

Tree

Dr. Seung Chul Han
Dept. Computer Engineering
Myongji University

Tree

- 계층형 자료구조 (hierarchical data structure)

- 트리의 순환적 정의

- 하나의 노드는 그 자체로 트리이면서 해당 트리의 루트가 됨
 - 만일 n 이 노드이고 T_1, T_2, \dots, T_k 가 n_1, n_2, \dots, n_k 를 루트로 갖는 트리라고 할때

- n 을 부모로 n_1, n_2, \dots, n_k 를 연결 - 새로운 트리 생성

- n 은 루트(root)

- T_1, T_2, \dots, T_k 는 루트 n 의 서브트리(subtree)

- n_1, n_2, \dots, n_k 는 노드 n 의 자식들(children)



2



- 용어

- Node/Edge, Root, Parent/Child, Siblings, Ancestor/Descendant, Leaf/Internal node, Level, Height/Depth, Degree (차수), Subtree

21세기

조사

자식

잎 노드

용어

▶ 노드(node) : 데이터와 링크를 통합적으로 표현

▶ 노드의 차수(degree)

▶ 한 노드가 가지고 있는 서브 트리의 수

▶ A의 차수: 3, B의 차수: 2, C의 차수: 0

▶ 리프(leaf), 단말 또는 터미널(terminal) 노드

▶ 차수가 0인 노드

▶ 비단말(nonterminal) 노드

▶ 차수가 1 이상인 노드

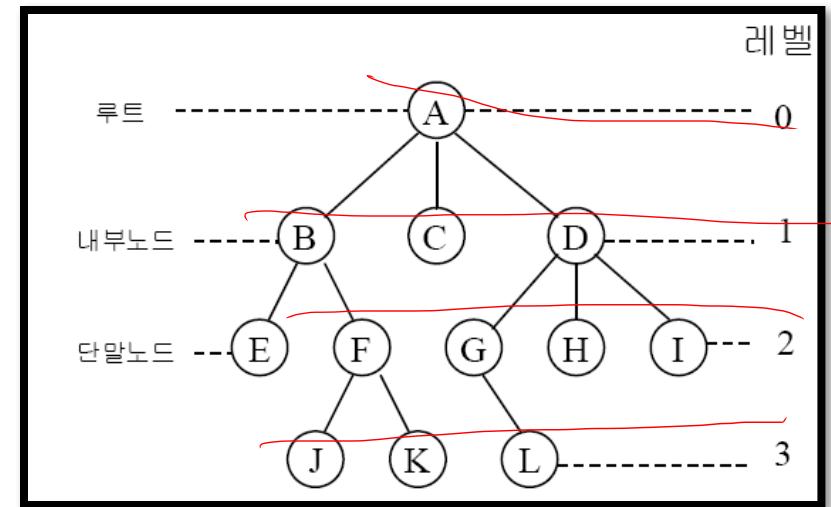
▶ 노드의 parent/children 구조

▶ 자식(children) : 노드 x의 서브 트리 루트들

▶ 부모(parent) : 노드 x

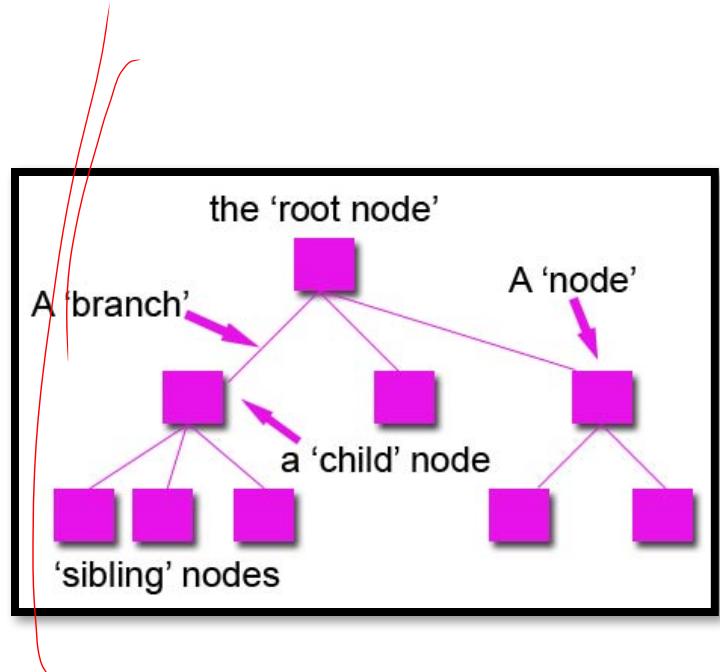
▶ 노드 D의 자식들: G, H, I

▶ D의 부모: A



용어

- ▶ 형제(siblings)
 - ▶ 한 부모의 자식들
 - ▶ 노드 G, H, I는 형제들
- ▶ 트리의 차수
 - ▶ 그 트리에 있는 노드의 최대차수
 - ▶ 트리 T의 차수 : 3



용어

- ▶ 노드의 레벨(level)
 - ▶ 루트 : 0
 - ▶ 한 노드가 레벨 l 에 속하면, 그 자식들은 레벨 $l+1$ 에 속함
 - ▶ $\text{level}(v)$

```
if ( $v = \text{root}$ ) then return 0;  
else return ( $1 + \text{level}(v\text{의 부모})$ );  
end level()
```
- ▶ 트리의 높이(height) 또는 깊이(depth)
 - ▶ 그 트리의 최대 레벨
 - ▶ 트리 T 의 높이: 3
- ▶ 노드의 레벨순서(level order)
 - ▶ 트리의 노드들에 레벨별로 위에서 아래로,
 - ▶ 같은 레벨 안에서는 왼편에서 오른편으로 차례로 순서를 매긴 것

Binary Tree(이진트리)

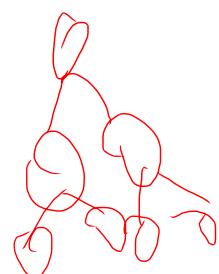
▶ 특징

- ▶ 컴퓨터 응용에서 가장 많이 사용하는 아주 중요한 트리 구조
- ▶ 모든 노드가 정확하게 두 서브트리를 가지고 있는 트리
 - ▶ 서브트리는 공백이 될 수 있음
 - ▶ 리프 노드 (leaf node) : 두 공백 서브트리를 가지고 있는 노드
- ▶ 왼쪽 서브 트리와 오른쪽 서브 트리를 분명하게 구별
- ▶ 이진트리 자체가 노드가 없는 공백이 될 수 있음

▶ 정의 : 이진 트리(binary tree: BT)

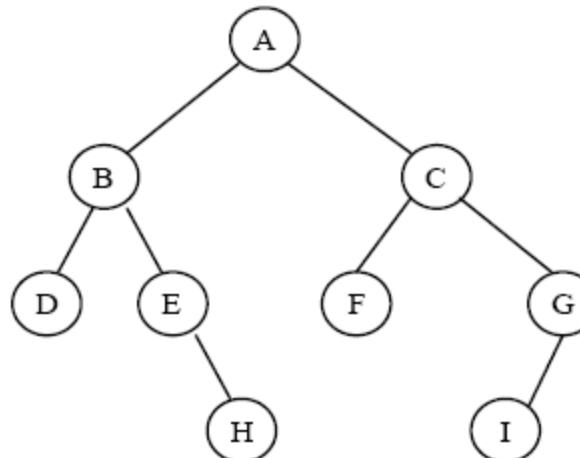
- ▶ 노드의 유한집합
- ▶ 공백이거나 루트와 두 개의 분리된 이진 트리인 왼쪽 서브 트리와 오른쪽 서브 트리로 구성

내가 이진

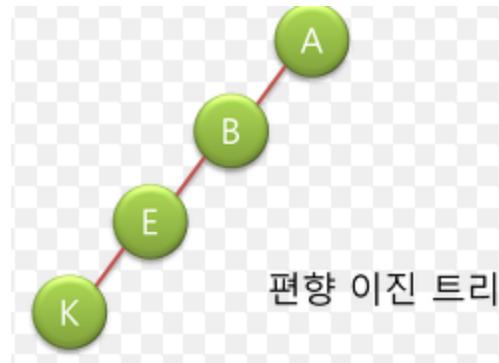
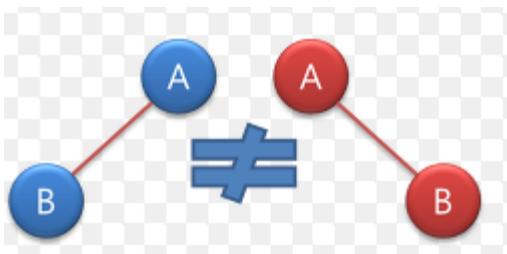


