자료구조\_과제6\_01

AVL Tree

본 과제는 C++을 기반으로 작성하였습니다.



자료구조(2058)

김성열교수님

소프트웨어학과

201811182

박원준

2019.05.21

1. 소스코드

#include <iostream>

#include <queue>

using namespace std;

struct Node

{

        int data, bf; //bf=balance factor

        Node \*leftChild, \*rightChild;

        Node(int element, Node \*left = 0, Node \*right = 0) :data(element), bf(0), leftChild(left), rightChild(right)

        {

        }

};

class AVL

{

private:

        Node \*root;

        void Insert(Node \*&ptr, int element)

        {

               if (ptr == 0)

                       ptr = new Node(element);

               else if (element < ptr->data)

               {

                       Insert(ptr->leftChild, element);

                       //균형 맞추는 코드

                       setBF(root);

                       if (getBF(ptr) > 1 || getBF(ptr) < -1)

                       {

                              switch (getBF(ptr))

                              {

                              case -2: //오른쪽 서브트리 높이가 왼쪽 서브트리 높이보다 2 높을 때

                                      if (getBF(ptr->rightChild) == -1) //오른쪽 자식의 bf가 -1일 경우 rr

                                              ptr = rrRotation(ptr);

                                      else //오른쪽 자식의 bf가 1일 경우 rl

                                              ptr = rlRotation(ptr);

                                      break;

                              case 2: //왼쪽 서브트리 높이가 오른쪽 서브트리보다 2 높을 때

                                      if (getBF(ptr->leftChild) == 1) //왼쪽 자식의 bf가 1일 경우 ll

                                              ptr = llRotation(ptr);

                                      else //왼쪽 자식의 bf가 -1일 경우 lr

                                              ptr = lrRotation(ptr);

                                      break;

                              }

                       }

               }

               else if (element > ptr->data)

               {

                       Insert(ptr->rightChild, element);

                       setBF(root);

                       if (getBF(ptr) > 1 || getBF(ptr) < -1)

                       {

                              switch (getBF(ptr))

                              {

                              case -2:

                                      if (getBF(ptr->rightChild) == -1)

                                              ptr = rrRotation(ptr);

                                      else

                                              ptr = rlRotation(ptr);

                                      break;

                              case 2:

                                      if (getBF(ptr->leftChild) == 1)

                                              ptr = llRotation(ptr);

                                      else

                                              ptr = lrRotation(ptr);

                                      break;

                              }

                       }

               }

               else

                       ptr->data = element;

        }

        void Delete(Node \*&ptr, int element)

        {

               Node \*tmpptr;

               Node \*tmpdaddyptr;

               if (ptr == 0)

               {

                       cout << "해당 노드를 찾을 수 없습니다" << endl;

                       return; //그런 노드 없으므로 그냥 return

               }

               if (element < ptr->data)

               {

                       Delete(ptr->leftChild, element);

                       setBF(root);

                       if (getBF(ptr) > 1 || getBF(ptr) < -1)

                       {

                              switch (getBF(ptr))

                              {

                              case -2:

                                      if (getBF(ptr->rightChild) == -1)

                                              ptr = rrRotation(ptr);

                                      else

                                              ptr = rlRotation(ptr);

                                      break;

                              case 2:

                                      if (getBF(ptr->leftChild) == 1)

                                              ptr = llRotation(ptr);

                                      else

                                              ptr = lrRotation(ptr);

                                      break;

                              }

                       }

               }

               else if (element > ptr->data)

               {

                       Delete(ptr->rightChild, element);

                       setBF(root);

                       if (getBF(ptr) > 1 || getBF(ptr) < -1)

                       {

                              switch (getBF(ptr))

                              {

                              case -2:

                                      if (getBF(ptr->rightChild) == -1)

                                              ptr = rrRotation(ptr);

                                      else

                                              ptr = rlRotation(ptr);

                                      break;

                              case 2:

                                      if (getBF(ptr->leftChild) == 1)

                                              ptr = llRotation(ptr);

                                      else

                                              ptr = lrRotation(ptr);

                                      break;

                              }

                       }

               }

               else //ptr 노드가 바로 지울 노드인 경우

               {

                       if (!ptr->leftChild && !ptr->rightChild) //자식이 없다면

                       {

                              delete ptr;

