컴퓨터알고리즘과실습 실습7

2020112046 이재원

```
#include <algorithm>
#include <iostream>
#include <random>
#include <vector>

using namespace std;

typedef int itemType;
typedef int infoType;

class BST {
   private:
    struct node {
        int key;
        infoType info;
        struct node *left, *right;
        node(int k, infoType i, struct node *l, struct node *r) {
            key = k;
            info = i;
            left = l;
            right = r;
        }
    };
    struct node *head, *z;

   public:
    BST() {
        z = new node(0, 0, 0, 0);
        head = new node(0, 0, z, z);
    }
    ~BST(){};
    infoType BSTsearch(int v, int &count);
    void inorderSearch(struct node *x);
    void BSTinsert(int v, infoType info);
};

infoType BST::BSTsearch(int v, int &count) {
    struct node *x = head->right;
    while (v != x->key) {
        count++;
        // 키값이 같은 지와, 큰 지 작은 지 비교횟수 각각 1번씩으로 침.
        count++;
        if (v < x->key)
            x = x->left;
        else
```

```
            x = x->right;
    }
    return x->key;
}

void BST::BSTinsert(int v, infoType info) {
    struct node *p, *x;
    p = head;
    x = head->right;
    while (x != z) {
        p = x;
        if (v < x->key)
            x = x->left;
        else
            x = x->right;
    }
    x = new node(v, info, z, z);
    if (v < p->key)
        p->left = x;
    else
        p->right = x;
}

int inorderLevel = 0;
int inorderArr[2000];
int inorderIndex = 0;
// inorder traversal 결과는 inorderArr에 저장
void BST::inorderSearch(struct node *x) {
    if (!inorderLevel) x = head->right;
    inorderLevel++;
    if (x->left != z) inorderSearch(x->left);
    inorderArr[inorderIndex++] = x->key;
    if (x->right != z) inorderSearch(x->right);
}

class RBTree {
   private:
    struct node {
        int key;
        infoType info;
        struct node *left, *right, *p;
        bool color;
        node(int k, infoType i, struct node *l, struct node *r, struct node *pp, bool
c) {
            key = k;
            info = i;
            left = l;
            right = r;
            p = pp;
            color = c;
        }
    };
    struct node *head, *z;

   public:
    RBTree() {
        z = new node(0, 0, 0, 0, 0, 0);
        head = new node(0, 0, z, z, z, 0);
    }
    ~RBTree(){};
    infoType RBsearch(int v, int &count);
    void inorderSearch(struct node *x);
    void RBinsert(int v, infoType info);
    void leftRotate(struct node *x);
    void rightRotate(struct node *x);
    void RBinsertFixup(struct node *x);
};

infoType RBTree::RBsearch(int v, int &count) {
    struct node *x = head->right;
    while (v != x->key) {
        count++;
        // 키값이 같은 지와, 큰 지 작은 지 비교횟수 각각 1번씩으로 침.
        count++;
        if (v < x->key)
            x = x->left;
        else
            x = x->right;
    }
    return x->key;
}

void RBTree::RBinsert(int v, infoType info) {
    struct node *p, *x;
    p = head;
    x = head->right;
    while (x != z) {
        p = x;
        if (v < x->key)
            x = x->left;
        else
            x = x->right;
    }
    x = new node(v, info, z, z, p, 1);
    if (v < p->key)
        p->left = x;
    else
        p->right = x;
    RBinsertFixup(x);
}

void RBTree::leftRotate(struct node *x) {
    struct node *y = x->right;
    x->right = y->left;
    if (y->left != z) y->left->p = x;
    y->p = x->p;
    if (x->p == head)
        head->right = y;
    else if (x == x->p->left)
        x->p->left = y;
    else
```

```
        x->p->right = y;
    y->left = x;
    x->p = y;
}

void RBTree::rightRotate(struct node *x) {
    struct node *y = x->left;
    x->left = y->right;
    if (y->right != z) y->right->p = x;
    y->p = x->p;
    if (x->p == head)
        head->right = y;
    else if (x == x->p->right)
        x->p->right = y;
    else
        x->p->left = y;
    y->right = x;
    x->p = y;
}

void RBTree::RBinsertFixup(struct node *x) {
    while (x->p->color) {
        if (x->p == x->p->p->left) {
            struct node *y = x->p->p->right;
            if (y->color) {
                x->p->color = 0;
                y->color = 0;
                x->p->p->color = 1;
                x = x->p->p;
            } else {
                if (x == x->p->right) {