▶ 이진 트리 예

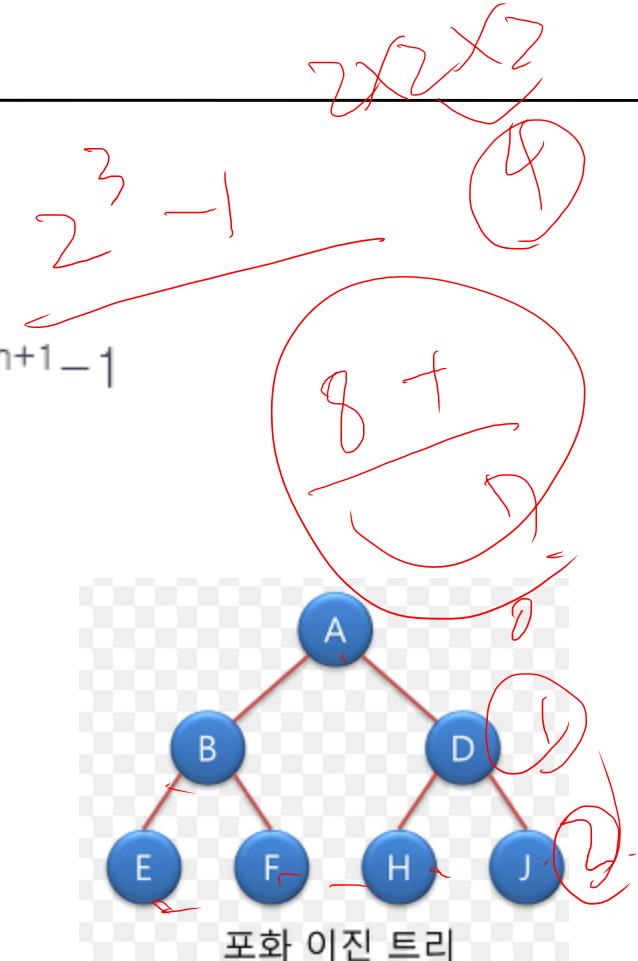
- ▶ 리프노드 (leaf node) : D, H, F, I
- ▶ 노드 E와 G : 공백이 아닌 서브트리를 하나씩 가지고 있음
- ▶ 노드 E : 공백 왼쪽 서브트리와 오른쪽 서브트리 H를 가지고 있음
- ▶ 노드 G : 왼쪽 서브트리 I와 공백 오른쪽 서브트리를 가지고 있음



- ▶ 이진 트리의 주요 성질
 - ▶ 레벨 i 의 최대 노드 수: 2^i
 - ▶ 높이가 h 인 이진 트리의 최대 노드수 : $2^{h+1}-1$

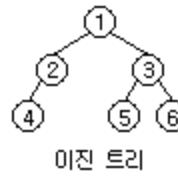


- ▶ 이진 트리에서는 서브 트리의 순서를 구분



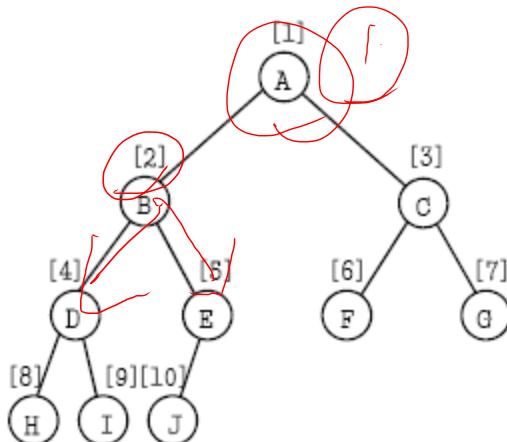
- ▶ 높이가 h 이고 노드수가 $2^{h+1}-1$ 인 이진 트리

- 완전이진트리
 - 리프노드를 제외한 모든 노드가 자식을 2개씩 갖고, 마지막 레벨의 노드들은 왼쪽부터 채워져 있음

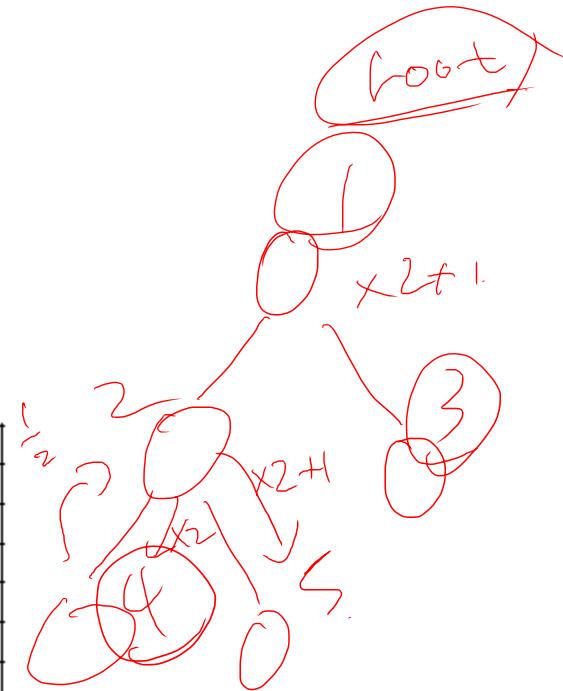


Tree Implementation 1: ARRAY

- ▶ 일차원 배열 표현
 - ▶ 포화 이진 트리 번호를 배열의 인덱스로 사용
 - ▶ 이진 트리의 순차표현
 - ▶ 인덱스 0: 실제로 사용하지 않음
 - ▶ 인덱스 1: 항상 루트 노드
 - ▶ 완전 이진 트리(T)의 순차 표현 예

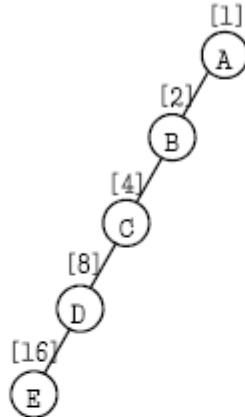


T
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]



$$i/2 = 2$$

▶ 편향 이진 트리(S)의 순차 표현 예



S
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
.
.
...
.
E
[16]

$$i/2$$

▶ 완전 이진 트리의 1차원 배열 표현에서의 인덱스 관계

목표 노드	인덱스 값	조건
노드 i의 부모	$i/2$	$i > 1$
노드 i의 왼쪽 자식	$2*i$	$2*i \leq n$
노드 i의 오른쪽 자식	$2*i + 1$	$(2*i + 1) \leq n$
마지막 노드	1	$0 < n$



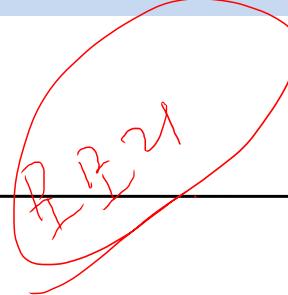
▶ 배열 T에 대한 인덱스 관계 적용 예

- ▶ 노드 I의 부모: 배열 T에서 $I=T[9]$, 그 부모는 $T[9/2]=T[4]$ 에 위치
 - ▶ 노드 C의 오른쪽 자식: $C=T[3]$, 그 오른쪽 자식은 $T[3*2+1]=T[7]$ 에 위치
 - ▶ 노드 C의 왼쪽 자식: $T[3*2]=T[6]$ 에 위치
-
- ▶ 순차 표현의 장단점
 - ▶ 장점
 - 어떤 이진 트리에도 사용 가능
 - 완전 이진 트리: 최적
 - ▶ 단점
 - 편향 이진 트리: 배열 공간을 절반도 사용하지 못할 수 있음
→ 높이가 k인 편향 이진 트리: $2^{k+1}-1$ 개의 공간이 요구될 때 $k+1$ 만 실제 사용
 - 트리의 중간에 노드의 삭제나 삽입시: 많은 다른 노드들의 이동 불가피

