
자료구조 설계과제 3 보고서

[자료구조 학생 관리 프로그램]

010-8578-7628

12131489 김영훈

12131489@inha.edu

1. 개요

이 프로그램은 인하대 재학생들의 정보를 학번으로 관리한다. AVL Tree 형태로 구현하였고 삽입, 탐색의 기능을 갖도록 설계하였다. AVL Tree는 높이를 일정하게 유지하는 트리로 삽입, 탐색 기능이 최악에 경우에도 $O(\log n)$ 의 수행시간이 걸리도록 한다. 프로그램 설계를 위해 학번은 8자리, 이름과 학과는 20바이트 이하, 학년은 1~4 사이의 값만 들어가도록 제한한다.

프로그램을 설계하기 위해, 프로그램은 운영체제 Windows 10에서 개발 도구 Microsoft Visual Studio Community 2015를 이용하여 C++ 언어로 개발하였다.

2. 필요한 자료구조 및 기능

Student 구조체	Tree 클래스
<pre>struct student { int number; string name; string department; int grade; };</pre>	<pre>class Tree { public: Node* root; int probe; bool limitLength(int number, string name, string department, int grade); void setNode(Node* node, int number, string name, string department, int grade); void insert(int number, string name, string department, int grade); void nodePosition(Node* newNode, Node* node); int height(Node* node); bool heightProblem(Node* node, int h); void restructuringCheak(Node* newNode); void restructuring(Node* newNode, Node* current); Node* xyz(Node* child, Node* parent); void print(int number); Node* search(int k, Node* v); Tree() { root = NULL; probe = 0; } };</pre>
Node 클래스	
<pre>class Node { public: Node* parent; Node* left; Node* right; student data; Node() { parent = NULL; left = NULL; right = NULL; } };</pre>	

Student 구조체			
int number	학생의 학번	string department	학생의 학과
string name	학생의 이름	int grade	학생의 학년

Node 클래스	
Node* parent	부모 노드를 가리키는 노드형 포인터 타입
Node* left	왼쪽자식 노드를 가리키는 노드형 포인터 타입
Node* right	오른쪽자식 노드를 가리키는 노드형 포인터 타입
student data	노드에 저장된 데이터(학번, 이름, 학과, 학년)

Tree 클래스	
Node* root	Tree의 root로 최상위 노드를 지정
int probe	탐사횟수를 나타내는 변수
bool limiteLength	입력 받은 학생정보가 초기조건과 안 맞을 경우 1을 반환
void setNode	사용자로부터 입력 받은 학생 정보로 노드의 데이터를 설정
void insert	새로운 노드를 생성하여 트리에 삽입
void nodePosition	노드가 삽입될 위치를 찾아 트리 형태로 연결
int gight	노드의 높이를 반환
bool heightProblem	노드의 양쪽 자식의 높이 차이가 2이상일 때 1을 반환
void restructuringCheak	트리의 높이를 적절하게 해주기 위해 구조를 변경
void restructuring	양쪽 자식 노드의 높이가 2 이상 차이나지 않도록 트리의 구조를 변경
Node* xyz	x, y, z를 지정
void print	입력한 학번의 노드의 데이터와 탐사횟수 출력
Node* search	입력한 학번의 노드를 찾아 반환

AVL Tree를 구현하기 위하여 Tree 클래스와 각 기능들을 하는 함수를 만들었다. Tree를 구성하는 Node를 만들고 각 Node 안에 학생의 데이터를 저장하였다. Binary Tree 형태가 될 수 있도록 부모와 왼쪽, 오른쪽 자식들을 가리키는 포인터형 변수를 갖고 있다. 또한 AVL Tree는 Binary search tree와 같이 왼쪽 자식노드에는 노드보다 작은 값(key)을 갖고, 오른쪽 자식노드에는 노드보다 큰 값을 갖는다. AVL Tree는 각 자식노드의 높이차이가 1이하이다. 그 이상의 높이차이가 발생 했을 경우 restructuring 함수로 인하여, 조건에 만족하도록 재구조화 된다.

사용자가 사용할 수 있는 기능으로 입력한 학번을 찾아 출력해주는 출력 기능, 입력한 학생정보를 프로그램에 저장해주는 삽입 기능이 있다.

3. 기능별 알고리즘 명세

- 각 기능들이 수행되는 과정과 사용된 알고리즘

1. 삽입

insert 함수 (노드 생성 및 삽입)

```
bool equals = 0; // 전역변수로 같은 값이 있을 때 1로 변경한다.
void Tree::insert(int number, string name, string department, int grade) {
    probe = 0; // 초기 탐사횟수 0
    if (limtelLength(number, name, department, grade) == 0) {
        Node* newNode = new Node; // 새로운 노드를 생성한다.
        setNode(newNode, number, name, department, grade); // 노드의 데이터를 입력 받은 값으로 설정한다.

        if (root == NULL) { // 루트노드가 없으면 새로운 노드를 루트로 설정한다.
            root = newNode;
        }
        else
            nodePosition(newNode, root); // 노드의 위치를 정해준다.
        if (equals != 1) { // 같은 값이 없으면
            if (txtType == 1) // txtType이 command.txt일때
                outputFile << probe << endl; // 탐사횟수를 출력한다.
            restructuringCheak(newNode); // 구조의 변화가 필요한지 확인하는 함수로 간다.
        }
    }
    equals = 0; // 다시 equals를 0으로 초기화한다.
}
```

노드를 생성하여 삽입하는 역할을 한다. 이 함수에서는 삽입하는 노드의 위치를 정해주는 nodePosition 함수와 트리의 구조를 재구조화해야하는지 판단하는 restructuringCheak함수가 필요에 따라 호출된다.

