자료구조 HW12 AVL tree

C211123 이준선

2022년 12월 16일

목차

그림목차

코드목차

1 개요

AVL tree를 구현하였고, AVL tree의 핵심인 LL, LR, RR, RL 회전에 대한 설명을 담았다. 또한 클래스 구현 내용과 설계 내용을 담았다.

Note : LaTeX보고서의 tex 파일엔 tikz package의 graphdrawing.tree library가 담겼습니다. 이 라이브러리는 **'LuaLaTeX'** 컴파일러 환경에서만 작동하므로, 확인 시 LuaLaTeX컴파일러로 실행해주십시오. 일반적으로 많이 쓰이는 pdfLaTeX에서는 컴파일되지 않습니다.

2 Class 설계 내용 및 이유

```
#ifndef AVL_H
#define AVL_H
#include <queue>
#include <iostream>
using namespace std;

class Node {
public:
  int GetBF();
  int GetData();
  void SetData(const int data);
  void SetBF(const int bf);
  Node* GetLeftChild();
  Node* GetRightChild();
  Node* GetParent();
  void SetLeftChild(Node* leftChild);
  void SetRightChild(Node* rightChild);
  void SetParent(Node* parent);
  Node(int data, int bf, Node* leftChild, Node* rightChild, Node*
    parent);
  Node(int data);
  Node();

  Node*& GetLeftChildRef();
  Node*& GetRightChildRef();
  Node*& GetParentRef();
private:
  int data;
  int bf;
  Node* leftChild;
  Node* rightChild;
  Node* parent;
};

class bstree {
private:
  bool rotation(Node* start, Node* end);
```

```
37    inline void changeLeftSubtree(Node* main, Node* sub);
38    inline void changeRightSubtree(Node* main, Node* sub);
39    int GetHeight(Node* root);
40    int GetBalanceFactor(Node* root);
41    Node* rotateLL(Node* root);
42    Node* rotateRR(Node* root);
43    Node* rotateLR(Node* root);
44    Node* rotateRL(Node* root);
45    void Rebalance(Node*& root);
46    Node* findMin(Node* root);
47    Node* findNode(const int key, Node* root);
48    Node* InorderSucc(Node* current);
49    void ResetParent(Node*& root);
50    void ResetBF(Node*& root);
51    void remove(const int key, Node*& root);
52  public:
53    void visit(Node* ptr);
54    void insert(const int value, Node*& root);
55    void Showresult(Node* root);
56    bool Search(const int key, Node* root);
57    void del(const int key, Node*& root);
58    void clear(Node*& root);
59  };
60  #endif
```

코드 1: AVL.h

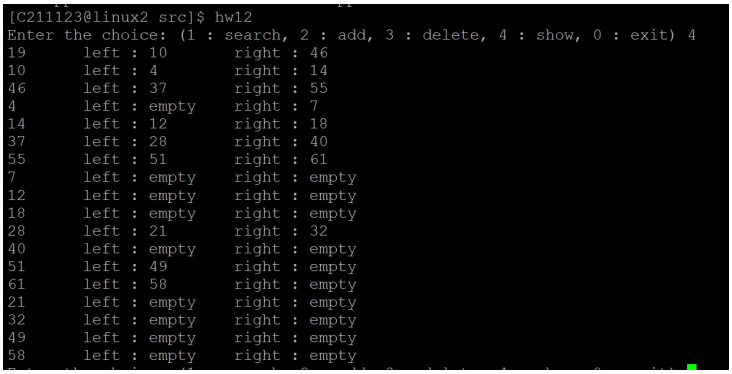
코드 1에 클래스 설계 내용을 담은 헤더 파일이 나와 있다. 명세서에서는 Node를 struct로 정의했지만 여기서는 클래스로 정의하는 것이 더 편하기에 클래스로 바꾸었다. 객체 지향 설계 원칙에 맞추어 내부 함수와 내부 변수는 최대한 정보를 감추고, 필요한 경우에만 외부 함수를 통해서 접근하도록 설계하였다. 먼저 Node 클래스의 경우, 데이터를 담는 int형 data, 그리고 balance factor를 의미하는 bf, 왼쪽 서브 트리, 오른쪽 서브 트리, 그리고 부모 서브 트리를 각각 의미하는 leftChild, rightChild, parent가 private 멤버로 존재한다. Node 클래스는 public 함수로 각각의 get, set 함수를 제공하여 이에 접근할 수 있도록 한다. 한편 특별히 leftChild, rightChild, parent에는 get, set 함수 이외에도 getref 함수가 존재한다. 이는 getref 함수를 통해서 private 변수에 직접 접근할 수 있으므로 객체 지향 설계에 위배될 수 있으나, bstree에서는 계속적으로 재귀 함수를 통해 구현을 이어가기에 편의성을 위해 만들었다. 따라서 getref 함수는 변수에 직접 접근할 수 있으므로 주의해서 사용하되 재귀적 구현에서 매우 유용한 도구가 된다[1]. parent는 노드 자체가 부모의 정보 역시 포함하도록 설계하였다. 이 경우 삽입, 삭제, 회전 등의 연산에서 매번 parent의 정보를 관리해주는 단점이 있으나, 그에 대응하는 여러 가지 장점들이 있다. 부모 정보를 포함하므로 InorderSuccessor() 함수 구현하기가 쉬워지며, 매번 parent 정보를 찾기 위해 root 노드를 가지고 다니며 새롭게 계산해줄 필요가 없어서 성능 개선의 여지가 있다.

bstree 노드의 경우 트리를 구성하는데 필요한 기능들을 담고 있다. 사용자에게 공개된 함수는 visit, insert, showresult, Search, del, clear[2] 함수 뿐이며, 나머지 함수들은 내부적 구현을 위해 private하게 숨겨져 있다.

