과제 13 - AVL 트리

자료구조 3분반
B411114 신재협

2017. 12. 21

<개요>

HW13은 AVL 트리 자료구조를 이용하여 사용자가 입력한 키 값을 트리에 알맞는 위치에 저장하는 과제입니다.

<요구사항>

주어진 키 값들을 main이 시작됨과 함께 AVL트리에 추가합니다. 이후에는 사용자의 선택에 따라 검색, 입력, 삭제의 명령을 수행합니다.

해당 입력, 삭제의 명령을 수행한 뒤에는 루트 노드부터 마지막 리프노드까지 각 노드와 노드의 왼쪽,오른쪽 자식 노드를 출력합니다. 그리고 검색의 명령을 수행할 때에는 루트 노드부터 검색하고자 하는 노드까지의 경로를 출력합니다.
일반적인 Binary Search Tree와 비슷한 입력, 삭제, 검색 방법이 진행되나 입력, 삭제시 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이를 고려하여 균형잡힌 AVL트리를 구성해야합니다.

다른 헤더파일 없이 hw13.cpp 소스파일 내에 트리의 노드에 해당하는 Node 클래스와 AVL 트리에 해당하는 AVL 클래스이 구현되어 있습니다.

<hw13.cpp 전체 코드>

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

template<class K> class AVL;

template<class K>
class Node
{
friend class AVL<K>;
private:
K key;
Node<K> *leftChild, *rightChild, *parent;
public:
Node(K k)
{
key = k;
leftChild = rightChild = parent = NULL;
}
};

template<class K>
class AVL
{
private:
Node<K> *root;
int count; // main에서 초기값으로 입력되는 19개의 숫자는 조회하지 않
도록 하는 카운트 변수
public:
//생성자
AVL()
{
root = NULL;
count = 0;
}

//LL회전
Node<K>* LL(Node<K>* node)
{
Node<K>* ptr;
ptr = node->leftChild;
```

```
node->leftChild = ptr->rightChild;
ptr->rightChild = node;

//기존의 부모 노드에 대한 관계 갱신
ptr->parent = node->parent;
if (ptr->parent->leftChild == node)
{
ptr->parent->leftChild = ptr;
}
else
{
ptr->parent->rightChild = ptr;
}
node->parent = ptr;

return ptr;
}
//RR회전
Node<K>* RR(Node<K>* node)
{
Node<K>* ptr;
ptr = node->rightChild;
node->rightChild = ptr->leftChild;
ptr->leftChild = node;

//기존의 부모 노드에 대한 관계 갱신
ptr->parent = node->parent;
if (ptr->parent->leftChild == node)
{
ptr->parent->leftChild = ptr;
}
else
{
ptr->parent->rightChild = ptr;
}
node->parent = ptr;

return ptr;
}
//LR회전
Node<K>* LR(Node<K>* node)
{
Node<K>* ptr;
ptr = node->leftChild;
```

```
node->leftChild = RR(ptr);

return LL(node);
}
//RL회전
Node<K>* RL(Node<K>* node)
{
Node<K>* ptr;
ptr = node->rightChild;
node->rightChild = LL(ptr);

return RR(node);
}
// 트리의 높이를 구하기 위한 함수
int GetH(Node<K> * node)
{
int leftH, //node의 왼쪽 서브트리 높이
rightH; //오른쪽 높이

if (node == NULL)
{
return 0;
}

leftH = GetH(node->leftChild);
rightH = GetH(node->rightChild);

if (leftH > rightH)
{
return leftH + 1;
}
else
{
return rightH + 1;
}
}

int GetBF(Node<K> * node)
{
int lh, //왼쪽 서브트리 높이
rh; //오른쪽 서브트리 높이

if (node == NULL)
{
```

```
return 0;
}

lh = GetH(node->leftChild);
rh = GetH(node->rightChild);

return lh - rh;
}
// AVL 균형을 맞추기 위한 밸런싱 함수
Node<K>* Balance(Node<K>* node)
{
if (node == NULL)
{
return node;
}

int bf = GetBF(node);

if (bf > 1) //왼쪽 서브 트리 방향으로 높이가 2 이상 크다면
{
if (GetBF(node->leftChild) >= 0)
{
node = LL(node);
}
else
{
node = LR(node);
}
}

if (bf < -1) //오른쪽 서브 트리 방향으로 높이가 2 이상 크다면
{
if (GetBF(node->rightChild) <= 0)
{
node = RR(node);
}
else
{
node = RL(node);
}
}
return node;
}
//탐색 함수
```

