

20121277 김주호 HW5 보고서

이번 과제에서 구현한 ADT는 크게 3가지이다. Singly Linked List, Double Linked List, Circular Linked List 가 그것인데, 이에 대한 ADT를 먼저 설명하도록 한다.

- Singly Linked List

Node
<pre>typedef int element; typedef struct ListNode { element data; struct ListNode * link; }ListNode;</pre>

- ① `ListNode* insert_first(ListNode* head, element val)`
head쪽에 노드를 삽입한다.
- ② `ListNode* insert_after_pre(ListNode* head, ListNode* pre, element val)`
인자로 들어온 pre노드 뒤에 새로운 노드를 삽입한다.
- ③ `ListNode* delete_first(ListNode* head)`
head쪽에 있는 노드를 삭제한다.
- ④ `ListNode* delete_after_pre(ListNode* head, ListNode* pre)`
pre의 link가 가리키는 노드를 삭제한다.
- ⑤ `ListNode* insert_last(ListNode* head, element val)`
tail 쪽에 노드를 삽입한다.
- ⑥ `ListNode* delete_last(ListNode* head)`
tail 쪽 노드를 삭제한다.

- Circular Linked List

Node
<pre>typedef char element; typedef struct ListNode { element data; struct ListNode * link; }ListNode;</pre>

- ① `void print_list(ListNode* tail)`
연결리스트 내의 항목을 출력한다.
- ② `void print_first_node(ListNode* tail)`
첫 번째 노드의 data를 출력한다.
- ③ `ListNode* insert_first(ListNode* tail, element data)`

head 쪽에 노드를 삽입한다.

④ ListNode* insert_last(ListNode* tail, element data)

tail 쪽에 노드를 삽입한다.

⑤ ListNode* delete_first(ListNode* tail)

head 쪽에 노드를 삭제한다.

⑥ ListNode* delete_after_pre(ListNode* tail, ListNode* pre)

인자로 들어온 pre 노드의 link가 가리키는 노드를 삭제한다.

⑦ ListNode* delete_last(ListNode* tail)

tail쪽의 원소를 삭제한다.

- Double Linked List

```
Node
typedef struct {
    short int row;
    short int col;
    short int dir;
}element;

typedef struct DListNode {
    element data;
    struct DListNode * llink;
    struct DListNode * rlink;
}DListNode;
```

① void init(DListNode* phead)

이중 연결리스트를 초기화하는 함수이다. llink와 rlink가 모두 head 노드를 가리키게 한다.

② is_empty(DListNode* phead)

이중 연결리스트의 공백 여부를 알려주는 함수이다. phead->llink == phead && phead->rlink이면 해당 연결리스트는 공백이 된다.

③ void print_dlist_reverse(DListNode* phead)

이중 연결 리스트의 노드를 tail에서 시작하여 head를 끝으로 역순으로 출력해주는 함수이다.

④ void d_insert(DListNode* pre, element data)

인자로 들어온 pre노드의 rlink가 가리키는 곳에 새로운 데이터를 가지는 노드를 삽입하는 함수이다.

⑤ void d_delete(DListNode* head, DListNode* removed)

인자로 들어온 removed 노드를 삭제하는 함수이다.

1번.

- ADT

Max heap을 array가 아닌 Tree로 구현하였는데, 그에 따라 node가 value 값 뿐만 아니라, parent, leftChild, rightChild의 정보를 갖도록 구성하였다.

Node of Max Heap Tree

```
typedef struct node * treePointer;
typedef struct node {
    int key;
    struct node * parent;
    struct node * leftChild;
    struct node * rightChild;
} node;
```

원하는 노드에 접근하기 위해, queue 자료구조를 연결리스트로 표현하였다.