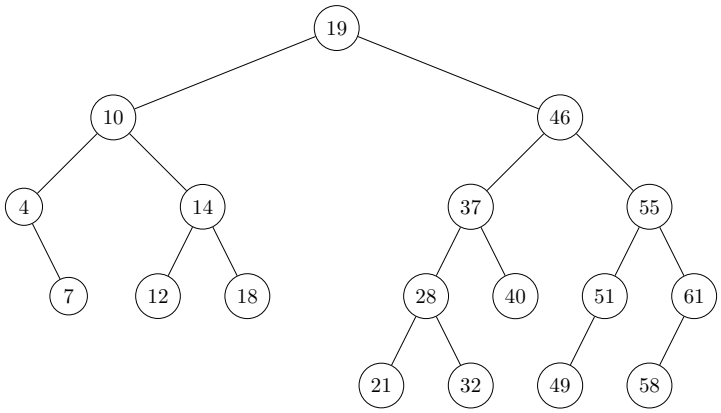
Node 내부 변수인 data, bf, 그리고 자식과 부모 노드의 경우 Node 클래스에서 자체적으로 이에 접근 및 수정하도록 함수를 구성하였고, bstree는 Node의 변수에는 직접적으로 접근하지는 않지만 Node를 이용해서 AVL tree를 구성하는데 필요한 모든 연산을 처리한다. 다른 방법으로는 Node를 bstree의 friend 클래스로 선언해서, bstree에서 Node의 모든 private 변수에 직접적으로 접근이 가능하게끔 구현하는 방법도 있다. Node가 현재로썬 확실하게 bstree에 귀속되는 클래스이기에 가능한

[1] 정확히 말하면 재귀적 구현에서 인자로 Node*& 형식의 참조를 매개 인자로 받을 때 사용되며, private한 변수에 직접 접근하여 수정할 수 있도록 하려면 getref 함수가 필수적으로 필요하다.

[2] 트리 내 모든 데이터를 지우는 함수, 과제 자체에는 필요 없는 내부적 테스트를 위한 함수이나 vector 클래스의 clear 함수처럼, 사용자에게 public하게 공개되기에 적절한 함수이며 테스트 과정 또한 과제의 일부라 판단되어 남겨두었다. 메뉴 선택 시 5번을 선택하면 clear 함수를 이용할 수 있다.



(a) Initial result



(b) initial result tree

그림 1: Initial Result

방법이다.

3 결과

그림 1는 명세서에서 주어진 초기 Insert input값에 대한 AVL tree이며, 1(b)는 1(a)의 출력 결과를 트리로 표현한 것이다. 1(b)에서 모든 서브트리의 BF가 2 미만으로 잘 균형이 맞아 떨어진 것을 확인할 수 있다.

그림 2에서 18을 검색하는 모습을 볼 수 있으며, 18까지의 경로인 19→10→14→18가 잘 출력된 것을 그림 2(a)에서 확인할 수 있다. 그림 2(b)은 그림 2(a)의 출력 결과를 tree에 시각화한 것이다.

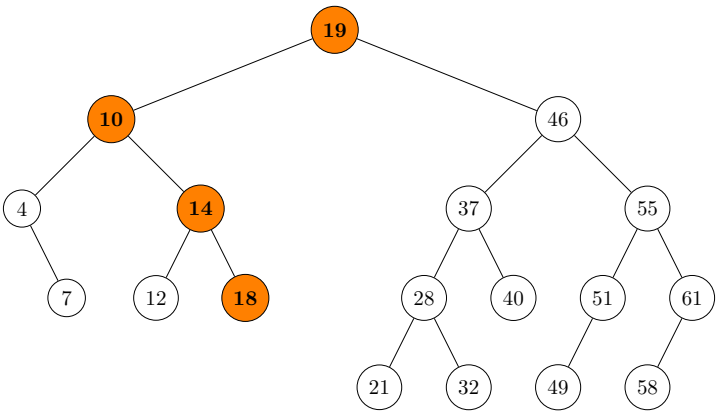
그림 3에서 9를 삽입했을 때의 결과를 볼 수 있다. 그림 3(b)에서 그림 3(a)의 출력 결과를 시각화한 것을 볼 수 있다. 일반 이진 검색 트리(Binary Search Tree, 이하 BST)에서는 9는 노드 7의 RightChild로 삽입됐을 것이지만, AVL tree에서는 4의 BF가 2가 되어 균형이 깨지므로, LR 회전하여 균형을 적절하게 이뤘음을 알 수 있다.