```
void Search(K k)
{
Node<K>* ptr = root;
if (ptr == NULL)
{
return;
}
while (ptr != NULL)
{
if (k == ptr->key)
{
cout << ptr->key << endl;
return;
}
else if (k < ptr->key)
{
cout << ptr->key << " -> ";
ptr = ptr->leftChild;
continue;
}
else if (k > ptr->key)
{
cout << ptr->key << " -> ";
ptr = ptr->rightChild;
continue;
}

}
cout << endl;
cout << "-------------해당 키가 존재하지 않습니다.-------------" << endl;
}
//삽입 함수
void Add(K k)
{
//트리에 어떠한 노드도 존재하지 않을 때
if (root == NULL)
{
root = new Node<K>(k);
return;
}

Node<K>* ptr = root; // 비어있는 노드일 때까지 탐색
Node<K>* pptr = NULL; //ptr의 부모 노드
```

```
  //삽입 위치에 대한 탐색
while (ptr != NULL)
{
pptr = ptr;
if (k < ptr->key)
{
ptr = ptr->leftChild;
continue;
}
if (k > ptr->key)
{
ptr = ptr->rightChild;
continue;
}
if (k == ptr->key)
{
//AVL트리에 대한 전체 레벨 순회
LevelOrder();
cout << "-------------해당 키가 이미 존재합니다.-------------" << endl;
return;
}
}
// 탐색한 삽입 위치에 키 값 k를 가지는 노드 삽입
if (pptr != NULL)
{
if (k < pptr->key)
{
pptr->leftChild = new Node<K>(k);
pptr->leftChild->parent = pptr;
}
else if (k > pptr->key)
{
pptr->rightChild = new Node<K>(k);
pptr->rightChild->parent = pptr;
}
else if (k == pptr->key)
{
LevelOrder();
return;
}
}

//AVL트리를 만들기 위해 균형 조정
while (pptr->parent != NULL)
```

```
{
pptr = Balance(pptr);
pptr = pptr->parent;
}

//삽입or삭제시 트리 조회시켜줘야함!!
//이곳에 큐 사용하여 레벨순회 하도록 하자!! 삭제의 경우에도 마찬가지.
if (count >= 17)
{
//AVL트리에 대한 전체 레벨 순회
LevelOrder();
}
count++; //한번 Add호출될 때 마다 count변수 1증가
}
//삭제 함수
void Delete(K k)
{
Node<K>* ptr = root; //삭제할 노드를 찾기 위한 ptr
Node<K> * pptr; //ptr의 부모 노드
Node<K> * tmpptr; //임시 노드

// 삭제 대상을 저장한 노드 탐색
while (ptr != NULL&&ptr->key != k)
{
pptr = ptr;
if (k < ptr->key)
{
ptr = ptr->leftChild;
}
else
{
ptr = ptr->rightChild;
}
}

//대상이 존재하지 않는다면,
if (ptr == NULL)
{
//AVL트리에 대한 전체 레벨 순회
LevelOrder();
cout << "-------------해당 키가 존재하지 않습니다.-------------" << endl;
return;
}
```

```
tmpptr = ptr;
//첫번째: 삭제할 노드가 자식을 갖지 않을 경우
if (tmpptr->leftChild == NULL&&tmpptr->rightChild == NULL)
{
if (pptr->leftChild == tmpptr)
{
pptr->leftChild = NULL;
}
else
{
pptr->rightChild = NULL;
}
}

//두번째: 하나의 자식을 갖는 경우
else if (tmpptr->leftChild == NULL || tmpptr->rightChild == NULL)
{
Node<K> * ctmpptr; // tmpptr의 자식
if (tmpptr->leftChild == NULL) //오른쪽 자식만을 갖는 경우
{
ctmpptr = tmpptr->rightChild;
}
else //왼쪽 자식만을 갖는 경우
{
ctmpptr = tmpptr->leftChild;
}

if (pptr->leftChild == tmpptr)
{
pptr->leftChild = ctmpptr;
ctmpptr->parent = pptr;
}
else
{
pptr->rightChild = ctmpptr;
ctmpptr->parent = pptr;
}
}

//세번째: 두 개의 자식을 갖는 경우
else
{
//lc는 지울 노드의 왼쪽 서브트리의 루트 노드를 가리킨다.
Node<K> *lc = tmpptr->leftChild;
```