처음 probe(탐사 횟수)를 0으로 설정한다.

limtelLength 함수를 통해 입력 받은 학생의 정보가 주어진 초기조건과 맞지 않은 경우 1을 반환하여 값이 삽입되지 않도록 한다.

그렇지 않을 경우 노드를 생성하고, 입력 받은 학생 데이터를 노드의 데이터로 설정한다.

root가 null일 경우 노드가 처음 생성된 경우로 노드를 root로 설정한다.

root가 null이 아니면, 노드의 위치를 지정해주는 nodePosition 함수를 호출한다.
(nodePosition 함수에서 같은 학번을 찾았을 경우 equals는 1이 되고 함수는 종료된다.)

같은 학번이 아니라면

txtType이 command.txt 일때 1이므로 command.txt일때만
탐사횟수를 출력시킨다.

삽입된 노드에 의해 AVL Tree의 구조의 균형이 맞지 않는지 확인해야 하므로
restructuringCheak 함수를 호출한다

equals를 0으로 초기화한다.

nodePosition 함수 (삽입할 노드의 위치 지정)

```
void Tree::nodePosition(Node* newNode, Node* node) {
    probe++;
    if (node != NULL) {
        if (newNode->data.number == node->data.number) {
            outputFile << "추가할 수 없음 " << probe << endl;
            equals = 1;
            delete newNode;
        }
        else {
            if (newNode->data.number < node->data.number) {
                if (node->left == NULL) {
                    node->left = newNode;
                    newNode->parent = node;
                }
                else {
                    nodePosition(newNode, node->left);
                }
            }
            else if (newNode->data.number > node->data.number) {
                if (node->right == NULL) {
                    node->right = newNode;
                    newNode->parent = node;
                }
                else {
                    nodePosition(newNode, node->right);
                }
            }
        }
    }
}
```

트리에서 노드가 삽입할 위치를 찾아주고 트리의 순서와 맞게 연결해준다. 생성된 노드를 root 노드부터 학번을 비교하며 작을 경우 왼쪽, 클 경우 오른쪽 노드를 탐색한다. 같은 값이 있을 경우 추가가 불가능하도록 하였다.

이 함수를 호출할 때마다 탐사 횟수를 1 증가시킨다. insert 함수에서 nodePosition 함수의 인자로 newNode과 root를 넣어줬다. root는 null이 아니므로, 처음 이 함수를 실행 시에는 if(node != NULL) 조건문이 항상 참이 된다

newNode의 데이터와 현재 탐색중인 node의 값이 일치하면 "추가할 수 없음"과 탐사횟수를 나타내고, equals를 1로 바꿔 insert 함수에서 다음 동작을 취하지 않게 하고, 생성된 노드를 다시 삭제하였다.

newNode의 데이터와 현재 탐색중인 node의 값이 일치하지 않을 때

newNode가 node의 학번보다 작은 값일때, node의 왼쪽자식이 비었으면, newNode를 node의 왼쪽자식으로 설정한다. 왼쪽자식이 존재하면, newNode는 node의 왼쪽자식에 대하여 이 함수를 호출한다.

newNode가 node의 학번보다 큰 값일 때, node의 오른쪽 자식이 비었으면, newNode를 node의 오른쪽자식으로 설정한다. 오른쪽 자식이 존재하면m newNode는 node의 오른쪽 사식에 대하여 이 함수를 호출한다.

restructuringCheak 함수 (재구조화 필요 확인)

```
void Tree::restructuringCheak(Node* newNode) {  
  
    int h = 0;  
    for (Node* current = newNode; current != NULL; current = current->parent) {  
        h++;  
        if (current->right == NULL && current->left == NULL)  
            continue;  
        else if (h >= 3 && current->right == NULL || h >= 3 && current->left == NULL)  
            {}  
        else if (heightProblem(current, h) == 1)  
        {  
            current = current->parent;  
        }  
        else  
            continue;  
  
        restructuring(newNode, current);  
        break;  
    }  
}
```

AVL 트리가 균형을 이루고 있는지 확인하고, 균형에 문제(오른쪽 노드와 왼쪽 노드의 높이가 2 이상 차이가 날 경우)가 있을 경우 트리의 높이가 적절하도록 구조를 바꿔주는 함수이다. insert에서 newNode가 인자로 넘어와 새로운 노드부터 탐색을 시작한다. 반복문이 실행될 때마다 높이를 1 증가시킨다. current를 node로 지정하고 반복 시 current는 node의 부모 노드로 하고, null이 될 때까지 반복한다. current의 자식 노드들이 null일 경우 외부노드이므로 계속하여 반복문을 돌린다. current의 높이가 3이상일때 자식 노드의 한쪽이 null이면 자식 간의 높이차이가 2 이상이므로 문제 노드이다. 또한 반대편 노드와 높이 차이가 2 이상일때 1을 반환해주는 heightProblem 함수를 이용하여, 문제가 있을 경우 부모가 문제 노드이므로 current를 currnet의 부모로 설정한다. 높이 문제 노드를 찾았을 경우 restructuring 함수를 이용하여 트리를 재구조화시킨다.

