알고리즘 과제(04)

201204025 김대래

과목 명 : 알고리즘
담당 교수 : 김동일
분반 : 02 분반

Index

1. 실습 환경

OS : MacOs Mojave 10.14
Language : C
Tool : Neovim, gcc

2. 과제 개요

임의의 나열된 수가 저장된 파일(DataN.txt)을 입력 받아 입력 방법(Insert, median insert, balanced median insert)에 따라 이진탐색트리를 구성하고 결과를 파일로 출력한다.

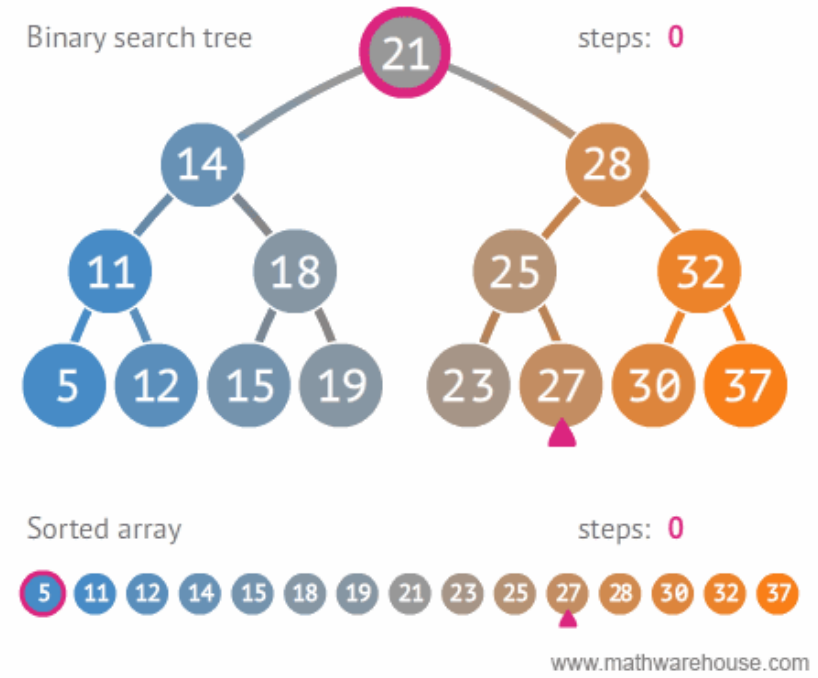
입력 예시 : Data1.txt / Data2.txt
>> Input test number [1: Data1.txt, 2: Data2.txt] :
>> Input insert mode [1 : insert, 2: median, 3: balance median] :

출력 예시 : inorderTree.txt

3. 주요 개념

3.1. 이진탐색트리(Binary Search Tree)

3.1.1. 개념



이진탐색트리란 이진탐색(binary search)과 연결리스트(linked list)를 결합한 자료구조의 일종으로 이진탐색의 효율적인 탐색 능력을 유지하면서도, 빈번한 자료 입력과 삭제를 가능하게끔 고안되었다.

예컨대 이진탐색의 경우 탐색에 소요되는 복잡도는 $O(\log n)$으로 빠르지만 자료 입력, 삭제가 불가능합니다. 연결리스트의 경우 자료 입력, 삭제에 필요한 복잡도는 $O(1)$로 효율적이지만 탐색하는 데에는 O(n)의 복잡도가 발생한다.

3.1.2. 삽입

개념의 이미지를 보면 알 수 있듯이 **노드의 왼쪽 서브트리는 노드의 값보다 반드시 작아야 하며 오른쪽 서브트리는 노드의 값보다 반드시 커야한다.**

삽입 시에도 마찬가지이며 root 노드에서부터 자식까지 값의 크기를 비교하며 위치를 찾아 삽입을 한다.