2 0-11 0-2(방법)
2F

K+1

Tree Implementation 2: LIST



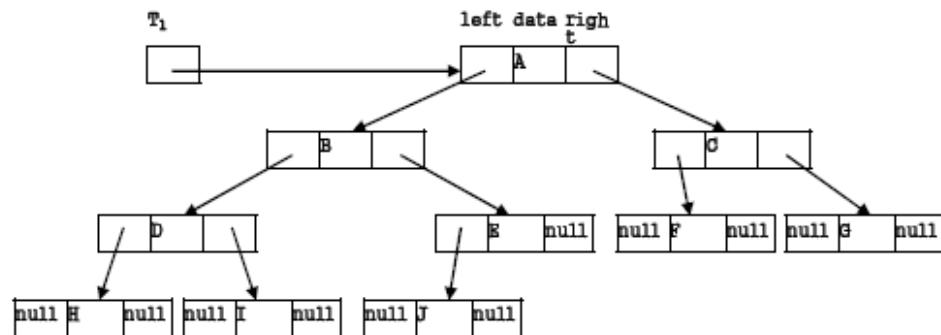
연결 리스트 표현

- 각 노드를 3개의 필드 left, data, right로 구성



고지모리 낭비가
심한

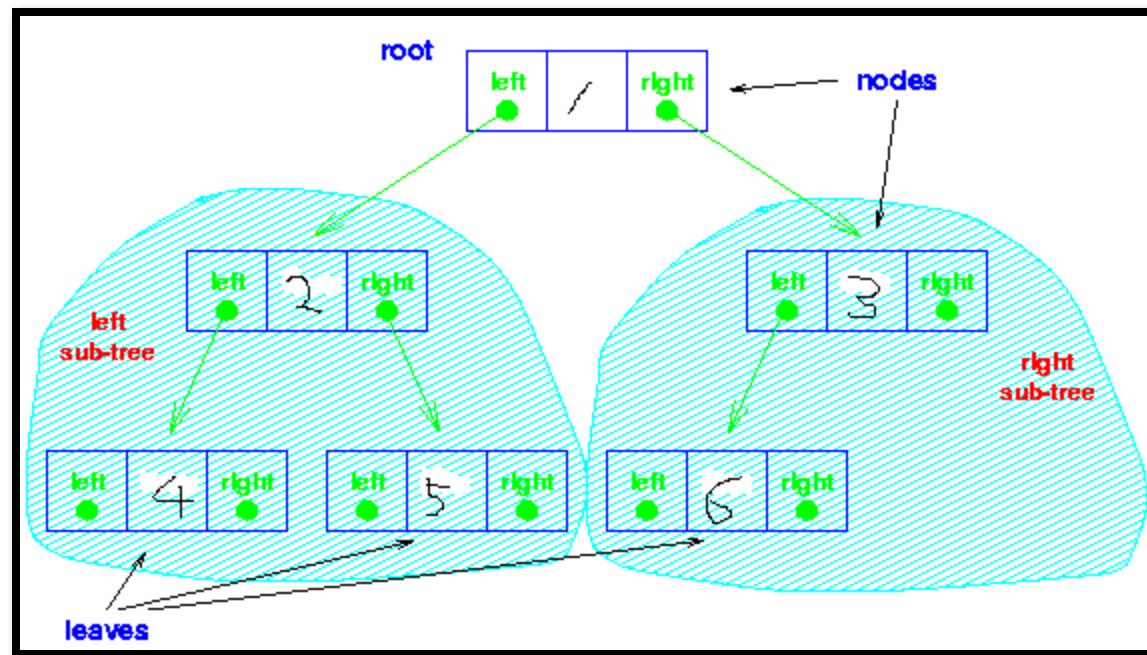
- left와 right : 각각 왼쪽 서브트리와 오른쪽 서브트리를 가리키는 링크
- 필요시 부모를 가리키는 parent 필드 추가
- 완전 이진 트리와 편향 이진 트리의 연결 표현



Tree Implementation 2: LIST

- 다음 tree를 코딩해 보자

```
typedef struct treeNode {  
    int data;  
    struct treeNode* left;  
    struct treeNode* right;  
} treeNode;
```



Tree Traverse (트리순회)

- Tree내의 노드들을 모두 방문
 - ▶ n개 노드시 노드들의 가능한 순회 순서: $n!$
 - ▶ 한 노드에서 취할 수 있는 조치
 - ▶ 왼쪽 서브 트리로의 이동 (L)
 - ▶ 현재의 노드 방문 (D)
 - ▶ 오른쪽 서브 트리로의 이동 (R)
 - ▶ 한 노드에서 취할 수 있는 순회 방법
 - ▶ LDR, LRD, DLR, RDL, RLD, DRL 등 6가지
 - ▶ 왼편을 항상 먼저 순회한다고 가정시 : LDR, LRD, DLR 3가지
→ 이것을 중위(inorder), 후위(postorder), 그리고 전위(preorder) 순회라 함 : 데이터 필드의 위치기준

▶ 중위 순회 (inorder traversal)

▶ 중위 순회 방법의 순환식 기술

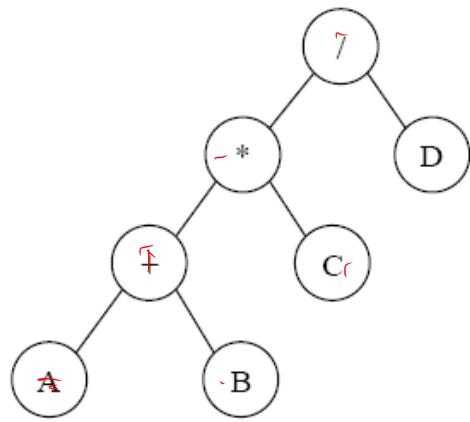
- i) 왼편 서브 트리(left subtree)를 중위 순회한다.
- ii) 루트 노드(root node)를 방문한다.
- iii) 오른편 서브 트리(right subtree)를 중위 순회한다.

▶ 순환 함수 표현

inorder(T)

```
if (T != null) then {
    inorder(T.left);
    visit T.data;
    inorder(T.right);
}
end inorder()
```

▶ 수식 이진 트리의 예



▶ 수식 이진 트리의 중위 순회 결과: A+B*C/D

*A+B*C/D*

```
struct node {  
    int data;  
    struct node *leftChild;  
    struct node *rightChild;  
};
```

```
struct node *root = NULL;
```

```
void inorder_traversal(struct node* root) {  
    if(root != NULL) {  
        inorder_traversal(root->leftChild);  
        printf("%d ",root->data);  
        inorder_traversal(root->rightChild);  
    }  
}
```

*A+B*C/D*

▶ 후위 순회 (postorder traversal)

▶ 후위 순회 방법의 순환식 기술

- i) 왼편 서브 트리(left subtree)를 후위 순회한다.
- ii) 오른편 서브 트리(right subtree)를 후위 순회한다.
- iii) 루트 노드(root node)를 방문한다.

▶ 순환 함수 표현

postorder(T)

```
if (T != null) then {
    postorder(T.left);
    postorder(T.right);
    visit T.data;
}
end postorder()
```

```
void post_order_traversal(struct node* root) {
    if(root != NULL) {
        post_order_traversal(root->leftChild);
        post_order_traversal(root->rightChild);
        printf("%d ", root->data);
    }
}
```

▶ 수식 이진 트리의 후위 순회 결과: AB+C*D/

▶ 전위 순회 (preorder traversal)

▶ 전위 순회 방법의 순환식 기술

- i) 루트 노드(root node)를 방문한다.
- ii) 왼편 서브 트리(left subtree)를 전위 순회한다.
- iii) 오른편 서브 트리(right subtree)를 전위 순회한다.