                    x = x->p;
                    leftRotate(x);
                }
                x->p->color = 0;
                x->p->p->color = 1;
                rightRotate(x->p->p);
            }
        } else {
            struct node *y = x->p->p->left;
            if (y->color) {
                x->p->color = 0;
                y->color = 0;
                x->p->p->color = 1;
                x = x->p->p;
            } else {
                if (x == x->p->left) {
                    x = x->p;
                    rightRotate(x);
                }
                x->p->color = 0;
                x->p->p->color = 1;
                leftRotate(x->p->p);
            }
        }
    }
    head->right->color = 0;
```

```cpp
}

int RBinorderLevel = 0;
int RBinorderArr[2000];
int RBinorderIndex = 0;
// RBT inorder traversal 결과는 RBinorderArr에 저장
void RBTree::inorderSearch(struct node *x) {
    if (!RBinorderLevel) x = head->right;
    RBinorderLevel++;
    if (x->left != z) inorderSearch(x->left);
    RBinorderArr[RBinorderIndex++] = x->key;
    if (x->right != z) inorderSearch(x->right);
}

// Hash Table with division method and chaining
// Hash Table size = 1009 (prime number)
class HT {
   private:
    struct node {
        int key;
        struct node *next;
        node(int k, struct node *n) {
            key = k;
            next = n;
        }
    };
    struct node *HashNode[1009];

   public:
    HT() {
        for (int i = 0; i < 1009; i++) {
            HashNode[i] = NULL;
        }
    }
    ~HT(){};
    void HTsearch(int v, int &count);
    void HTinsert(int v);
};

void HT::HTsearch(int v, int &count) {
    int index = v % 1009;
    struct node *x = HashNode[index];
    count++;  // 비교횟수 1번 (HashNode[index]->key = v인지 확인)
    while (v != x->key) {
        count++;  // 비교횟수 1회 추가 (while문 조건)
        x = x->next;
    }
}

void HT::HTinsert(int v) {
    int index = v % 1009;
    struct node *x = HashNode[index];
    if (x == NULL) {
        HashNode[index] = new node(v, NULL);
    } else {
        while (x->next != NULL) {
            x = x->next;
        }
        struct node *temp = new node(v, NULL);
        x->next = temp;
    }
}

int main() {
    BST T1;
    vector<int> a;
    int N = 2000;

    for (int i = 0; i < N; i++) {
        a.push_back(i);
    }

    // shuffle the vector
    random_device rd;
    default_random_engine rng(rd());
    shuffle(a.begin(), a.end(), rng);

    // insert
    for (int i = 0; i < N; i++) {
        cout << a[i] << " ";
        T1.BSTinsert(a[i], 1);
    }

    // inorder traversal
    T1.inorderSearch(NULL);
    cout << endl;



    // print inorder traversal result
    for (int i = 0; i < N; i++) {
        cout << inorderArr[i] << " ";
    }




    // insert inorderArr in BST T2
    BST T2;
    for (int i = 0; i < N; i++) {
        T2.BSTinsert(inorderArr[i], 1);
    }

    // insert inorderArr in RBTree T3
    RBTree T3;
    for (int i = 0; i < N; i++) {
        T3.RBinsert(inorderArr[i], 1);
    }
    int compareCntT1 = 0, compareCntT2 = 0, compareCntT3 = 0;
    for (int i = 0; i < N; i++) {
        T1.BSTsearch(i, compareCntT1);
        T2.BSTsearch(i, compareCntT2);
        T3.RBsearch(i, compareCntT3);
    }

    cout << endl
         << "---------------------( 7 - 1 )---------------------" << endl;
    // print Average compareCnts
    cout << endl
         << "Average compareCnts of BST T1 : " << (double)compareCntT1 / (double)N <<
endl;
    cout << "Average compareCnts of BST T2 : " << (double)compareCntT2 / (double)N <<
endl;
    cout << "Average compareCnts of RBTree T3 : " << (double)compareCntT3 / (double)N
<< endl
         << endl;

    // insert rand() in Hash Table T4
    HT T4;
    for (int i = 0; i < N; i++) {
        T4.HTinsert(a[i]);
    }

    // search rand() in Hash Table T4
    int compareCntT4 = 0;
    for (int i = 0; i < N; i++) {
        T4.HTsearch(a[i], compareCntT4);
    }

    cout << "---------------------( 7 - 2 )---------------------" << endl;
    // print Average compareCnts
    cout << endl
         << "Average compareCnts of Hash Table T4 : " << (double)compareCntT4 / (double)N
<< endl
         << endl;

    return 0;
}
```