Node of Queue

```
typedef struct queue {
    struct queue * next;
    treePointer element;
} queue;
```

- treePointer create_a_node();

node 하나를 생성하는 함수이다. parameter로 들어온 int x를 node의 key값으로 주고, leftChild와 rightChild에는 NULL, parent에는 parameter로 들어온 parent를 할당하게 된다.

input : int x, treePointer parent

return : treePointer nd;

- queue* create_a_queue();

queue의 원소 하나를 생성하는 함수이다. queue의 원소는 treePointer element와 다음 노드를 가리키는 링크인 next로 이루어진 구조체이다.

input : node* element

return : queue* q

- void free_queue();

queue의 활용이 끝나면, 동적 메모리 할당을 해제해주기 위한 함수이다.

input : queue** q

return : 없음

- int search_key();

max heap tree가 원소 x를 저장하고 있는지 조회하는 함수이다. max heap tree 안에 있는 node들을 bfs 탐색하면서, 원소 x를 저장하고 있는지 살핀다.

input : treePointer* root, int x

return : 해당 원소가 max heap tree 안에 있으면 1, 없으면 0

- void insert_a_node_into_Q();

leftChild 노드와 rightChild 노드를 Q 자료구조에 삽입하는 함수이다.

input : treePointer n, queue** last

return : 없음

- void insertion();

parameter로 들어오는 max heap tree에 원소 x를 삽입하는 함수이다. max heap tree의 끝에 저장된 node에 접근하여 삽입한 뒤, heapify를 진행하여, max heap의 구조를 유지한다.

input : treePointer* root, int x

return : 없음

- void deletion();

max heap tree에서 최대원소(root에 저장되어 있음)를 제거하는 함수이다. 가장 끝에 저장되어있는 노드에 접근하여 이를 root에 할당한 뒤, heapify를 진행하여 max heap의 구조를 유지한다.

input : treePointer* root

return : 없음

- 함수 설명

- void print_heap();

과제의 해결에 직접 사용하지 않지만, max heap에 원소들이 추가되고 삭제됨에 따라 max heap tree에 저장된 정보들을 살펴볼 수 있는 함수이다. max heap의 정보를 출력할 때, bfs의 방식으로 노드를 방문하게 된다.

input : treePointer* root

return : 없음

- int str_split_to_int_arr();

사용자가 node의 원소 값을 입력할 때, 각 원소를 띄어쓰기로 구분하여 입력하게 된다. 이 정보를 띄어쓰기 기준하여 parsing하여 배열에 저장하는 함수이다.

input : int arr[], int max_arr_size, char* str

return : 배열에 저장된 원소의 개수

- 코드

```
#include <stdio.h >
#include <stdlib.h >
#include <time.h >

typedef struct node * treePointer;
typedef struct node {
    int key;
    struct node * parent;
    struct node * leftChild;
    struct node * rightChild;
} node;
// 연결 리스트로 queue를 표현
typedef struct queue {
    struct queue * next;
    treePointer element;
} queue;

treePointer create_a_node(int x, treePointer parent);
queue * create_a_queue(node * element);
void free_queue(queue ** q);
int search_key(treePointer * root, int x);
void insert_a_node_into_Q(treePointer n, queue ** last);
void insertion(treePointer * root, int x);
void deletion(treePointer * root);
void print_heap(treePointer * root);
int str_split_to_int_arr(int arr[], int max_arr_size, char * str);

// node 하나를 생성
treePointer create_a_node(int x, treePointer parent) {
    treePointer nd = (treePointer)malloc(sizeof(node));
    if (nd == NULL) {
        printf("동적할당 err!");
        exit(1);
    }
    nd->key = x;
    nd->leftChild = NULL;
    nd->rightChild = NULL;
    nd->parent = parent;
    return nd;
}
// queue 원소(treePointer와 링크) 하나를 생성
queue * create_a_queue(node * element) {
    queue * q = (queue *)malloc(sizeof(queue));
```

```

    if (q == NULL) {
        printf("동적할당 err!");
        exit(1);
    }
    q->next = NULL;
    q->element = element;
    return q;
}
// free queue
void free_queue(queue ** q) {
    queue * current_q = *q;
    while (current_q != NULL) {
        *q = current_q ->next;
        free(current_q);
        current_q = *q;
    }
}
int search_key(treePointer * root, int x) {
    if (*root != NULL) {
        queue * q = create_a_queue(*root);
        queue * current_q = q;
        queue * last_q = q;
        treePointer current_node;
        while (current_q != NULL) {
            current_node = current_q ->element;
            insert_a_node_into_Q(current_node, &last_q);
            // printf("%d ", current_node->key);
            if (current_node ->key == x) {
                return 1;
            }
            current_q = current_q ->next;
        }
        free_queue(&q);
    }
    return 0;
}
// 큐에 insert
void insert_a_node_into_Q(treePointer n, queue ** last) {
    if (n ->leftChild != NULL) {
        (*last)->next = create_a_queue(n ->leftChild);
        (*last) = (*last)->next;
    }
    if (n ->rightChild != NULL) {
        (*last)->next = create_a_queue(n ->rightChild);
        (*last) = (*last)->next;
    }
}
// 힙에 insert
void insertion(treePointer * root, int x) {
    if (*root == NULL) { // 힙이 비어있다면,
        *root = create_a_node(x, NULL);
    }
    else { // 힙이 비어있지 않다면, node가 들어갈 적절한 위치를 탐색한 뒤 넣는다.
        queue * q = create_a_queue(*root);