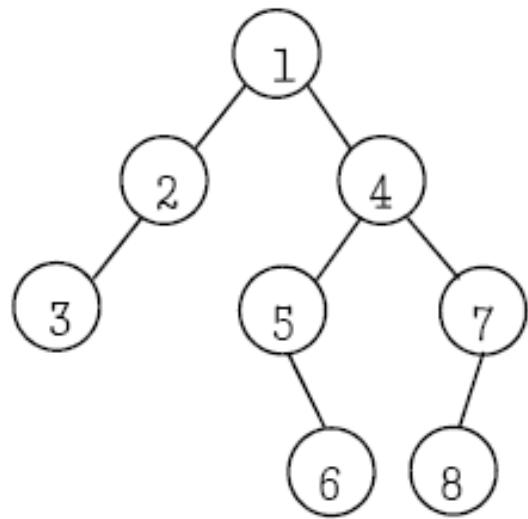
▶ 순환 함수 표현

```
preorder(T)
  if (T != null) then {
    visit T.data;
    preorder(T.left);
    preorder(T.right);
  }
end preorder()
```

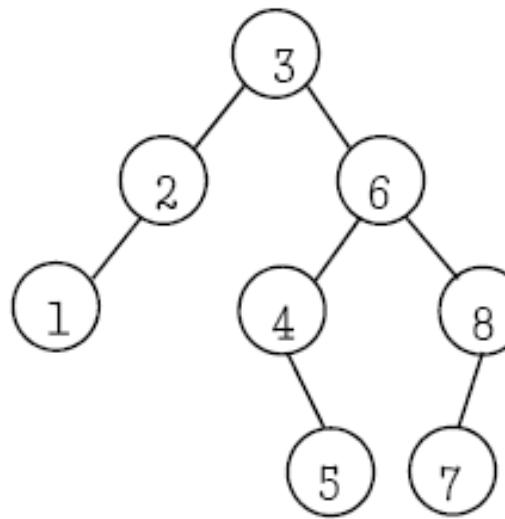
```
void pre_order_traversal(struct node* root) {
  if(root != NULL) {
    printf("%d ",root->data);
    pre_order_traversal(root->leftChild);
    pre_order_traversal(root->rightChild);
  }
}
```

▶ 수식 이진 트리의 전위 순회 결과: /*+ABCD

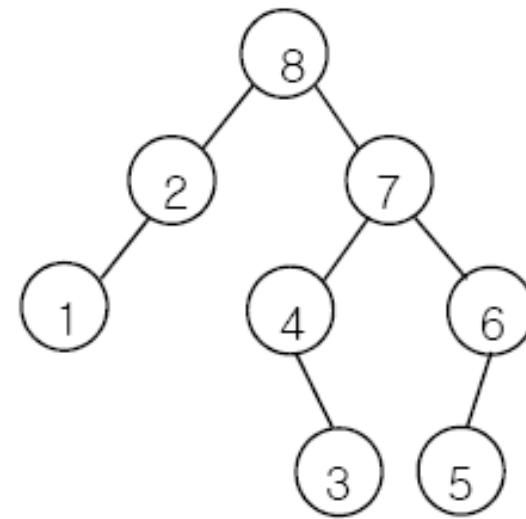
▶ 전위, 중위, 후위 순회 방법 경로의 종합적인 표현



(a) 전위 순회



(b) 중위 순회



(c) 후위 순회

https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal_in_c.htm

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

struct node {
    int data;

    struct node *leftChild;
    struct node *rightChild;
};

struct node *root = NULL;
```

```
void insert(int data) {
    struct node *tempNode = (struct node*) malloc(sizeof(struct node));
    struct node *current;
    struct node *parent;

    tempNode->data = data;
    tempNode->leftChild = NULL;
    tempNode->rightChild = NULL;

    //if tree is empty
    if(root == NULL) {
        root = tempNode;
    } else {
        current = root;
        parent = NULL;

        while(1) {
            parent = current;

            //go to left of the tree
            if(data < parent->data) {
                current = current->leftChild;

                //insert to the left
                if(current == NULL) {
                    parent->leftChild = tempNode;
                    return;
                }
            } //go to right of the tree
            else {
                current = current->rightChild;

                //insert to the right
                if(current == NULL) {
                    parent->rightChild = tempNode;
                    return;
                }
            }
        }
    }
}
```

```

struct node* search(int data) {
    struct node *current = root;
    printf("Visiting elements: ");

    while(current->data != data) {
        if(current != NULL)
            printf("%d ", current->data);

        //go to left tree
        if(current->data > data) {
            current = current->leftChild;
        }
        //else go to right tree
        else {
            current = current->rightChild;
        }

        //not found
        if(current == NULL) {
            return NULL;
        }
    }

    return current;
}

```

```

void pre_order_traversal(struct node* root) {
    if(root != NULL) {
        printf("%d ", root->data);
        pre_order_traversal(root->leftChild);
        pre_order_traversal(root->rightChild);
    }
}

void inorder_traversal(struct node* root) {
    if(root != NULL) {
        inorder_traversal(root->leftChild);
        printf("%d ", root->data);
        inorder_traversal(root->rightChild);
    }
}

void post_order_traversal(struct node* root) {
    if(root != NULL) {
        post_order_traversal(root->leftChild);
        post_order_traversal(root->rightChild);
        printf("%d ", root->data);
    }
}

```

```
int main() {
    int i;
    int array[7] = { 27, 14, 35, 10, 19, 31, 42 };

    for(i = 0; i < 7; i++)
        insert(array[i]);

    i = 31;
    struct node * temp = search(i);

    if(temp != NULL) {
        printf("[%d] Element found.", temp->data);
        printf("\n");
    }else {
        printf("[ x ] Element not found (%d).\n", i);
    }

    i = 15;
    temp = search(i);

    if(temp != NULL) {
        printf("[%d] Element found.", temp->data);
        printf("\n");
    }else {
        printf("[ x ] Element not found (%d).\n", i);
    }

    printf("\nPreorder traversal: ");
    pre_order_traversal(root);

    printf("\nInorder traversal: ");
    inorder_traversal(root);

    printf("\nPost order traversal: ");
    post_order_traversal(root);

