BinaryTree 자료 구조
구현 및 분석

작성자 : Lee Daero(skseofh@naver.com)

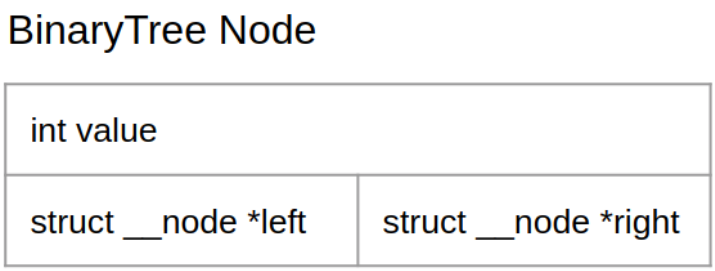
<Binary Tree 구현 및 분석>

[목적]
- BinaryTree 구조를 그림을 통해 분석한다.

[Binary Tree]
- 하나의 노드에 오른쪽, 왼쪽 2개의 자식만 가지는 tree 노드
- 구현에서는 data의 값에 따라 현재 노드의 data 보다 작은 경우는 왼쪽 자식 노드에, 큰 경우에는 오른쪽 자식 노드에 위치하도록 하였다.

(1) node 구조

```
typedef struct __node
{
    int value;
    struct __node * left;
    struct __node * right;

}node;
```



- value : data를 담는 int형 변수
- left : 왼쪽 자식 노드를 가리키는 struct __node *형 left 포인터
- right : 오른쪽 자식 노드를 가리키는 struct __node * 형 right 포인터

(2) 구현 함수

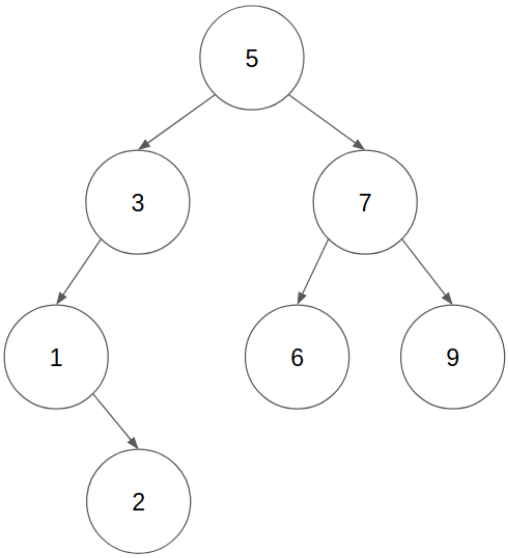
```
node * get_node(void);
void insert_tree(node ** top, int value);
int delete_tree(node ** top, int value);
void find_max(node ** top, int * max);
node * change_node(node ** top);

void print_preorder(node * top);
void print_inorder(node * top);
void print_postorder(node * top);
```

- get_node : 동적 할당으로 tree 구조체를 할당 받아서 포인터를 return
- insert_tree : value값에 의해 삽입 위치를 찾아서 tree 구조에 삽입
- delete_tree : value값에 해당하는 노드를 삭제
- find_max : delete_tree 함수에서 삭제할 노드를 대체할 노드를 찾는 과정, 삭제할 노드 왼쪽 노드의 자식 노드 중에서 가장 큰 value를 가지는 노드를 찾기 위한 함수.
- change_node : delete_tree 함수에서 삭제할 노드를 자식 노드 중 하나로 대체하는 함수.

- print_preorder, print_inorder, print_postorder : 전위, 중위, 후위 순으로 value값 출력

(3) main() 함수 동작 설명

```c
int main(void)
{
    int arr[7] = {5, 3, 7, 1, 2, 9, 6};
    int i;
    node * top = NULL;

    for(i=0; i<7; i++)
        insert_tree(&top, arr[i]);

    print_preorder(top);
    printf("\n");

    print_inorder(top);
    printf("\n");

    print_postorder(top);
    printf("\n");

    //왼쪽 자식만 있는 노드 삭제
    delete_tree(&top, 3);

    print_preorder(top);
    printf("\n");

    //오른쪽 자식만 있는 노드 삭제
    delete_tree(&top, 1);

    print_preorder(top);
    printf("\n");

    //왼쪽, 오른쪽 자식 모두 있는 노드 삭제
    delete_tree(&top, 7);

    print_preorder(top);
    printf("\n");

    //자식이 없는 노드 삭제
    delete_tree(&top, 9);

    print_preorder(top);
    printf("\n");

    return 0;
}
```



- arr[7]에 tree에 삽입할 원소들을 저장
- node * top으로 tree 구조를 가리키는 포인터 top 생성 후 NULL로 초기화
- insert_tree로 arr[]의 원소 삽입
- print_preorder, print_inorder, print_postorder로 전위, 중위, 후위 출력
- delete_tree 함수를 검증하기 위해, 4가지 경우로 나누어 삭제 검사, 검사 후 출

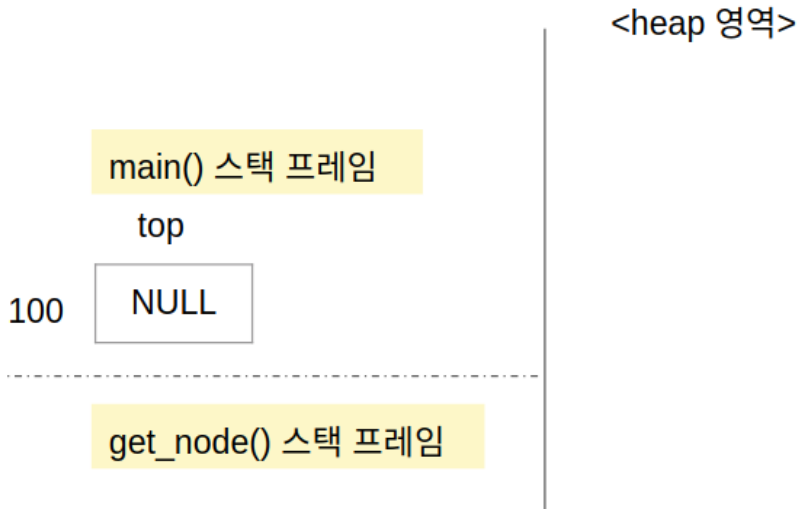
```
5 3 1 2 7 6 9
1 2 3 5 6 7 9
2 1 3 6 9 7 5
delete value = 3
5 1 2 7 6 9
delete value = 1
5 2 7 6 9
delete value = 7
5 2 6 9
delete value = 9
5 2 6
```

(4) 함수 분석

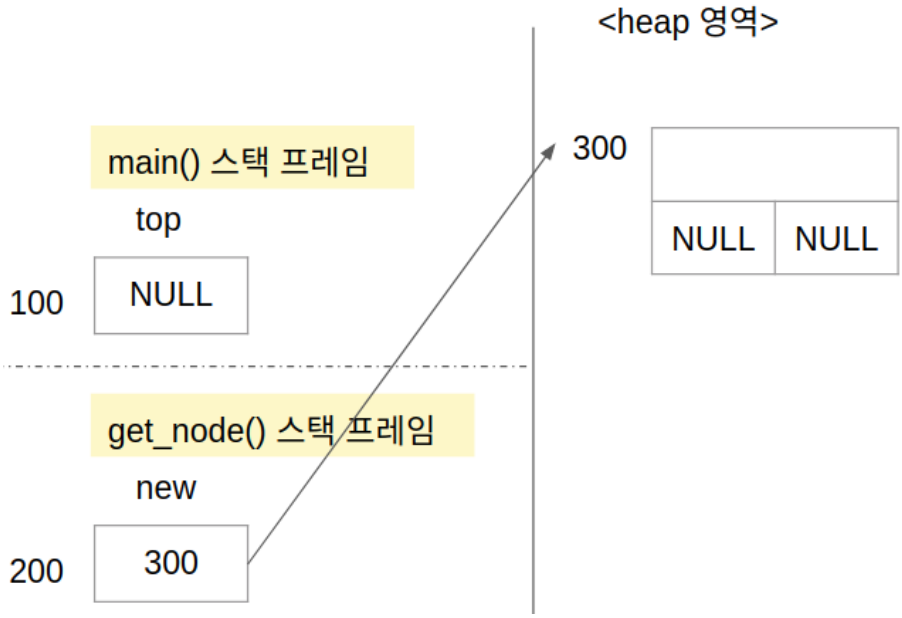
1) get_node : 동적 할당으로 tree 구조체를 할당 받아서 포인터를 return

```
node * get_node(void)
{
    node * new = (node *)malloc(sizeof(node));
    new->left = NULL;
    new->right = NULL;

    return new;
}
```

- main()에서 node * top = NULL 이후 get_node() 함수로 넘어왔을 때



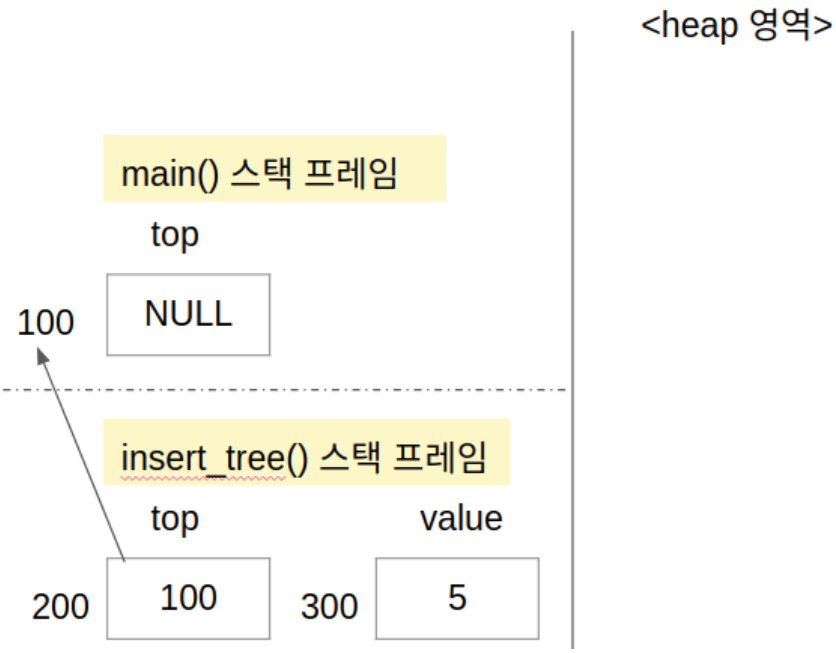
- node * new = (node *)malloc(sizeof(node));
: 동적할당으로 node 크기의 구조체 할당
- new→left = NULL;
- new→right = NULL;
: 새로 생성된 노드의 왼쪽, 오른쪽 자식 노드는 NULL로 초기화