```
if (!lc->rightChild)
{
tmpptr->key = lc->key;
tmpptr->leftChild = lc->leftChild;
delete lc;
}
else
{
//삭제할 노드를 대체할 노드 탐색
while (lc->rightChild != NULL)
{
pptr = lc;
lc = lc->rightChild;
}
//대체할 노드에 저장된 값을 삭제할 노드에 대입한다.
tmpptr->key = lc->key;
pptr->rightChild = lc->leftChild;
delete lc;
}
}
//AVL트리를 만들기 위해 균형 조정
while (pptr->parent != NULL)
{
pptr = Balance(pptr);
pptr = pptr->parent;
}
//AVL트리에 대한 전체 레벨 순회
LevelOrder();
}
//트리 조회를 위한 레벨 순회 함수
void LevelOrder()
{
queue< Node<K>* > q; //레벨 순회를 위한 큐
q.push(root);
while (!q.empty())
{
Node<K>* tmp = q.front();
cout << tmp->key;
if (tmp->leftChild != NULL)
{
q.push(tmp->leftChild);
cout << " left: " << tmp->leftChild->key;
}
if (tmp->rightChild != NULL)
```

```
{
q.push(tmp->rightChild);
cout << " right: " << tmp->rightChild->key;
}
cout << endl;
q.pop();
}
}
};

int main()
{
AVL<int> avlT;
//주어진 초기값을 직접 AVL트리에 입력
//19,10,46,4,14,37,55,7,12,18,28,40,51,61,21,32,49,58 순서
avlT.Add(19);
avlT.Add(10);
avlT.Add(46);
avlT.Add(4);
avlT.Add(14);
avlT.Add(37);
avlT.Add(55);
avlT.Add(7);
avlT.Add(12);
avlT.Add(18);
avlT.Add(28);
avlT.Add(40);
avlT.Add(51);
avlT.Add(61);
avlT.Add(21);
avlT.Add(32);
avlT.Add(49);
avlT.Add(58);

while (1)
{
int sel1;
cout << "AVL Tree" << endl;
cout << "1. Search" << endl;
cout << "2. Add" << endl;
cout << "3. Delete" << endl;
cin >> sel1;