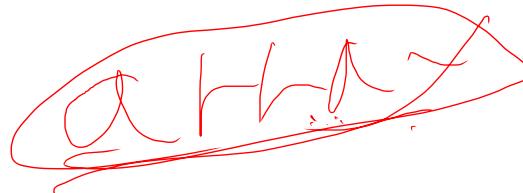
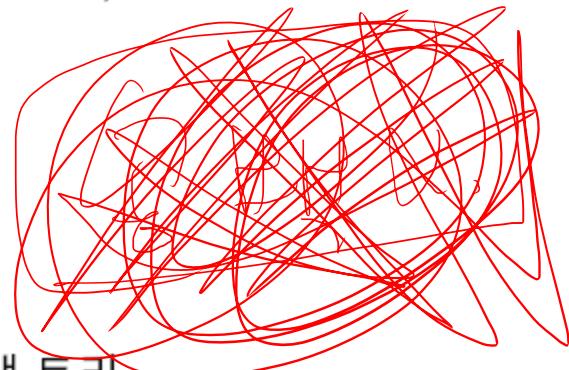
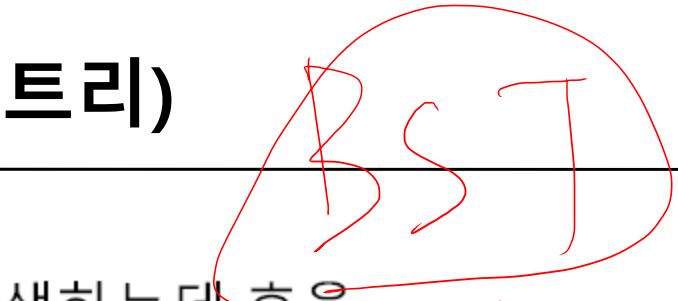
    return 0;
}
```

Binary Search Tree (이진탐색트리)

- ▶ 임의의 키를 가진 원소를 삽입, 삭제, 검색하는데 효율적인 자료구조
- ▶ 모든 연산은 키값을 기초로 실행

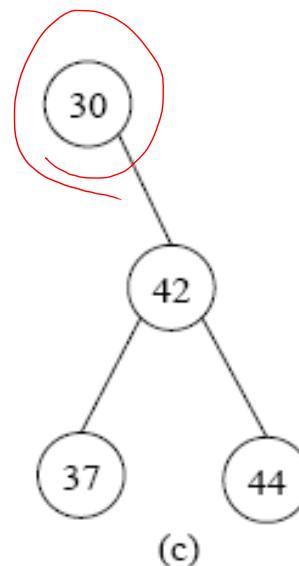
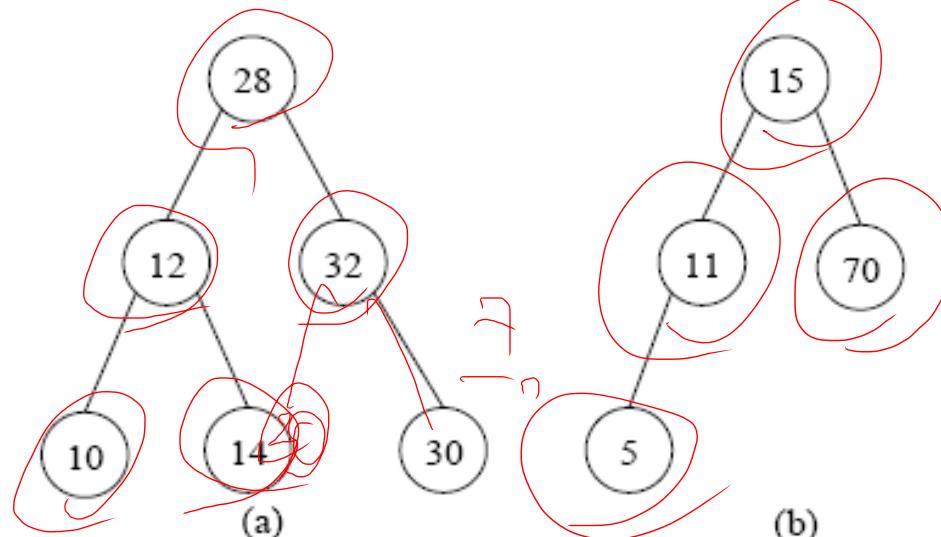
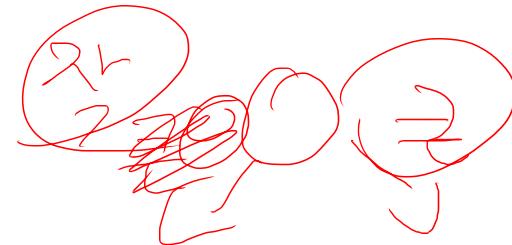
- ▶ 정의 : 이원 탐색 트리(binary search tree:BST)

- ▶ 이진 트리
- ▶ 공백이 아니면 다음 성질을 만족
 - ▶ 모든 원소는 상이한 키를 갖는다.
 - ▶ 왼쪽 서브 트리 원소들의 키 < 루트의 키
 - ▶ 오른쪽 서브 트리 원소들의 키 > 루트의 키
 - ▶ 왼쪽 서브 트리와 오른쪽 서브 트리 : 이원 탐색 트리



▶ 예

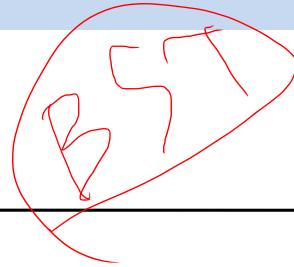
- ▶ 그림 (a): 이원 탐색 트리가 아님
- ▶ 그림 (b), (c): 이원 탐색 트리임



(b)

(c)

Search in BST



- ▶ 이원 탐색 트리에서의 탐색 (순환적 기술)
 - ▶ 키값이 x 인 원소를 탐색
 - ▶ 시작 : 루트
 - ▶ 이원 탐색 트리가 공백이면, 실패로 끝남
 - ▶ 루트의 키값 = x 이면, 탐색은 성공하며 종료
 - ▶ 키값 $x <$ 루트의 키값이면, 루트의 왼쪽 서브트리만 탐색
 - ▶ 키값 $x >$ 루트의 키값이면, 루트의 오른쪽 서브트리만 탐색
 - ▶ 연결 리스트로

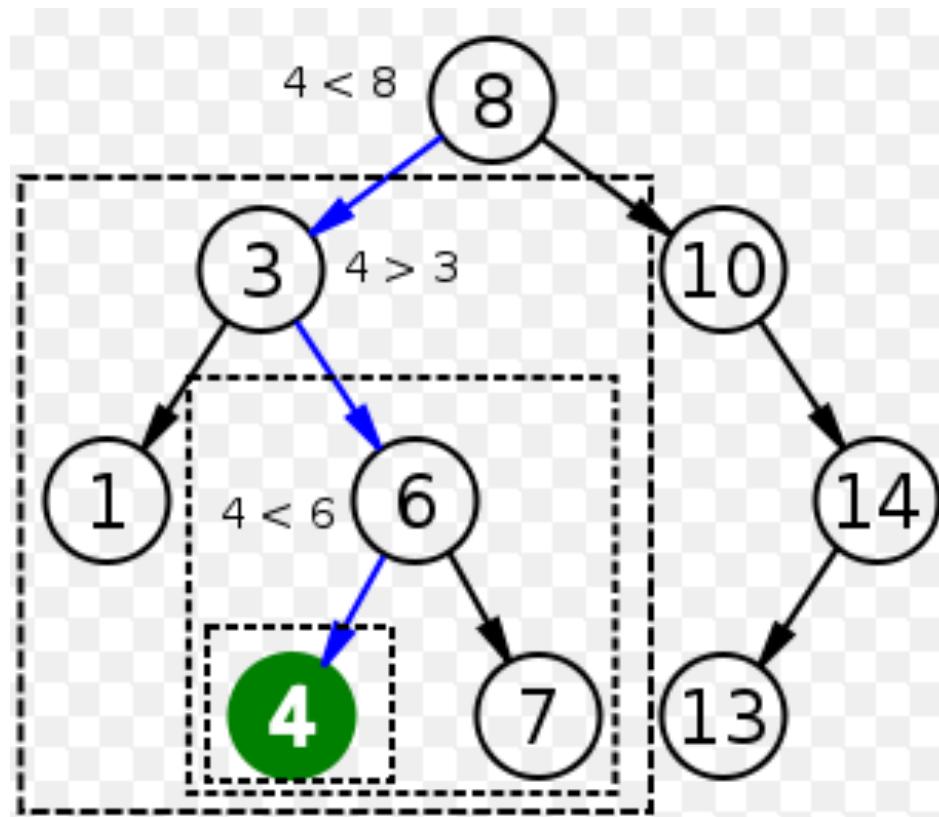
left	key	right
------	-----	-------

 - ▶ 노드 구조 :

- Search Algorithm

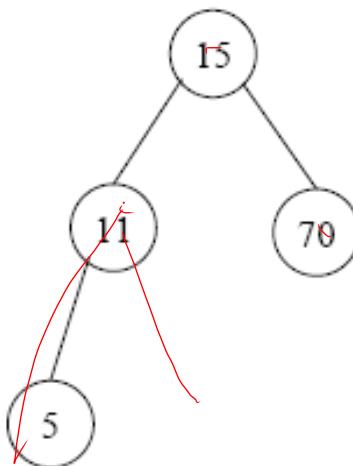
```
searchBST(B, x)
    // B는 이원 탐색 트리
    // x는 탐색 키값
    p ← B;
    if (p = null) then           // 공백 이진 트리로 실패
        return null;
    if (p.key = x) then          // 탐색 성공
        return p;
    if (p.key < x) then          // 오른쪽 서브트리 탐색
        return searchBST(p.right, x);
    else return searchBST(p.left, x); // 왼쪽 서브트리 탐색
end searchBST()
```

- Example

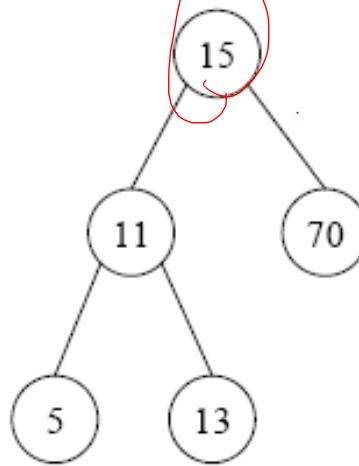


Insert in BST

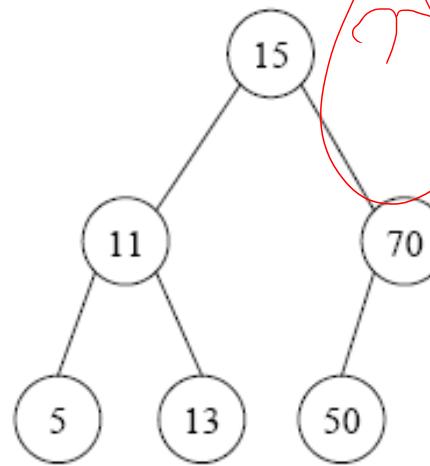
- ▶ 이원 탐색 트리에서의 삽입
 - ▶ 키값이 x인 새로운 원소를 삽입
 - ▶ x를 키값으로 가진 원소가 있는가를 탐색
 - ▶ 탐색이 실패하면, 탐색이 종료된 위치에 원소를 삽입
 - ▶ 예 : 키값 13, 50의 삽입 과정



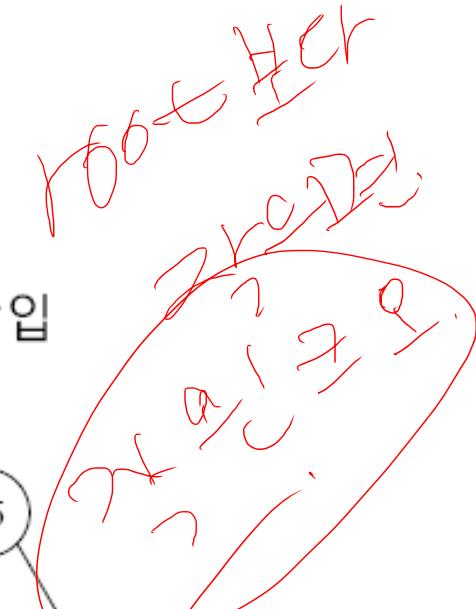
(a) 이진 탐색 트리



(b) 원소 13을 삽입



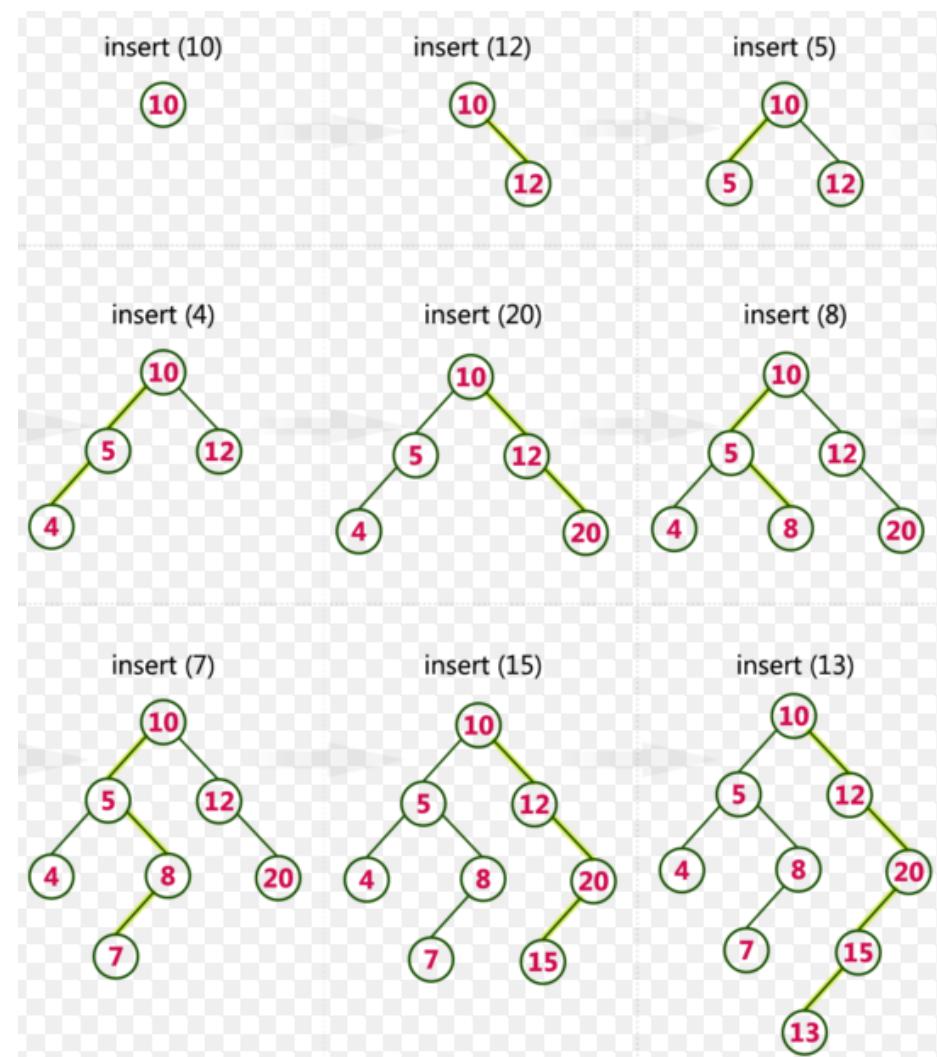
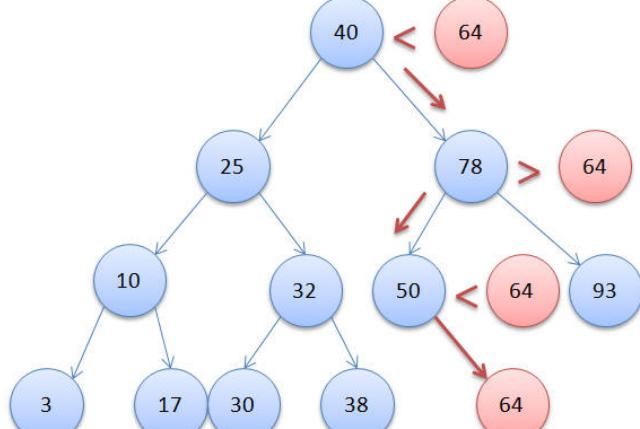
(c) 원소 50을 삽입



• Insert Algorithm