그림 4에서 19를 삭제했을 때 결과를 확인할 수 있다. 그림 4(b)에서 19의 Successor인 21이 19와 자리를 교환한 것을 알 수 있다.

그림 5에서 빈 트리에서의 삽입과 삭제 과정이 정상적으로 이루어지고 있음을 확인할 수 있다. 5번은 모든 트리를 비우는 Clear 기능이다. 이 이후에 순서대로 3, 5, 7의 삽입이 이루어지고, 5, 7, 3의 순서대로 삭제가 이루어진다. 3, 5, 7의 삽입 과정에서 RR 회전이 이루어졌음을 확인할 수 있다. 삭제 과정에서도 정상적으로 모든 노드를 삭제하였고, root 노드가 빈 상태에서도 정상적으로 프로그램이 작동함을 알 수 있다. root 노드가 빈 상태에서도 새롭게 insert하면 정상적으로 작동한다.

```
Enter the choice: (1 : search, 2 : add, 3 : delete, 4 : show, 0 : exit) 1
Enter node to search: 18
19 -> 10 -> 14 -> 18
```

(a) Result of searching 18

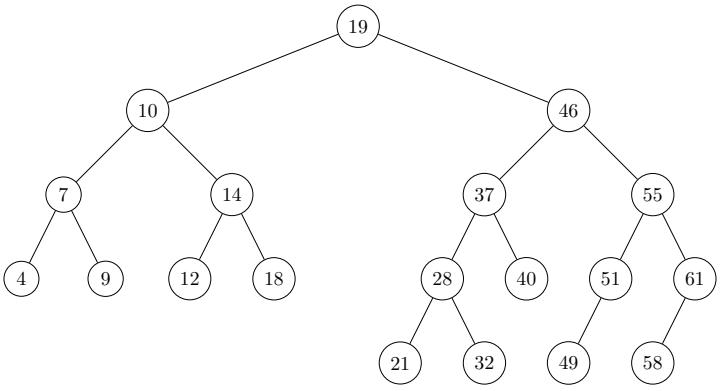


(b) Path to 18

그림 2: Search : 18

```
Enter the choice: (1 : search, 2 : add, 3 : delete, 4 : show, 0 : exit) 2
Enter a new value: 9
19      left : 10       right : 46
10      left : 7        right : 14
46      left : 37       right : 55
7       left : 4        right : 9
14      left : 12       right : 18
37      left : 28       right : 40
55      left : 51       right : 61
4       left : empty    right : empty
9       left : empty    right : empty
12      left : empty    right : empty
18      left : empty    right : empty
28      left : 21       right : 32
40      left : empty    right : empty
51      left : 49       right : empty
61      left : 58       right : empty
21      left : empty    right : empty
32      left : empty    right : empty
49      left : empty    right : empty
58      left : empty    right : empty
```

(a) Result of insert 9

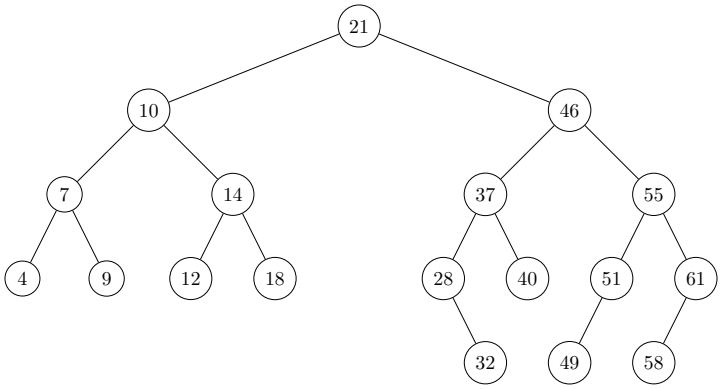


(b) Result of insert 9 tree

그림 3: After insert 9

```
Enter the choice: (1 : search, 2 : add, 3 : delete, 4 : show, 0 : exit) 3
Enter node to delete: 19
21      left : 10       right : 46
10      left : 7        right : 14
46      left : 37       right : 55
7       left : 4        right : 9
14      left : 12       right : 18
37      left : 28       right : 40
55      left : 51       right : 61
4       left : empty    right : empty
9       left : empty    right : empty
12      left : empty    right : empty
18      left : empty    right : empty
28      left : empty    right : 32
40      left : empty    right : empty
51      left : 49       right : empty
61      left : 58       right : empty
32      left : empty    right : empty
49      left : empty    right : empty
58      left : empty    right : empty
```