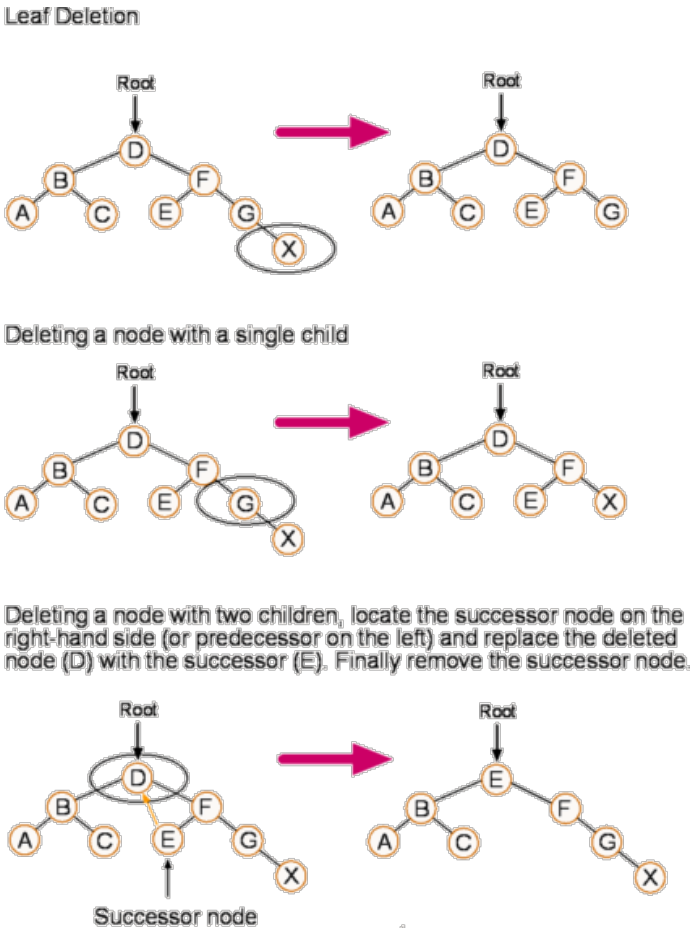
3.1.3. 삭제

이진탐색트리의 삭제는 3가지의 경우의 수를 고려해야한다.

case 1:
    자식이 없는 경우
case 2:
    자식이 하나일 경우
case 3:
    자식이 둘일 경우

case 1, 2의 경우 어렵지 않게 삭제를 하거나(case 1) 삭제 후 자식 노드를 부모에게 연결(case2)하면 된다.

case 3일 때, 자신보다 큰 가장 작은 값(Successor)를 찾아 값을 교환해주고 삭제를 하는 방식이다.



3.1.4. 탐색

이진 탐색 트리의 경우 root노드에서부터 해당 값을 찾기 위해 자식을 탐색해 나가는 방식으로 최악의 경우 높이 h만큼 탐색해야하기 때문에 복잡도가 $O(h)$이며 이진 트리(Binary Tree)의 특성에 따라 n개의 원소의 최대 높이인 $O(logn)$의 복잡도를 가진다.

4. 과제 4-1 Binary Search Tree - Insert

4.1. main() - input data

```
input_FD = freopen(inputFileName, "r", stdin);

int *arr = (int *)malloc(sizeof(int) * test_num * 10);

if(mode == 1)
    start = clock();
int i;
for(i = 0; i < (test_num * 10); i++){
    int x;
    scanf("%d", &x);
    if(mode == 1){
        if(i == 0){
            root = tree_insert(root, x);
        }
        else{
            tree_insert(root, x);
        }
    }
    else if(mode == 2 || mode == 3){
        arr[i] = x;
    }
    else{
        printf("[!] Mode Error!");
        exit(1);
    }
}
```

I. test_num (1 or 2) 를 받아서 각 파일의 크기만큼 (* 10) 반복하며 파일을 읽어 tree_insert(node, data) 함수를 통해 이진 탐색 트리 구조로 삽입

II. 만약 root 가 NULL이면 삽입을 하고 해당 원소를 루트로 지정

4.2. newNode(node) / tree_insert(node, data)

```
Node* newNode(int data){
    Node *new_node = (Node *)malloc(sizeof(Node));
    if(new_node == NULL)
    {
        fprintf (stderr, "error memory!! (create_node)\n");
        exit(1);
    }
    new_node->data = data;
    new_node->left = NULL;
    new_node->right = NULL;
    new_node->parent = NULL;

    return new_node;
}

Node* tree_insert(Node *node, int data){
    Node *temp = (Node *)malloc(sizeof(Node));

    // if the tree is empty, return a new node
    if (node == NULL) return newNode(data);

    if(data < node->data){
        temp = tree_insert(node->left, data);
        node->left = temp;
        temp->parent = node;
    }
    else if(data > node->data){
        temp = tree_insert(node->right, data);
        node->right = temp;
        temp->parent = node;
    }

    return node;
}
```