switch (sel1)
```

```
{
int sel2;
case 1:
cout << "Select : ";
cin >> sel2;
cout << "[Result]" << endl;
avlT.Search(sel2);
break;
case 2:
cout << "Add : ";
cin >> sel2;
cout << "[Result]" << endl;
avlT.Add(sel2);
break;
case 3:
cout << "Delete : ";
cin >> sel2;
cout << "[Result]" << endl;
avlT.Delete(sel2);
break;
}
}
}
```

<class설계 내용 및 이유>

Node 클래스: AVL클래스에 대해 friend선언을 하여 AVL트리에서 Node클래스를 이용할 수 있도록 했습니다.
private멤버로는 해당 노드가 갖는 키 값을 선언했습니다. 또한 여러가지 함수에 사용하기 위한 왼쪽 자식 노드, 오른쪽 자식 노드, 부모 노드에 대한 포인터 값을 선언했습니다.
public 멤버로는 생성자만을 선언했습니다. 주요 함수들은 AVL클래스에서 선언하였습니다.

AVL 클래스: AVL트리를 나타내기 위한 클래스입니다.
private멤버로는 전체 트리에 대한 루트 노드를 표현하기 위한 Node형 포인터 root와 초기값으로 입력되는 19개의 입력은 따로 조회(레벨순회)하지 않도록 하는 카운트 변수 count를 선언했습니다.
public멤버로는 root를 NULL로 초기화하는 생성자가 있습니다. 또한 삽입, 삭제, 검색, 순회를 위한 함수들과 AVL트리로서의 균형을 맞추기 위한 Balance 함수와 이에 이용되는 균형인자BF를 구하는 함수, 주어진 노드를 루트로 하는 트리에 대한 높이를 구하는 함수, AVL트리의 균형을 맞출 때 균형인자에 따라 작동하는 LL, RR, LR, RL 회전 함수가 있습니다.

<AVL트리 연산 설명>

AVL트리를 의미하는 AVL 클래스는 일반적인 삽입,삭제,검색,순회 함수와 다양한 함수를 통해 AVL트리의 균형을 유지하도록 합니다.

- 삽입 함수 void Add(K k)
: 키 값으로 k를 갖는 노드를 트리의 알맞는 위치에 삽입하는 함수입니다. 만약 트리가 비어있다면 k값을 키 값으로 하는 루트 노드를 만들고 함수를 종료합니다. 비어있지 않다면 반복문을 통해 삽입할 위치를 탐색합니다. 탐색을 위해서 키 값의 비교를 진행하는데 이미 존재하는 키 값이라면 탐색을 멈추고 전체 트리에 대한 순회만을 진행하고 함수를 종료합니다. 이미 존재하는 키 값이 아니라면 그 삽입할 위치에 키 값 k를 가지는 노드를 생성하고 새로 생성된 노드의 부모 노드를 설정합니다. 삽입이 완료된 후에는 삽입한 노드의 부모 노드부터 루트 노드까지 올라가면서 올라갈 때마다 Balance함수를 통해 그 노드를 루트 노드로 하는 서브트리의 균형을 맞춥니다. 균형을 다 맞추고 난 뒤에는 전체 트리에 대한 레벨 순회를 합니다. 과제에서 초기 값으로 주어진 키 값들은 굳이 순회할 필요가 없다고 생각하여 기존에 입력된 18개의 입력은 순회하지 않도록 구현하였습니다.

- 삭제 함수 void Delete(K k)
: 키 값으로 k를 갖는 노드를 삭제하는 함수입니다. 처음에는 삭제 대상을 탐색합니다. 만약 삭제 대상이 존재하지 않는다면 전체 트리에 대한 순회만을 마치고 함수를 종료합니다.
삭제 대상이 존재한다면 3가지 경우로 나누어 삭제를 진행합니다. 첫번째로 삭제할 노드가 자식을 갖지 않는 경우에는 삭제 대상 노드만을 삭제합니다. 두번째로 하나의 자식을 갖는 경우에는 그 자식이 왼쪽 자식인지 오른쪽 자식인지를 구별하여 삭제를 진행합니다. 마지막으로 두 개의 자식 노드를 갖는 경우에는 삭제하고자 하는 노드의 왼쪽 서브트리에서 가장 키 값이 큰 노드를 찾아서 삭제 대상 노드에 대입합니다.
삭제가 마무리 되면 삽입 함수와 마찬가지로 삽입한 노드의 부모 노드부터 루트 노드까지 올라가면서 올라갈 때마다 Balance함수를 통해 그 노드를 루트 노드로 하는 서브트리의 균형을 맞춥니다. 균형을 다 맞추고 난 뒤에는 전체 트리에 대한 레벨 순회를 합니다.

- 탐색 함수 void Search(K k)
: 키 값으로 k를 갖는 노드를 탐색하는 함수입니다. 트리가 비어있다면 함수를 종료합니다. 비어있지 않다면 키 값의 비교 연산을 통해 일반적인 BST와 같이 탐색을 진행하여 탐색 대상의 노드가 존재한다면 노드의 키 값을 출력합니다. 존재하지 않는다면 함수를 종료합니다.

- 순회 함수 void LevelOrder()