```

```

    queue * current_q = q;
    queue * last_q = q;
    treePointer current_node;
    if (search_key(root, x)) {
        printf("Exist Number\n");
        return;
    }
    while (current_q !=NULL) {
        // Q에서 노드하나 꺼냄.
        current_node = current_q ->element;
        if (current_node ->leftChild ==NULL) {
            current_node->leftChild = create_a_node(x, current_node);
            current_node = current_node ->leftChild;
        }
        else if (current_node ->rightChild ==NULL) {
            current_node->rightChild = create_a_node(x, current_node);
            current_node = current_node ->rightChild;
        }
        else {
            insert_a_node_into_Q(current_node, &last_q);
            current_q = current_q ->next;
            continue;
        }
        // heap tree를 heapify한다.
        while (current_node ->parent !=NULL && current_node ->parent ->key
< current_node ->key) {
            // swap the value
            int temp = current_node ->parent ->key;
            current_node->parent ->key = current_node ->key;
            current_node->key = temp;
            current_node = current_node ->parent;
        }
        break;
    }
    free_queue(&q);
}
printf("Insert %d\n", x);
return;
}

void deletion(treePointer * root) {
    int tmp;
    if (*root !=NULL) {
        tmp = (*root)->key;
        queue * q = create_a_queue(*root);
        queue * current_q = q;
        queue * last_q = q;
        queue * previous_q = (queue *)malloc(sizeof(queue));
        treePointer current_node;
        if (previous_q ==NULL) {
            printf("동적할당 err!");
            exit(1);
        }
        // 마지막 노드를 찾는다.
        while (current_q !=NULL) {

```

```

current_node = current_q ->element;
insert_a_node_into_Q(current_node, &last_q);
previous_q = current_q;
current_q = current_q ->next;
}
current_node = previous_q ->element;
free_queue(&q);
// 가장 끝 노드를 free 하고, 이 노드 정보를 root로 옮긴다.
if (current_node ->parent ==NULL) {
    free(current_node);
    *root =NULL;
}
else {
    (*root)->key = current_node ->key;
    current_node = current_node ->parent;
    if (current_node ->rightChild !=NULL) {
        free(current_node ->rightChild);
        current_node->rightChild =NULL;
    }
    else {
        free(current_node ->leftChild);
        current_node->leftChild =NULL;
    }
}

// 끝 노드를 무작정 root에 옮겼으니, heapify를 진행한다.
int cur_key, l_key, r_key;
current_node =*root;
while (1) {
    if (current_node ->leftChild ==NULL) {
        // (current->right == NULL) 가 true임을 함의한다.
        break;
    }
    else if (current_node ->rightChild ==NULL) {
        cur_key = current_node ->key;
        l_key = current_node ->leftChild ->key;
        if (cur_key < l_key) {
            current_node->key = l_key;
            current_node->leftChild ->key = cur_key;
            current_node = current_node ->leftChild;
        }
        else {
            break;
        }
    }
    else {
        cur_key = current_node ->key;
        l_key = current_node ->leftChild ->key;
        r_key = current_node ->rightChild ->key;
        //cur_key와 l_key, r_key를 비교
        if (cur_key >= l_key && cur_key >= r_key) { // cur_key가 가장
크다면,
            break;
        }
        else if (l_key > cur_key && l_key >= r_key) { // l_key가 가장