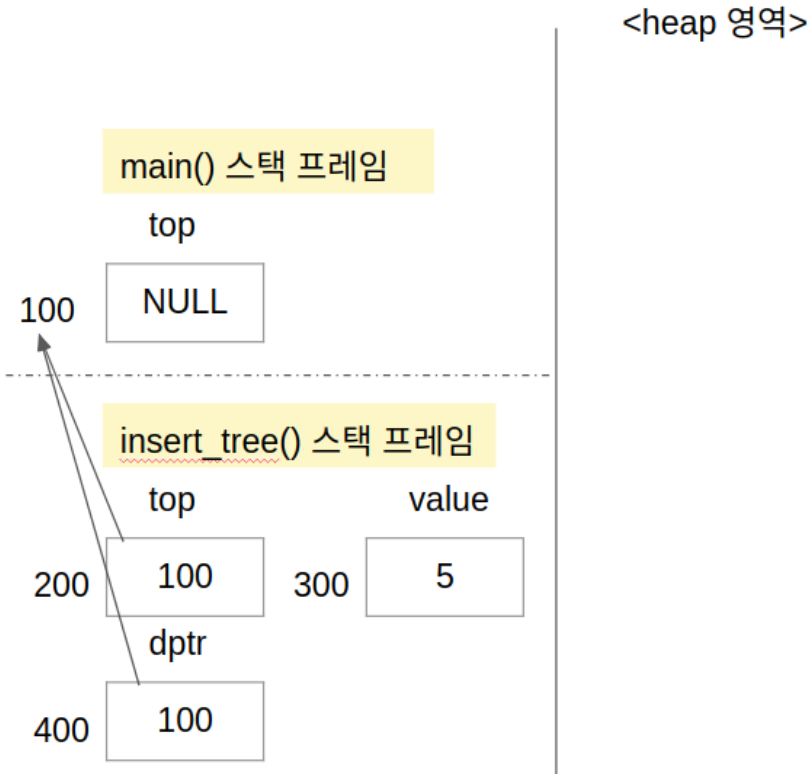
2) insert_tree : value값에 의해 삽입 위치를 찾아서 tree 구조에 삽입

```
void insert_tree(node ** top, int value)
{
    node ** dptr = top;

    while(*dptr)
    {
        if(value < (*dptr)->value)
        {
            dptr = &((*dptr)->left);
        }
        else
        {
            dptr = &((*dptr)->right);
        }
    }

    *dptr = get_node();
    (*dptr)->value = value;
}
```

- mian()에서 insert_tree()로 넘어왔을 때
##main()에서 삽입한 arr[]의 원소 순서(5, 3, 7, 1, ...)대로 삽입한다고 가정##

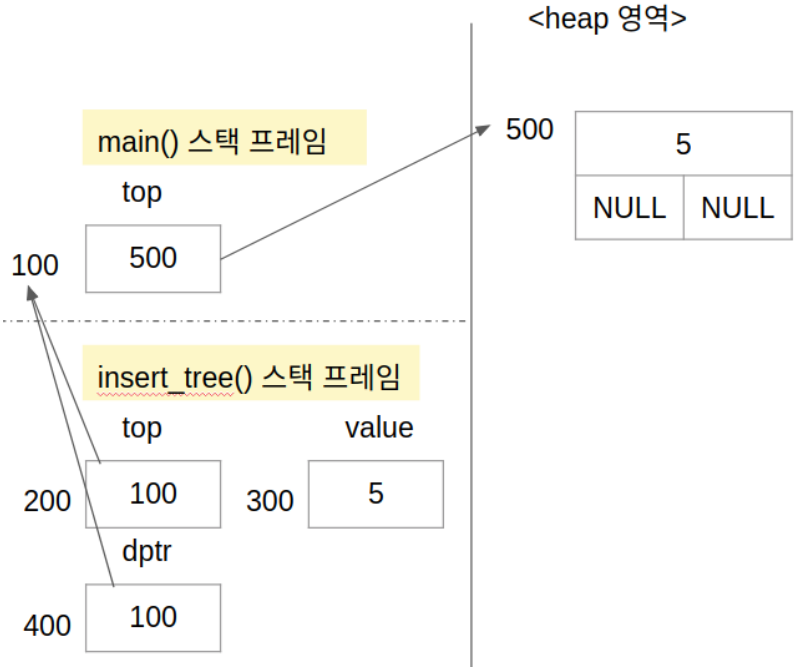


- node ** dptr = top
 : tree 구조가 생기는 top 변수를 참조하기 위해 node ** dptr를 선언

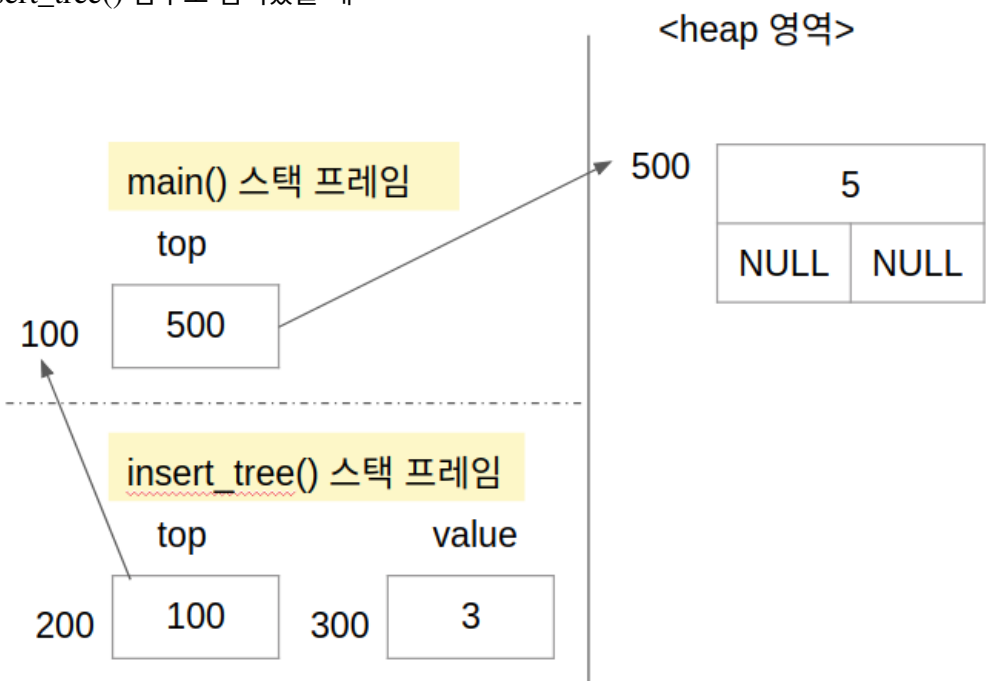


- while 문
 : *dptr이 NULL일 때까지 다음 tree 노드를 타고 내려감
현재는 *dptr이 NULL이므로 실행되지 않음

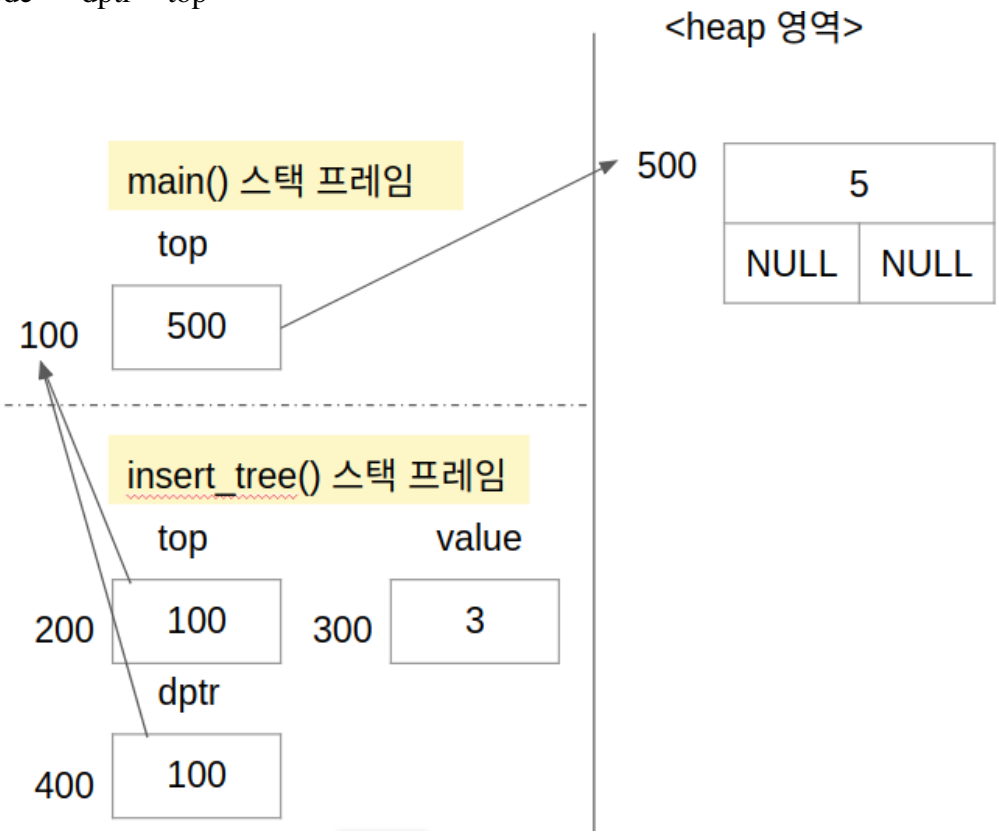
- *dptr = get_node();
- (*dptr)→value = value;
 : *dptr이 가리키고 있는 곳에 새로운 노드를 만듦, value는 인자로 넘어온 value값을 저장