height 함수 (높이 지정)

```
int Tree::height(Node* node) {  
    int left, right;  
  
    if (node == NULL)  
        return 0;  
  
    left = height(node->left);  
    right = height(node->right);  
  
    if (left > right)  
        return left + 1;  
    else  
        return right + 1;  
}
```

// 높이를 반환하는 함수
// 노드의 왼쪽, 오른쪽 자식의 높이를 나타내는 변수
// 노드가 비었으면 외부노드이므로 h = 0이다. 따라서 0을 반환한다.
// 왼쪽 자식의 높이를 구하기 위해, 노드의 왼쪽 자식의 높이에 대한 함수를 적용한다.
// 오른쪽 자식의 높이를 구하기 위해, 노드의 오른쪽 자식의 높이에 대한 함수를 적용한다.
// 왼쪽 자식의 높이가 오른쪽 자식의 높이보다 클 때
// 이 노드의 높이는 왼쪽자식의 높이 +1 이므로, 값을 반환한다.
// 이 노드의 높이는 오른쪽자식의 높이 +1 이므로, 값을 반환한다.

이 노드의 높이를 계산하여 반환해주는 함수이다. 이 함수가 실행되면 left, right 함수로 노드의 각 자식노드의 높이를 계산하는 변수를 만든다. nodo가 비었을때 높이 0을 반환한다. 각 자식들의 높이를 계산하기 위해 자식의 높이를 height 함수를 호출한다. 함수가 호출되면 각 자식들의 외부노드(node == null)가 나올 때까지 함수를 계속 호출한다. 외부노드의 호출이 종료되면, 그 외부노드의 부모노드로 올라와 함수가 실행된다. 왼쪽자식의 높이가 오른쪽자식의 높을 경우 왼쪽 높이의 1을 더한 값 반환하고, 아니면 오른쪽 자식의 높이의 1을 더해서 반환한다. 노드의 sub tree의 높이가 모두 계산되면 left, right의 값이 결정되고, 비교하여 노드의 높이가 반환된다.

hightProblem 함수(문제노드를 탐색)

```
bool Tree::heightProblem(Node* node, int h) {
// 해당 노드의 부모가 문제노드인지 확인하는 함수
//(부모의 각 자식들의 높이를 비교했을때 2 이상 차이나면 부모는 문제노드, h는 노드의 높이)

    int opposite;                                // 노드의 반대편 자식의 높이

    if (node->parent == NULL)                    // 노드가 부모가 없으면 비교할 값이 없으므로 0 반환
        return 0;

    if (node->parent->left == node)               // 노드가 부모의 왼쪽자식이면
        opposite = height(node->parent->right); // 오른쪽 자식의 높이를 구한다.
    else                                         // 노드가 부모의 오른쪽자식이면
        opposite = height(node->parent->left);  // 왼쪽 자식의 높이를 구한다.

    if (h - opposite > 1 || opposite - h < -1) // 높이가 2이상 차이가 날 경우
        return 1;                             // 문제가 있음을 나타내는 1 반환(부모노드에 문제가 있음)
    else
        return 0;
}
```

부모가 각 자식들의 높이를 비교했을 때 2 이상 차이나면 '문제노드'라고 한다. 이 함수에서는 해당 노드의 부모가 문제인지 확인하는 함수이다. 노드(node)와 노드의 높이(h)를 인자로 불러 이 노드에 문제가 있을 시 1을 반환한다.

노드의 반대편 자식(노드의 부모의 다른 자식노드)의 높이를 opposite로 한다. 노드의 부모가 없으면 비교값이 없으므로 0을 반환한다. 그리고 노드의 반대편 자식의 높이를 구해 opposite에 넣는다.

노드의 반대편 노드의 높이인 opposite와 자신의 높이인 h를 비교하여 2 이상 차이가 날 경우 1을 반환하고 아닐경우 0을 반환한다.

restructuring함수

개요

AVL 트리의 균형을 맞춰주기 위해 트리의 구조를 변형하는 함수이다. 먼저 문제노드를 z로 한다. z의 자식 중 newNode가 포함된 쪽으로 y는 z의 자식, x는 y의 자식으로 설정한다. 이 x, y, z의 학번을 비교하여 작은 값 순으로 a, b, c를 지정한다. 각 노드들의 서브 트리를 t0, t1, t2, t3로 지정하고 3개의 노드의 형태를 바꾼 뒤 바뀐 형태에 따라 서브 트리들을 다시 메달아 준다.

x,y,z 지정

```
void Tree::restructuring(Node* newNode, Node* current) {
    Node* a = NULL;    // 가장 큰 수의 노드
    Node* b = NULL;    // 두번째 큰 수의 노드
    Node* c = NULL;    // 가장 작은 수의 노드
    Node* x = NULL;    // 문제노드의 자손
    Node* y = NULL;    // 문제노드의 자식
    Node* z = NULL;    // 문제노드

    //x,y,z를 지정
    z = current;
    y = xyz(newNode, z);
    x = xyz(newNode, y);
}
```

x,y,z, a,b,c 를 null 로 초기화하고 z,y,x 를 xyz 함수를 통해 결정한다.