```


크다면,

```
        current_node->leftChild ->key = cur_key;
        current_node->key = l_key;
        current_node = current_node ->leftChild;
    }
    else { // r_key가 가장 크다면,
        current_node->rightChild ->key = cur_key;
        current_node->key = r_key;
        current_node = current_node ->rightChild;
    }
}
}
}
printf("Delete %dWn", tmp);
}
else {
    printf("The heap is emptyWn");
    return;
}
}
void print_heap(treePointer * root) {
    if (*root !=NULL) {
        queue * q = create_a_queue(*root);
        queue * current_q = q;
        queue * last_q = q;
        treePointer test;
        while (current_q !=NULL) {
            test = current_q ->element;
            insert_a_node_into_Q(test, &last_q);
            printf("%d ", test ->key);
            current_q = current_q ->next;
        }
        free_queue(&q);
        printf("Wn");
    }
}
int str_split_to_int_arr(int arr[], int max_arr_size, char * str) {
    /*공백을 포함한 str을 입력받아 공백 기준 split하여 arr에 저장하는 함수이다.*/
    int * start = arr;
    int num =0;
    // 숫자 분리 상태를 기억 (0이면 진행 안됨, -1이면 음수 진행, 1이면 양수 진행)
    int state =0;
    while (*str == ' ') str ++;; // 앞쪽 공백 제거
    for (; *str; str ++) {
        if (*str != ' ') {
            if (state ==0) { // 숫자 분리가 진행되지 않았다면,
                if (*str == '-') {
                    state =-1; // 음수라는 표시,
                    str++;
                }
                else state =1; // 양수라는 표시
            }
            // 문자를 숫자로 변경
            num = num *10 +*str -'0';
        }
    }
}
```

```

    }
    else {
        if ((arr - start) < max_arr_size) {
            *arr += num * state;
            num = 0;
            state = 0;
            // 공백 문자 제거
            while (*(str + 1) == ' ') str ++;
        }
        else return arr - start;
    }
}
// 숫자를 분리중에 반복문이 중단되었으면 해당 숫자를 arr에 추가한다.
if (state)*arr += num * state;;
return arr - start;
}

int main() {
    treePointer root = NULL;
    int for_getchar;
    char command;
    int num;
    char str[10];
    int arr[10];
    while (1) {
        printf("명령을 입력하세요. (삽입 : i [삽입을 원하는 정수], 삭제 : d, 종료 q)");
        if (scanf("%[^Wn]s", str) == -1) {
            return 0;
        }
        str_split_to_int_arr(arr, 2, str);
        command = (char)(arr[0] + '0');
        num = arr[1];
        switch (command) {
            case 'i':
                insertion(&root, num);
                break;
            case 'd':
                deletion(&root);
                break;
            case 'q':
                exit(1);
                break;
            default:
                printf("잘못된 명령을 입력하였습니다. (i - insert, d - delete, q - 종");
        }
        //print_heap(&root);
        for_getchar = getchar();
    }
    return 0;
}

```

- 실행 결과

```
Microsoft Visual Studio 디버그 콘솔
명령을 입력하세요. (삽입 : i [삽입을 원하는 정수], 삭제 : d, 종료 q) i 3
Insert 3
명령을 입력하세요. (삽입 : i [삽입을 원하는 정수], 삭제 : d, 종료 q) i 2
Insert 2
명령을 입력하세요. (삽입 : i [삽입을 원하는 정수], 삭제 : d, 종료 q) i 2
Exist Number
명령을 입력하세요. (삽입 : i [삽입을 원하는 정수], 삭제 : d, 종료 q) i 1
Insert 1
명령을 입력하세요. (삽입 : i [삽입을 원하는 정수], 삭제 : d, 종료 q) d
Delete 3
명령을 입력하세요. (삽입 : i [삽입을 원하는 정수], 삭제 : d, 종료 q) d
Delete 2
명령을 입력하세요. (삽입 : i [삽입을 원하는 정수], 삭제 : d, 종료 q) d
Delete 1
명령을 입력하세요. (삽입 : i [삽입을 원하는 정수], 삭제 : d, 종료 q) d
The heap is empty
명령을 입력하세요. (삽입 : i [삽입을 원하는 정수], 삭제 : d, 종료 q) d
The heap is empty
명령을 입력하세요. (삽입 : i [삽입을 원하는 정수], 삭제 : d, 종료 q) et
잘못된 명령을 입력하였습니다. (i - insert, d - delete, q - 종료)

명령을 입력하세요. (삽입 : i [삽입을 원하는 정수], 삭제 : d, 종료 q) q

C:\Users\juho3\Desktop\hw5\Debug\hw5.exe (프로세스 6560개)이(가) 종료되었습니다(코드: 12)
이 창을 닫으려면 아무 키나 누르세요...
```

2번.

