[백준]탐색트리

imhyun · 방금 전

백준 알고리즘 자료구조

https://www.acmicpc.net/problem/1620

> 문자열로 구성된 포켓몬 이름을 저장해, 숫자가 주어지면 해당 순서에 해당하는 이름을, 이름이 주어지면 몇 번째 순서인지 출력하는 문제이다.

아이디어

숫자를 기준으로 AVL 트리를 작성하게 되면, 문자열을 찾는 데에 어려움이 생기고, 문자열을 기준으로 사전을 구현한다면, 숫자를 찾는 데에 어려움이 생긴다. 따라서 문자열을 기반으로 한 AVL 트리 사전을 만들고 이름을 갖고 있는 배열을 하나 추가적으로 만들어주겠다.
포켓몬의 이름이 주어지면 AVL 트리에서 찾아 O(log n) 속도로 번호를 찾을 수 있다. 번호가 주어졌을 시에는 단순히 배열의 인덱스로 접근한다.

개념 - AVL 트리

이진탐색트리(Binary search tree)

> u,v,w 모두 트리 노드이며, u와 w가 **v의 왼쪽과 오른쪽 부트리**에 존재할 때 다음에 성립.
> **중위순회**하면 키가 **증가**하는 순서로 방문한다.

$$key(u) < key(v) \leq key(w)$$

이진탐색트리의 성능

> 높이 h의 이진탐색트리를 사용하여 n항목의 사전을 가정하면,
> O(n) 공간 사용
> findElement,insertItem,removeElement 모두 O(h)의 시간 수행

최악의 경우 O(n)

- 편향이진트리

최선의 경우 O(logn)

- 균형이진트리

코드

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include<time.h>
#pragma warning (disable:4996)

//노드 구조체
typedef struct Node {
    struct Node* lChild;
    struct Node* rChild;
    struct Node* parent;
    int key;
}Node;

typedef struct Tree {
    Node* root;
}Tree;

//트리
Tree BST;

//함수
void init(Tree* T);

Node* makeNode();

int isExternal(Node* node);

void findItem(int k);

Node* treeSearch(Node* r, int k);

void insertItem(int k);
```

```
void priorOrder(Node* root);

Node* sibling(Node* node);

void reduceExternal(Node* z);

Node* inOrderSucc(Node* node);

void deleteItem(int k);

int main(void)
{

    init(&BST);

    //ADT
    char type;
    int k;

    while (1)
    {
        scanf("%c", &type);

        if (type == 'i') {
            scanf("%d", &k);
            insertItem(k);
        }

        else if (type == 'd') {
            scanf("%d", &k);
            deleteItem(k);
        }

        else if (type == 's') {
            scanf("%d", &k);
            findItem(k);
        }

        else if (type == 'p') {
            priorOrder(BST.root);
            printf("\n");
        }

        else break;

        getchar();

    }



}

void init(Tree* T) {
    T->root = makeNode();

    T->root->lChild = NULL;
```

```
        T->root->rChild = NULL;
        T->root->parent = NULL;

    }

    Node* makeNode() {
        Node* node = (Node*)malloc(sizeof(Node));

        node->parent = NULL;
        node->lChild = NULL;
        node->rChild = NULL;

        return node;
    }

    int isExternal(Node* node) {
        if ((node->lChild == NULL) && (node->rChild == NULL))
            return 1;

        else
            return 0;
    }

    Node* treeSearch(Node* r, int k) {
        if (isExternal(r))
            return r;

        if (r->key == k)
            return r;

        else if (r->key > k)
            return treeSearch(r->lChild, k);

        else
            return treeSearch(r->rChild, k);
    }

    void findItem(int k) {
        Node* node = treeSearch(BST.root, k);

        if (isExternal(node))
            printf("X\n");

        else
            printf("%d\n",k);
    }

    void insertItem(int k) {
        Node* node = treeSearch(BST.root, k);

        if (isExternal(node)) {
            node->key = k;

            node->lChild = makeNode();
            node->lChild->parent = node;

            node->rChild = makeNode();
            node->rChild->parent = node;
```

```
        }

        else return;
    }

    void priorOrder(Node* root) {
        if (isExternal(root))
            return;

        printf(" %d", root->key);
        priorOrder(root->lChild);
        priorOrder(root->rChild);