실습과정 / 결과

Vector를 이용해 0~1999 사이의 숫자를 랜덤하게 배열하여 랜덤한 2000개의 자료 생성.

생성한 자료를 BST(T1)에 insert, inorder traversal(inorder traversal 결과는 inorderArr에

저장. 0~1999의 숫자가 순서대로 담기게 됨) 결과 출력

BSTree(T2) -> inorderArr을 이용해 새로운 트리를 생성.

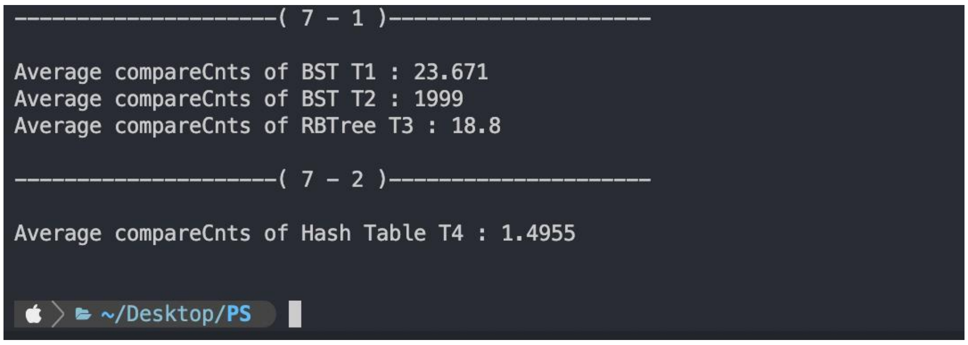
RBTree(T3) 구현 -> 234Tree 생성, T1을 inorder traversal한 결과인 inorderArr을 이용해 순서대로 insert, 234-Tree를 먼저 생성하고 Rotate, Fixup 등을 통해 RBTree로 고쳐나가는 형식으로 구현.

Hash Table(T4) 구현 -> division method, chaining 방법을 이용해 hash table을 구현.

collision을 고의로 발생시키기 위해, size는 2000 이하로 설정.

1000보다 큰 최소의 prime number인 1009를 size로 설정.

T1, T2, T3, T4 의 search에서 key의 평균 비교 회수 결과



```
---------------------( 7 - 1 )---------------------

Average compareCnts of BST T1 : 23.671
Average compareCnts of BST T2 : 1999
Average compareCnts of RBTree T3 : 18.8

---------------------( 7 - 2 )---------------------

Average compareCnts of Hash Table T4 : 1.4955

 ~/Desktop/PS
```

-> T1의 평균 비교 회수는 실행할 때마다 값이 바뀌나(실행 시마다 섞인 형태가 다르므로)

T3의 경우 균형 트리의 특성상 평균 비교 회수가 18.8로 계속 동일했다.