Data Structure AVL Tree Implementation

Header

|  |
| --- |
| \_NODE 트리에 들어가는 노드. 각 노드별로   1. 어떠한 자료도 담을 수 있는 void pointer를 이용한 데이터 포인터 2. 왼쪽 자식 노드 3. 왼쪽 트리와 오른쪽 트리 간의 높이를 판단할 balance 값. 4. 오른쪽 자식 노드 |
| \_AVL\_TREE 노드가 있는 트리.   1. 노드 갯수 2. 각 노드의 데이터를 비교할 수 있는 포인터함수 3. 루트노드 |

Create AVL TREE

|  |
| --- |
| AVL TREE를 이용하기 위해 root 노드를 초기화하여 트리 형태로 리턴함.   1. 동적할당을 통해 TREE 메모리 할당. 2. 할당된 TREE에 각 데이터 초기화. 3. 매개변수 포인터함수를 담은 tree 리턴 |

Insert

|  |
| --- |
| 트리에 노드를 삽입해주는 함수. 삽입할 데이터 포인터와 삽입 대상 트리 포인터를 매개변수로 받는다.   1. 삽입할 노드의 초기화값은 자식노드 모두 NULL, 밸런스 값 EH, 그리고 보이드 포인터. 2. 이후 \_insert 함수를 불러 트리에 저장. 3. Insert 함수에서는 1. 베이스케이스   2. Insert할 노드 데이터 값 < 노드 데이터 값  3. Insert할 노드 데이터 값 >= 노드 데이터 값으로 나뉨.   * 2번에 해당할 경우 왼쪽 트리에 Insert함수 재귀적 이용.   삽입 후 \*taller 값이 참일 때 ( 높이가 커져 밸런스 조정이 필요함을 의미)  Root->bal  1. LH 일경우 -> 왼쪽이 높았는데 왼쪽에 삽입했으므로 오른쪽으로 회전.  2. EH일 경우 -> 왼쪽이 더 높음으로 바꿈.  3. RH일 경우 -> 오른쪽이 높았다가 왼쪽에 추가했으므로 EH로 변경.   * 3번에 해당할 경우 왼쪽 트리에 해당한 함수를 대칭하여 적용. |

InsertBalance

|  |
| --- |
| 왼쪽트리의 높이가 높은 상황에서 왼쪽 트리에 노드를 삽입할 경우 밸런스 조정을 해주는 함수.  밸런스 조정은 Simple Balance와 Complex Balance 두가지가 존재한다. Simple Balance는 한번만 회전하면 되는 경우이고 Complex Balance는 두번에 걸친 회전이 필요.  왼쪽트리의 밸런스:   1. LH인 경우 : 왼쪽-왼쪽 높은 상태이기 때문에 오른쪽으로 회전.   그 이후 root-balance와, leftTree-balance 모두 EH로 변경.   1. EH인 경우 : 왼쪽 트리에서 불균형이 생겨 밸런스 함수를 불러왔는데 왼쪽 트리가 EH인 경우는 존재하지 않음. 따라서 오류 프린트. 2. RH인 경우 : 다시 왼쪽트리 우측 트리의 루트 밸런스 값에 따라 경우를 나눈다.   왼쪽트리 -> 오른쪽 루트 밸런스 값  LH인 경우 : 회전 후 root 노드에 담겨지는 노드 높이가 낮으므로 root 노드의 밸런스는 RH, 왼쪽 트리의 루트는 회전 후 전체 트리의 노드가 되며 양 트리의 높이가 동일해지므로 EH.  EH인 경우 : 삽입된 노드는 왼쪽->오른쪽 노드가 아니라 왼쪽->왼쪽 노드. 삽입후 EH가 될 경우 밸런스가 필요하지 않기 때문. 그러므로 회전 후 leftTree->bal 은 왼쪽 트리가 더 높으며 root->bal는 EH.  RH인 경우 : 회전 후 root 노드에 담겨지는 노드가 가장 높이가 크기 때문에 root->bal은 EH, leftTree->bal은 LH가된다.  이후 root 노드가 되는 rightTree->bal 은 EH가된다.  오른쪽 트리 밸런스는 대칭으로 적용. |

Rotate

|  |
| --- |
| 회전하는 함수. 왼쪽으로 회전할 경우 오른쪽 트리의 루트가 새로운 루트. 새로운 루트의 오른쪽 자식 노드는 기존 트리 루트의 오른쪽 노드의 왼쪽 노드. 그리고 새로운 노드의 왼쪽노드는 기존 노드가 된다.  오른쪽 회전 은 대칭하여 적용. |

Delete

|  |
| --- |
| 트리에서 원하는 값을 제거한다.   1. 해당하는 값을 찾았을 경우. 2. 제거하고자하는 값이 루트노드보다 작은경우 왼쪽 트리를 대상으로 재귀 함수. 3. 제거하고자하는 값이 루트노드보다 큰 경우 오른쪽 트리를 대상으로 재귀 함수.   제거 이후 밸런스 조정 함수를 부른다. 그리고 왼쪽 트리에서 가장 작은 값을 루트 노드로 삼고 왼쪽 트리에서 그 노드를 제거한다.  밸런스 조정시 인서트 함수에서 사용한 함수를 사용할 수는 없다. 왜냐하면 인서트 함수에서는 삽입한 후이기 때문에 높아진 상태를 전제로 트리를 조정하지만 딜리트 함수에서는 제거한 후이기 때문에 낮아진 상태를 전제로 트리를 조정하기 때문이다. 따라서 각 밸런스 케이스에 따른 경우가 달라지기 때문에 다시 함수를 경우에 따라 회전시켜 적용한다. |

Search

|  |
| --- |
| 트리에서 원하는 데이터 값을 찾아 존재하면 그 값을 출력하는 함수.  각 노드에 해당하는 데이터 값과 원하는 값을 비교하여 노드 값이 작으면 오른쪽 트리를 대상으로, 노드 값이 크면 왼쪽 트리를 대상으로 재귀적 함수 호출. 해당하는 데이터를 찾으면 데이터를 리턴하며 없을 경우 NULL값을 리턴한다. |

Traverse

|  |
| --- |
| 트리에서 모든 노드에 특정 작업을 거친다.  재귀함수의 특징을 이용.   1. 왼쪽 트리 2. 해당 노드 3. 오른쪽 트리   의 과정을 거쳐 모든 노드에 특정 작업을 적용한다.   * 모든 노드의 데이터 값을 출력하는 print 함수를 traverse 함수를 이용하여 구현했다. |

Destroy

|  |
| --- |
| 모든 노드에 대한 free함수 적용. 트리에 대한 free함수 적용 |

Count

|  |
| --- |
| Tree에 저장된 노드 갯수를 리턴. |

Empty

|  |
| --- |
| 트리에 저장된 count 값이 0일 경우 true, 아닐 경우 false 리턴 |

Full

|  |
| --- |
| 동적 메모리 할당에 실패할 경우 true, 아닐 경우 false 리턴. |