    }

    Node* sibling(Node* node) {
        if (node == BST.root)
            return NULL;

        else {
            Node* p = node->parent;

            if (p->lChild == node)
                return p->rChild;

            else
                return p->lChild;
        }
    }

    void reduceExternal(Node* z) {
        Node* w = z->parent;
        Node* zs = sibling(z);

        if (w == BST.root) {
            zs->parent = NULL;
            BST.root = zs;
        }

        else {
            Node* pp = w->parent;
            zs->parent = pp;

            if (pp->lChild == w)
                pp->lChild = zs;
            else
                pp->rChild = zs;
        }

        free(z);
        free(w);
    }

    Node* inOrderSucc(Node* node) {
        Node* p = node->rChild;

        if (isExternal(p))
            return p;
```

```
        while (!isExternal(p->lChild)) {
            p = p->lChild;
        }

        return p;
    }

    void deleteItem(int k) {
        Node* delete = treeSearch(BST.root, k);

        if (isExternal(delete))
            printf("X\n");

        else {
            int e = delete->key;

            Node* z = delete->lChild;

            if (!isExternal(z)) {
                z = delete->rChild;
            }

            if (isExternal(z))
                reduceExternal(z);

            else {
                Node* y = inOrderSucc(delete);
                delete->key = y->key;
                z = y->lChild;
                reduceExternal(z);
            }

            printf("%d\n", e);
        }
    }
```

AVL 트리

> 편향이진트리를 막기 위해, **높이의 균형성을 보장**하기 위한 AVL 트리. (탐색,삽입,삭제 ADT의 효율이 좋아짐.)
> 모든 내부노드 v에 대해 **v의 자식들의 좌우 높이 차이가 1을 넘지 않**는다.

- **높이 균형 속성**
  - AVL 트리의 부트리 역시 AVL 트리
  - **높이** 정보는 각 **내부노드에 저장**
  - AVL 트리의 높이 : O(logn)

> **삽입 및 삭제** 작업은 이진탐색트리의 ADT와 비슷하나, 높이균형속성이 파괴될 수도 있다.
> 따라서 생겼을지도 모를 **불균형**을 "**찾아서 수리**"해야 한다.
> 1. **불균형 찾기** : **균형검사**를 통해 찾는다.
> 2. 불균형 **수리** : **개조**를 통해 높이 균형 속성을 회복한다.

AVL 트리에서의 삽입

```
Alg inserItem(k,e)
    input AVL tree T,key k,element e
    output none
1. w<-treeSearch(root(),k)
2.if(isInternal(w))
        return
    else
        Set node w to (k,e)
        expandExternal(w)
        searchAndFixAfterInsertion(w)
        return


Alg searchAndFixAfterInsertion(w)
    input internal node w
    output none
1. w에서 **T의 루트로 향해 올라가**다 **처음 만나는 불균형** 노드 z
2. z의 높은 자식을 y
    **{y는 w의 조상}**
3. y의 높은 자식을 x
4. restructure(x,y,z)
    {x,y,z 모두 전역적으로나 지역적으로나 모두 복구됨}
5. return
```

개조

```
Alg restructure(x,y,z)
    input a node x of a binary search tree T that has both a parent y and a grandparent z
    output tree T after restructuring involving nodes x,y,and z