- ADT

Node
<pre>typedef int element; typedef struct TreeNode { element data; struct TreeNode * left, * right; }TreeNode;</pre>

- 함수 설명

- `TreeNode* search()`
parameter로 들어오는 key 값을 해당 이진 탐색트리가 저장하고 있는지 탐색하는 함수이다.
input : `TreeNode * node, int key`
return : key가 이진 탐색 트리에 있다면 해당 node를 반환, 없다면 NULL을 반환
- `TreeNode* new_node()`
parameter로 들어오는 item 값을 data로 하는 node 하나를 생성하는 함수이다.
input : `int item`

return : 생성된 node

- `TreeNode* insert_node()`

parameter로 들어오는 key값을 해당 이진 탐색 트리에 삽입하는 함수이다. 이진 탐색 트리의 구조를 훼손하지 않게 적절한 위치를 찾아 해당 key 값을 할당하게 된다. 이 때, recursive call을 활용하게 된다.

input : `TreeNode * root`, `int key`

return : `TreeNode*` 이진 탐색 트리의 root

- `int sum_nodes_having_0_or_1_child()`

0 또는 1개의 child node를 가지는 node들의 data를 temp 변수에 누적하여 쌓아 올린다. 해당 함수를 한번 실행할 때, `root->left`와 `root->right`를 재귀로 call 하게 되어, 모든 노드를 방문하게 된다.

input : `TreeNode* root`

return : 0개 또는 1개의 child node를 가지는 node 들의 data 합.

- 코드

```
#include <stdio.h >
#include <stdlib.h >

typedef int element;
typedef struct TreeNode {
    element data;
    struct TreeNode * left, * right;
}TreeNode;

// 순환적인 탐색 함수
TreeNode* search(TreeNode * node, int key) {
    if (node ==NULL) return NULL;
    if (key == node ->data) return node;
    else if (key < node ->data) {
        return search(node ->left, key);
    }
    else {
        return search(node ->right, key);
    }
}

TreeNode* new_node(int item) {
    TreeNode* temp = (TreeNode *)malloc(sizeof(TreeNode));
    if (temp ==NULL) {
        printf("동적 할당 err!");
        exit(1);
    }
    temp->data = item;
    temp->left = temp ->right =NULL;
    return temp;
}

TreeNode* insert_node(TreeNode * root, int key) {
    // 트리가 공백이면 새로운 노드를 반환한다.
```

```

    if (root ==NULL) return new_node(key);
    // 그렇지 않으면, 순환적으로 트리를 내려간다.
    if (key < root ->data) {
        root->left = insert_node(root ->left, key);
    }
    else if (key > root ->data) {
        root->right = insert_node(root ->right, key);
    }
    // 변경된 루트 포인터를 반환한다.
    return root;
}

int sum_nodes_having_0_or_1_child(TreeNode * root)
{
    int temp =0;
    if (root ==NULL)
        return 0;
    //만약 노드가 없으면 0을 반환한다
    else
    {
        if ((root ->left !=NULL && root ->right ==NULL)
            || (root ->left ==NULL && root ->right !=NULL)
            || (root ->left ==NULL && root ->right ==NULL))
            temp += root ->data;
        //만약 노드의 자식이 1. 오른쪽만 있거나 2. 왼쪽만 있거나 3. 없으면
        temp에 값을 쌓는다.
    }
    return temp + sum_nodes_having_0_or_1_child(root ->left) +
sum_nodes_having_0_or_1_child(root ->right);
}

void main() {
    TreeNode* node =NULL;
    FILE* fp;
    int size;
    int data;
    fp = fopen("input.txt", "rt");
    if (fp ==NULL) {
        printf("파일 읽기 err!");
        exit(1);
    }
    if (fscanf(fp, "%d", &size) ==-1) {
        printf("파일 입력 err!");
        exit(1);
    }
    for (int i =0; i <size; i ++) {
        if (fscanf(fp, "%d", &data) ==-1) {
            printf("파일 입력 err!");
            exit(1);
        }
        if (search(node, data) !=NULL) {
            printf("동일한 원소가 이미 이진 탐색 트리에 있습니다.");
            exit(1);
        }
        node = insert_node(node, data);
    }
}