xyz 함수

```
Node* Tree::xyz(Node* child, Node* ancestor) {    // 두 노드의 값을 비교하여 child가 ancestor의 어느쪽 자식에 속한지 확인
    if (child->data.number > ancestor->data.number) // 자식의 수가 부모의 수보다 더 크면
        return ancestor->right;                  // 조상의 오른쪽노드를 자식으로 지정한다.
    else
        return ancestor->left;                    // 조상의 왼쪽노드를 자식으로 지정한다.
}
```

x, y, z 는 newNode 가 존재하는 쪽으로 지정하도록 만들었다.

xyz 함수 : xyz 함수에 인자로 newNode 와 z 를 넣어 학번의 값을 비교한다. newNode 의 학번이 z 의 학번보다 더 큰 값이면 z 의 오른쪽에 newNode 가 삽입됐으므로 y 는 z 의 오른쪽 노드이다. 값이 더 작으면 부모의 y 는 z 의 왼쪽 노드이다.

A,B,C 지정

```
//z, y, x의 학번을 비교하여 작은 순으로 a, b, c를 정한다.
if (z->data.number > y->data.number && z->data.number > x->data.number) { // z > y,x
    c = z;
    if (y->data.number > x->data.number) {
        b = y;
        a = x;
    }
    else {
        b = x;
        a = y;
    }
}
else if (y->data.number > z->data.number && y->data.number > x->data.number) { // y > z,x
    c = y;
    if (z->data.number > x->data.number) {
        b = z;
        a = x;
    }
    else {
        b = x;
        a = z;
    }
}
else { // x > y,z
    c = x;
    if (y->data.number > z->data.number) {
        b = y;
        a = z;
    }
    else {
        b = z;
        a = y;
    }
}
```

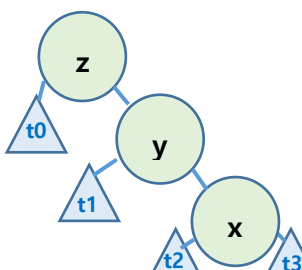
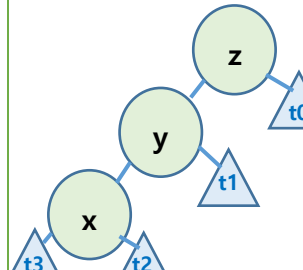
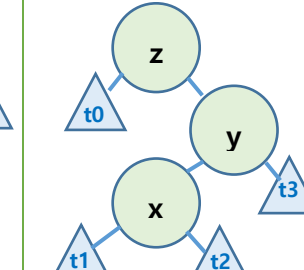
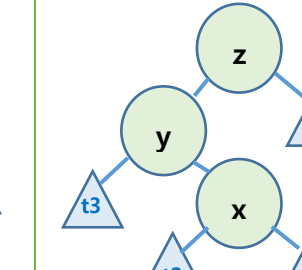
앞서 구한 z, y, x 노드의 각 학번의 크기를 비교하여 작은 순으로 a, b, c 를 지정한다. if 문을 이용하여 세 노드 중에서 가장 큰 값의 노드를 먼저 찾고, if 문 안에 if 문을 사용해 다음으로 큰 값을 찾아 a, b, c 를 지정하였다.

서브트리 t0, t1, t2, t3 설정

```
//a,b,c의 각 서브트리를 재조정하여 각 노드에 붙여준다.
Node* t0 = NULL;
Node* t1 = NULL;
Node* t2 = NULL;
Node* t3 = NULL;

if (z->right == y && y->right == x) {
    t0 = z->left;
    t1 = y->left;
    t2 = x->left;
    t3 = x->right;
}
else if (z->left == y && y->left == x) {
    t0 = z->right;
    t1 = y->right;
    t2 = x->right;
    t3 = x->left;
}
else if (z->right == y && y->left == x) {
    t0 = z->left;
    t1 = x->left;
    t2 = x->right;
    t3 = y->right;
}
else if (z->left == y && y->right == x) {
    t0 = z->right;
    t1 = x->right;
    t2 = x->left;
    t3 = y->left;
}
}
```

x,y,z 위치에 따른 4 가지 종류

z->right == y y->right == y	z->left == y y->left == x	z->right == y y->left == x	z->left == y y->right == x
			

t0 ~ t3 까지의 서브트리를 z, y, x의 순서에 따라 다르게 설정한다.