1. x,y,z의 중위순회 방문 순서 나열 (a,b,c)
2. x,y,z의 부트리 가운데 x,y,z를 루트로 하는 부트리를 제외한 4개의 부트리 중위 순서 (T0,T1,T2,T3)
3. z를 루트로 하는 부트리를 b를 루트로 하는 부트리로 대체
4. T0와 T1을 각각 a의 왼쪽, 오른쪽 부트리로 만든다.
5. T2와 T3을 각각 c의 왼쪽, 오른쪽 부트리로 만든다.
6. a와 c를 각각 b의 왼쪽 및 오른쪽 자식으로 만든다.
7. return b
```

- 3 -노드 개조 (회전이라고도 불림)
    - x,y,z의 중위순회 순서 a<b<c를 회전축으로 하여 수행

- **단일 회전**
    - b=y면, y를 중심으로 z을 회전
- **이중 회전**
    - b=x이면, x를 중심으로 y를 회전한 후, 다시 x를 중심으로 z을 회전

AVL 트리의 성능

- restructure **O(1)**
    - 연결 이진트리 사용을 전제로
    - 단순히 부모 자식 관계만 수정
- 탐색 ADT - 개조 X **O(log n)**
- 삽입, 삭제 ADT **O(log n)**
    - 개조를 수행하여 높이 균형을 회복 **O(log n)**

코드

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include<time.h>
#pragma warning (disable:4996)

//노드 구조체
typedef struct Node {
    struct Node* lChild;
    struct Node* rChild;
    struct Node* parent;

    int key;
    int height;
}Node;

typedef struct Tree {
    Node* root;
}Tree;

//트리
Tree BST;

//함수
void init(Tree* T);

Node* makeNode();

int isExternal(Node* node);

void findItem(int k);

Node* treeSearch(Node* r, int k);
```

```
int updateHeight(Node* w);

int isBalanced(Node* z);

void searchAndFixAfterInsertion(Node * node);

void insertItem(int k);

void priorOrder(Node* root);

Node* sibling(Node* node);

void reduceExternal(Node* z);

Node* inOrderSucc(Node* node);

void deleteItem(int k);

int main(void)
{

    init(&BST);

    //ADT
    char type;
    int k;

    while (1)
    {
        scanf("%c", &type);

        if (type == 'i') {
            scanf("%d", &k);
            insertItem(k);
        }

        else if (type == 'd') {
            scanf("%d", &k);
            deleteItem(k);
        }

        else if (type == 's') {
            scanf("%d", &k);
            findItem(k);
        }

        else if (type == 'p') {
            priorOrder(BST.root);
            printf("\n");
        }

        else break;

        getchar();

    }
```

```
}

void init(Tree* T) {
    T->root = makeNode();

    T->root->lChild = NULL;
    T->root->rChild = NULL;
    T->root->parent = NULL;

}

Node* makeNode() { //외부노드를 형성
    Node* node = (Node*)malloc(sizeof(Node));

    node->parent = NULL;
    node->lChild = NULL;
    node->rChild = NULL;

    node->height = 0;

    return node;
}

int isExternal(Node* node) {
    if ((node->lChild == NULL) && (node->rChild == NULL))
        return 1;

    else
        return 0;
}

Node* treeSearch(Node* r, int k) {

    if (isExternal(r))
        return r;

    if (r->key == k)
        return r;

    else if (r->key > k)
        return treeSearch(r->lChild, k);

    else
        return treeSearch(r->rChild, k);

}

void findItem(int k) {
    Node* node = treeSearch(BST.root, k);

    if (isExternal(node))
        printf("X\n");

    else
        printf("%d\n",k);
}

int isBalanced(Node* z) {
```

```
        int dif = z->lChild->height - z->rChild->height;

        if (dif >= 2 || dif <= -2)
            return 0;
        else
            return 1;
    }

    int updateHeight(Node* w) {

        int height;

        if (w->lChild->height > w->rChild->height) {

            height = w->lChild->height + 1;

        }

        else {

            height = w->rChild->height + 1;

        }

        if (height != w->height) {

            w->height = height;

            return 1;

        }

        else {

            return 0;

        }

    }

    Node* restructure(Node* x, Node* y, Node* z) {
        Node* a, * b, * c;
        Node* T0, * T1, * T2, * T3;

        if ((z->key < y->key) && (y->key < x->key)) {
            a = z;
            b = y;
            c = x;

            T0 = a->lChild;
            T1 = b->lChild;
            T2 = c->lChild;
            T3 = c->rChild;
        }

        else if ((z->key > y->key) && (y->key > x->key)) {
            a = x;
            b = y;
```

```
        c = z;

        T0 = a->lChild;
        T1 = a->rChild;
        T2 = b->rChild;
        T3 = c->rChild;
    }