```

```

    }
    printf("자식 노드의 개수가 하나 or 0개인 노드 data합 : %d\n",
sum_nodes_having_0_or_1_child(node));
    fclose(fp);
    return;
}

```

- 실행 결과

Microsoft Visual Studio 디버그 콘솔

자식 노드의 개수가 하나 or 0개인 노드 data합 : 38

C:\Users\juho3\Desktop\hw5\Debug\hw5.exe (프로세스 20156개)이(가) 종료되었습니다. 이 창을 닫으려면 아무 키나 누르세요...

input - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

5
15
23
2
8
5

Microsoft Visual Studio 디버그 콘솔

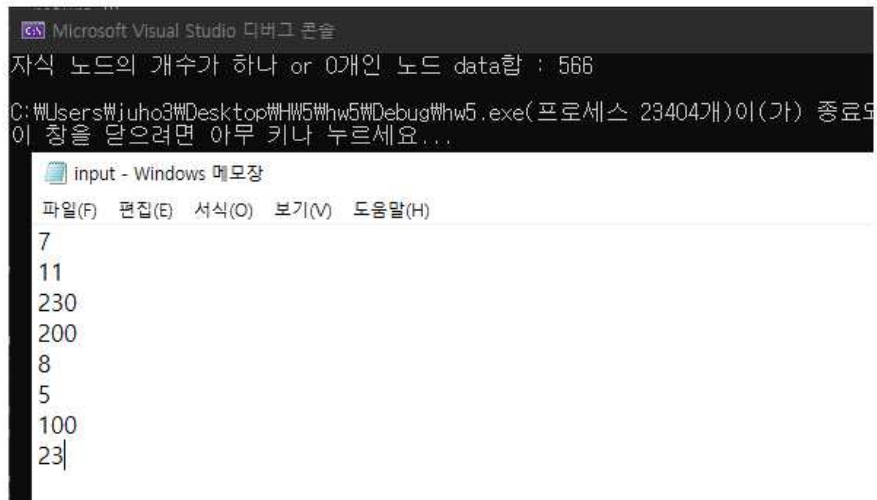
동일한 원소가 이미 이전 탐색 트리에 있습니다.

C:\Users\juho3\Desktop\hw5\Debug\hw5.exe (프로세스 28300개)이(가) 종료되었습니다. 이 창을 닫으려면 아무 키나 누르세요...

input - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

7
11
23
200
8
5
100
23



3번

- ADT

2번과 동일하다.

Node
<pre>typedef int element; typedef struct TreeNode { element data; struct TreeNode * left, * right; }TreeNode;</pre>

과제 해결에 사용하지는 않지만, 이진 탐색 트리의 기본 연산으로서 삭제 함수를 추가로 정리하였다.

- `TreeNode* min_value_node()`

아래 서술할 `delete_node()` 함수에서 활용되는 함수이다. 이진 탐색 트리에서 자식 노드를 왼쪽과 오른쪽 2개 모두 가지고 있는 노드가 있다. 이러한 노드를 삭제할 때는, 삭제된 자리에 이를 대신할 적절한 노드를 옮겨야 하는데, 이때 이진 탐색 트리의 구조가 훼손되지 않아야 한다. 삭제된 노드의 오른쪽 트리에서 가장 작은 원소를 택하여, 삭제된 자리에 할당하면, 이진 탐색 트리의 구조가 유지된다. 이진 탐색 트리에서 특정 노드를 기준으로 left 링크만을 타고 내려가 가장 작은 원소를 가지고 있는 node를 얻어내는 함수이다.

input : `TreeNode * node`

return : 이진 탐색 subtree에서 가장 작은 원소를 가지고 있는 node

- `TreeNode* delete_node()`

parameter로 들어오는 key 값을 가진 노드를 삭제하는 함수이다. recursive call을 이용하여 해당 key를 가진 노드를 찾아내고, 이를 삭제한다. 삭제하려는 node가 자