b를 부모노드로 설정(구조 변경)

```

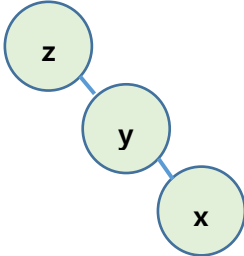
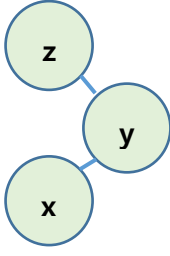
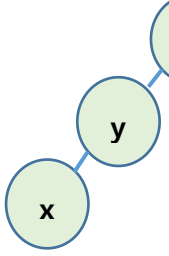
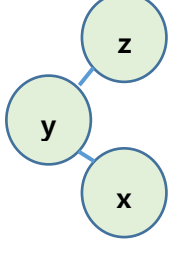
bool rotationType = 0;           // 회전종류를 나타내는 변수
if (y == z->right) {             // y가 z의 오른쪽 자식이면
    rotationType = 1;           // rotationType을 1로 바꾼다.
}

if (z != root) {                // z가 루트노드가 아니고
    if (z == z->parent->right) {  // z가 z의 부모의 오른쪽 자식이면
        z->parent->right = b;    // z의 부모노드의 오른쪽 자식을 b로 재설정한다.
    }
    else
        z->parent->left = b;
}

b->parent = z->parent;           // a < b < c 작은값 왼쪽, b가 a,b의 부모 노드가 된다.
b->left = a;                    // b의 부모를 z의 부모로 b를 부모노드로 바꿈
b->right = c;                  // b의 각 자식노드에 a, b를 연결한다.
a->parent = b;                 // a,b의 부모를 b로 설정한다.
c->parent = b;

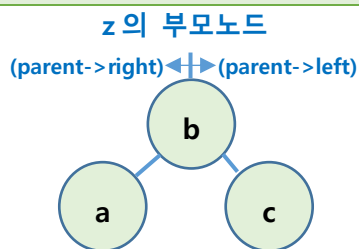
```

x,y,z 위치에 따른 4 가지 종류

y 가 z 의 오른쪽 자식 (rotationType = 1)		y 가 z 의 왼쪽 자식 (rotationType = 0)	
			

x,y,z의 위치에 따라 4 가지의 케이스가 존재한다. y가 z의 오른쪽인지, 왼쪽자식인지에 따라 각 노드의 서브트리를 다르게 처리해줘야하므로 y가 z의 어느 쪽 자식인지 저장하는 변수가 필요하다.

코드 실행 후 상태



b가 a, b의 부모노드로 구조를 바꾸는 단계이다.

z가 루트노드가 아니고 z의 부모의 오른쪽 자식이면 z의 부모노드의 오른쪽 자식을 b로 재설정한다. b가 a, b의 부모노드가 되고, b의 부모노드를 z의 부모노드로 바꾼다. b의 각 자식노드에 a, b를 연결한다.

각 서브트리를 재구조화된 세 노드들에 연결

```
1   if (rotationType == 1) {  
    //a->left = t0;  
    a->right = t1;  
    if (t1 != NULL)  
        t1->parent = a;  
    c->left = t2;  
    if (t2 != NULL)  
        t2->parent = c;  
    //c->right = t3;  
    }  
2   else if (rotationType == 0) {  
    //a->left = t3;  
    a->right = t2;  
    if (t2 != NULL)  
        t2->parent = a;  
    c->left = t1;  
    if (t1 != NULL)  
        t1->parent = c;  
    //c->right = t0;  
    }  
    if (root == z)  
        root = b;  
}
```

각 노드의 서브트리를 바꿔 달아주는 단계이다.

y가 z의 오른쪽 자식이면, a의 오른쪽 자식을 t1으로 달아주고, c의 왼쪽 자식을 t2로 달아준다.

y가 z의 왼쪽 자식이면, a의 오른쪽 자식을 t2로 달아주고, c의 왼쪽자식을 t1으로 달아준다.

t0와 t3는 변함 없이 a, c에 달려 있으므로 설정해 줄 필요가 없다.

만약 z가 root 였다면, 부모노드가 b가 됐으므로 root로 b를 지정한다.

2. 출력

print 함수

```
void Tree::print(int number) {  
    probe = 0;  
    Node* node;  
    node = search(number, root);  
    if (node != NULL) {  
        outputFile << node->data.number << " " << node->data.name << " " <<  
            node->data.department << " " << node->data.grade << " " << probe << endl;  
    }  
    else  
        outputFile << "없음 " << probe << endl;  
}
```

print 함수 : 입력한 학번과 일치한 노드를 찾기위해 노드형 포인터변수를 만든다. 노드를 search 함수를 통해 주소값을 입력 받는다. node가 받은 값이 null이 아닐 경우 노드의 데이터를 출력한다.

node가 null 일 경우 "없음"을 출력한다.

search 함수

```
Node* Tree::search(int k, Node* v) {  
    if (v == NULL)  
        return v;  
    probe++;  
  
    if (k < v->data.number) {  
        search(k, v->left);  
    }  
    else if (k == v->data.number)  
        return v;  
    else {  
        search(k, v->right);  
    }  
}
```

search 함수 : 학번에 해당 되는 키(key) 값과 노드의 주소를 인자로 받는다. insert에서 찾고자하는 학번과 root를 넣어줬다. 따라서 root부터 탐색을 시작한다. root가 null일 경우 null을 반환한다. 탐사 횟수를 1회 증가시킨다. 키 값이 root의 키 값보다 크다면 root의 왼쪽 자식노드, 작다면 오른쪽 자식노드에 대하여 search 함수를 실행 해 작업을 반복한다. 찾고자 하는 키 값과 같은 키 값의 노드를 찾았을 때 이 노드를 반환하고, null 값이 나왔다면 외부노드이므로 null을 반환한다.