(a) Result of remove 19

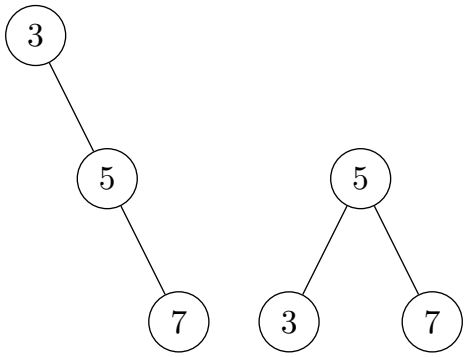


(b) Result of remove 19 tree

그림 4: After remove 19

```
Enter the choice: (1 : search, 2 : add, 3 : delete, 4 : show, 0 : exit) 5
Clear the tree
Enter the choice: (1 : search, 2 : add, 3 : delete, 4 : show, 0 : exit) 2
Enter a new value: 3
3       left : empty    right : empty
Enter the choice: (1 : search, 2 : add, 3 : delete, 4 : show, 0 : exit) 2
Enter a new value: 5
3       left : empty    right : 5
5       left : empty    right : empty
Enter the choice: (1 : search, 2 : add, 3 : delete, 4 : show, 0 : exit) 2
Enter a new value: 7
5       left : 3        right : 7
3       left : empty    right : empty
7       left : empty    right : empty
Enter the choice: (1 : search, 2 : add, 3 : delete, 4 : show, 0 : exit) 3
Enter node to delete: 5
7       left : 3        right : empty
3       left : empty    right : empty
Enter the choice: (1 : search, 2 : add, 3 : delete, 4 : show, 0 : exit) 3
Enter node to delete: 7
3       left : empty    right : empty
Enter the choice: (1 : search, 2 : add, 3 : delete, 4 : show, 0 : exit) 3
Enter node to delete: 3
```

그림 5: 빈 트리에서 3, 5, 7의 삽입과 삭제



(a) RR 회전 전 (b) RR 회전 후

그림 6: RR 회전

4 AVL tree에서의 회전

AVL tree의 핵심은 회전이다. AVL tree에서는 삽입된 노드와 밸런스가 깨진 노드 사이의 위치 관계를 기준으로 다음 4가지 상황을 정의하고 있다.

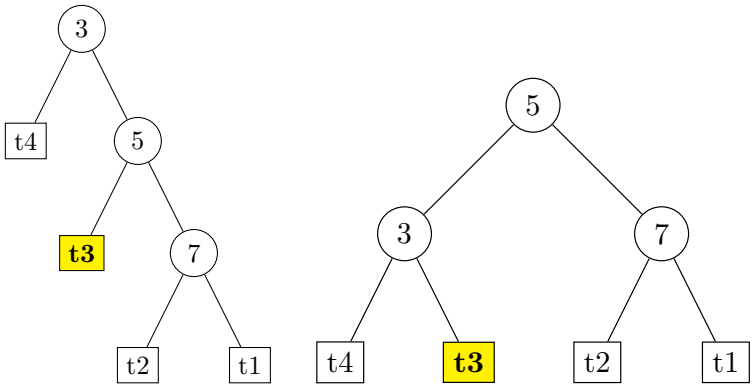
- LL : 임의의 노드 A를 기준으로 왼쪽 Subtree의 왼쪽 서브트리에 의해 A의 균형 인수 BF가 +2 이상으로 균형이 깨진 경우
- RR : 임의의 노드 A를 기준으로 오른쪽 Subtree의 오른쪽 서브트리에 의해 A의 균형 인수 BF가 -2 이하로 균형이 깨진 경우
- LR : 임의의 노드 A를 기준으로 왼쪽 Subtree의 오른쪽 서브트리에 의해 A의 균형 인수 BF가 +2 이상으로 균형이 깨진 경우
- RL : 임의의 노드 A를 기준으로 오른쪽 Subtree의 왼쪽 서브트리에 의해 A의 균형 인수 BF가 -2 이하로 균형이 깨진 경우

4.1 RR 회전

구현 시에도 알 수 있겠지만, LL과 RR, LR과 RL은 각각 서로 대칭이다. 먼저 그림 5의 예시를 보며 RR 회전을 알아보자. RR 회전은 왼쪽 방향(반시계방향) 회전이라고도 불리며, 균형 인수가 -2인 A 노드의 자식 노드를 기준으로 왼쪽 방향 회전하며 그 자식 노드가 서브 트리의 루트 노드가 되었기 때문이다. 그림 6을 보자. 그림 6(a)에서 루트 노드인 3은 BF가 -2이고 자식 노드인 5의 BF는 -1이다. 한편 5의 오른쪽 자식 노드인 7에 의해 균형이 깨졌으므로, RR 상황이다. 이 경우 왼쪽 방향으로 회전하며 5가 루트로, 3이 5의 왼쪽 자식으로 편입되면 그림 6(b)이 된다. RR 회전의 결과이다.

한편 RR 회전을 일반화할 경우, 그림 7으로 표현할 수 있다. 즉 각 서브 트리가 t1, t2, t3, t4의 서브트리를 갖고 있는 경우이다. 다른 서브트리는 앞서 보인 RR 회전을 적용할 경우 신경 쓸 필요는 없지만, 그림 7(a)에서 주목해야 할 서브트리는 t3이다. t3는 5의 왼쪽 서브트리인데, 5의 왼쪽 서브 트리는 RR 회전 후 3이 차지해야 하기 때문이다. 이 경우 그림 7(b)에서 t3는 5의 부모 노드인 3의 오른쪽 자식으로 편입됨을 알 수 있다. 이렇게 이동해도 t3는 여전히 7보다는 비교 순위가 작고 5보다는 크니, 이진 탐색 트리의 기준에는 부합한다는 것을 관찰할 수 있다.

```
1 Node* bstree::rotateRR(Node* root) {
2   Node* parentNode = root;
3   Node* childNode = root->GetRightChild();
4
5   changeRightSubtree(parentNode, childNode->GetLeftChild());
6   changeLeftSubtree(childNode, parentNode);
7
8   return childNode;
```