```
insertBST(B, x)
    // B는 이원 탐색 트리, x는 삽입할 키값
    p ← B;
    while (p ≠ null) do {
        if (x = p.key) then return;           // 키값을 가진 노드가 이미 있음
        q ← p;                            // q는 p의 부모 노드를 지시
        if (x < p.key) then p ← p.left;
        else p ← p.right;
    }
    newNode ← getNode();           // 삽입할 노드를 만듦
    newNode.key ← x;
    newNode.right ← null;
    newNode.left ← null;
    if (B = null) then B ← newNode; // 공백 이원 탐색 트리인 경우
    else if (x < q.key) then      // q는 탐색이 실패하여 종료된 원소
        q.left ← newNode;
    else
        q.right ← newNode;
    return;
end insertBST()
```

- Example

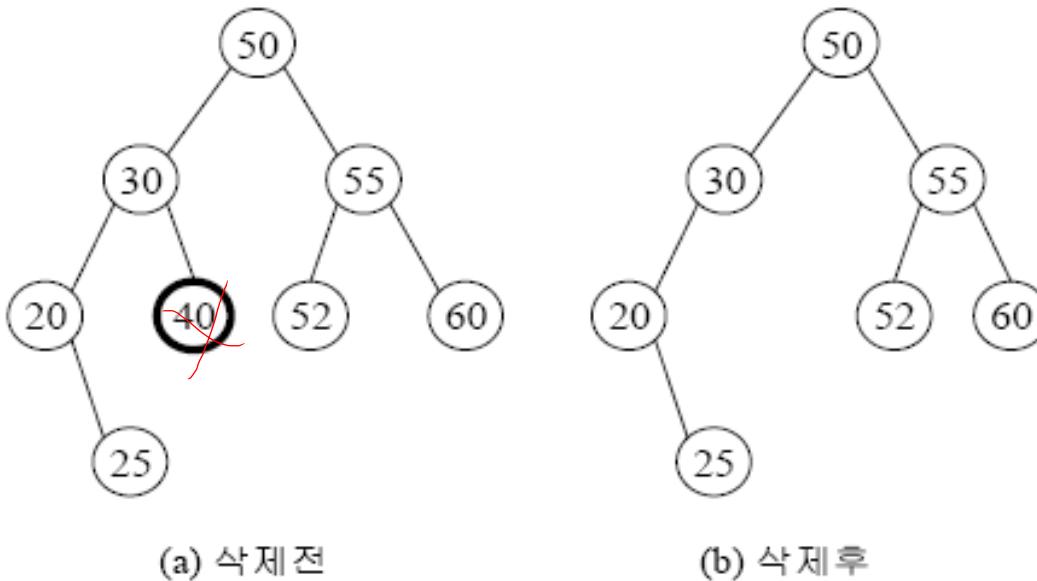


Delete in BST

- ▶ 이원 탐색 트리에서의 원소 삭제
 - ▶ 삭제하려는 원소의 키 값이 주어졌을 때
 - ▶ 이 키 값을 가진 원소를 탐색
 - ▶ 원소를 찾으면 삭제 연산 수행
 - ▶ 해당 노드의 자식수에 따른 세가지 삭제 연산
 - ▶ 자식이 없는 리프 노드의 삭제
 - ▶ 자식이 하나인 노드의 삭제
 - ▶ 자식이 둘인 노드의 삭제

▶ 자식이 없는 리프 노드의 삭제

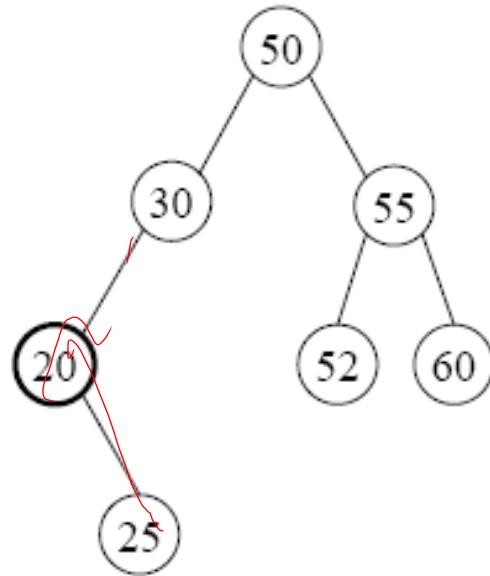