3을 insert_tree() 했을 때
- insert_tree() 함수로 넘어왔을 때



- node ** dptr = top

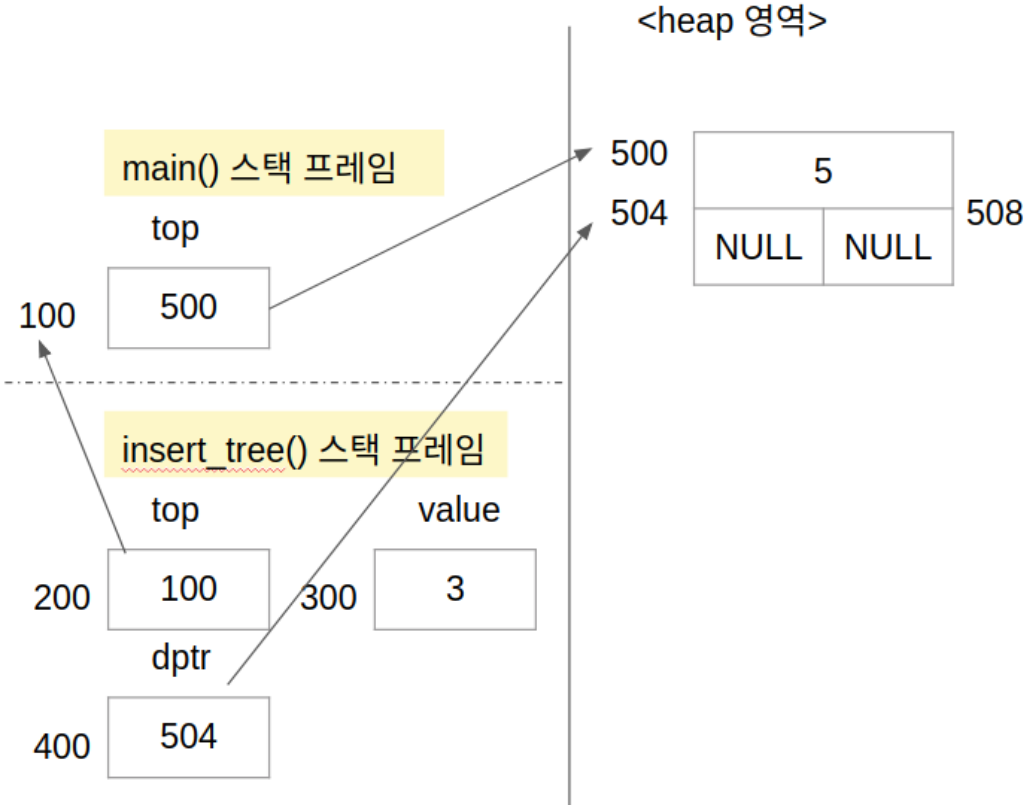


- while 문
 : *dptr값이 500이므로 while문 실행됨

- value < (*dptr)→value or value > (*dptr)→value
 : value변수가 가지고 있는 값(3)이 *dptr이 가리키고 있는 값(5)와 비교하여 왼쪽, 오른쪽으로 갈 지를 결정
  → 3 < 5 이므로 if 문이 실행됨

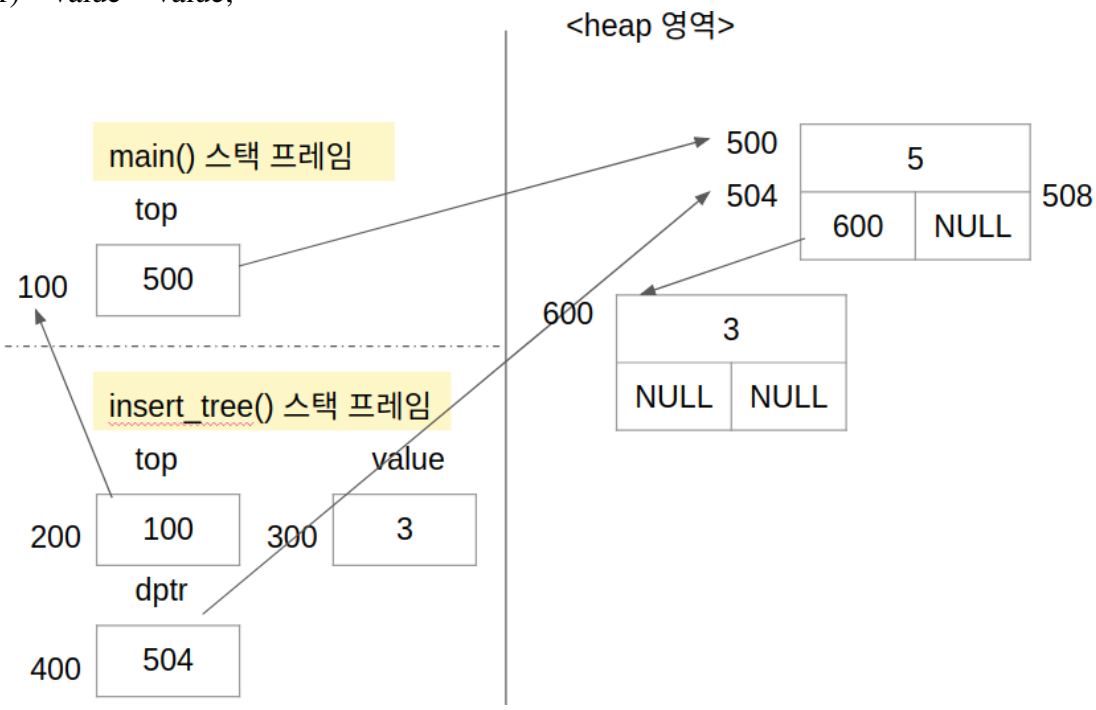
- dptr = &((*dptr)→left) or dptr = &((*dptr)→right)
 : dptr이 왼쪽, 오른쪽 자식 노드의 구조체를 가리키도록 설정
dptr은 node ** 로 노드의 주소를 직접 담고 있는 것이 아니라 노드를 가리키는 노드의 주소를 담고 있음

dptr = &((*dptr)→left)가 실행되어서 다음 그림처럼 됨.
 dptr은 value가 5인 노드의 node * left의 시작 주소를 가리키게 되므로 504가 저장됨
(int는 4byte, node * 는 주소를 담는 자료형이므로 4byte로 가정해서 계산함)

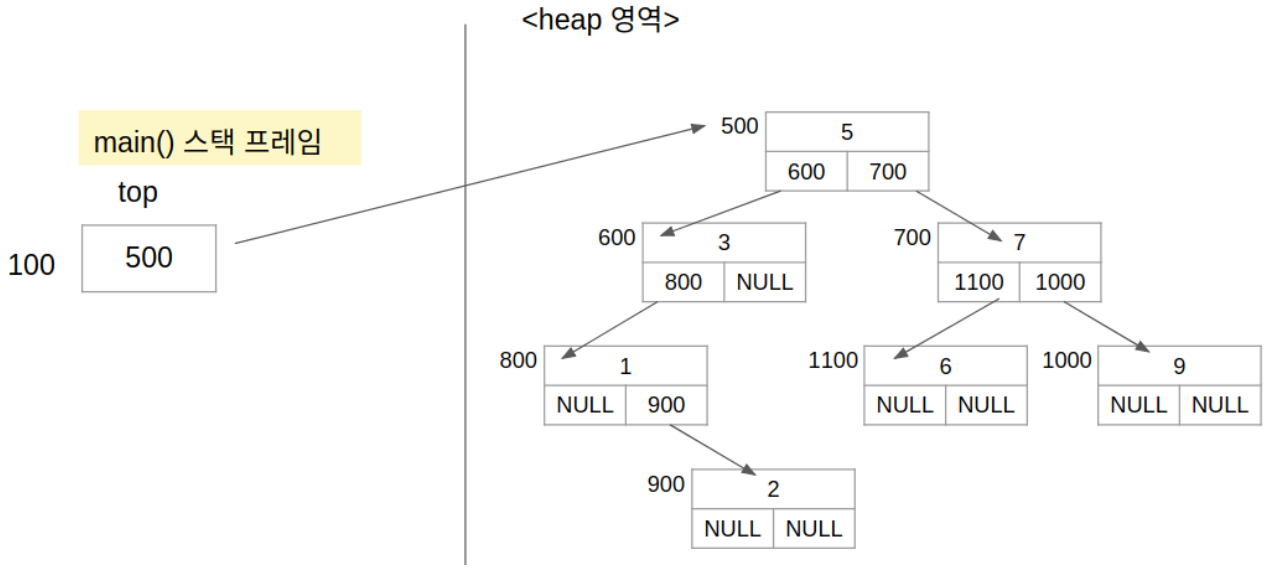


- 다음 while 문에서 *dptr이 NULL 이므로 while문을 빠져 나옴.

- *dptr = get_node();
  (*dptr)→value = value;



- 5, 3, 7, 1, 2, 9, 6 순으로 입력했을 때 tree 구조

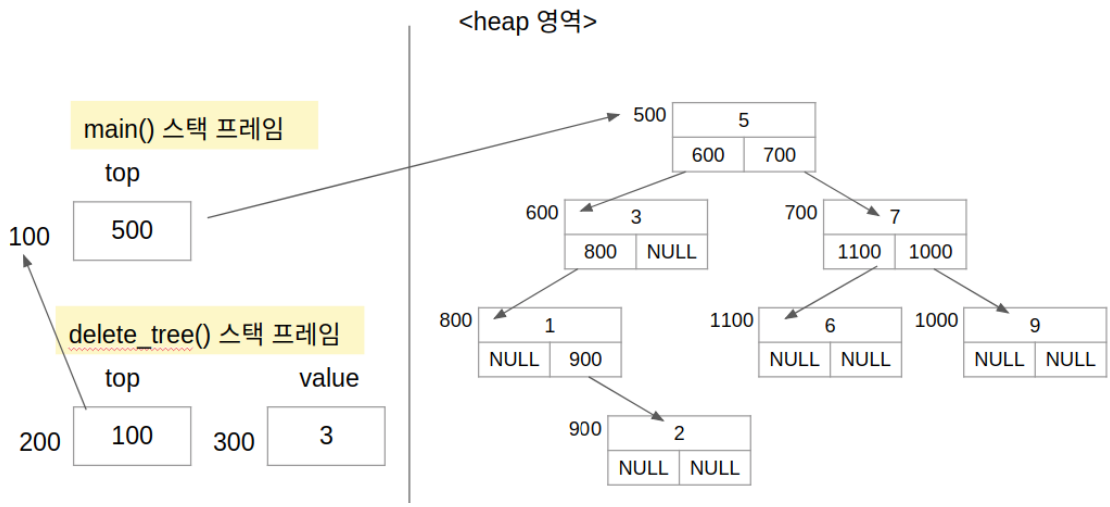


3) delete_tree : value값에 해당하는 노드를 삭제

```c
int delete_tree(node ** top, int value)
{
    int max, dvalue;
    node ** dptr = top;

    while(*dptr)
    {
        if(value < (*dptr)->value)
            dptr = &((*dptr)->left);
        else if(value > (*dptr)->value)
            dptr = &((*dptr)->right);
        else
            break;
    }

    if(!(*dptr))
    {
        printf("No matching node\n");
        return 0;
    }

    dvalue = (*dptr)->value;
    printf("delete value = %d\n", (*dptr)->value);

    if(((*dptr)->left) && ((*dptr)->right))
    {
        find_max(&(*dptr)->left, &max);
        (*dptr)->value = max;
    }
    else if((*dptr)->left)
    {
        (*dptr) = (*dptr)->left;
    }
    else
    {
        (*dptr) = (*dptr)->right;
    }
    return dvalue;
}
```