(a) RR 회전 전 (b) RR 회전 후

그림 7: RR 회전의 일반화

```
}
```

코드 2: RR rotation function

위 RR 회전의 과정을 코드 상으로 구현하면 코드 2으로 표현할 수 있다[3]. 즉 parent는 루트 노드, child는 루트 노드의 오른쪽 서브트리라 할 때 parent의 왼쪽 서브트리를 child의 왼쪽 서브트리(그림 7에서 t3 서브트리)로 만들어준 다음, child의 왼쪽 서브트리로 parent를 설정해준다.

4.2 LL 회전

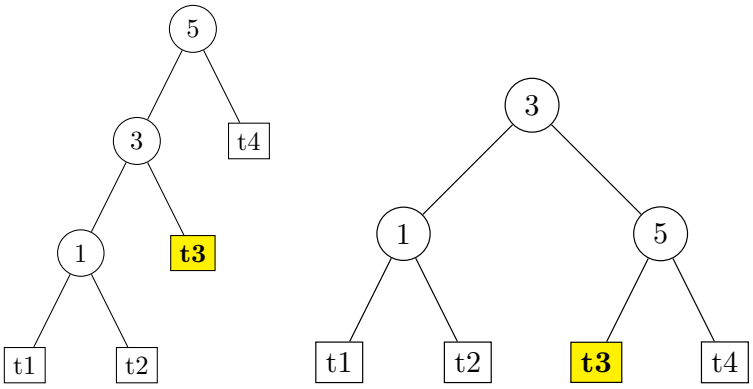
LL 회전은 그 반대이다. 그림 8에서 확인할 수 있다. LL 회전은 오른쪽 방향(시계 방향) 회전이라고도 불린다. 그림 8(a)에서 8(b)로 회전할 때 역시나 t3의 위치 관계만 바뀌는 것에만 주의한다면 큰 문제는 없다. 코드 4에서 LL 회전을 구현한 코드를 확인할 수 있는데, RR 회전과 대칭을 이루고 있음을 확인할 수 있다.

```
Node* bstree::rotateLL(Node* root) {
  Node* parentNode = root;
  Node* childNode = root->GetLeftChild();

  changeLeftSubtree(parentNode, childNode->GetRightChild());
  changeRightSubtree(childNode, parentNode);

  return childNode;
```

[3] 이후에도 계속해서 나올 changeRightSubtree(Node A, Node B)와 changeLeftSubtree(Node A, Node B)는 각각 A의 왼쪽 혹은 오른쪽 서브 트리를 B로 바꾸고, B의 부모 노드를 A로 갱신하는 함수이다. 함수의 코드는 각각 코드 3에 나와 있다.

```
inline void bstree::changeLeftSubtree(Node* main, Node* sub) {
  main->SetLeftChild(sub);
  if (main->GetLeftChild()) {
    main->GetLeftChild()->SetParent(main);
  }
}
inline void bstree::changeRightSubtree(Node* main, Node* sub) {
  main->SetRightChild(sub);
  if (main->GetRightChild()) {
    main->GetRightChild()->SetParent(main);
  }
}
```