- 기능별 시간&공간 복잡도

삽입

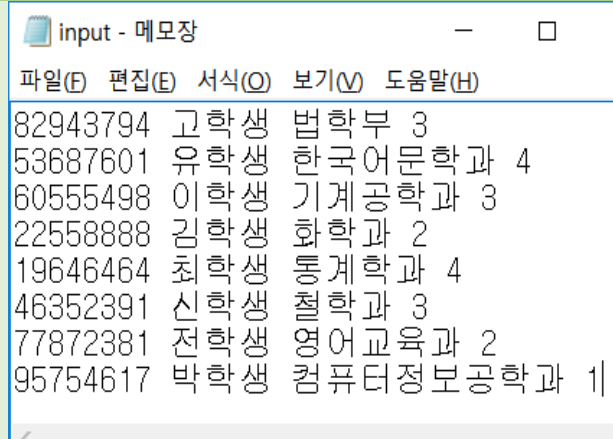
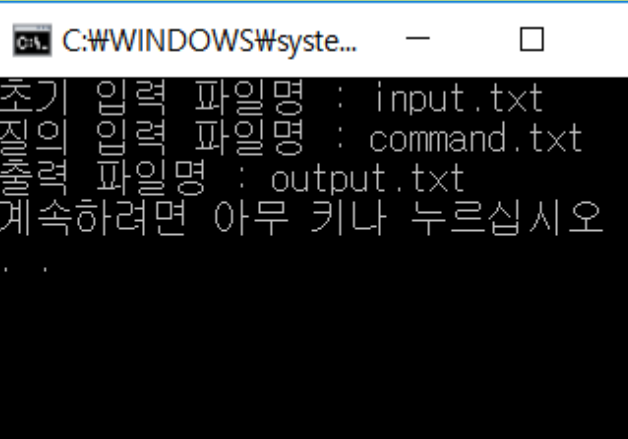
- **시간복잡도** : 최악의 경우 tree의 높이만큼 탐색을 수행하고 삽입한다. 노드의 개수가 n 일때 트리의 높이는 $\log n$ 에 가까이 유지되므로 $O(\log n)$ 의 수행시간이 걸린다. 삽입 시 발생하는 재구조화는 최대 1번만 수행하기 때문에 $O(1)$ 이므로, 삽입 시 최악의 경우 걸리는 총 수행시간은 $O(\log n)$ 이다.
- **공간복잡도** : 삽입 시 노드의 크기만큼 새로운 공간이 필요하므로 $O(1)$ 이다. 삽입 후 노드가 삽입될 위치를 찾을 때 사용하는 nodePosition 함수에서 최악의 경우 tree의 높이만큼 함수를 호출해야하므로 $O(\log n)$ 만큼의 공간이 필요하다. 그리고 재구조화가 일어나는 판단하기 위해 노드의 각 자식들의 높이를 비교해야 한다. 이 때 height 함수가 재귀함수이므로 최악의 경우 $O(\log n)$ 만큼의 공간을 사용한다. 따라서 총 $O(\log n)$ 만큼의 공간이 필요하다.

탐색

- **시간복잡도** : 최악의 경우 tree의 높이만큼 탐색을 수행하고 찾는다. 노드의 개수가 n 일때 트리의 높이는 $\log n$ 에 가까이 유지되므로 최악의 경우 $O(\log n)$ 의 수행시간이 걸린다.
- **공간복잡도** : search 함수가 재귀함수이므로 최악의 경우 트리의 높이만큼 호출되기 때문에 $O(\log n)$ 이다.

4. 인터페이스 및 사용법

초기 입력 파일 설정,


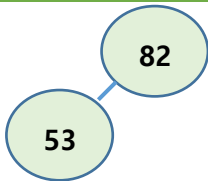
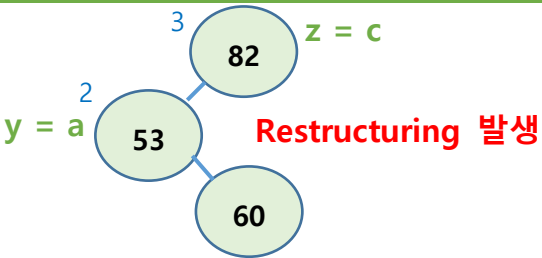
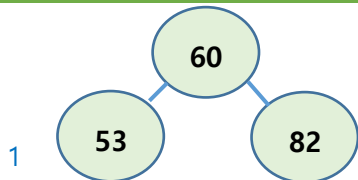
input.txt	cmd
 <p>input - 메모장</p> <p>파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)</p> <p>82943794 고학생 법학부 3 53687601 유학생 한국어문학과 4 60555498 이학생 기계공학과 3 22558888 김학생 화학과 2 19646464 최학생 통계학과 4 46352391 신학생 철학과 3 77872381 전학생 영어교육과 2 95754617 박학생 컴퓨터정보공학과 1</p>	 <p>C:\WINDOWS\system...</p> <p>초기 입력 파일명 : input.txt 질의 입력 파일명 : command.txt 출력 파일명 : output.txt 계속하려면 아무 키나 누르십시오 . . .</p>

실행창에서 초기 및 질의 입력, 출력 파일명 이름을 입력한다. 메인 함수에서 초기 입력 파일을 먼저 불러들여 각 학생정보를 차례대로 받아서 AVL 트리 형태로 만든다. 이후 질의 입력 파일을 불러 각 단계를 수행한다. 그리고 수행하여 나온 값을 출력 파일에 저장한다.