I. data를 비교하며 자리를 찾아가면서 자식을 재귀로 data의 위치 탐색
II. Node가 NULL이면, 즉 해당 노드가 들어가야할 자리이면 노드를 생성

4.3. tree_search(node, data)

```
/*Recursive Search*/
Node* tree_R_search(Node *node, int data){
    if(node){
        if(data == node->data)
            return node;
        if(data < node->data)
            return tree_R_search(node->left, data);
        else
            return tree_R_search(node->right, data);
    }
    else
        return NULL;
}

/*Iterative Search*/
Node* tree_I_search(Node *node, int data){
    while (node != NULL && data != node->data){
        if(data < node->data)
            node = node->left;
        else
            node = node->right;
    }
    return node;
}
```

I. 재귀 혹은 반복을 통하여 data의 위치를 찾는 함수
II. 현재 노드의 값보다 작으면 왼쪽 서브트리로, 크면 오른쪽 서브트리로
III. 값을 찾으면 해당 노드를 반환

4.4. Successor / Predecssor

```
Node* succsesor(Node *node){
    if(node->right != NULL){
        return tree_minimum(node->right);
    }

    Node *y = node->parent;
    while( y != NULL && node == y->right){
        node = y;
        y = y->parent;
    }
    return y;
}

Node* tree_minimum(Node *node){
    Node *current = node;
    while(current->left != NULL){
        current = current->left;
    }

    return current;
}

Node* predecessor(Node *node){
    if(node->left != NULL){
        return tree_maximum(node->left);
    }
    Node *y = node->parent;
    while(y != NULL && node == y->left){
        node = y;
        y = y->parent;
    }

    return y;
}

Node* tree_maximum(Node *node){
    Node *current = node;
    while(current->right != NULL){
        current = current->right;
    }
    return current;
}
```

I. 해당 노드보다 큰 가장 작은 수인 Successor를 찾는 함수
II. 해당 노드보다 작은 가장 큰 수인 Predecessor를 찾는 함수
III. tree_minimum/maximum은 해당 노드를 기준으로 가장 작은/큰 값을 찾는 함수
IV. 이를 통해 해당 노드로 부터 전임자와 후임자를 찾을 수 있음

4.5. tree_delete / transplant

```
void tree_delete(Node *node, int data){
    Node *z = (Node *)malloc(sizeof(Node));
    z = tree_R_search(node, data);
    //z = tree_I_search(node, data);
    if(z->left == NULL)
        transplant(node, z, z->right);
    else if(z->right == NULL)
        transplant(node, z, z->left);
    else{
        Node *y = tree_minimum(z->right);
        if(y->parent != z){
            transplant(node, y, y->right);
            y->right = z->right;
            y->right->parent = y;
        }
        transplant(node, z, y);
        y->left = z->left;
        y->left->parent = y;
    }
}

void transplant(Node *node, Node *u, Node *v){
    //if u == root
    if(u->parent == NULL)
        root = v;
    else if (u == u->parent->left)
        u->parent->left = v;
    else
        u->parent->right = v;
    if(v != NULL)
        v->parent = u->parent;
}
```

I. 주요 개념 삭제 부분을 참고하면 case 1, 2, 3이 있는데 그에 따라 if문으로 구분하여 삭제가 진행 (z = 삭제할 노드)
II. z->left == NULL이면 z->right를 z의 위치에 이식(z->right가 NULL이어도 NULL을 부모의 부모에게 연결
III. 마찬가지로 z->right가 NULL이면 z->left를 z의 위치에 이식
IV. 만약 자식이 둘 다 있으면 오른쪽 서브트리의 최소값(Successor)를 y로 지정하여 z에 이식하고 y의 자식을 z의 자식을 해당 자리에 연결

