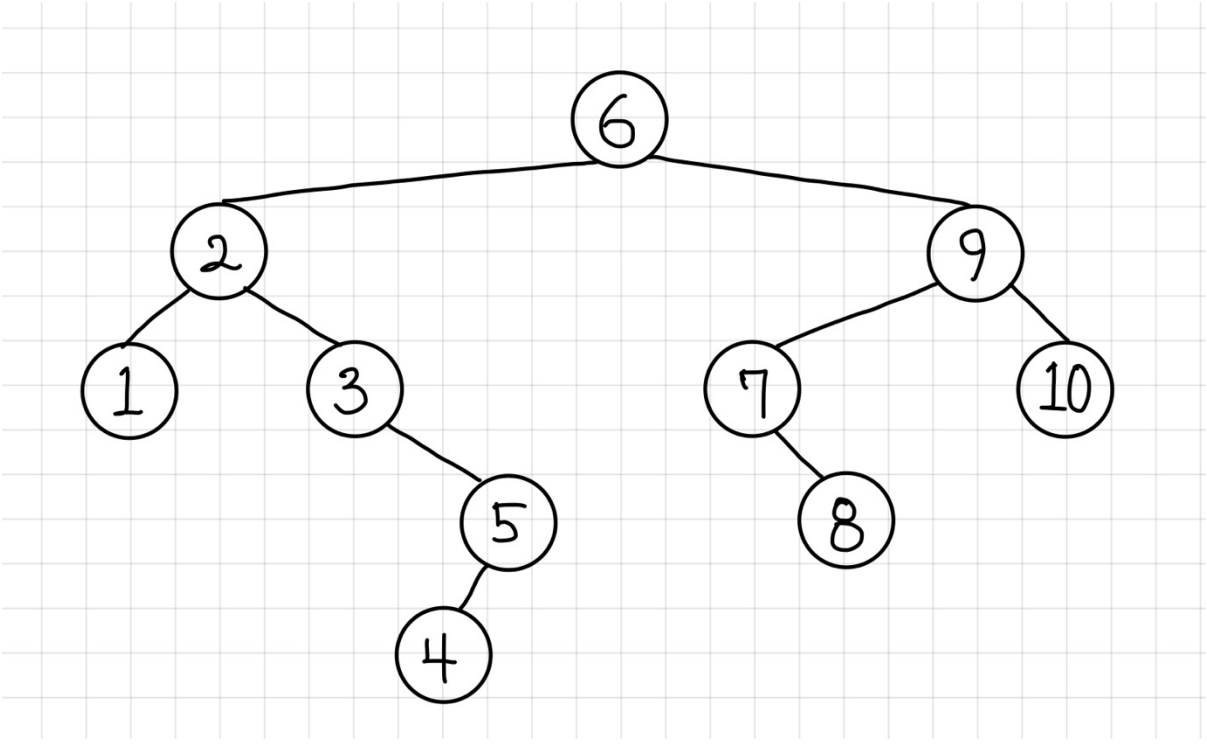
하지만 결과를 보면 뭔가 이상한 점이 있었다. 특정 높이 이상으로는 값이 아예 나오지 않는 다는 것이다. 위의 그래프는 값을 의도적으로 자른 것이 아니라, 최소 높이부터 값이 0이 아닌 마지막 높이까지를 보여주게끔 만든 것이다. (높이가 N인 결과가 하나라도 있고 그 이상의 결과가 하나도 없다면, 최소 높이부터 N까지 출력한다.) 즉, 저 그래프를 보면 높이가 높은 값이 아예 나오지 않는 다는 것을 알 수 있다.

그래서 특정 높이 이상으로 나오지 않는 것 에 대해 간단하고 직관적으로 접근해보았다.

이진 탐색 트리의 경우 왼쪽에는 현재 노드보다 작은 값, 오른쪽에는 현재 노드보다 큰 값이 들어가게 된다. 간단하게 아래의 순서대로 값들이 들어갔다고 생각해보자.

