Binary-Search-Tree

1115 이정우

목차

1 자료구조 개요

BST는 이진트리의 한 종류로, 각 노드가 최대 두개의 자식 노드를 가지며 왼쪽 서브 트리에는 해당 노드보다 작은값들이, 오른쪽 서브트리에는 대당 노드보다 큰 값들이 저장되는 구조를 가진 자료구조이다.

2 자료구조 상세

2.1 변수와 ADT

2.1.1 변수

- `root : int`
  - 트리 전체의 시작 노드이다.
- `Node : struct`
  - data, left, right를 저장하는 변수이다.

2.1.2 ADT

- `insert() -> void`
  - 입력받은 값을 이진 탐색 트리의 규칙에 따라 삽입하는 함수이다.
- `remover() -> void`
  - 주어진 값을 가진 노드를 트리에서 찾아 삭제하며 그와 동시에 이진탐색트리의 성질을 유지하도록 하는 함수이다.

2.1.3 Exception

- 특별한 예외 상황은 존재하지 않는다.

2.2 구현 상세 및 핵심 코드

2.2.1 `insert()`

N을 매개변수로 받아 N만큼 이진탐색트리에 데이터를 삽입하는 함수이다. 반복문으로 삽입할 위치를 찾는다.

```
void insert(){
    int N;
    scanf("%d", &N);
    for(int i=1; i<=N; i++){
        struct Node *cur = (struct Node*)malloc(sizeof(struct Node));
        scanf("%d", &cur->data);
        cur->left = NULL;
        cur->right = NULL;
        if(root==NULL){
            root=cur;
            continue;
        }
        struct Node *temp = root;
        while(1){
            if(cur->data < temp->data){
                if(temp->left == NULL){
                    temp->left = cur;
                    break;
                }
                temp = temp->left;
            }
            else{
                if(temp->right == NULL){
                    temp->right = cur;
                    break;
                }
                temp = temp->right;
            }
        }
    }
}
```

2.2.2 `remover()`

remover 함수는 입력한 데이터를 가진 노드를 삭제하는 연산을 하는 함수로, 노드의 자식이 없을경우, 0/1개인 경우, 2개인 경우를 나눠 BST형식에 맞게 재정렬한다.

```
void remover() {
    int RV;
    scanf("%d", &RV);

    struct Node *P = NULL;
    struct Node *RD = root;

    while (RD && RD->data != RV) {
        P = RD;
        if (RV < RD->data)
            RD = RD->left;
        else
            RD = RD->right;
    }
    if (!RD) return;

    if (RD->left && RD->right) {

        struct Node *sp = RD;
        struct Node *s = RD->right;
        while (s->left) {
            sp = s;
            s = s->left;
        }

        RD->data = s->data;
        struct Node *child = s->right;
        if (sp->left == s)
            sp->left = child;
        else
            sp->right = child;

        free(s);
        return;
    }

    struct Node *child = (RD->left) ? RD->left : RD->right;
```

```
    if (P == NULL)
        root = child;
    else if (P->left == RD)
        P->left = child;
    else
        P->right = child;

    free(RD);
}
```

2.3 복잡도 분석

- 위 코드를 바탕으로 시간복잡도를 분석한다면 최초 삽입시 O(1)의 시간복잡도를 가지며 일반적으로 O(N log N)의 시간 복잡도를 가진다. 한쪽으로 치우친 트리가 되면 O(N^2)의 시간복잡도를 가진다. 또한 remover함수는 삭제할 노드를 찾고 포인터를 조정하는 과정이 트리 높이에 비례하므로 평균 O(log N), 최악 O(N) 시간이 걸린다. 그러므로 전체 코드의 시간복잡도는 O(N log N)을 평균으로 가지며 최악의 경우 O(N^2)의 시간복잡도를 가진다..

2.4 전체 코드

```
#include <stdio.h>
#include <stdlib.h>
struct Node{
    int data;
    struct Node *left;
    struct Node *right;
};

struct Node *root = NULL;

void remover() {
    int RV;
    scanf("%d", &RV);

    struct Node *P = NULL;
    struct Node *RD = root;

    while (RD && RD->data != RV) {
        P = RD;
```

```
        if (RV < RD->data)
            RD = RD->left;
        else
            RD = RD->right;
    }
    if (!RD) return;

    if (RD->left && RD->right) {

        struct Node *sp = RD;
        struct Node *s = RD->right;
        while (s->left) {
            sp = s;
            s = s->left;
        }

        RD->data = s->data;
        struct Node *child = s->right;
        if (sp->left == s)
            sp->left = child;
        else
            sp->right = child;

        free(s);
        return;
    }

    struct Node *child = (RD->left) ? RD->left : RD->right;

    if (P == NULL)
        root = child;
    else if (P->left == RD)
        P->left = child;
    else
        P->right = child;

    free(RD);
}


void insert(){
    int N;
```

```
    scanf("%d", &N);
    for(int i=1; i<=N; i++){
        struct Node *cur = (struct Node*)malloc(sizeof(struct Node));
        scanf("%d", &cur->data);
        cur->left = NULL;
        cur->right = NULL;
        if(root==NULL){
            root=cur;
            continue;
        }
        struct Node *temp = root;
        while(1){
            if(cur->data < temp->data){
                if(temp->left == NULL){
                    temp->left = cur;
                    break;
                }
                temp = temp->left;
            }
            else{
                if(temp->right == NULL){
                    temp->right = cur;
                    break;
                }
                temp = temp->right;
            }
        }
    }
}

int main(){
    insert();
    remover();
}
```

3 활용 방안

1. 빠른 탐색이 필요할때
    - BST를 활용한다면 생성된 BST를 중위탐색하여 정렬된 데이터를 출력할수 있으며 특정 값을 찾는것 또한 쉽게 가능하다.

4 인상깊었던 점

이진검색트리를 생각할때면 저걸 어떻게 하지라는 생각으로 망막하게 생각했던때가 생각난다. 그러나 뭔가 코드로 동작을 하나하나 짜보니 그다지 어렵지 않았고 연결리스트보다 효율적이라는 사람들의 말에 자주 애용할것만같은 느낌이들었다.

5 참고문헌

이진검색트리(Binary Search Tree)