식노드를 가지지 않는 경우, 하나의 서브트리만 갖는 경우, 2개의 서브트리를 가지고 있는 세가지 경우에 따라 이진 탐색 트리의 구조를 유지하기 위한 다른 작업을 해준다.

input : `TreeNode * root`, `int key`

return : 새로운 루트노드

```
TreeNode* min_value_node(TreeNode * node) {
    TreeNode* current = node;
    // 맨 왼쪽 단말 노드를 찾으러 내려감
    while (current ->left !=NULL) {
        current = current ->left;
    }
    return current;
}

// 이진 탐색 트리와 키가 주어지면 키가 저장된 노트를 삭제하고,
// 새로운 루트 노드를 반환한다.
TreeNode* delete_node(TreeNode * root, int key) {
    if (root ==NULL) {
        return root;
    }
    // 만약 키가 루트보다 작으면 왼쪽 서브 트리에 있는 것임
    if (key < root ->data) {
        root ->left = delete_node(root ->left, key);
    }
    // 만약 키가 루트보다 크면 오른쪽 서브 트리에 있는 것임
    else if (key > root ->data) {
        root ->right = delete_node(root ->right, key);
    }
    else {
        // 첫번째나 두번째 경우
        if (root ->left ==NULL) {
            TreeNode* temp = root ->right;
            free(root);
            return temp;
        }
        else if (root ->right ==NULL) {
            TreeNode* temp = root ->left;
            free(root);
            return temp;
        }
        // 세 번째 경우
        TreeNode* temp = min_value_node(root ->right);
        // 중위 순회시 후계노드를 복사한다.
        root ->data = temp ->data;
        root ->right = delete_node(root ->right, temp ->data);
    }
    return root;
}
```

- 함수 구현

- void inorder()

이진 탐색 트리를 중위 순회하는 함수이다.

input : TreeNode * root

return : 없음

- void postorder()

이진 탐색 트리를 후위 순회하는 함수이다.

input : TreeNode * root

return : 없음

- 코드

```
#include <stdio.h >
#include <stdlib.h >
#define MAX_SIZE 500
typedef int element;
typedef struct TreeNode {
    element data;
    struct TreeNode * left, * right;
}TreeNode;
// 순환적인 탐색 함수
TreeNode* search(TreeNode * node, int key) {
    if (node ==NULL) return NULL;
    if (key == node ->data) return node;
    else if (key < node ->data) {
        return search(node ->left, key);
    }
    else {
        return search(node ->right, key);
    }
}

TreeNode* new_node(int item) {
    TreeNode* temp = (TreeNode *)malloc(sizeof(TreeNode));
    if (temp ==NULL) {
        printf("동적 할당 err!");
        exit(1);
    }
    temp->data = item;
    temp->left = temp ->right =NULL;
    return temp;
}

TreeNode* insert_node(TreeNode * root, int key) {
    // 트리가 공백이면 새로운 노드를 반환한다.
    if (root ==NULL) return new_node(key);
    // 그렇지 않으면, 순환적으로 트리를 내려간다.
    if (key < root ->data) {
        root->left = insert_node(root ->left, key);
    }
}
```

```

        else if (key > root ->data) {
            root->right = insert_node(root ->right, key);
        }
        // 변경된 루트 포인터를 반환한다.
        return root;
    }
}
void inorder(TreeNode * root)
{
    if (root !=NULL)
    {
        inorder(root->left);
        printf("%d ", root ->data);
        inorder(root->right);
    }
}
void postorder(TreeNode * root)
{
    if (root !=NULL)
    {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root ->data);
    }
}
}
int str_split_to_int_arr(int arr[], int max_arr_size, char * str) {
    /*공백을 포함한 str을 입력받아 공백 기준 split하여 arr에 저장하는 함수이다.*/
    int * start = arr;
    int num =0;
    // 숫자 분리 상태를 기억 (0이면 진행 안됨, -1이면 음수 진행, 1이면 양수 진행)
    int state =0;
    while (*str == ' ') str ++; // 앞쪽 공백 제거
    for (; *str; str ++) {
        if (*str != ' ') {
            if (state ==0) { // 숫자 분리가 진행되지 않았다면,
                if (*str == '-') {
                    state =-1; // 음수라는 표시,
                    str++;
                }
                else state =1; // 양수라는 표시
            }
            // 문자를 숫자로 변경
            num = num *10 +*str -'0';
        }
        else {
            if ((arr - start) < max_arr_size) {
                *arr += num * state;
                num =0;
                state =0;
                // 공백 문자 제거
                while (*(str +1) == ' ') str ++;
            }
            else return arr - start;
        }
    }
}