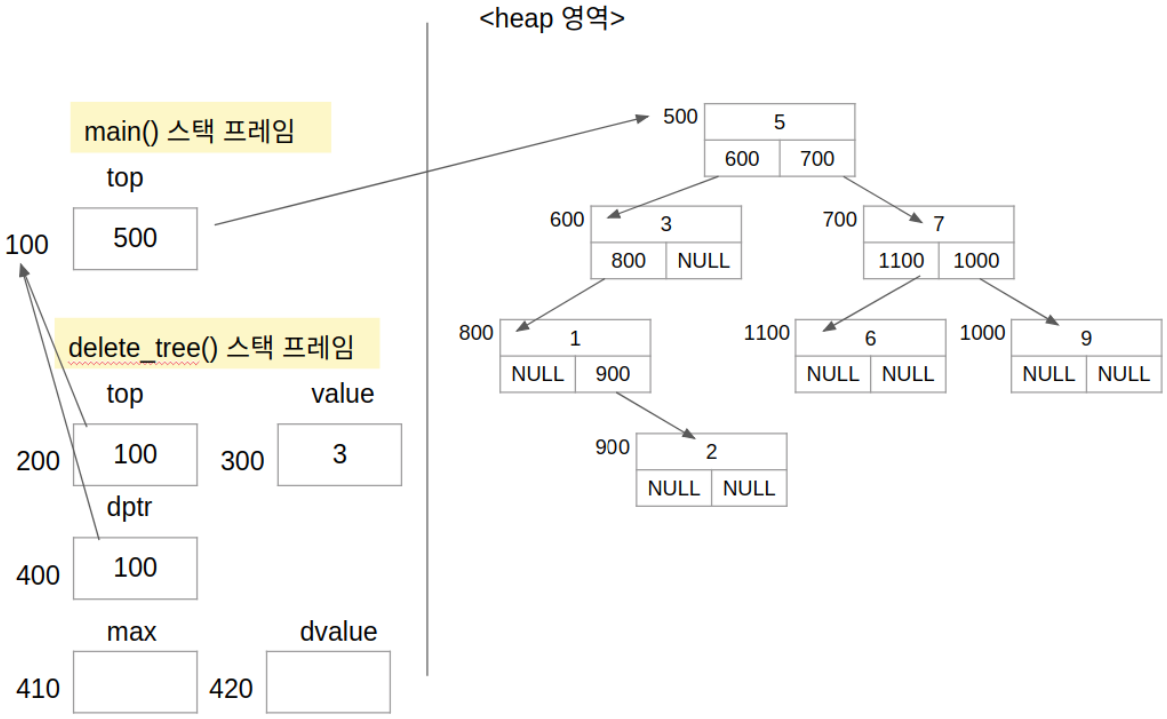
##오른쪽 자식만 있는 3을 지운다고 가정

- delete_tree()으로 넘어왔을 때,

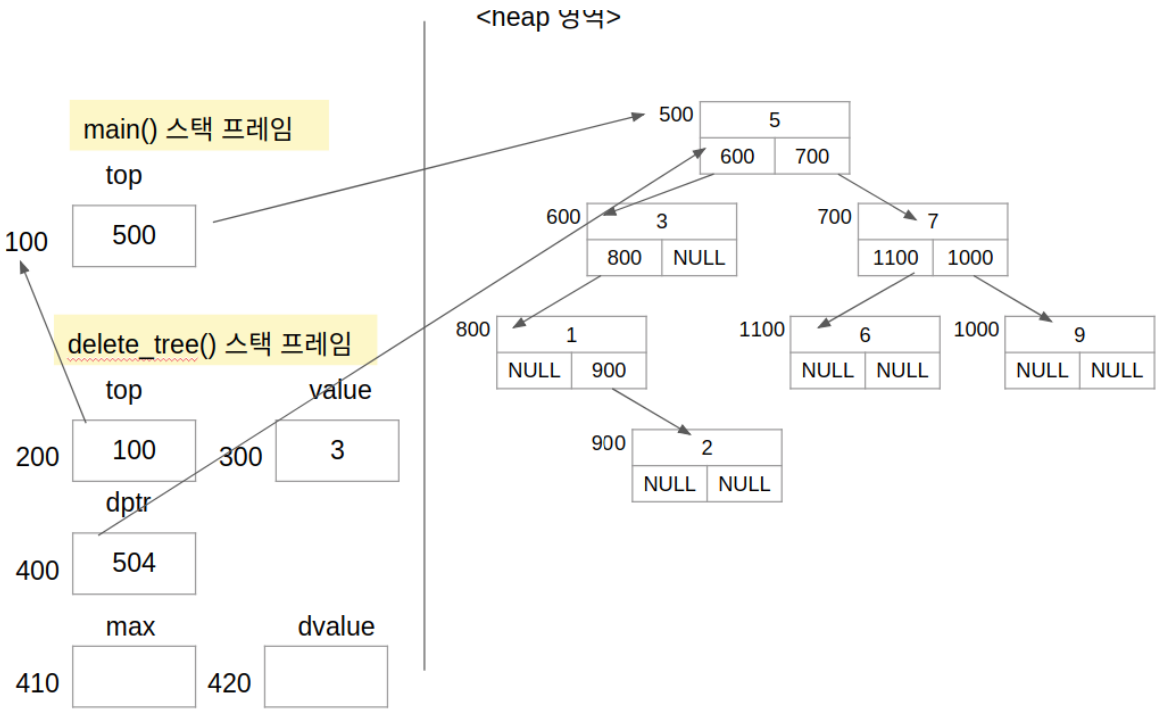


- int max, dvalue
  node ** dptr = top;

max : 양쪽 자식 노드가 둘 다 존재하는 노드를 삭제 할 때. 삭제할 노드를 대체할 노드를 찾아서 저장하는 변수
dvalue : 삭제하는 value
dptr : Tree 구조인 top을 가리키는 node ** 인 dptr



- while 문
 : *dptr이 가리키고 있는 노드의 value와 인자로 넘어온 value(여기서는 3)을 비교하여 같을 때까지 자식노드를 타고 내려간다.
 : value < (*dptr)→value 인 경우는 좌측 자식 노드로, value > (*dptr)→value 인 경우는 우측 자식 노드로 내려간다.



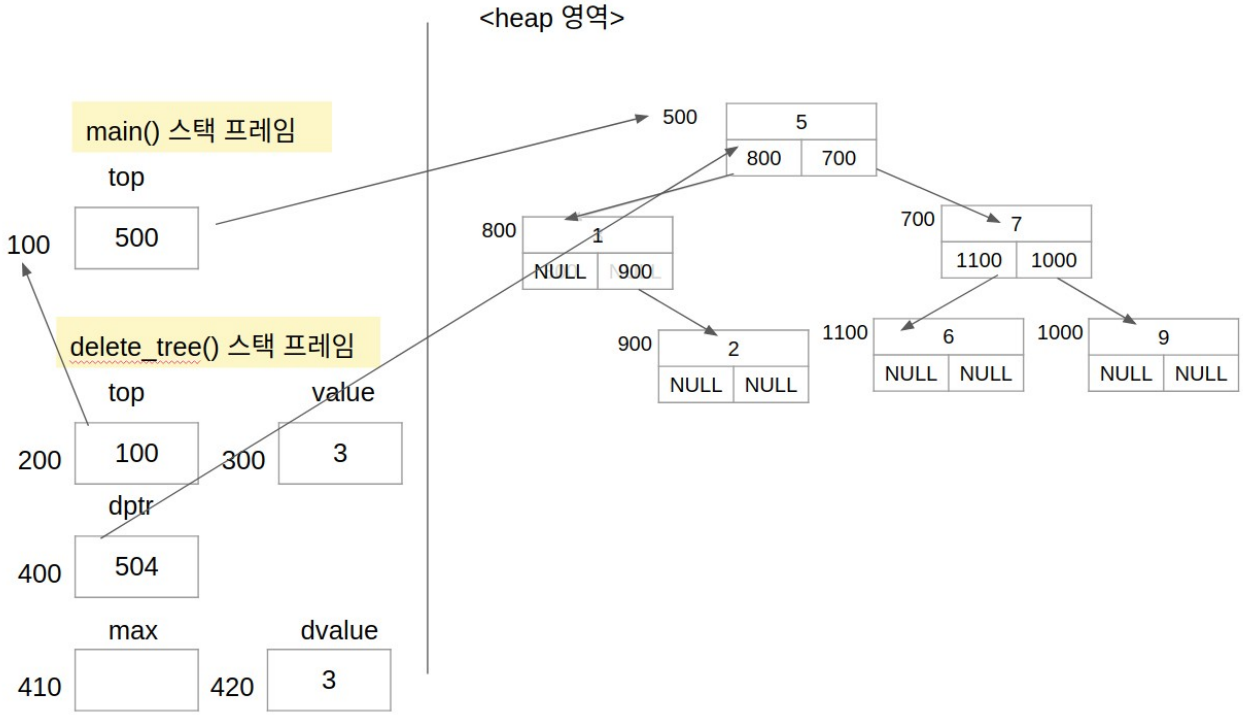
- 현재 *dptr이 가리키는 노드인 500의 value가 5로 인자로 넘어온 value 3이 작으므로
  dptr에 왼쪽 자식 노드의 주소를 저장해 *dptr이 왼쪽 자식 노드를 가리킬 수 있도록 한다.
- while문을 다시 돌 때, 현재 (*dptr)->value가 3 이므로 while문을 빠져나온다.