코드 3: changeLeft/RightSubtree



(a) LL 회전 전 (b) LL 회전 후

그림 8: LL 회전의 일반화

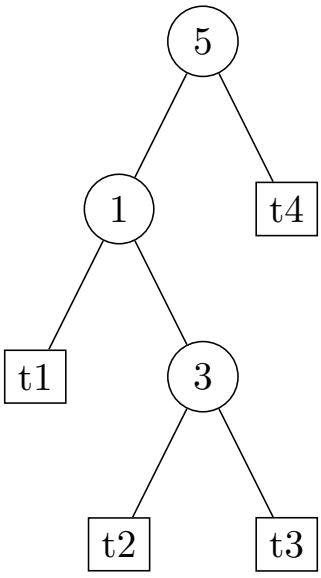


그림 9: LR 상태

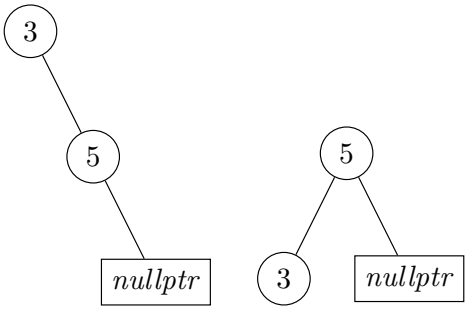
```
}
```

코드 4: LL rotation function

4.3 LR 회전

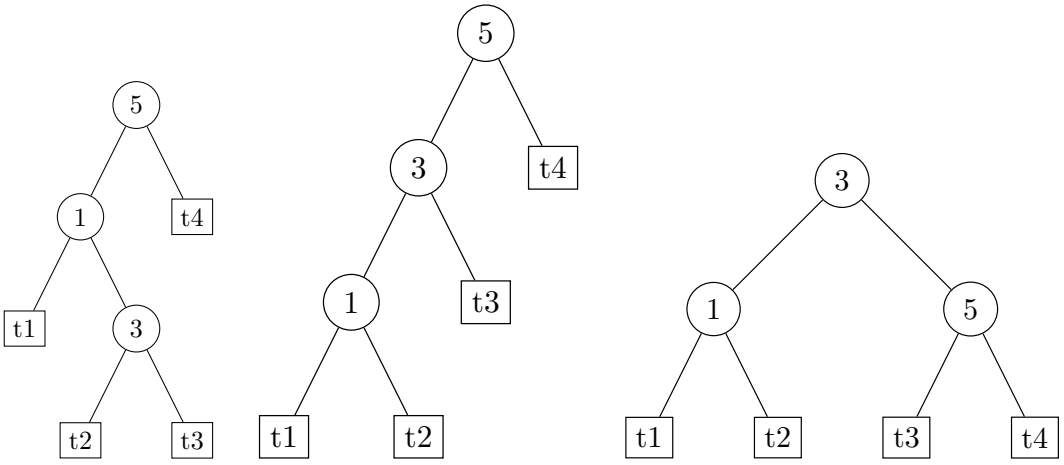
LR과 RL 회전은 LL 회전과 RR 회전의 조합으로 만들어진다. LR, RL 상태는 각각 LL 회전과 RR 회전으로 균형이 잡히는 LL 상태나 RR 상태로 다시 만들 수 있기 때문이다. 따라서 LR, RL 회전은 두 번의 회전이 필요하며, LR과 RL의 회전 순서는 반대이다. 먼저 LR부터 살펴보자. 그림 9을 간단하게 하려면, 결론적으로 RR 회전을 통해 LL 상태로 만들어야 한다. 이는 RR 회전의 부수적인 효과를 이용한 것으로 해석할 수 있는데, 이를 위해 먼저 RR 회전을 다시 한 번 관찰해보자. 그림 6의 과정에서 7이 저장된 노드를 nullptr로 치환해보자. 이 과정을 다시 보면 그림 10으로 표현할 수 있다. 곧 부모 노드와 자식 노드의 관계를 역전시키는 것을 RR 회전의 부수적인 효과로 볼 수 있으며, 노드가 2개만 존재하는 상황에서도 RR 회전은 가능하다. 즉 그림 9을 다시 보자면 그림 11(a)에서 11(b)로 RR 회전을 통해 LL 상태로 만든다. 이후 LL 회전을 통해 그림 11(c)로 만들어주면 끝난다.

```
Node* bstree::rotateLR(Node* root) {
```



(a) RR 회전 전 (b) RR 회전 후

그림 10: nullptr을 포함하는 RR 회전



(a) RR 회전 전 (b) RR 회전 후 (c) LL 회전 후

그림 11: RR 회전 후 LL 회전을 통한 LR 회전

```
  Node* parentNode = root;
  Node* childNode = root->GetLeftChild();

  changeLeftSubtree(parentNode, rotateRR(childNode)); // 부분적 RR 회전

  return rotateLL(parentNode); // LL 회전
}
```

코드 5: LR rotation function

코드 5에서 LR 회전 함수를 확인할 수 있다. parentNode는 root 노드로, root노드의 왼쪽 서브트리를 childNode라 할 때, 먼저 rotateRR(childNode)를 통해 childNode 서브트리를 RR 회전한다. 이 결과를 parentNode의 왼쪽 서브트리에 저장한다(그림 11(a)에서 11(b)로). 이후 parentNode를 기준으로 LL 회전한 결과를 반환한다(그림 11(b)에서 11(c)로).

4.4 RL 회전

```
Node* bstree::rotateRL(Node* root) {
  Node* parentNode = root;
  Node* childNode = root->GetRightChild();

  changeRightSubtree(parentNode, rotateLL(childNode)); // 부분적 LL 회전

  return rotateRR(parentNode); // RR 회전
}
```

코드 6: RL rotation function

RL 회전은 LR 회전과 대칭이다. 즉 코드 6에서 parentNode의 오른쪽 서브 트리를 childNode라 할 때 이 childNode를 기준으로 LL 회전을 한 뒤 parentNode의 오른쪽 서브트리에 다시 이어 붙인 후, parentNode를 기준으로 RR 회전한다.

정리하자면 다음과 같다.