                              ptr = 0;

                              return;

                       }

                       else if (ptr->leftChild && !ptr->rightChild) //왼쪽 자식만 있다면

                       {

                              //그 자식을 ptr이 가리키게하고 현재 ptr이 가리키는 노드 지움

                              tmpptr = ptr;

                              ptr = ptr->leftChild;

                              delete tmpptr;

                              return;

                       }

                       else if (!ptr->leftChild&&ptr->rightChild) //오른쪽 자식만 있다면

                       {

                              tmpptr = ptr;

                              ptr = ptr->rightChild;

                              delete tmpptr;

                              return;

                       }

                       else

                       {

                              //두 자식 모두 있음:루트가 rc인 우측트리에서 제일작은 노드 찾자

                              Node \*rc = ptr->rightChild; //rc가 루트인 subtree

                              if (!rc->leftChild) //rc가 왼쪽자식이 없으면 rc가 그 노드

                              {

                                      ptr->data = rc->data;

                                      ptr->rightChild = rc->rightChild;

                                      delete rc;

                                      return;

                              }

                              //rc의 왼쪽 자식이 있는 경우, rc의 왼쪽 자식의 왼쪽 자식의 식으로 왼쪽 자식을 끝까지 쫓아가, 가장 작은 키 갖는 노드를 찾는다

                              //그 노드의 key/element를 ptr노드로 옮기고 그 노드의 rightChild는 그 노드의 부모의 leftChild에 저장한 다음 그 노드를 지움

                              else

                              {

                                      while (1)

                                      {

                                              if (rc->leftChild)

                                              {

                                                     tmpdaddyptr = rc;

                                                     rc = rc->leftChild;

                                              }

                                              else

                                                     break;

                                      }

                                      ptr->data = rc->data;

                                      tmpdaddyptr->leftChild = rc->rightChild;

                                      delete rc;

                                      return;

                              }

                       }

               }

        }

        //회전은 그림으로 설명

        Node \*llRotation(Node \*&ptr)

        {

               Node \*temp = ptr->leftChild;

               ptr->leftChild = temp->rightChild;

               temp->rightChild = ptr;

               return temp;

        }

        Node \*rrRotation(Node \*&ptr)

        {

               Node \*temp = ptr->rightChild;

               ptr->rightChild = temp->leftChild;

               temp->leftChild = ptr;

               return temp;

        }

        Node \*lrRotation(Node \*&ptr)

        {

               Node \*temp = ptr->leftChild;

               Node \*temp2 = temp->rightChild;

               ptr->leftChild = temp2->rightChild;

               temp2->rightChild = ptr;

               temp->rightChild = temp2->leftChild;

               temp2->leftChild = temp;

               return temp2;

        }

        Node \*rlRotation(Node \*&ptr)

        {

               Node \*temp = ptr->rightChild;

               Node \*temp2 = temp->leftChild;

               ptr->rightChild = temp2->leftChild;

               temp2->leftChild = ptr;

               temp->leftChild = temp2->rightChild;

               temp2->rightChild = temp;

               return temp2;

        }

        int getHeight(Node \*ptr) //높이 반환

        {

               int height = 0;

               if (ptr != NULL)

               {

                       int leftHeight = getHeight(ptr->leftChild);

                       int rightHeight = getHeight(ptr->rightChild);

                       int result = leftHeight > rightHeight ? leftHeight : rightHeight; //leftHeight과 rightHeight 중 큰 높이 반환

                       height = result + 1;

               }

               return height;

        }

        int getBF(Node \*ptr) //해당 노드의 bf 반환

        {

               return ptr->bf;

        }

        void setBF(Node \*ptr) //모든 노드의 bf를 초기화

        {

               queue<Node\*> q; //큐를 이용

               Node \*currentNode = ptr;

               while (currentNode)

               {

                       int leftHeight = getHeight(currentNode->leftChild);

                       int rightHeight = getHeight(currentNode->rightChild);

                       currentNode->bf = leftHeight - rightHeight;

                       if (currentNode->leftChild)

                              q.push(currentNode->leftChild);

                       if (currentNode->rightChild)

                              q.push(currentNode->rightChild);

                       if (q.empty())

                              break;

                       currentNode = q.front();

                       q.pop();

               }

        }

public:

        AVL()

        {

               root = NULL;