- if 문에서는 *dptr이 NULL인지 아닌지를 검사하는 데, *dptr == NULL 이면 Tree의 가장 말단으로 내려와서 삭제할 value를 찾지 못했다는 의미이므로 에러 메세지를 출력하고 0을 return 한다.

- dvalue = (*dptr)→value;
  printf 문
: 삭제할 value 값을 반환하기 위해 dvalue에 저장하고, printf문 실행한다.

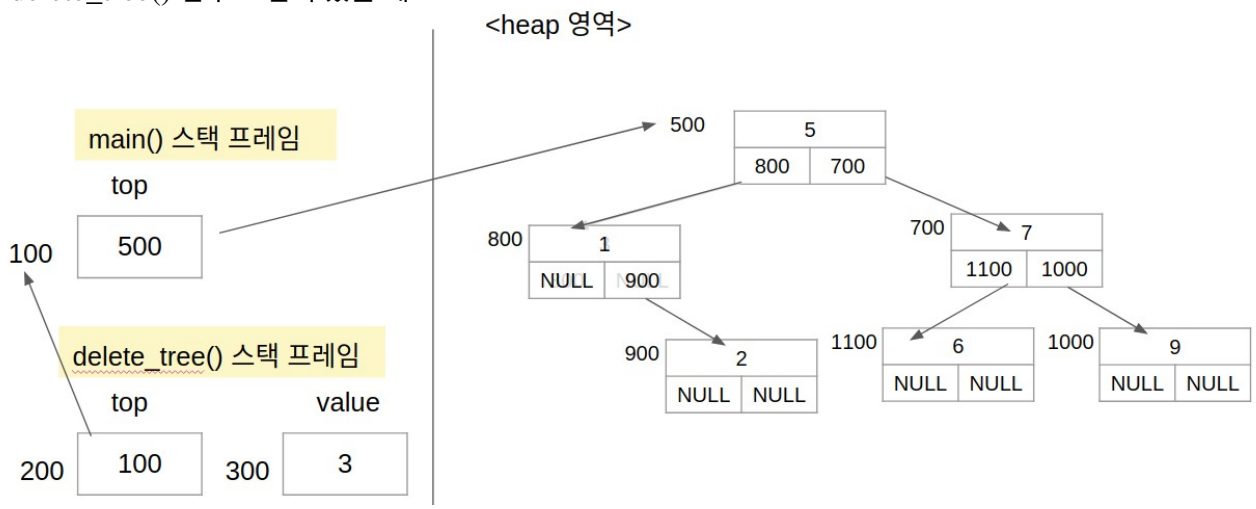
- if- else if – else 문
: 삭제할 노드의 자식 노드가 오른쪽 왼쪽 둘 다 존재하는 지(if), 왼쪽만 존재하는지(else if), 오른쪽만 존재하는 지/자식 노드가 없는 지(else)를 판단한다.
 : 자식 노드가 오른쪽, 왼쪽 둘 다 존재하는 경우에는, 대체할 노드를 찾아주는 과정으로 find_max를 사용한다. (find_max 함수는 잠시 후에 설명)
 : 자식 노드가 왼쪽에만 있는 경우는 왼쪽 자식 노드로 대체한다.
 : 자식 노드가 오른쪽에만 있거나 자식 노드가 없는 경우는 오른쪽 자식 노드로 대체한다.

이번에 삭제할 노드는 value가 3인 노드로 왼쪽에만 자식 노드가 있다. 따라서 else if 문으로 동작한다.

