실습과제 7.1

[과제 설명]

트리(Tree)는 계층적 관계를 표현하는 자료구조이다. 가장 위쪽(index가 1)인 노드를 루트 노드라고 하고, 마지막에 있는 노드는 단말 노드 등으로 칭하며, 이 두 노드 사이에 위치한 노드들은 내부 노드라고 한다. 노드들을 잇는 선을 간선(edge) 또는 branch라고 한다.

이런 트리 중에는 자식 노드의 개수가 유동적인 것도 있지만, 이진트리(Binary Tree)는 루트 노드를 기준으로 두 개의 서브 트리로 구성되어있고, 자식 노드의 개수는 모두 2개다. 이것을 왼쪽 자식과 오른쪽 자식으로 구분한다. 또, 최대 2개의 자식 노드를 가져서, 노드가 존재하지 않는 공간에는 공집합 노드가 존재하는 것으로 간주한다. 이진트리를 사용하는 이유에는 크게 3가지가 있는데 구현이 용이하고 오류의 가능성이 적으며, 효과적인 탐색이 가능하기 때문이다.
효과적인 탐색이 가능한 이유는, 자식 노드가 불규칙적이면 주변 노드의 인덱스를 예상해서 사용할 수 없다. 따라서 원하는 노드의 인덱스를 예상할 수 있는 이진트리의 부모와 자식의 관계는 다음과 같다.

-i번째 노드의 부모 인덱스(p)

$p = i \div 2$

-p번째 노드의 왼쪽 자식 인덱스(ls)

$ls = 2 \times p$

-p번째 노드의 오른쪽 자식 인덱스(rs)

$rs = 2 \times p + 1$

[코드]

```
#include <iostream>
#include <cmath>
using namespace std;

void PrintTree2Matrix(int** M, int* bTree, int size, int idx, int col, int row, int height) {
  if (idx > size) return;
  M[row][col] = bTree[idx];
  PrintTree2Matrix(M, bTree, size, idx * 2, col - pow(2, height - 2), row + 1, height - 1);
  PrintTree2Matrix(M, bTree, size, idx * 2 + 1, col + pow(2, height - 2), row + 1, height - 1);
}
```

```
void TreePrinter(int* bTree, int size) {
  int h = (int)ceil(log2(size + 1));
  int col = (1 << h) - 1;
  int** M = new int* [h];
  for (int i = 0; i < h; i++) {
    M[i] = new int[col];
  }
  for (int i = 0; i < h; i++) {
    for (int j = 0; j < col; j++) {
      M[i][j] = 0;
    }
  }
  for (int j = 0; j < col; j++) {
    printf("==");
  }
  printf("\n");
  PrintTree2Matrix(M, bTree, size, 1, col / 2, 0, h);
  for (int i = 0; i < h; i++) {
    for (int j = 0; j < col; j++) {
      if (M[i][j] == 0)
        printf(" ");
      else
        printf("%2d", M[i][j]);
    }
    printf("\n");
  }
  for (int j = 0; j < col; j++) {
    printf("==");
  }
  printf("\n");
}
//아래 함수를 구현

//i번째 노드의 모든 조상 노드를 출력하는 함수
void PrintAncestor(int* bTree, int size, int idx) {
  printf("%d의 모든 조상 노드: ", bTree[idx]);

  int flag = idx;
  while (flag > 1) {
    flag = flag / 2;
    //n번째 노드의 부모 인덱스는 n/2의 바닥 함수값이다.
    //정수형끼리의 나눗셈은 몫만 남기 때문에 n/2로 가능
    printf("%d ", bTree[flag]);
  }
  puts("");
}

//i번째 노드의 모든 왼쪽 후손 노드들을 출력하는 함수
void PrintLeftDescendant(int* bTree, int size, int idx) {
  printf("%d의 모든 왼쪽 후손 노드: ", bTree[idx]);

  int flag = idx;
  while (2 * flag <= size) { //전체 크기보다 커지면 종료
    flag = 2 * flag;
    //n번째 노드의 왼쪽 자식의 인덱스는 2 * n이다.
    printf("%d ", bTree[flag]);
  }
  puts("");
}

//i번째 노드의 모든 오른쪽 후손 노드들을 출력하는 함수
void PrintRightDescendant(int* bTree, int size, int idx) {
```