```

```

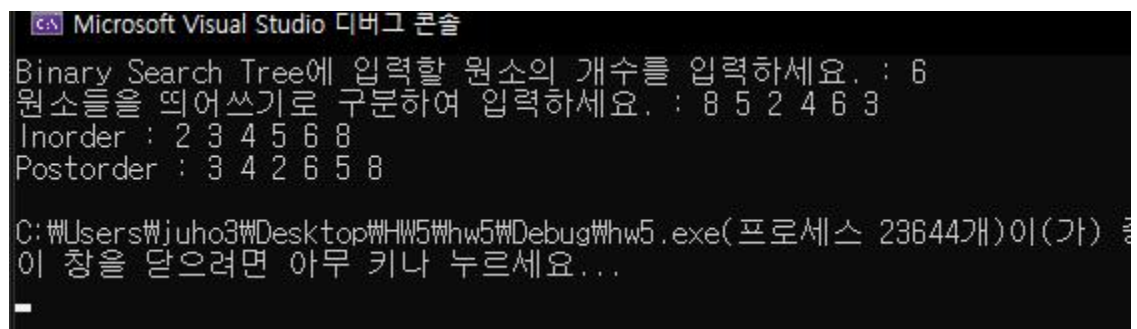
    }
}
// 숫자를 분리중에 반복문이 중단되었으면 해당 숫자를 arr에 추가한다.
if (state)*arr += num * state;;
return arr - start;
}

void main() {
    TreeNode* node =NULL;
    int for_getchar;
    int size;
    char string[MAX_SIZE];
    int arr[MAX_SIZE];
    printf("Binary Search Tree에 입력할 원소의 개수를 입력하세요. : ");
    if (scanf("%d", &size) ==-1) {
        printf("입력 err!");
        exit(1);
    }
    for_getchar = getchar();
    printf("원소들을 띄어쓰기로 구분하여 입력하세요. : ");
    if (scanf("%[^\\n]s", string) ==-1) {
        printf("입력 err!");
        exit(1);
    }
    str_split_to_int_arr(arr, MAX_SIZE, string);
    for (int i =0; i <size; i ++) {
        if (search(node, arr[i]) !=NULL) {
            printf("cannot construct BST");
            exit(1);
        }
        node = insert_node(node, arr[i]);
    }

    printf("Inorder : "); inorder(node); printf("\\n");
    printf("Postorder : "); postorder(node); printf("\\n");
    return;
}

```

- 실행 결과



```

Microsoft Visual Studio 디버그 콘솔
Binary Search Tree에 입력할 원소의 개수를 입력하세요. : 6
원소들을 띄어쓰기로 구분하여 입력하세요. : 8 5 2 4 6 3
Inorder : 2 3 4 5 6 8
Postorder : 3 4 2 6 5 8

C:\Users\juho3\Desktop\hw5\Debug\hw5.exe(프로세스 23644개)이(가) 종료되었습니다.
이 창을 닫으려면 아무 키나 누르세요...

```

Microsoft Visual Studio 디버그 콘솔

Binary Search Tree에 입력할 원소의 개수를 입력하세요. : 5

원소들을 띄어쓰기로 구분하여 입력하세요. : -1 0 300 2 5

Inorder : -1 0 2 5 300

Postorder : 5 2 300 0 -1

C:\Users\juho3\Desktop\HW5\Debug\hw5.exe(프로세스 27712개)이(가) 종료
이 창을 닫으려면 아무 키나 누르세요...

Microsoft Visual Studio 디버그 콘솔

Binary Search Tree에 입력할 원소의 개수를 입력하세요. : 3

원소들을 띄어쓰기로 구분하여 입력하세요. : 1 1 2

cannot construct BST

C:\Users\juho3\Desktop\HW5\Debug\hw5.exe(프로세스 26760개)이(가) 종료
이 창을 닫으려면 아무 키나 누르세요...