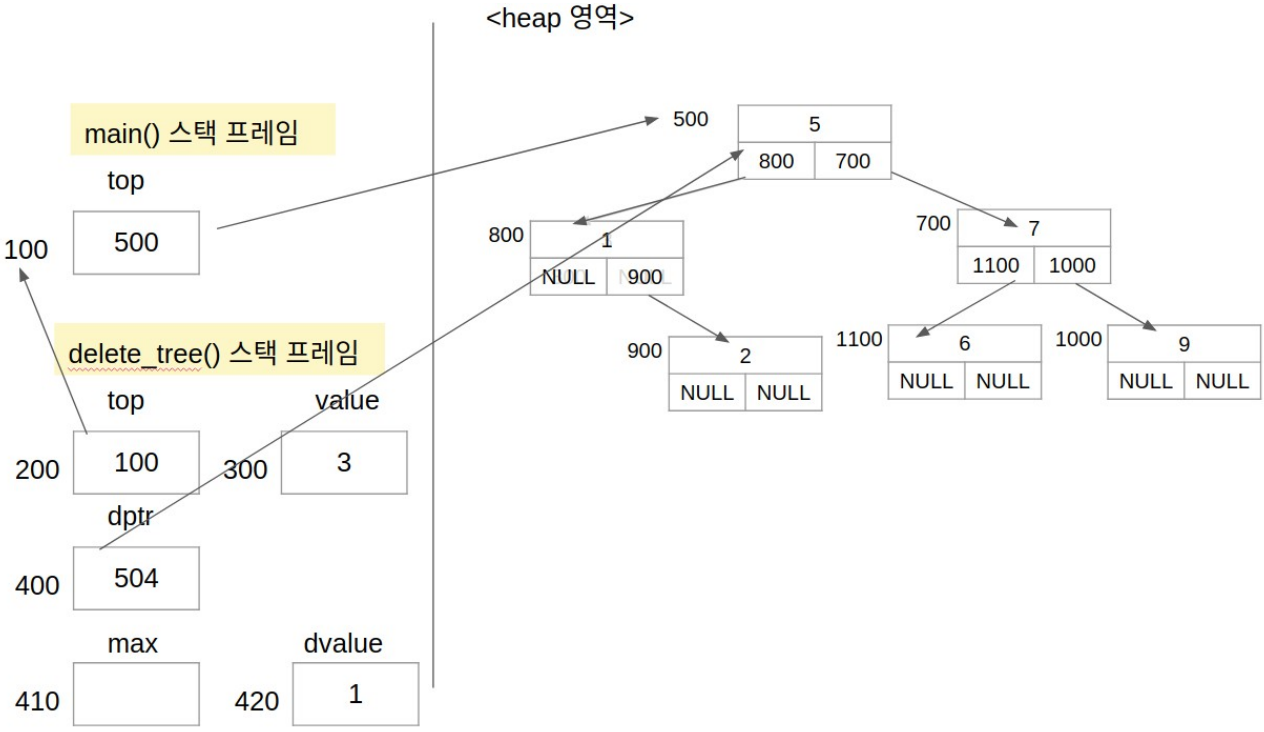


오른쪽 자식만 있는 1을 삭제

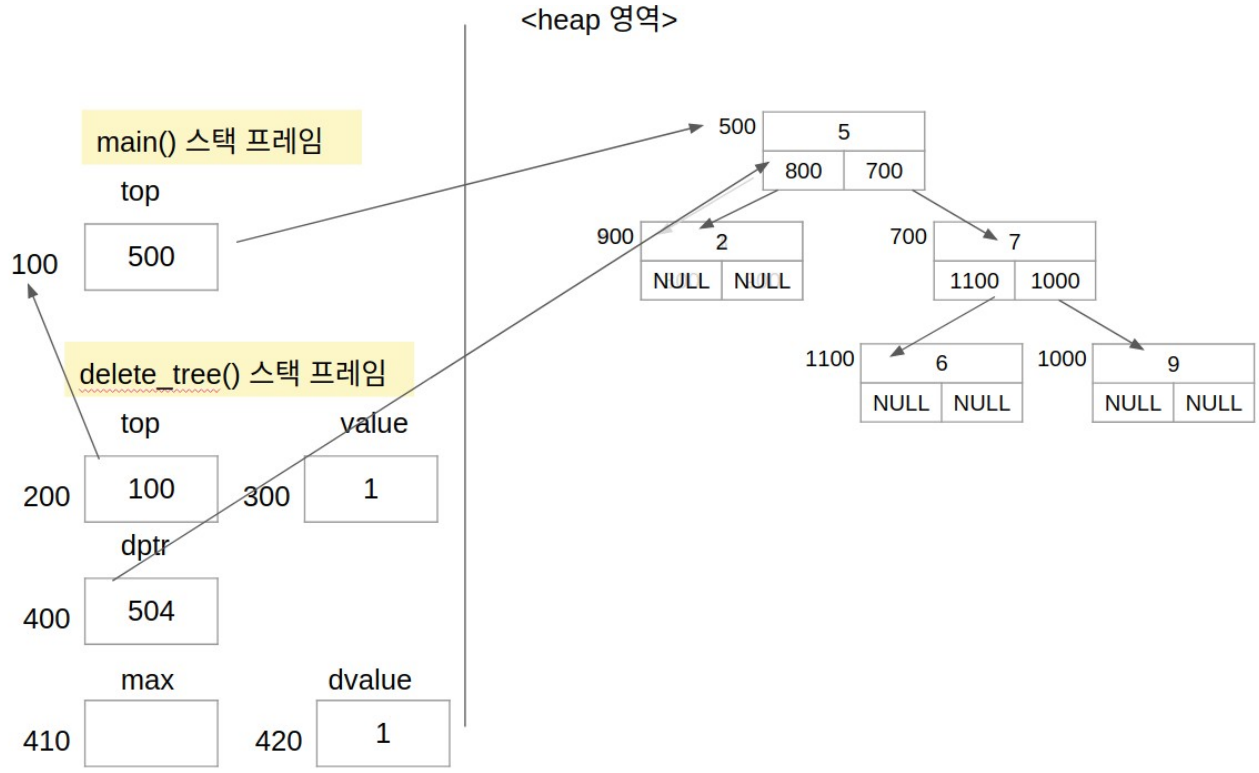
- delete_tree() 함수로 넘어 왔을 때



- while문으로 삭제할 노드 찾기



- value가 1인 노드는 자식 노드가 오른쪽에만 있으므로 else 문 동작

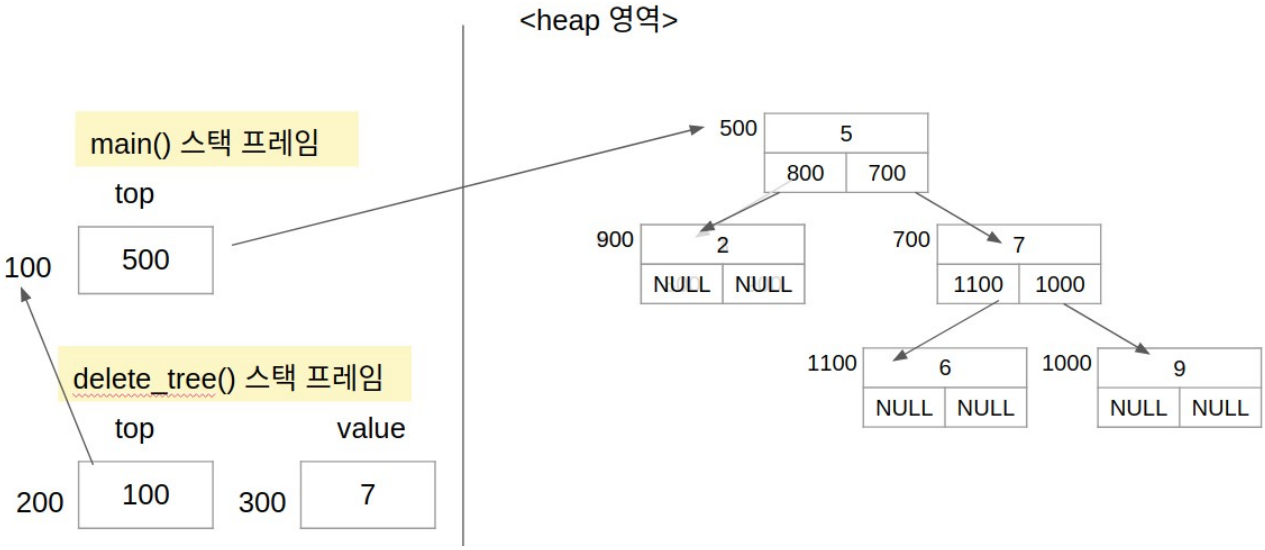


왼쪽, 오른쪽 모두 자식 노드가 있는 경우

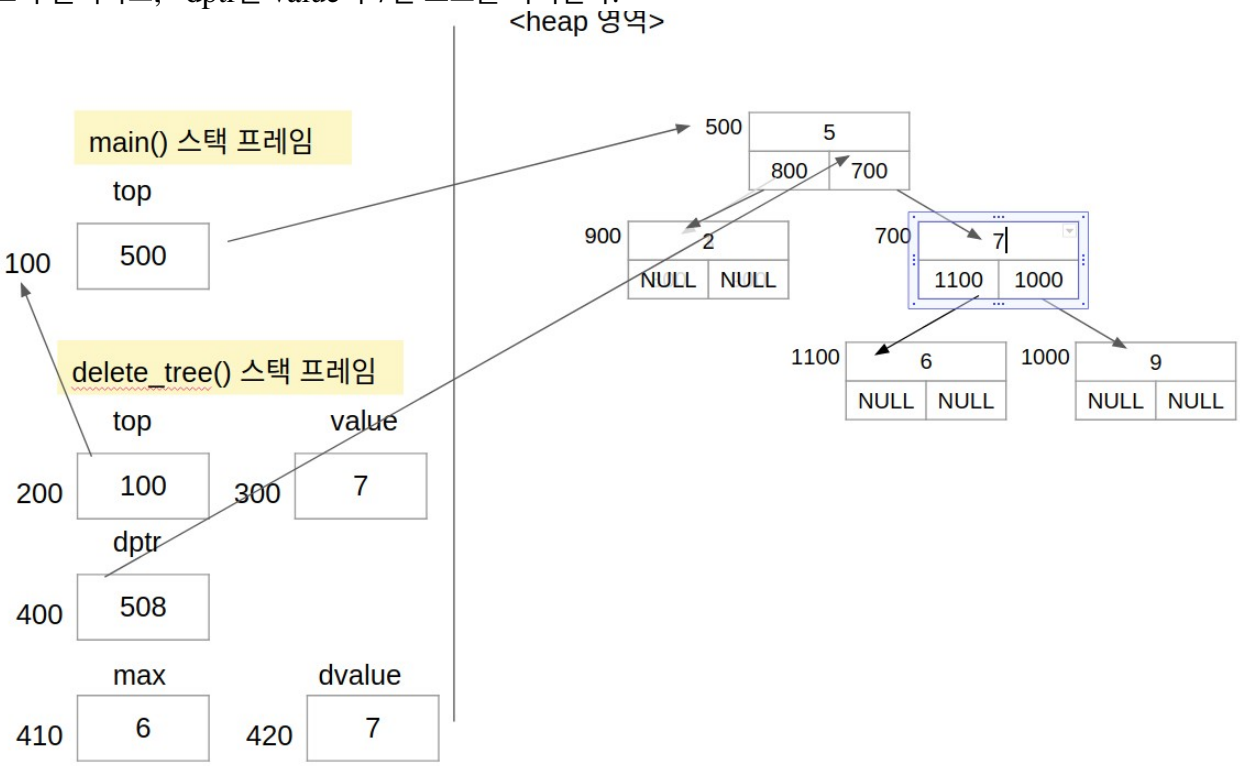
(1) 삭제할 노드의 왼쪽 자식 노드 중 value가 가장 큰 노드(@1)를 찾는다.
(2) @1의 value를 max에 저장한다.
(3) @1노드를 @1의 자식 노드로 대체한다.
(4) max값을 삭제할 노드의 value에 대체한다.

왼쪽, 오른쪽 모두 자식 노드가 있는 경우 → 7을 지우는 경우 생각해보자

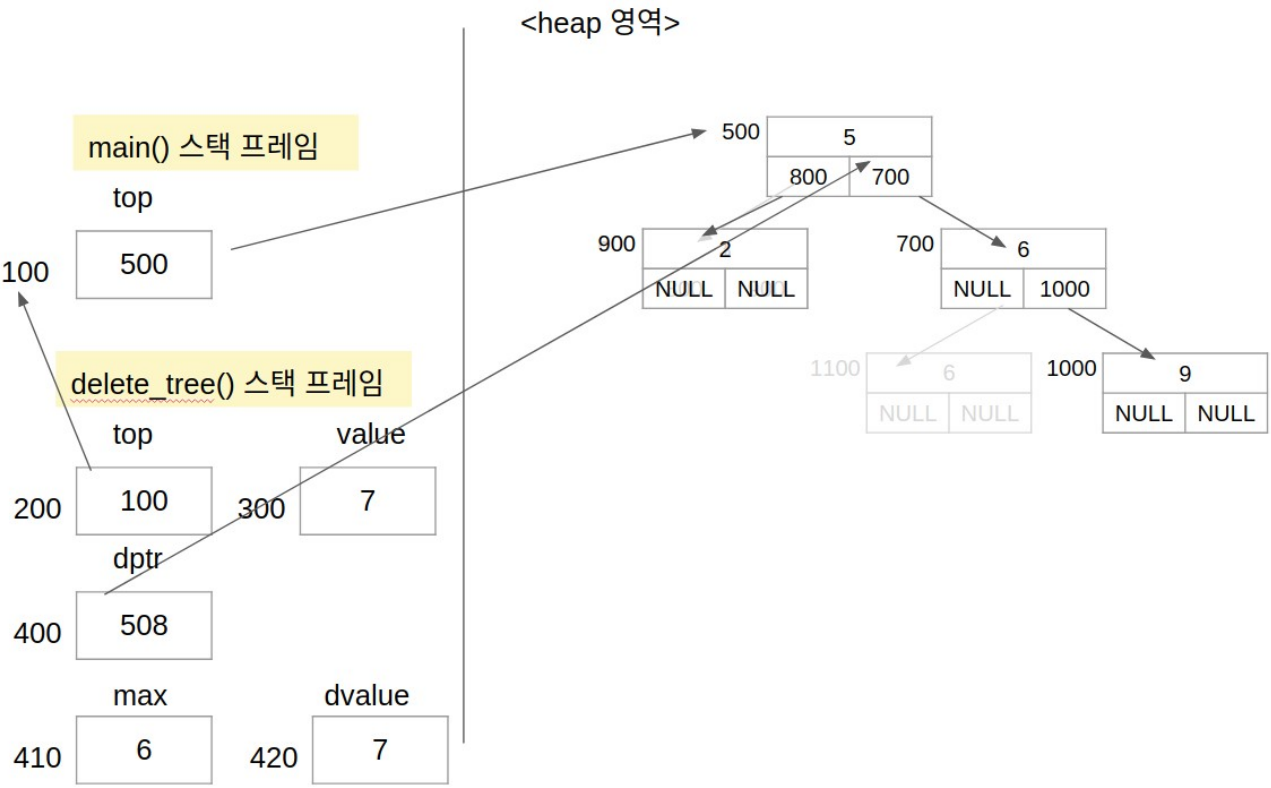
- delete_tree로 넘어왔을 때



- while문으로 삭제할 노드 찾기
: value가 7인 노드는 value가 5인 노드의 오른쪽에 있으므로 dptr에는 value가 5인 노드의 오른쪽 노드의 주소가 들어가고, *dptr은 value기 7인 노드를 가리킨다.



- value가 7인 노드는 오른쪽 왼쪽에 모두 자식 노드가 있으므로 if문이 실행된다.
→ find_max() 함수는 삭제할 노드의 왼쪽 자식 노드에서 value가 가장 큰 노드(@1)의 value를 찾아 삭제할 노드에 대체하고, @1 노드의 자식 노드로 @1 노드를 바꿔준다.



4) find_max : delete_tree 함수에서 삭제할 노드를 대체할 노드를 찾는 과정, 삭제할 노드 왼쪽 노드의 자식 노드 중에서 가장 큰 value를 가지는 노드를 찾기 위한 함수.