: 과제에서 요구하는 출력을 위해서 구현한 레벨 순회 함수입니다. STL queue를 이용하여 루트 노드부터 마지막 리프노드까지 노드의 키 값과 그 노드의 왼

쪽, 오른쪽 자식 노드의 키 값을 출력하도록 했습니다.

- AVL트리의 균형을 맞추는 함수 Node<K>* Balance(Node<K>* node)
:트리가 비어있다면 매개변수 node 그대로를 반환합니다. 비어있지 않다면 균형 인자 bf를 이용하여 node가 AVL트리로서 균형이 적당한 지를 구분하고 적당하지 않다면 그에 맞는 회전 함수를 호출합니다. 여기서 균형 인자 bf는 "왼쪽 서브 트리 높이 - 오른쪽 서브 트리 높이" 값입니다.

- 균형 인자를 구하는 함수 int GetBF(Node<K> * node)
: 왼쪽 서브 트리의 높이와 오른쪽 서브 트리의 높이의 차를 의미하는 균형 인자 bf를 반환하는 함수입니다.

- 트리 높이를 구하는 함수 int GetH(Node<K> * node)
: 재귀를 통해 트리의 높이를 의미하는 왼쪽 서브 트리의 높이와 오른쪽 서브 트리의 높이 중 가장 큰 값을 출력하는 함수입니다.

- LL회전 Node<K>* LL(Node<K>* node)
: 매개변수 node를 기준으로 균형 인자가 1보다 크고 node 왼쪽 자식의 균형 인자가 0보다 크다면 실행되는 LL회전 함수입니다. 우선 node의 왼쪽 자식 노드를 ptr이 가리키도록 합니다. 그리고 ptr의 오른쪽 자식 노드를 node의 왼쪽 자식 노드로 설정합니다. 마지막으로 node를 ptr의 오른쪽 자식노드로 설정해줍니다. 그 뒤에는 노드의 부모 노드를 의미하는 parent를 회전한 결과에 맞도록 설정하고 ptr을 반환합니다.

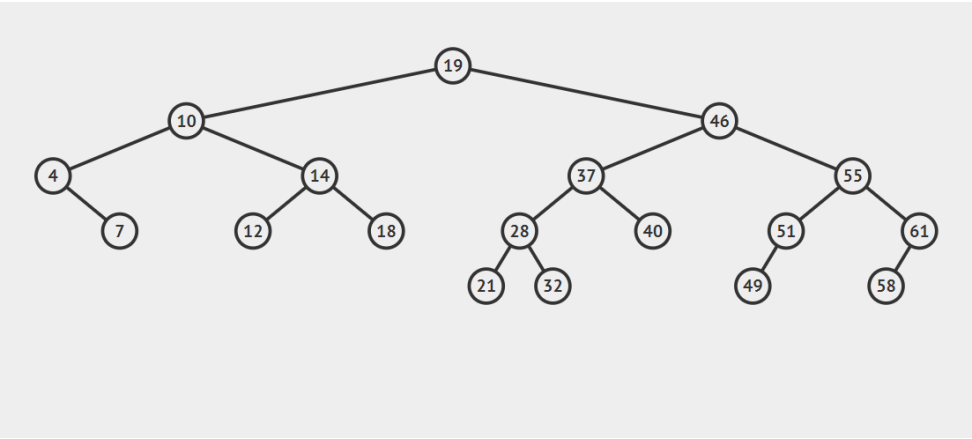
- RR회전 Node<K>* RR(Node<K>* node)
: 매개변수 node를 기준으로 균형 인자가 -1보다 작고 node 오른쪽 자식의 균형 인자가 0보다 작다면 실행되는 RR회전 함수입니다. 우선 node의 오른쪽 자식 노드를 ptr이 가리키도록 합니다. 그리고 ptr의 왼쪽 자식 노드를 node의 오른쪽 자식 노드로 설정합니다. 마지막으로 node를 ptr의 왼쪽 자식노드로 설정해줍니다. 그 뒤에는 노드의 부모 노드를 의미하는 parent를 회전한 결과에 맞도록 설정하고 ptr을 반환합니다.

- LR회전 Node<K>* LR(Node<K>* node)
: 우선 ptr이 매개변수 node의 왼쪽 자식 노드를 가리키도록 합니다. 그리고 ptr에 대해 부분적인 RR회전을 수행하여 LR회전이였던 상태를 LL상태로 만듭니다. 마지막으로 node에 대해 LL회전을 진행하고 LL회전의 반환 값을 반환합니다.

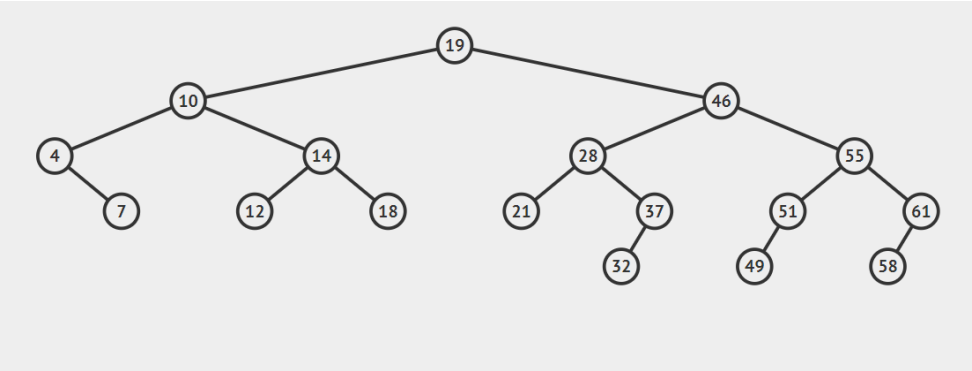
- RL회전 Node<K>* RL(Node<K>* node)
: 우선 ptr이 매개변수 node의 오른쪽 자식 노드를 가리키도록 합니다. 그리고 ptr에 대해 부분적인 LL회전을 수행하여 RL회전이였던 상태를 RR상태로 만듭니다. 마지막으로 node에 대해 RR회전을 진행하고 RR회전의 반환 값을 반환합니다.

<Delete 연산 전,후 비교>

키 값이 40인 노드 Delete 전 트리의 모습



키 값이 40인 노드 Delete 후 트리의 모습



<결과 분석>

실행된 프로그램에서 탐색을 하기 위해 숫자 1을 입력하고 찾고자 하는 키 값을 입력하면 루트 노드부터 찾고자 하는 노드까지의 경로를 출력합니다. 만약 찾고자 하는 노드가 트리 내에 존재하지 않는다면 그 노드를 찾기 위한 경로와 함께 끝내 찾지 못했다는 출력문이 출력 됩니다.
노드를 추가(입력)하기 위해 숫자 2를 누르고 추가하고자 하는 키 값을 입력하면 AVL 트리에 알맞는 위치로 노드가 추가되고 출력된 이후의 전체 트리에 대한 값이 출력됩니다. 이때 루트 노드와 그에 연결된 왼쪽,오른쪽 자식 노드의 키 값이 조회됩니다. 만약 추가하고자 하는 키 값이 이미 존재한다면 전체 트리에 대한 내용과 함께 이미 존재하는 키 값이라는 출력문을 출력합니다.
노드를 삭제하기 위해 숫자 3을 누르고 삭제하고자 하는 키 값을 입력하면 AVL 트리의 균형을 고려한 삭제가 진행되고 삭제된 이후의 전체 트리에 대한 내용이 출력됩니다. 만약 삭제하고자 하는 키 값이 존재하지 않는다면 전체 트리에 대한 내용과 함께 해당 키 값이 존재하지 않는다는 출력문을 출력합니다.

<결과 값 스크린 샷>

앞에서 설명드린 각 명령에 대한 예시 출력입니다.

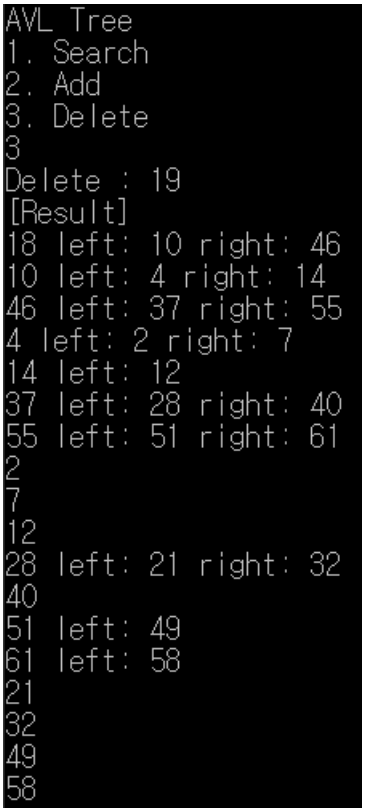
키 값이 18인 노드 Search 결과