- RR 회전 : 왼쪽 방향(반시계 방향) 회전
- LL 회전 : 오른쪽 방향(시계 방향) 회전
- LR 회전 : RR 회전 후 LL 회전
- RL 회전 : LL 회전 후 RR 회전

5 삽입

```
void bstree::insert(const int value, Node*& root)
{
  if (root == NULL) {
    root = new Node(value);
    root->SetParent(root->GetParent());
    root->SetBF(GetBalanceFactor(root));
  }
  else if (value < root->GetData()) {
    insert(value, root->GetLeftChildRef());
    Rebalance(root);

  }
  else if (value > root->GetData()) {
```

```
14      insert(value, root->GetRightChildRef());
15      Rebalance(root);
16    }
17  }
```

코드 7: insert

```
1  void bstree::Rebalance(Node*& root) {
2    if (root == NULL) return;
3    ResetBF(root);
4    int balanceFactor = root->GetBF();
5
6    // BF가 2 이상이면 LL, LR 회전
7    if (balanceFactor > 1) {
8      if (root->GetLeftChild()->GetBF() > 0) { // LL
9        root = rotateLL(root);
10     }
11     else { // LR
12       root = rotateLR(root);
13     }
14   }
15   else if (balanceFactor < -1) { // 오른쪽 서브트리가 더 높을 때
16     if (root->GetRightChild()->GetBF() < 0) { // RR
17       root = rotateRR(root);
18     }
19     else { // RL
20       root = rotateRL(root);
21     }
22   }
23   ResetParent(root);
24   ResetBF(root);
25 }
```

코드 8: Rebalance

AVL tree에서 삽입 과정은 BST tree에서와 동일하게 삽입한 후, 재귀적으로 하나씩 부모로 거슬러 올라가면서 BF를 체크해 리밸런싱을 진행한다(코드 7, 8). 최종적으로 Root 노드에 이르기까지 리밸런싱을 검사하므로 삽입 후 결과는 AVL tree임이 보장된다.

6 삭제

```
1  Node* bstree::InorderSucc(Node* current)
2  {
3    if (current == NULL) return NULL;
4
5    // case 1: has right subtree -> then find the min value from right
       subtree
6    if (current->GetRightChild() != NULL) {
7      return findMin(current->GetRightChild());
8    }
9
10   // case 2: no right subtree -> then find the first ancestor that is
       greater than current node
11   else {
```

```
12      // current has no right subtree, and no parent -> then current is
        the root node
13      if (current->GetParent() == NULL) return NULL;
14
15      Node* ancestor = current->GetParent();
16      Node* successor = current;
17      while (ancestor != NULL && successor == ancestor->GetRightChild())
        {
18        successor = ancestor;
19        ancestor = ancestor->GetParent();
20      }
21      return ancestor;
22    }
23  }
```

코드 9: InorderSuccessor

```
1  void bstree::del(const int key, Node*& root) {
2    remove(key, root);
3
4    if (root == NULL) return;
5    // recursive하게 successor를 삭제하느라, successor의 parent를 다시
       연결해주어야 한다.
6    root->SetParent(NULL);
7    ResetParent(root);
8    ResetBF(root);
9
10   // 마지막으로 모든 노드 순회하면서 밸런스 체크
11   recursive_balance_checker(root);
12 }
13
14 void bstree::remove(const int key, Node*& root)
15 {
16   if (root == NULL) return;
17
18   // 과정 1: 먼저 key에 해당하는 Node 찾기
19   Node* target = findNode(key, root);
20   // if target is null
21   if (target == NULL) {
22     cout << "I cannot delete " << key << " as " << key << " is not
       exist in tree" << endl;
23     return;
24   }
25
26   Node* parent = target->GetParent();
27
28   // If target is root node, then make virtual_parent.
29   Node* virtual_parent = NULL;
30   if (parent == NULL) {
31     virtual_parent = new Node(root->GetData() - 1, 0, NULL, root, NULL)
       ;
32     root->SetParent(virtual_parent);
33     parent = virtual_parent;
34   }
```

```
35
36    // 과정 2: 지우기
37    // case 1 : target is leaf node
38    if (target->GetRightChild() == NULL && target->GetLeftChild() == NULL
       ) {
39      bool isTargetEqualsRoot = (target == root);
40      // if target is leftnode of parent node
41      if (parent->GetLeftChild() == target) {
42        parent->SetLeftChild(NULL);
43        delete target;
44      }
45      else {
46        parent->SetRightChild(NULL);
47        delete target;
48      }
49
50      if (isTargetEqualsRoot) {
51        root = NULL;
52      }
53      else {
54        ResetBF(root);
55        ResetParent(root);
56        Node* cur = parent;
57        while (cur != NULL && cur != root) {
58          if (cur->GetBF() > 1 || cur->GetBF() < -1) {
59            Rebalance(cur);
60            ResetBF(root);
61            ResetParent(root);
62          }
63          cur = cur->GetParent();
64        }
65      }
66    }
67    // case 2 : target has one child
68    else if (target->GetLeftChild() == NULL || target->GetRightChild() ==
        NULL) {
69      Node* target_child = NULL; // child of target
70
71      if (target->GetLeftChild() != NULL)
72        target_child = target->GetLeftChild();
73      else
74        target_child = target->GetRightChild();
75
76      // 이 경우 target_child가 의사 successor가 된다.
77      // successor를 target->parent에게 연결시켜주면 된다.
78      if (parent->GetLeftChild() == target) {
79        parent->SetLeftChild(target_child);
80        target_child->SetParent(parent);
81        delete target;
82      }
83      else {
84        parent->SetRightChild(target_child);
85        target_child->SetParent(parent);
```

```
86        delete target;
87      }
88
89      // 의사 successor인 target_child로부터 root까지 balance 검사
90      ResetBF(root);
91      ResetParent(root);
92      Node* cur = target_child;
93      while (cur != NULL && cur != root) {
94        if (cur->GetBF() > 1 || cur->GetBF() < -1) {
95          Rebalance(cur);
96          ResetBF(root);
97          ResetParent(root);
98        }
99        cur = cur->GetParent();
100     }
101   }
102   // case 3 : target has two child
103   else {
104     Node* successor = InorderSucc(target);
105     target->SetData(successor->GetData());
106
107     Node* cur_parent = successor->GetParent(); // parent of successor
108
109     // 이제 successor를 삭제한다.
110     remove(successor->GetData(), successor);
111
112     // 삭제되었던 successor에서 root까지 올라가면서 밸런스를 맞춘다.
113     ResetBF(root);
114     ResetParent(root);
115     while (cur_parent != NULL && cur_parent != root) {
116       if (cur_parent->GetBF() > 1 || cur_parent->GetBF() < -1) {
117         Rebalance(cur_parent);
118         ResetBF(root);
119         ResetParent(root);
120       }
121       cur_parent = cur_parent->GetParent();
122     }
123   }
124
125   // 삭제된 노드가 루트 노드인 경우에 대한 처리
126   if (virtual_parent != NULL) {
127     root = virtual_parent->GetRightChild();
128     if (root != NULL) root->SetParent(NULL);
129     delete virtual_parent;
130     return;
131   }
132
133   if (root) {
134     Rebalance(root);
135   }
136 }
```