               Initialize();

        }

        void Initialize(void) //초기 설정

        {

               Insert(19);

               Insert(10);

               Insert(46);

               Insert(4);

               Insert(14);

               Insert(37);

               Insert(55);

               Insert(7);

               Insert(12);

               Insert(18);

               Insert(28);

               Insert(40);

               Insert(51);

               Insert(61);

               Insert(21);

               Insert(32);

               Insert(49);

               Insert(58);

        }

        void Insert(int element)

        {

               Insert(root, element);

        }

        void Delete(int element)

        {

               Delete(root, element);

        }

        void Search(int element)

        {

               //지나가는 경로를 큐에 저장하고 순차적으로 출력

               Node \*ptr = root;

               queue<Node\*> q;

               bool find = false;

               q.push(ptr);

               if (element == root->data) //root의 데이터를 찾을 경우

               {

                       cout << root->data << endl;

                       return;

               }

               while (ptr && element!=root->data) //root가 아닌 노드의 데이터를 찾는 경우

               {

                       if (element < ptr->data)

                       {

                              ptr = ptr->leftChild;

                              q.push(ptr);

                       }

                       else if (element > ptr->data)

                       {

                              ptr = ptr->rightChild;

                              q.push(ptr);

                       }

                       else if (element==ptr->data)

                       {

                              find = true;

                              break;

                       }

               }

               if (find)

               {

                       while (!q.empty())

                       {

                              cout << q.front()->data;

                               q.pop();

                              if (!q.empty())

                                      cout << " -> ";

                       }

                       cout << endl;

               }

               else

                       cout << "찾으시는 데이터가 없습니다" << endl;

        }

        void Visit(Node \*ptr)

        {

               //해당 노드와 자식 노드가 존재한다면 자식 노드의 데이터까지 출력

               cout << ptr->data << " ";

               if (ptr->leftChild)

                       cout << "left : " << ptr->leftChild->data << " ";

               if (ptr->rightChild)

                       cout << "right: " << ptr->rightChild->data << " ";

               cout << endl;

        }

        //아래 주석으로 처리한 코드는 balance factor가 잘 출력되나 확인하기 위해 작성한 코드

        /\*

        void Visit2(Node \*ptr)

        {

               cout << ptr->data<<":"<<ptr->bf << endl;

        }

        void printBF()

        {

               queue<Node \*> q;

               Node \*currentNode = root;

               while (currentNode)

               {

                       Visit2(currentNode);

                       if (currentNode->leftChild)

                              q.push(currentNode->leftChild);

                       if (currentNode->rightChild)

                              q.push(currentNode->rightChild);

                       if (q.empty())

                              break;

                       currentNode = q.front();

                       q.pop();

               }

               cout << endl;

        }

        \*/

        friend ostream &operator<<(ostream &os, AVL &a);

};

ostream &operator<<(ostream &os, AVL &a)

{

        queue<Node\*> q;

        Node \*currentNode = a.root;

        while (currentNode)

        {

               a.Visit(currentNode);

               if (currentNode->leftChild)

                       q.push(currentNode->leftChild);

               if (currentNode->rightChild)

                       q.push(currentNode->rightChild);

               if (q.empty())

                       break;

               currentNode = q.front();

               q.pop();

        }

        os << endl;

        return os;

}

Main.cpp

#include "avl.h"

int main(void)

{

        AVL avl;

        int sel, data;

        while (1)

        {

               cout << "AVL Tree" << endl;

               cout << "1. Search" << endl;

               cout << "2. Add" << endl;

               cout << "3. Delete" << endl;

               cout << "선택: ";

               cin >> sel;

               if (sel == 4)

                       break;

               switch (sel)

               {

               case 1:

                       cout << "Search: ";

                       cin >> data;

                       cout << "[Result]" << endl;

                       avl.Search(data);

                       break;

               case 2:

                       cout << "Add: ";

                       cin >> data;

                       cout << "[Result]" << endl;

                       avl.Insert(data);

                       cout << avl;

                       break;

               case 3:

                       cout << "Delete: ";

                       cin >> data;

                       cout << "[Result]" << endl;

                       avl.Delete(data);

                       cout << avl;

                       break;

               default:

                       cout << "다시 입력하세요(1~3)" << endl;

                       break;

               }

        }

        return 0;

}