    else if ((z->key < x->key) && (y->key > x->key)) {
        a = z;
        b = x;
        c = y;

        T0 = a->lChild;
        T1 = b->lChild;
        T2 = b->rChild;
        T3 = c->rChild;
    }

    else {
        a = y;
        b = x;
        c = z;

        T0 = a->lChild;
        T1 = b->lChild;
        T2 = b->rChild;
        T3 = c->rChild;
    }

    if (z->parent == NULL) { //z를 루트로 하는 부트리 b를 루트로하는 부트리로 대체

        BST.root = b;

        b->parent = NULL;

    }

    else if (z->parent->lChild == z) {

        z->parent->lChild = b;

        b->parent = z->parent;

    }

    else if (z->parent->rChild == z) {

        z->parent->rChild = b;

        b->parent = z->parent;

    }

    a->lChild = T0;
    a->rChild = T1;
    T0->parent = a;
    T1->parent = a;
```

```
        updateHeight(a);

        c->lChild = T2;
        c->rChild = T3;
        T2->parent = c;
        T3->parent = c;
        updateHeight(c);

        b->lChild = a;
        b->rChild = c;
        a->parent = b;
        c->parent = b;
        updateHeight(b);

        return b;
    }

    void searchAndFixAfterInsertion(Node * w) {
        Node* x, * y, * z;

        if (w->parent == NULL)
            return;

        z = w->parent;

        while (updateHeight(z)&&isBalanced(z)) { //조사
            if (z->parent == NULL)
                return;

            z = z->parent;

        }

        if (isBalanced(z))
            return;

        if (z->lChild->height >= z->rChild->height)
            y = z->lChild;

        else
            y = z->rChild;

        if (y->lChild->height > y->rChild->height)
            x = y->lChild;

        else
            x = y->rChild;

        //개조
        restructure(x, y, z);

    }

    void insertItem(int k) {
        Node* node = treeSearch(BST.root, k);

        if (isExternal(node)) {
            node->key = k;
```

```
            node->height = 1;

            node->lChild = makeNode();
            node->lChild->parent = node;

            node->rChild = makeNode();
            node->rChild->parent = node;

            searchAndFixAfterInsertion(node);
        }

        else return;
    }

    void priorOrder(Node* root) {
        if (isExternal(root))
            return;

        printf(" %d", root->key);
        priorOrder(root->lChild);
        priorOrder(root->rChild);

    }

    Node* sibling(Node* node) {
        if (node == BST.root)
            return NULL;

        else {
            Node* p = node->parent;

            if (p->lChild == node)
                return p->rChild;

            else
                return p->lChild;
        }
    }

    void reduceExternal(Node* z) {
        Node* w = z->parent;
        Node* zs = sibling(z);

        if (w == BST.root) {
            zs->parent = NULL;
            BST.root = zs;
        }

        else {
            Node* pp = w->parent;
            zs->parent = pp;

            if (pp->lChild == w)
                pp->lChild = zs;
            else
                pp->rChild = zs;
        }
```

```
        free(z);
        free(w);
    }

    Node* inOrderSucc(Node* node) {
        Node* p = node->rChild;

        if (isExternal(p))
            return p;

        while (!isExternal(p->lChild)) {
            p = p->lChild;
        }

        return p;
    }

    void deleteItem(int k) {
        Node* delete = treeSearch(BST.root, k);

        if (isExternal(delete))
            printf("X\n");

        else {
            int e = delete->key;

            Node* z = delete->lChild;

            if (!isExternal(z)) {
                z = delete->rChild;
            }

            if (isExternal(z))
                reduceExternal(z);

            else {
                Node* y = inOrderSucc(delete);
                delete->key = y->key;
                z = y->lChild;
                reduceExternal(z);
            }

            printf("%d\n", e);
        }
    }
```