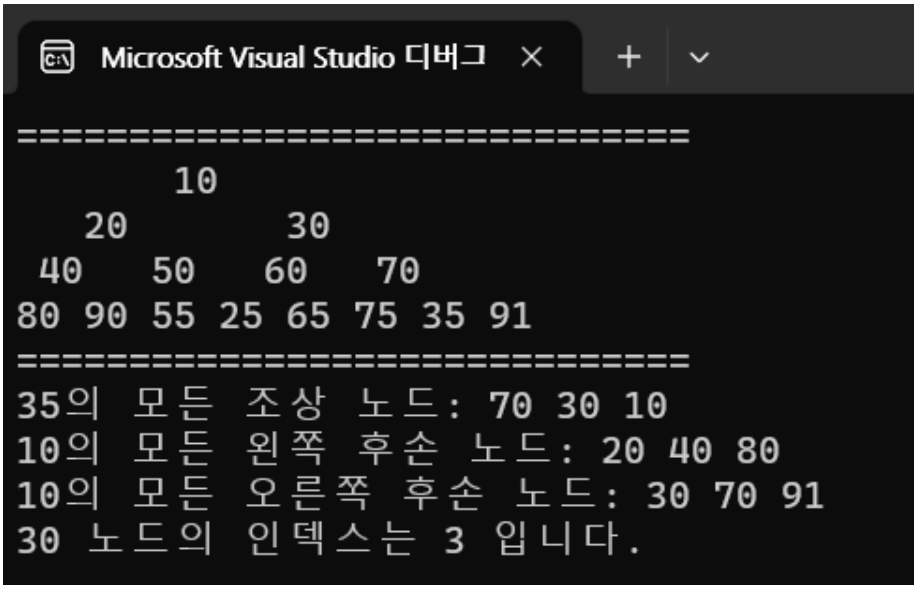
```
  printf("%d의 모든 오른쪽 후손 노드: ", bTree[idx]);

  int flag = idx;
  while (2 * flag + 1 <= size) { //전체 크기보다 커지면 종료
    flag = 2 * flag + 1;
    //n번째 노드의 왼쪽 자식의 인덱스는 2 * n + 1이다.
    printf("%d ", bTree[flag]);
  }
  puts("");
}

//해당 값을 가진 노드의 인덱스를 반환하는 함수
int FindNode(int* bTree, int size, int data) {
  int flag = 0;
  //초기화를 하지 않아도 상관은 없지만, 일종의 예외처리(찾을 수 없다면 0이 반환)
  for (int i = 1; i <= size; i++) {
    if (bTree[i] == data) {
      flag = i;
      break;
    }
  }//1부터 차례대로 탐색하다가 데이터와 일치하면 반환

  return flag;
}
int main() {
  int full_bTree[] = { 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 55, 25, 65,75,35,91 };
  int size = sizeof(full_bTree) / sizeof(full_bTree[0]) - 1;
  TreePrinter(full_bTree, size);
  PrintAncestor(full_bTree, size, 14);
  PrintLeftDescendant(full_bTree, size, 1);
  PrintRightDescendant(full_bTree, size, 1);
  printf("%d 노드의 인덱스는 %d 입니다. \n", 30, FindNode(full_bTree, size, 30));
  return 0;
}
```

[실행 결과]



[과제에 대한 고찰]

처음에 트리의 탐색을 할 때 단순 반복문을 이용해서 하나씩 찾는 것인 줄 모르고 트리 노드의 양 값을 비교해서 하는 방식으로 해보았다. 이런 건 나중에 하게 될 최소 힙 트리나 최대 힙 트리 같이 노드에 들어있는 자료의 크기가 일정해야 가능한 것이었고, 그렇지 않으면 단순 반복문을 사용하는 것보다 시간을 더 쓰게 되었다.

자식은 항상 2개보다 작거나 같은 이진트리의 특성 덕분에 조상 노드를 찾거나 후손 노드를 찾는 함수를 만들 때 (인덱스를 예상할 수 있어서) 생각보다 간단해진 것 같았다.