5. 과제 4-2 Binary Search Tree - Median Insert

*** 기본적인 삽입, 삭제, 탐색의 구조는 같다.**

5.1. Quick Sort를 통해 1차적인 정렬

```
    for(i = 0; i < (test_num * 10); i++){
        int x;
        scanf("%d", &x);
        if(mode == 1){
            if(i == 0){
                root = tree_insert(root, x);
            }
            else{
                tree_insert(root, x);
            }
        }
        else if(mode == 2 || mode == 3){
            arr[i] = x;
        }
        else{
            printf("[!] Mode Error!");
            exit(1);
        }
    }

    /* Median Insert */
    if(mode == 2 || mode == 3){
        start = clock();
        QuickSort(arr, 0, (test_num * 10) - 1);
        if(mode == 2)
            median_insert(arr);
        else if (mode == 3)
            balance_median_insert(arr, 0, (test_num * 10) - 1);
    }
```

I. 우선 배열에 원소들을 저장
II. 배열을 이전 과제 중 QuickSort를 통해 정렬 (- 함수 설명 생략)
III. 정렬된 배열을 통해 median insert를 실시

* 복잡도 비교를 위해 과제에 설명된 방식(median_insert)과 높이를 최소로 맞추기 위한 방식(balance_median_insert)으로 구분

5.2. median_insert

```
void median_insert(int arr[]){
    int i, j;
    int size = test_num * 10;
    for(i = (size - 1); i > 0; i--){
        int median = i / 2;
        if(root == NULL){
            root = tree_insert(root, arr[median]);
        }
        else
            tree_insert(root, arr[median]);
        for(j = median; j <= i; j++){
            int temp = arr[median];
            if( j == i)
                arr[j] = temp;
            else
                arr[j] = arr[j + 1];
        }
    }
}
```

I. 정렬된 배열의 중간 값들을 차례대로 입력하기 위하여 i가 배열의 마지막부터 줄어들며 median = i / 2로 정의
II. median을 트리에 삽입하고 해당 index를 배열의 마지막으로 보내고 median ~ i를 한 칸씩 왼쪽으로 shift하여 사용한 원소는 제외
III. 이 경우 한 쪽으로 치우치는 것을 방지할 수 있지만 가운데 원소 값을 기준으로 양쪽으로 치우치게 됨

5.3. balance_median_insert

```
void balance_median_insert(int arr[], int p, int r){
    if(p <= r){
        int median = (p + r) / 2;
        if(root == NULL)
            root = tree_insert(root, arr[median]);
        else
            tree_insert(root, arr[median]);
        balance_median_insert(arr, p, median - 1);
        balance_median_insert(arr, median + 1, r);
    }
}
```

I. 정렬된 배열의 가운데를 기점으로 재귀를 통해 왼쪽 서브트리와 오른쪽 서브트리의 중간값을 찾아 높이를 최소화하는 과정
II. Quick Sort와 유사한 방법으로 왼쪽과 오른쪽의 중간값들을 재귀 호출
III. 이 경우 어느 한 쪽으로도 치우치지 않는 balance를 갖출 수 있음

5.4. printInorder(root)

```
void printInorder(Node *node){
    if(node){
        printInorder(node->left);
        printf("%d\n", node->data);
        printInorder(node->right);
    }
}
```

I. 중위 순회를 위한 출력함수
II. 왼쪽 서브트리로 재귀 후 왼쪽에서 오른쪽으로 진행하며 출력
III. 이진 탐색 트리 구조에서는 중위 순회로 출력을 하면 데이터가 정렬된 상태로 출력됨을 확인할 수 있다.

6. 실행

6.1. BST-insert

```
~/Desktop/Algorithm/실 습 / [Al]실 습 5주 차
➜  [Al]실 습 5주 차  ./test
>> Input test number [1 : Data1.txt, 2: Data2.txt] : 1
>> Input insert mode [1 : insert, 2 : median inser, 3 : balanced median] : 1
Time : 0.000048
25
17 34
11 - 26 38
- 13 - - 32 - -
- 12 - - - 27 - - -
delete data : [exit : -1] -1

~/Desktop/Algorithm/실 습 / [Al]실 습 5주 차
➜  [Al]실 습 5주 차  cat inorderTree.txt
11
12
13
17
25
26
27
32
34
38
```

6.2. BST-median insert

```
~/Desktop/Algorithm/실 습 / [Al]실 습 5주 차
➜  [Al]실 습 5주 차  ./test
>> Input test number [1 : Data1.txt, 2: Data2.txt] : 1
>> Input insert mode [1 : insert, 2 : median inser, 3 : balanced median] : 2
Time : 0.000011
25
17 26
13 - - 27
12 - - - - 32
11 - - - - - - 34
- - - - - - - - - 38
delete data : [exit : -1] -1

~/Desktop/Algorithm/실 습 / [Al]실 습 5주 차
➜  [Al]실 습 5주 차  cat inorderTree.txt
11
12
13
17
25
26
27
32
34
38
```