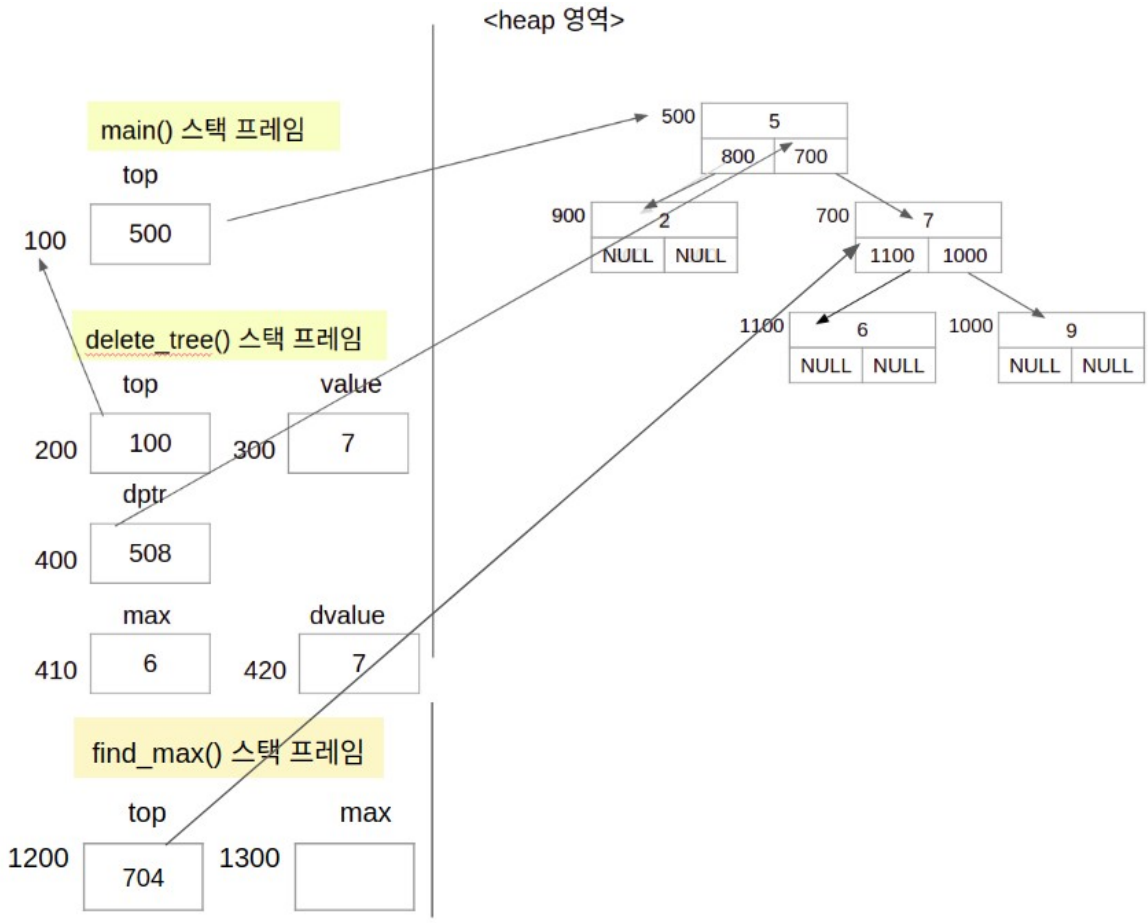
```
void find_max(node ** top, int * max)
{
    node ** tmp = top;

    while((*tmp)->right)
    {
        tmp = &((*tmp)->right);
    }

    *max = (*tmp)->value;
    change_node(tmp);

}
```

위에서 7을 지우는 예를 생각해보도록 하자

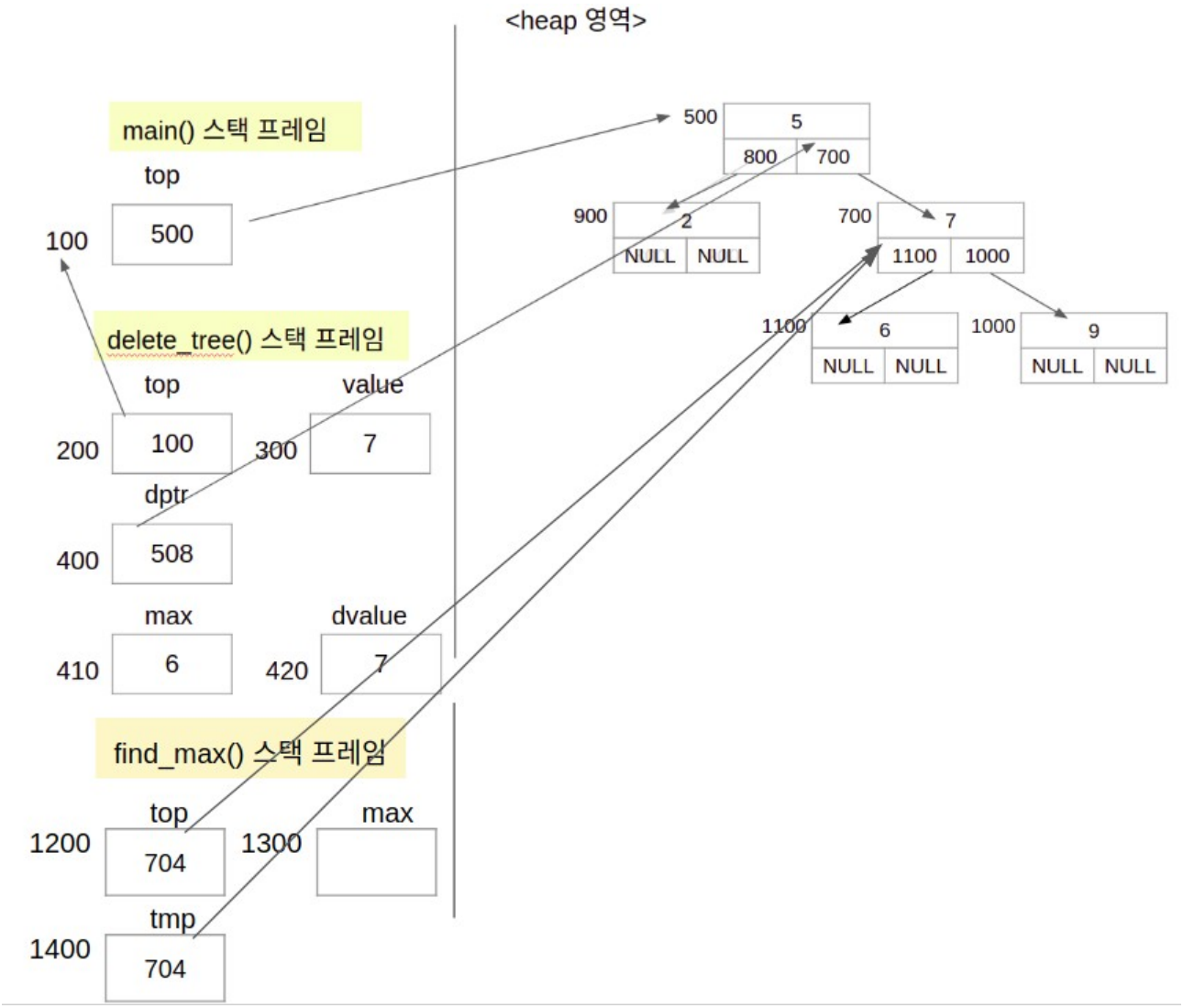
- find_max()에서 인자로 전달되는 값은 삭제할 노드를 가리키는 포인터의 주소(top),
삭제할 노드의 왼쪽 자식 노드 중에서 가장 큰 값을 저장할 max 두 개를 인자로 전달한다.

```
    if(((*dptr)->left) && ((*dptr)->right))
    {
        find_max(&(*dptr)->left, &max);
        (*dptr)->value = max;
    }
```

- find_max() 함수로 넘어 왔을 때



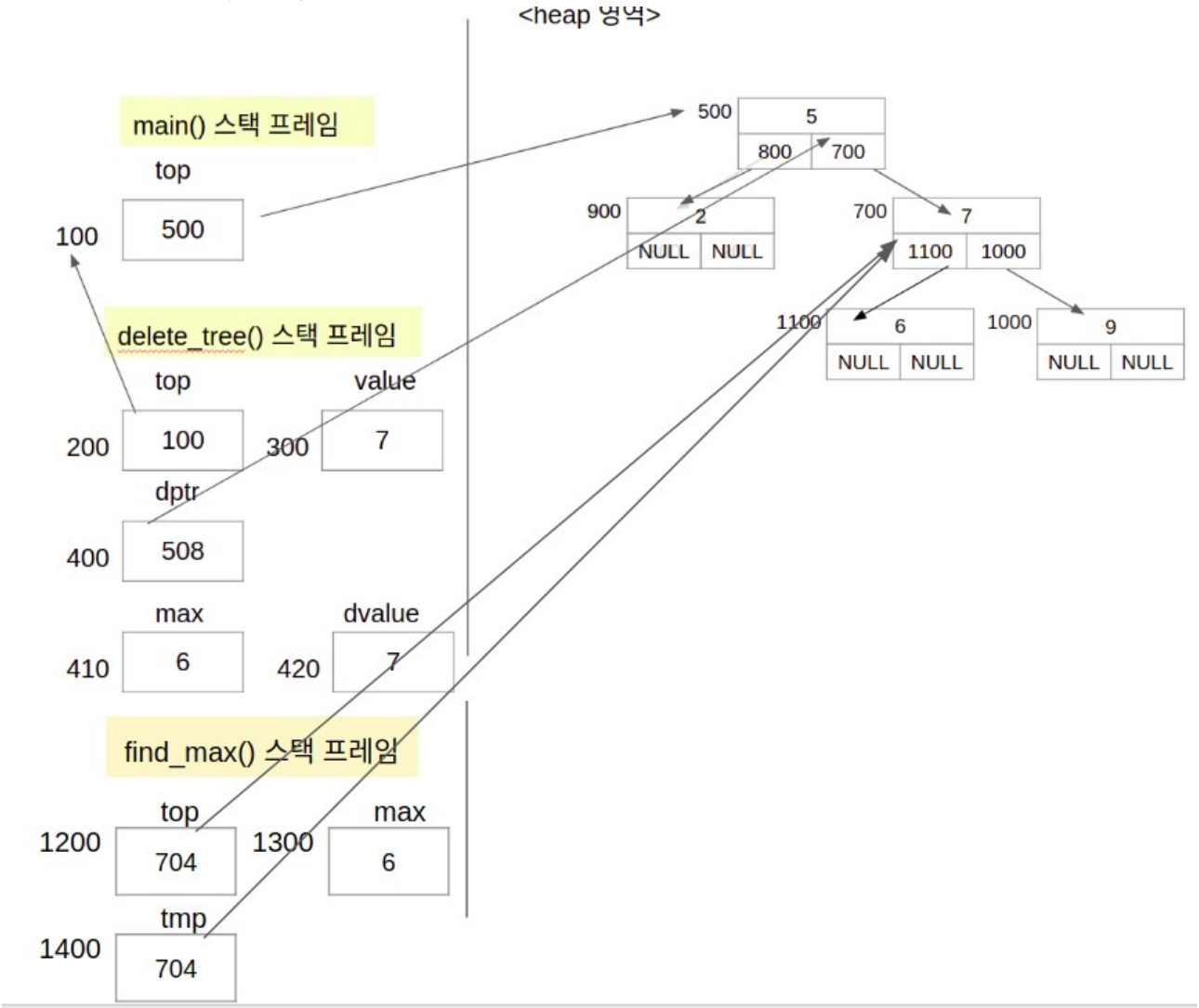
- node ** tmp = top
: tmp에 인자로 넘어온 값을 저장함. tmp도 삭제할 노드의 왼쪽 자식 노드를 가리키는 node * 의 주소를 저장한다.