삽입과정

파랑색 숫자 : 높이

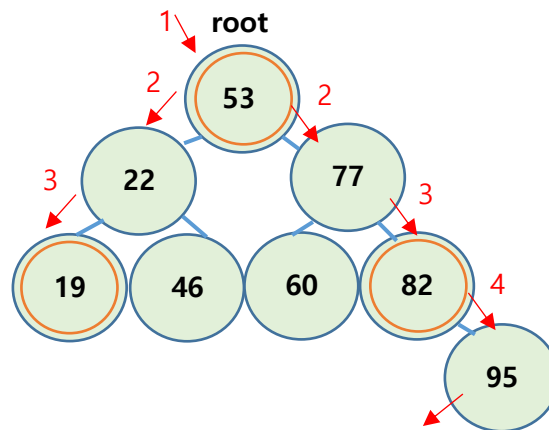
z : 자식의 높이가 2 이상 차이나는 문제노드 y : z의 자식 x : y의 자식
a, b, c : 노드의 값을 비교 ($a < b < c$)

82943794 삽입	53687601 삽입
	
60555498 삽입	구조 변경 후
	

22558888 삽입	19646464 삽입
<pre> graph TD 60((60)) --- 53((53)) 60 --- 82((82)) 53 --- 22((22)) </pre>	<pre> graph TD 60((60)) --- 53((53)) 60 --- 82((82)) 53 --- 22((22)) 53 --- 19((19)) 22 --- 19 style 19 fill:#d3d3d3 </pre> <p>z = c y = b x = a Restructuring 발생</p>
22558888 삽입	46352391 삽입
<pre> graph TD 60((60)) --- 22((22)) 60 --- 82((82)) 22 --- 19((19)) 22 --- 53((53)) </pre>	<pre> graph TD 60((60)) --- 22((22)) 60 --- 82((82)) 22 --- 19((19)) 22 --- 53((53)) 53 --- 46((46)) style 46 fill:#d3d3d3 </pre> <p>y = a z = c x = b Restructuring 발생</p>
	77872381 삽입
<pre> graph TD 53((53)) --- 22((22)) 53 --- 60((60)) 22 --- 19((19)) 22 --- 46((46)) 60 --- 82((82)) </pre>	<pre> graph TD 53((53)) --- 22((22)) 53 --- 60((60)) 22 --- 19((19)) 22 --- 46((46)) 60 --- 82((82)) 60 --- 77((77)) style 77 fill:#d3d3d3 </pre> <p>z = a y = c x = b Restructuring 발생</p>
	95754617 삽입
<pre> graph TD 53((53)) --- 22((22)) 53 --- 77((77)) 22 --- 19((19)) 22 --- 46((46)) 77 --- 60((60)) 77 --- 82((82)) </pre>	<pre> graph TD 53((53)) --- 22((22)) 53 --- 77((77)) 22 --- 19((19)) 22 --- 46((46)) 77 --- 60((60)) 77 --- 82((82)) 82 --- 95((95)) </pre>

Step 1. 학생 검색하기

command.txt	output.txt
<div>command - 메모장</div> <div>파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)</div> <div> s 53687601 s 19646464 s 82943794 s 94717719 </div>	<div>output - 메모장</div> <div>파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)</div> <div> 53687601 유학생 한국어문학과 4 1 19646464 최학생 통계학과 4 3 82943794 고학생 법학부 3 3 없음 4 </div>



외부노드이므로 probe 횟수 증가 X

s 입력을 통해 입력한 학번을 AVL 트리에서 찾을 수 있다.

- 53687601 : 입력한 학번을 찾기 위해 처음 루트노드부터 시작한다. 루트노드가 입력한 값과 동일하므로 학생의 정보를 출력한다. 탐사횟수는 루트노드에서 바로 찾았기 때문에 1번이다.
- 19646464 : 루트노드부터 탐색을 시작한다. 53보다 작은 숫자이기 때문에, 노드의 왼쪽 자식노드를 탐색한다. 22 보다도 작기 때문에 노드의 왼쪽 자식노드를 탐색한다. 탐색한 값의 학번과 동일하기 때문에 학생의 데이터를 출력한다. 이때의 탐사횟수는 총 3번이다.
- 82943794 : 루트노드 53 보다 큰 값이므로, 노드의 오른쪽 자식노드를 탐색한다. 77보다 큰 값이므로 오른쪽 자식노드를 탐색한다. 그 결과, 같은 학번이 나오므로 학생의 데이터를 출력한다. 이 때의 탐사횟수는 총 3번이다.
- 94717719 : 마찬가지로 루트노드부터 일치하는 값이 나올 때까지 자식노드로 내려가며 크기를 계속하여 비교한다. 53보다 크므로 77를 탐색하고, 77보다 크므로 82를 탐색한다. 82보다 크므로 95를 탐색한다. 95보다 작은 값이므로 95의 왼쪽자식노드를 탐색해야하지만, null값이므로 탐색을 중지하고, 탐색하는 노드가 없음을 출력한다. 탐사횟수는 53-77-82-95로 총 4번이다.