6.3. BST-balance median insert

```
~/Desktop/Algorithm/실 습 /[Al]실 습 5주 차
➜  [Al]실 습 5주 차  ./test
>> Input test number [1 : Data1.txt, 2: Data2.txt] : 1
>> Input insert mode [1 : insert, 2 : median inser, 3 : balanced median] : 3
Time : 0.000010
25
12 32
11 13 26 34
- - - 17 - 27 - 38
delete data : [exit : -1] -1

~/Desktop/Algorithm/실 습 /[Al]실 습 5주 차
➜  [Al]실 습 5주 차  cat inorderTree.txt
11
12
13
17
25
26
27
32
34
38
```

6.4. BST-delete

```
~/Desktop/Algorithm/실 습 /[Al]실 습 5주 차
➜  [Al]실 습 5주 차  ./test
>> Input test number [1 : Data1.txt, 2: Data2.txt] : 1
>> Input insert mode [1 : insert, 2 : median inser, 3 : balanced median] : 1
Time : 0.000046
25
17 34
11 - 26 38
- 13 - - 32 - -
- 12 - - - 27 - - -
delete data : [exit : -1] 25
26
17 34
11 - 32 38
- 13 - 27 - - -
- 12 - - - - - - -
delete 25
delete data : [exit : -1] 17
26
11 34
- 13 32 38
- 12 - 27 - - -
delete 17
delete data : [exit : -1] -1

~/Desktop/Algorithm/실 습 /[Al]실 습 5주 차  18s
➜  [Al]실 습 5주 차  cat inorderTree.txt
11
12
13
26
27
32
34
38
```

7. 결과

7.1. 삽입 시 트리 구조 비교

```
~/Desktop/Algorithm/실 습 /[Al]실 습 5주 차
➜  [Al]실 습 5주 차  ./test
>> Input test number [1 : Data1.txt, 2: Data2.txt] : 2
>> Input insert mode [1 : insert, 2 : median inser, 3 : balanced median] : 1
Time : 0.000075
36
30 37
11 35 - 44
4 21 - - - 41 46
1 8 15 23 - - - 40 - - 49
- - - - - 17 - 29 - - - - - - - - -
- - - - - 16 - - - - - - - - - - - -
delete data : [exit : -1] -1

~/Desktop/Algorithm/실 습 /[Al]실 습 5주 차
➜  [Al]실 습 5주 차  ./test
>> Input test number [1 : Data1.txt, 2: Data2.txt] : 2
>> Input insert mode [1 : insert, 2 : median inser, 3 : balanced median] : 2
Time : 0.000029
29
23 30
21 - - 35
17 - - - - 36
16 - - - - - - 37
15 - - - - - - - - 40
11 - - - - - - - - - - 41
8 - - - - - - - - - - - - 44
4 - - - - - - - - - - - - - - 46
1 - - - - - - - - - - - - - - - - 49
delete data : [exit : -1] -1

~/Desktop/Algorithm/실 습 /[Al]실 습 5주 차
➜  [Al]실 습 5주 차  ./test
>> Input test number [1 : Data1.txt, 2: Data2.txt] : 2
>> Input insert mode [1 : insert, 2 : median inser, 3 : balanced median] : 3
Time : 0.000025
23
15 37
4 16 30 44
1 8 - 17 29 35 40 46
- - - 11 - - 21 - - - 36 - 41 - 49
delete data : [exit : -1] -1
```

- 동일한 파일(Data2.txt)로 삽입
- 삽입, 삭제 모두 해당 데이터의 자리를 찾기위한 탐색의 복잡도가 필요
- 탐색은 $O(h)$의 복잡도
- 이진 탐색 트리는 높이를 기준으로 복잡도의 차이가 커지기 때문에 balance를 맞추는게 중요
- 최악의 경우 한 쪽으로 치우치게 되어 $O(n)$의 복잡도
- 일반 삽입의 경우 height가 6, 중간 값 삽입은 9, 균형 중간값 삽입은 4