- while 문
: *tmp의 오른쪽 자식 노드가 NULL이 아닐 때까지 타고 내려가서 삭제할 노드의 왼쪽 자식 노드 중에서 가장 큰 값을 찾는다.

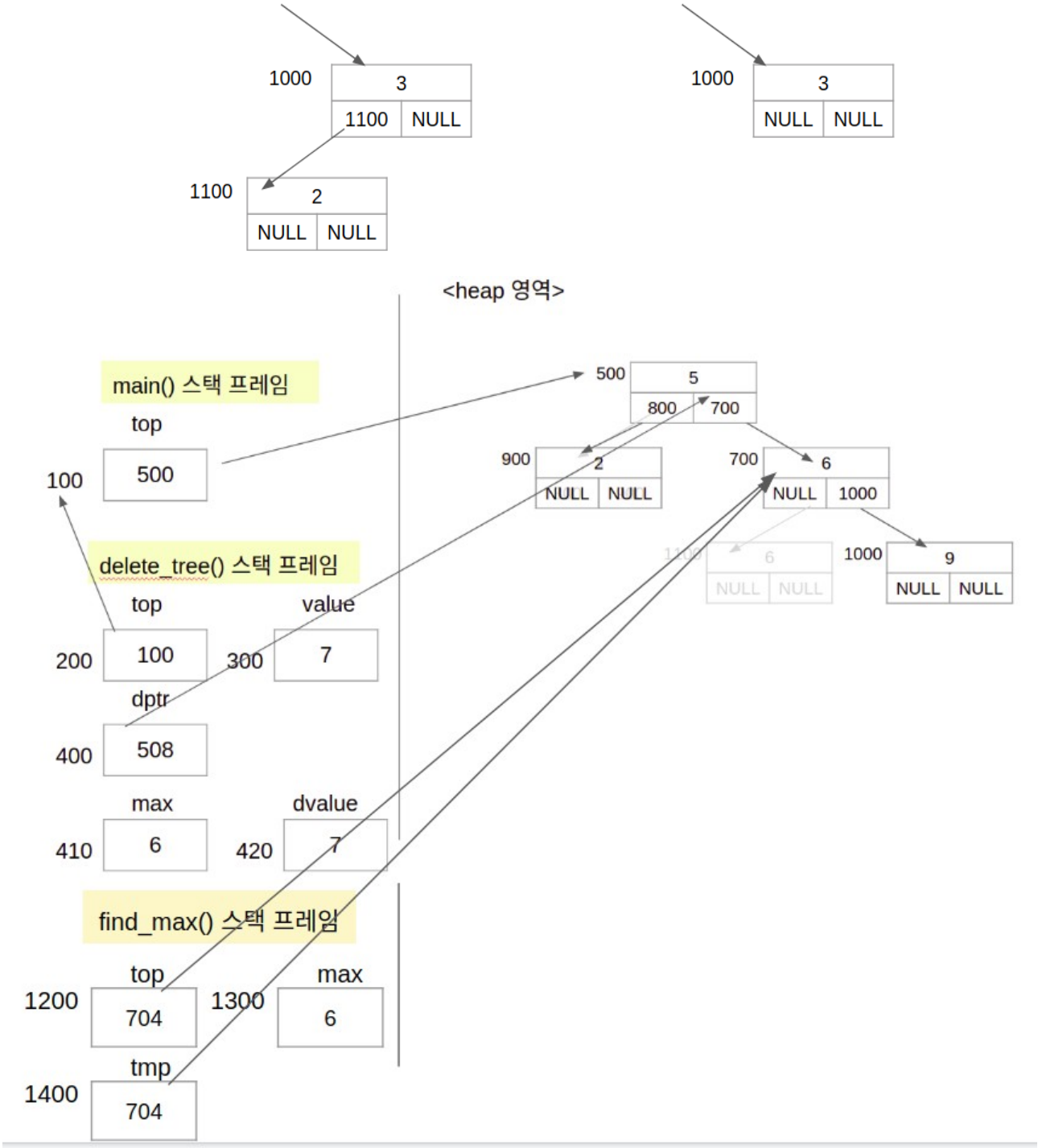
지금 예제에서는 value가 6인 노드의 오른쪽 자식 노드가 NULL이므로 while문이 실행되지 않음

- *max = (*tmp)→value;
: max에 찾은 값을 저장한다.



- change_node(tmp)
 : find_max() 함수에서 삭제할 노드의 왼쪽 자식 노드에서 value가 가장 큰 노드를 찾았을 때, 다음 두 가지 경우가 발생한다.

value가 가장 큰 노드가 3을 value로 하는 노드 인 경우
→ 노드의 왼쪽 자식 노드가 존재 or 자식 노드가 존재하지 않는 경우
→ value가 3인 노드는 삭제할 노드를 대체할 것이므로 value가 3인 노드는 자식 노드로 대체해야 한다.
→ 따라서 왼쪽 자식 노드가 존재하든 하지 않든 모두 왼쪽 자식 노드로 대체해주면 코드를 간단하게 할 수 있다.



5) change_node : delete_tree 함수에서 삭제할 노드를 자식 노드 중 하나로 대체하는 함수.

```
node * change_node(node ** top)
{
    node ** tmp = top;

    if((*top)->left)
        top = &(*top)->left;
    else
        top = &(*top)->right;

    *tmp = *top;

    free(*top);
    return *tmp;
}
```

- node ** tmp = top
 : tmp에 top의 값을 저장

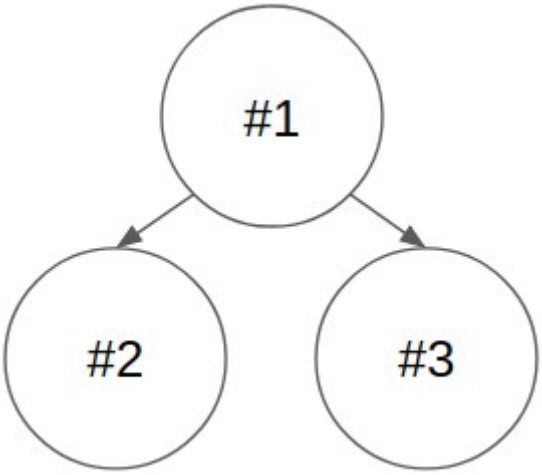
- if 문
: (*tmp)→left 가 있으면 top에 left의 주소값을 저장
: (*tmp)->right가 있으면 top에 right의 주소값을 저장

- *tmp = *top
 free(*top)
 return *tmp
: *top이 가리키는 노드를 *tmp에 저장하고 *top이 가리키는 노드를 free한다.

6) print_preorder, print_inorder, print_postorder : 전위, 중위, 후위 순으로 value값 출력

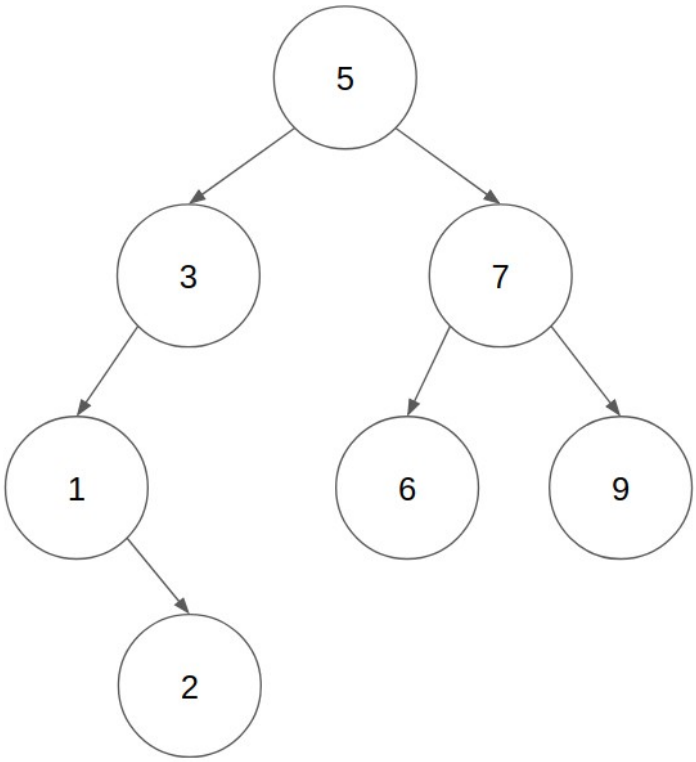
```
void print_preorder(node * top)
{
    node * tmp = top;

    if(!tmp)
        return;

    printf("%d ", tmp->value);
    print_preorder(tmp->left);
    print_preorder(tmp->right);
}

void print_inorder(node * top)
{
    node * tmp = top;

    if(!tmp)
        return;

    print_inorder(tmp->left);
    printf("%d ", tmp->value);
    print_inorder(tmp->right);

}

void print_postorder(node * top)
{
    node * tmp = top;

    if(!tmp)
        return;

    print_postorder(tmp->left);
    print_postorder(tmp->right);
    printf("%d ", tmp->value);

}
```

아래 그림처럼 Tree가 있을 때



- 전위 순회
: #1 → #2 → #3 순으로 방문
- 중위 순회
: #2 → #1 → #3 순으로 방문
- 후위 순회
: #2 → #3 → #1 순으로 방문

위 예제에서 아래 그림과 같이 Tree를 형성 했을 때,




```
5 3 1 2 7 6 9
1 2 3 5 6 7 9
2 1 3 6 9 7 5
```

- 전위, 중위, 후위 순으로 출력