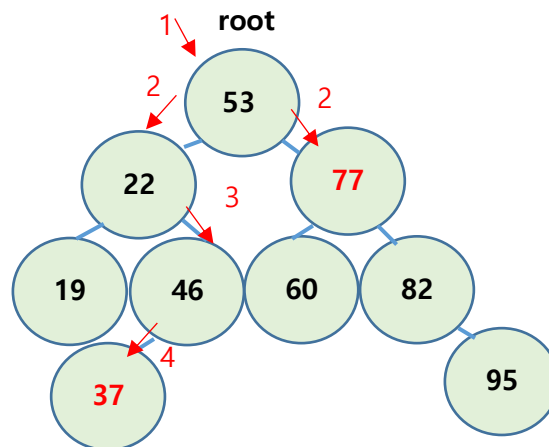
Step 2. 학생 추가하기

command.txt

```
command - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
s 53687601
s 19646464
s 82943794
s 94717719
i 37470006 정학생 정치외교학과 4
i 77872381 허학생 정보공학계열 1
```

output.txt

```
output - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
53687601 유학생 한국어문학과 4 1
19646464 최학생 통계학과 4 3
82943794 고학생 법학부 3 3
없음 4
3
추가할 수 없음 2
```



i 입력을 통해 입력한 학생의 정보를 AVL 트리에 추가시킬 수 있다.

- 37470006 : 루트노드부터 값을 비교하여, 값이 더 작기 때문에 왼쪽 노드로 간다. 왼쪽노드보다 값이 크므로 오른쪽노드로 간다, 오른쪽 노드보다 값이 작으므로 왼쪽노드로 간다. 왼쪽노드는 null이므로 이 자리에 노드를 트리의 형태에 맞게 연결 시킨다. 이후 노드에 문제가 없는지 확인한다. 노드의 부모들의 높이가 2 이상 차이나는 구간이 없으므로 root노드까지 확인 후 insert를 종료시킨다.

- 77872381 : 루트노드와 비교했을 때 더 크므로 오른쪽 노드로 간다. 오른쪽 노드에 동일한 학번이 존재하므로, 노드를 추가시킬 수 없다.

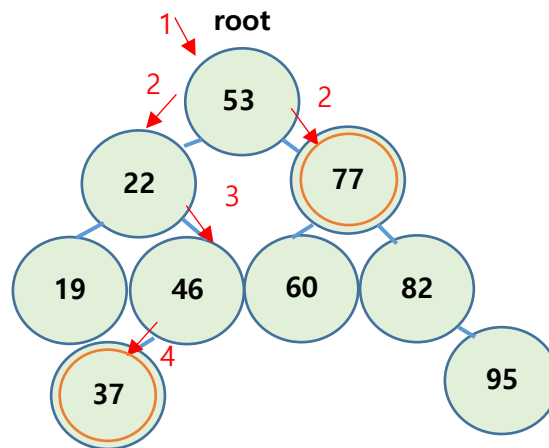
Step 3. 학생 추가 작업 수행 후 재검색

command.txt

```
command - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
s 53687601
s 19646464
s 82943794
s 94717719
i 37470006 정 학생 정치외교학과 4
i 77872381 허 학생 정보공학계열 1
s 37470006
s 77872381
```

output.txt

```
output - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
53687601 유 학생 한국어문학과 4 1
19646464 최 학생 통계학과 4 3
82943794 고 학생 법학부 3 3
없음 4
3
추가할 수 없음 2
37470006 정 학생 정치외교학과 4 4
77872381 전 학생 영어교육과 2 2
```



s 입력을 통해 입력한 학번을 AVL 트리에서 찾을 수 있다.

- 37470006 : AVL트리의 구조가 변하지 않았으므로, 루트부터 비교하여 노드가 삽입된 자리까지 찾아간다. 이 때 탐사 횟수는 총 4번이다.

- 77872381 : 이 학생의 노드 또한 이미 있는 값으로 추가 되지 않았기 때문에 노드의 위치는 변동이 없다.

5. 평가 및 개선 방향

1. 장점

AVL Tree를 이용하면 높이가 $\log n$ 에 가까이 유지된다. 재구조화(restructuring)는 최악의 경우에도 $O(1)$ 의 수행시간이 걸려 탐색, 삽입, 삭제 모두 최악의 경우 $O(\log n)$ 수행시간에 해결할 수 있다. 이로써 세가지 연산이 모두 자주 일어나고, 최악의 경우에도 빠른 속도를 보여야할 경우에 이 프로그램은 최적화 되어있다.

2. 단점

삽입 시 균형이 맞는지 확인하는 `restructuringCheck` 함수에서 많은 수행과정이 일어나고, 매번 문제가 있는지 서브트리들의 높이를 확인하기 때문에 수행시간이 지연될 수 있다. 또한 Binary Search Tree에 비해 구현이 복잡하고 여러가지 경우의 수를 나눠줘야 한다.

3. 향후 개선 방향

삽입 시 발생하는 재구조화 과정에서 각 노드들의 서브트리인 `t0`와 `t3`는 `a`, `c`노드에 위치가 변하지 않고 그대로 매달려있다. 명확한 이해를 위해 `t0`와 `t3`를 선언하였지만 사실상 필요가 없는 변수이다. 그리고 여러 함수들에서 인자의 이름을 `node`로 지정하여, 무엇을 지정한 것인지 명확성이 떨어진다. 인자의 이름들을 역할에 맞게 지어줄 필요성이 있다.

삽입과정이 최악의 경우에도 $O(\log n)$ 의 수행시간이 걸리지만, 균형을 확인하고 재구조화하는 과정에서 복잡한 수행과정을 거치기 때문에 단순화하고 효율적으로 수행하는 방법을 고민해봐야 할 것이다.