해결 코드

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include<time.h>
#pragma warning (disable:4996)

//노드 구조체
```

```
typedef struct Node {
    struct Node* lChild;
    struct Node* rChild;
    struct Node* parent;

    char name[21];
    int height;
    int key;
}Node;

typedef struct Tree {
    Node* root;
}Tree;

//트리
Tree BST;

//함수
void init(Tree* T);

Node* makeNode();

int isExternal(Node* node);

void findNum(char *ch);

int my_atoi( char* str) {
    int ret = 0;
    int i;
    for (i = 0; str[i] != '\0'; i++)    //NULL문자만나면 for문 종료
        ret = ret * 10 + (str[i] - '0');    //숫자 문자에 '0'을 빼면 레알 숫자가 구해짐
    return ret;
}

void findName(Node *r, char *ch) {
    if (isExternal(r))
        return;

    int k = my_atoi(ch);

    if (r->key == k) {
        printf("%s\n", r->name);
        return;
    }

    else {
        findName(r->lChild,ch);
        findName(r->rChild,ch);
    }

}

Node* treeSearch(Node* r, char* ch);

void insertItem(char *ch,int k);

void inOrder(Node* root);
```

```
int main(void)
{
    int N, M;
    char ch[21];
    int k=1;
    char question[21];

    init(&BST);

    scanf("%d %d", &N,&M);

    for (int i = 0; i < N; i++) {
        scanf("%s", ch);
        insertItem(ch,k);
        k++;
    }

    for (int i = 0; i < M; i++) {
        scanf("%s", &question);

        if ((question[0] >= '1') && (question[0] <= '9')) //숫자
            findName(BST.root,question);

        else //알파벳
            findNum(question);

    }
}

void init(Tree* T) {
    T->root = makeNode();

    T->root->lChild = NULL;
    T->root->rChild = NULL;
    T->root->parent = NULL;


}

Node* makeNode() { //외부노드를 형성

    Node* node = (Node*)malloc(sizeof(Node));

    node->parent = NULL;
    node->lChild = NULL;
    node->rChild = NULL;

    node->height = 0;

    node->key = 0;

    return node;
}

int isExternal(Node* node) {
    if ((node->lChild == NULL) && (node->rChild == NULL))
        return 1;
```

```
        else
            return 0;
    }

    Node* treeSearch(Node* r,char* ch) {

        if (isExternal(r))
            return r;

        if (strcmp(r->name, ch) < 0)
            return treeSearch(r->rChild,ch);

        else if (strcmp(r->name, ch) == 0)
            return r;

        else
            return treeSearch(r->lChild, ch);

    }

    void findNum(char* ch) {
        Node* find = treeSearch(BST.root, ch);

        if (isExternal(find))
            return;
        else
            printf("%d\n", find->key);
    }

    void insertItem(char *ch,int k) {
        Node* node = treeSearch(BST.root, ch);

        if (isExternal(node)) {
            strcpy(node->name, ch);
            node->height = 1;
            node->key = k;

            node->lChild = makeNode(0);
            node->lChild->parent = node;

            node->rChild = makeNode(0);
            node->rChild->parent = node;
        }

        else return;
    }

    void inOrder(Node* root) {
        if (isExternal(root))
            return;

        inOrder(root->lChild);
        printf(" %s", root->name);
        inOrder(root->rChild);

    }
```

문제점

올바르게 출력이 되나, 시간초과가 떴다. 이유는 2가지 문제점 때문인 거 같다.

1. AVL 트리 X
2. findName 함수 시간복잡도 O(n^2)

AVL 트리를 사용하기로 했으나, 위에 구현한 코드는 삽입 시에 개조를 하지 않는 단순 이진탐색트리를 구현하였다. 이로 인해 findItem의 시간복잡도가 높았다.
또한, findName은 트리가 이름 순으로 사전을 구현해 탐색 시에 전체를 훑어야 했다.

박시현

이전 포스트
[백준] 사전

0개의 댓글

댓글을 작성하세요

댓글 작성