- ▶ 부모 노드의 해당 링크 필드를 널(null)로 만들고 삭제한 노드 반환
- ▶ 예 : 키값 40을 가진 노드의 삭제시



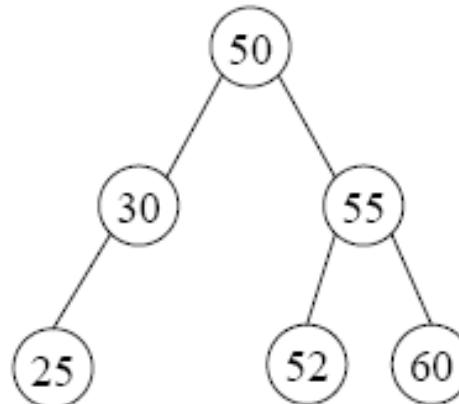
(a) 삭제전

(b) 삭제후

- ▶ 자식이 하나인 노드의 삭제
 - ▶ 삭제되는 노드 자리에 그 자식 노드를 위치
 - ▶ 예 : 원소 20을 삭제



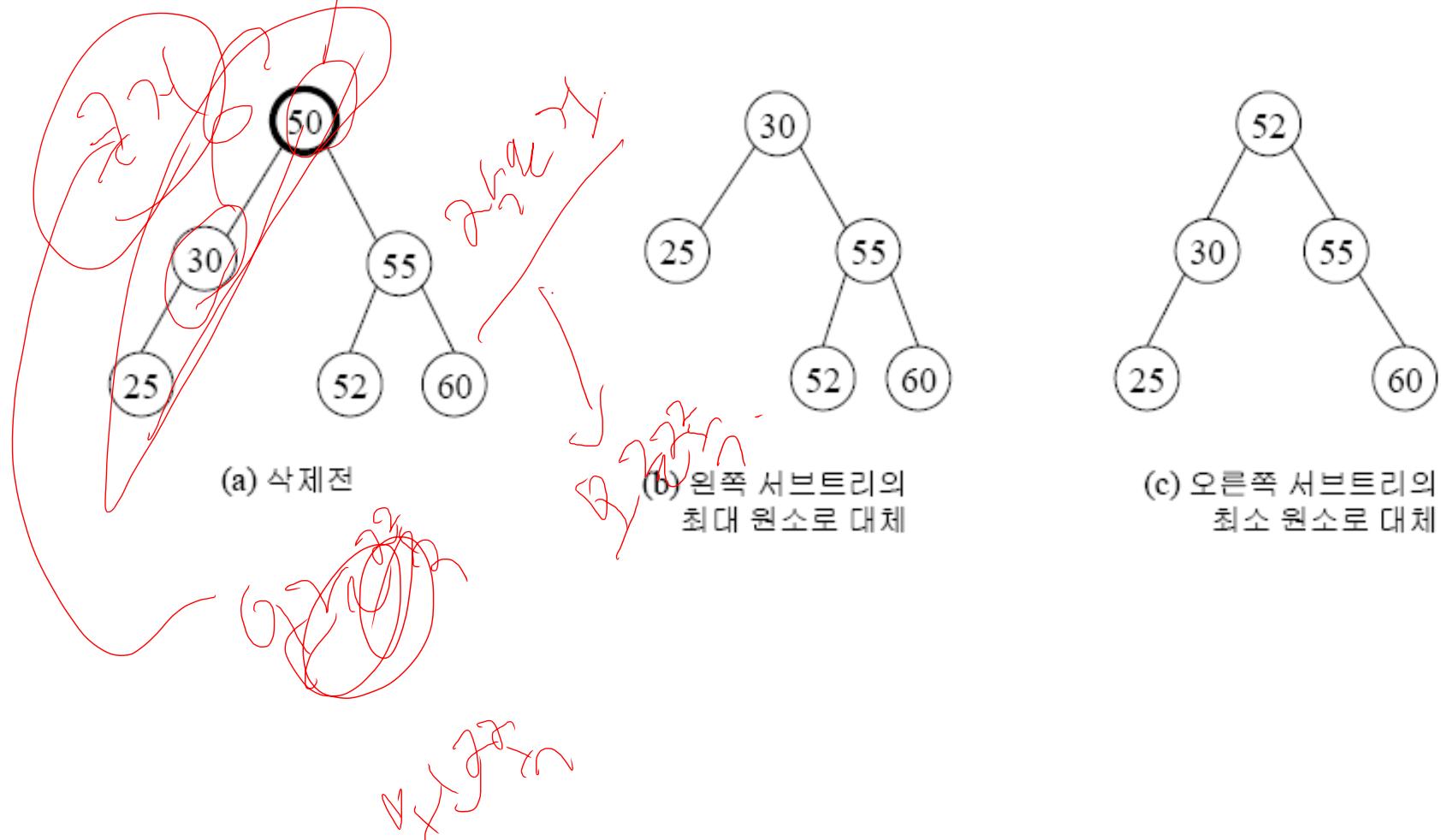
(a) 삭제전

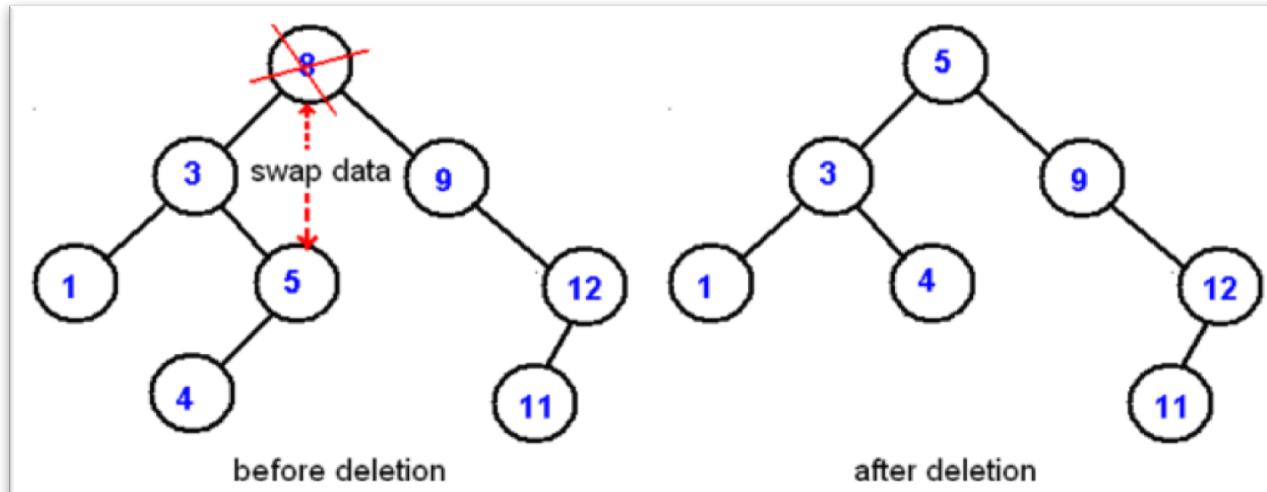
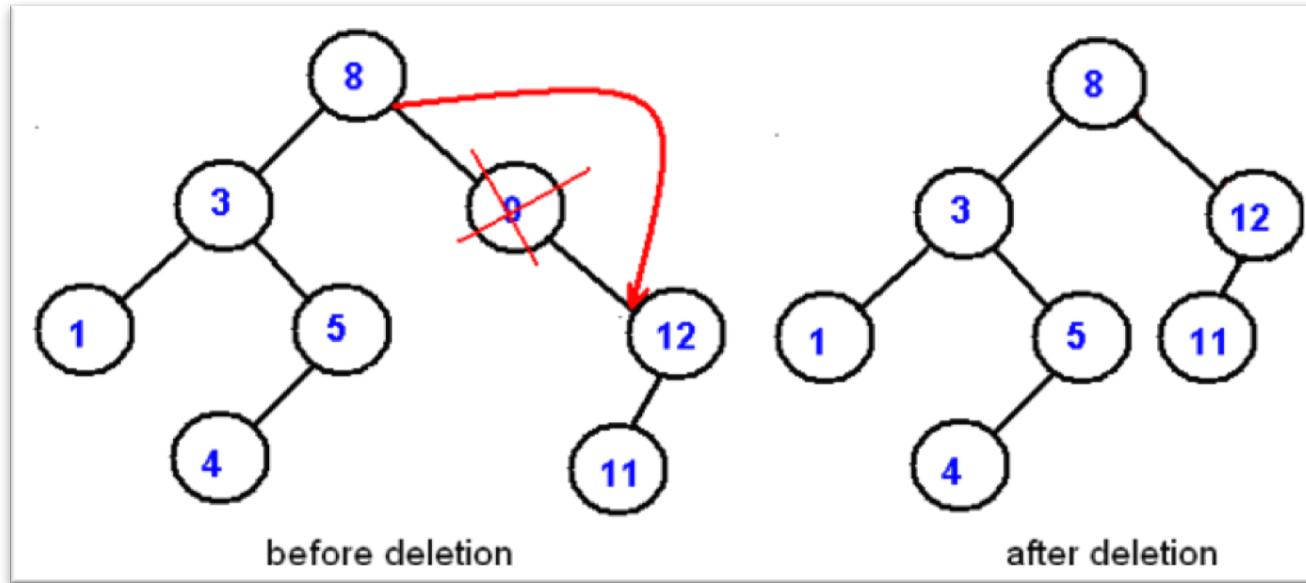


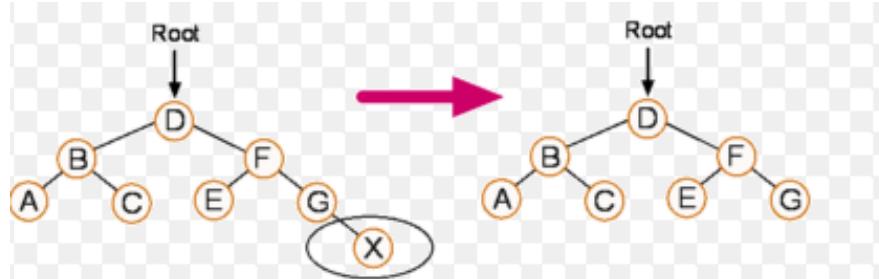
(b) 삭제후

-
- ▶ 자식이 둘인 노드의 삭제
 - ▶ 삭제되는 노드 자리
 - ▶ 왼쪽 서브 트리에서 제일 큰 원소
 - ▶ 또는 오른쪽 서브 트리에서 제일 작은 원소로 대체
 - ▶ 해당 서브 트리에서 대체 원소를 삭제
 - ▶ 대체하게 되는 노드의 차수는 1 이하가 됨

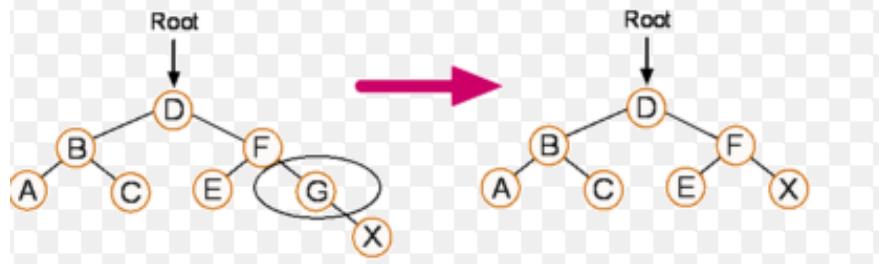
▶ 예 : 키값이 50인 루트 노드의 삭제시



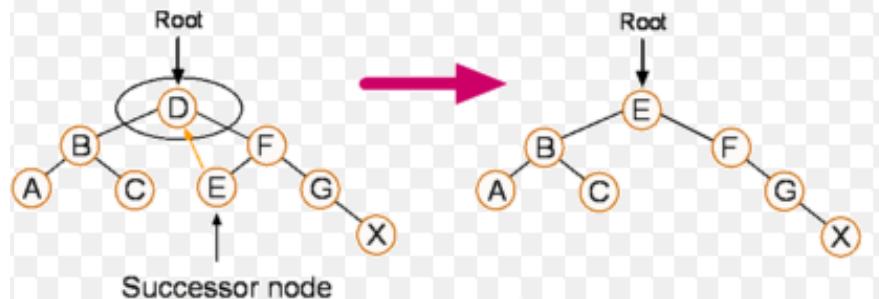




Deleting a node with a single child



Deleting a node with two children, locate the successor node on the right-hand side (or predecessor on the left) and replace the deleted node (D) with the successor (E). Finally remove the successor node.



• Delete Algorithm