```
AVL Tree
1. Search
2. Add
3. Delete
1
Select : 18
[Result]
19 -> 10 -> 14 -> 18
```

키 값이 2인 노드 Add 결과

```
AVL Tree
1. Search
2. Add
3. Delete
2
Add : 2
[Result]
19 left: 10 right: 46
10 left: 4 right: 14
46 left: 37 right: 55
4 left: 2 right: 7
14 left: 12 right: 18
37 left: 28 right: 40
55 left: 51 right: 61
2
7
12
18
28 left: 21 right: 32
40
51 left: 49
61 left: 58
21
32
49
58
```

키 값이 18인 노드 Delete 결과



<구현하면서 어려웠던 부분>

일반적인 삽입, 삭제, 탐색의 구현은 BST과제에서 연습해봤기 때문에 어렵지 않았지만 삽입, 삭제 연산 뒤에 AVL트리의 균형을 맞춰주는 연산은 구현하는 과정에서 갖가지 오류가 발생하는 등 쉽지 않았습니다.
그에 대한 해결책으로 각 노드에 대한 부모 노드인 Node포인터 parent를 선언함으로써, 새로 삽입하거나 삭제한 노드 혹은 새로 삽입한 노드의 부모 노드부터 루트 노드까지 조상 노드를 따라가면서 AVL 균형을 맞추는 진행을 구현해낼 수 있었습니다.

또한 AVL 트리의 균형을 맞추는 함수 Balance 함수를 구현할 때 상황에 따라 필요한 회전을 어떻게 구분해야 할지, 그리고 사용되는 트리의 높이를 구하기 위해서 어떻게 구현해야 할지에 대해 고심했습니다.
그에 대한 해결책으로 강의록과 전공 서적을 통해 회전 원리와 균형인자(Balance Factor)의 사용법에 대해 다시 공부하고 이해함으로써 구현할 수 있었습니다.