코드 10: remove

AVL tree에서 삭제 과정은 BST tree의 삭제와 유사하나, Inorder successor를 통해 좀 더 최적화할 수

있다. 코드 9는 Inorder Successor를 찾는 함수이다. 삭제 과정은 트리 거 함수인 del 함수와 재귀적으로 구현되는 remove 함수로 이루어진다(코드 10). 삭제 대상이 단말(리프) 노드인지, 하나의 자식을 갖는지 두 개의 자식을 갖는 지에 대해 각각 상황을 나눠 구현하였다. 단말 노드의 경우 삭제하고, 하나의 자식 노드를 갖는 경우 그 자식 노드가 의사 successor(sudo successor)가 되어 삭제 대상의 부모와 연결된다. 두 개의 자식 노드를 갖는 경우 Inorder successor를 찾아 successor가 그 자리에 대신 온다[4]. 이후 successor의 부모로부터 루트까지 거슬러 올라가며 BF를 확인하고 리밸런싱을 단행한다. 재귀적 구현의 최종으로 루트까지 거슬러 올라가며 리밸런싱을 하기에, 최종 결과는 AVL tree 임이 보장된다. 재귀적으로 구현한 이유는 successor의 삭제를 다시 한 번 remove 함수를 통해 실행하기 때문이다. del 함수에서는 재귀적 구현이 끝난 후, 다시 한 번 bf와 parent를 reset 함수를 통해 갱신한다. 그리고 다시 한 번 bf를 확인해서 전체에 대한 리밸런싱을 진행한다.

그 외에 여러 가지 조건문들이 존재하는데, 이는 대부분 1개, 2개일 때 노드의 삭제 과정에서, 트리가 비거나 노드가 하나만 남았을 때에도 정상적으로 작동하게끔 가정을 해놓은 것이다.

7 어려웠던 부분들

회전의 경우 다양한 강의 자료와, 이전에 학습한 경험이 있어 구현에 어려움은 없었으나, 삭제 기능 구현이 매우 어려웠다. 삭제 대상이 leaf 노드인 경우를 제외하곤 InorderSucc 함수를 통해 successor를 찾아서 이를 대체하는 식으로 구현하려 했는데, 다소 생각해야 할 것들이 많았고, 특히 삭제 대상이 하나의 자식 서브 트리를 가질 때 successor를 이용하면 안 된다는 것을 여러 번의 상황 시뮬레이션 끝에 깨닫고서 재구현했다. 결국 하나의 자식 서브 트리를 가지는 경우 진짜 Successor 대신 의사 Successor로 자식 노드를 설정하는 방식을 택했고, 두 개의 서브트리를 갖는 경우만 Inorder successor를 사용했다. 한편 구현 과정에서 parent 노드를 Node가 저장하게끔 구현을 변경하는 등 여러 번의 설계 변화가 있었다. 그리고 모든 노드가 제거되었을 때, 또는 단 두 개의 노드만 남았을 때 삭제 과정이 매우 어려웠다. 특히 가상의 부모 노드인 virtual_parent 노드를 도입하는 아이디어는 창의적이었으나 이 방법을 본인의 코드에 적용하는데 있어서 parent의 관계가 무너지게 된다는 단점이 있었다. 결국 모든 parent, 그리고 bf의 관계가 깨지는 상황마다 reset 함수를 통해 parent와 bf를 갱신하였고, virtual_parent 노드 역시 여러 번의 시도 끝에 메모리 오류가 발생하지 않게끔 조심히 구현되었다. 여러 번의 시행 착오 끝에 노드가 한 개, 두 개, 세 개 있을 때 등의 상황에서도 노드의 삭제가 자유롭게 구현되었다.

[4] 이 경우 유효한 BST임을 보장한다.