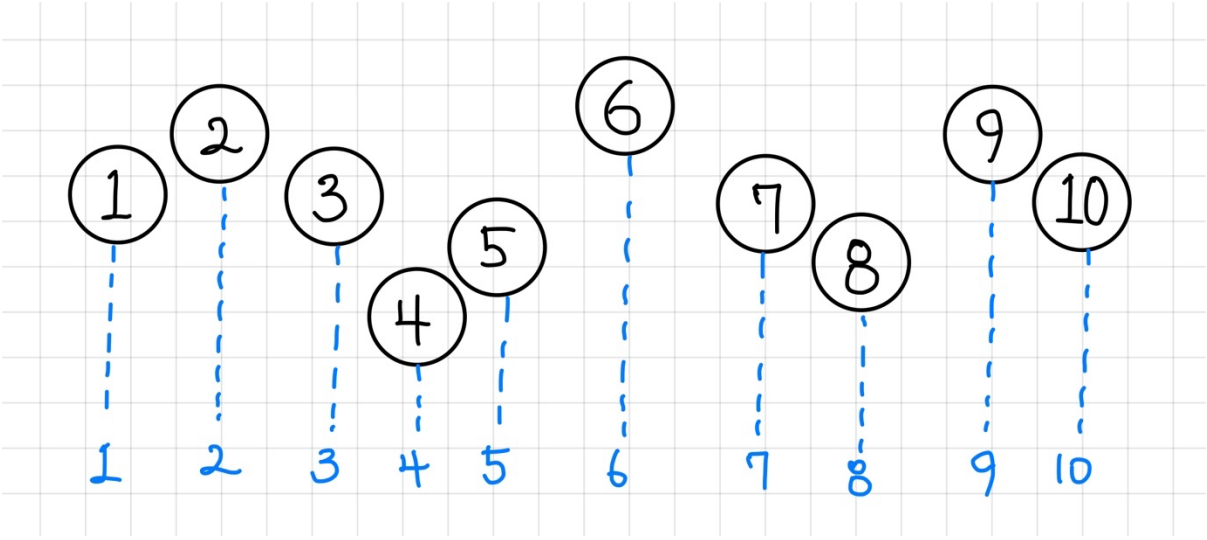
[6, 9, 2, 7 ,3, 5, 4, 10, 1, 8]

그러면 아래와 같이 트리가 구성될 것이다. 그런데 트리를 구성할 때 두 값의 크기 차이만큼 거리를 둔다고 해보자.

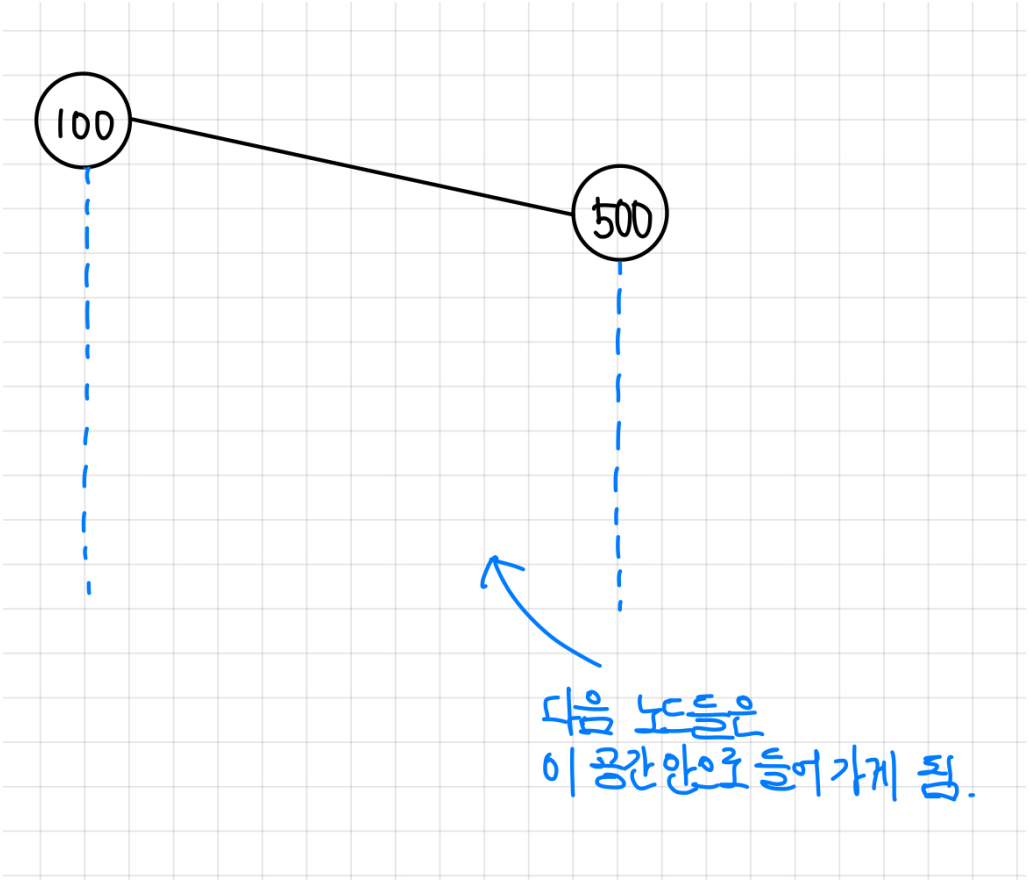



위의 규칙대로 크기 차이만큼 거리를 두어 구성한 트리를 가장 윗변으로 밀착 시킨다고 생각해보자. 즉, 이 트리를 위로 쭉 밀어서 일렬로 만든다는 것이다.

이렇게 트리를 밀어서 일렬로 만들면, 왼쪽부터 크기 순서대로 정렬되게 된다. 이진 탐색 트리를 이런 관점에서 보면, 공간 상으로 봤을 때 순서대로 정렬되게 된다.




이 정보에서 무엇을 알 수 있는가 하면, 부모와 자식 노드의 크기 차이가 크면 넓은 공간으로 트리가 구성되게 된다. 그런데 이렇게 만들어진 넓은 공간 안으로 다른 노드들이 들어오게 되면, 넓은 공간 안을 채우기 위해 높이가 빠르게 높아지지 않게 된다.




하지만 부모 노드와 자식 노드의 차이가 크지 않으면, 채울 공간이 부족하게 되어 계속해서 탑을 쌓

듯이 높이를 키우게 된다. 계속해서 부모 노드와 근접한 값들로 채워지게 되면 높이가 상승하게 된다.

위의 전제를 기반으로 확률적으로 접근을 해본다고 해보자. 확률적으로 생각을 했을 때, 부모 노드의 값과 근접한 값들이 연속적으로 들어갈 확률은 굉장히 낮다. 이전에 들어간 값이 50이라고 하고, 값의 범위가 500이라고 했을 때, 다음에 들어갈 값이 45 ~ 55의 값 중 하나일 확률은 50분의 1이다. 이런 확률이 연속적으로 여러 번 일어날 확률은 더 희박하다. 그렇기에 위의 전제가 맞고, 확률적인 계산에도 오류가 없다면, 위의 가정들에 의해 높이가 30 이상으로 나올 확률은 굉장히 희박해진다.

사실 이보다 더 직관적인 접근 방식은 Catalan Number 를 이용한 간단하고, 대략적인 확률 계산이다. N이 50일 때의 Catalan Number는 1978261657756160653623774456로, 엄청난 경우의 수를 보여준다. 이 때 가장 극단적인 경우로, 높이가 50이 되는 경우의 수를 구해본다고 해보자. 높이가 50일 경우의 수는 2^50으로 적지않은 경우의 수지만, 전체 Catalan Number와 비교를 하면 상대적으로 엄청나게 적은 수 임을 알 수 있다. 확률을 구해보면, 0.000000000582795로 엄청나게 희박한 확률 임을 알 수 있다. 물론 높이가 20이나 30인 경우는 이 경우보다는 훨씬 많겠지만, Catalan Number와 비교하면 역시나 엄청나게 작은 수가 될 것이다. (0.000000122773804 이 수가 로또 당첨 확률인데, 높이가 50이 될 확률은 로또 당첨 확률 보다 210배 낮다.)

# V. 마치며

앞에서 기술한 실행 결과에 대한 고찰의 경우 검증된 논리 전개 방식도 아닐뿐더러, 필자가 단순하게 생각해 본 것이라 그저 단순한 궤변으로 보일 수 있다. 보이기만 하면 다행이지만, 매우 높은 확률로 궤변일 것이다. 하지만 트리에 대해서 다른 방식, 다른 관점으로 접근할 수 있는 계기가 되었다. 이전에는 이진 탐색 트리에 대해서 단순히 검색 속도를 빠르게 하기 위해 고안한 트리의 일종이라고만 생각했는데, 트리의 값에 따라 거리를 다르게 하여 보면 다른 정보가 보이는 것이 놀라웠다.

트리의 구현에서도 프로그래밍 방식에 대해서 조금 더 생각해보게 된 계기가 되었다. 논리적으로 코드를 짜는 것도 좋지만, 가독성 있게 코드를 짜는 것도 중요하다는 걸 다시 한 번 느꼈다. 필자 스스로가 코드를 짠 것을 다시 보니 가독성이나 구현 방식에서 부족한 부분이 많이 보였다. 한 구조체에 변수를 몰아서 선언한 것이 특히 그랬다.

자료구조가 중요하고, 프로그래밍의 기본적인 소양이라고 많이 말하기도 하고, 다른 프로젝트나 알고리즘 문제를 풀면서 이를 느껴 보긴 했지만, 이론적이나 수학적으로 자료구조가 어떤 식으로 동작하고 왜 이런 시간복잡도를 보이는 지는 몰랐다. 이번 실험 과정에서 많은 것을 배웠다는 생각이 든다.