```
deleteBST(B, x)
    p ← the node to be deleted;      // 주어진 키값 x를 가진 노드
    parent ← the parent node of p;    // 삭제할 노드의 부모 노드
    if (p = null) then return;       // 삭제할 원소가 없음
    case {
        p.left = null and p.right = null:          // 삭제할 노드가 리프 노드인 경우
            if (parent.left = p) then parent.left ← null;
            else parent.right ← null;
        p.left = null or p.right = null:           // 삭제할 노드의 차수가 1인 경우
            if (p.left ≠ null) then {
                if (parent.left = p) then parent.left ← p.left;
                else parent.right ← p.left;
            } else {
                if (parent.left = p) then parent.left ← p.right;
                else parent.right ← p.right;
            }
        p.left ≠ null and p.right ≠ null:          // 삭제할 노드의 차수가 2인 경우
            q ← maxNode(p.left);                  // 왼쪽 서브트리에서 최대 키값을 가진 원소를 찾기
            p.key ← q.key;
            deleteBST(p.left, p.key);
    }
end deleteBST()
```

C implementation

```
#include<stdio.h>

typedef struct TreeNode {
    char key;
    struct TreeNode* left;
    struct TreeNode* right;
} treeNode;

treeNode* insertKey(treeNode* p, char x) {
    /* insert() 함수가 사용하는 보조 순환 함수 */
    treeNode* newNode;
    if (p == NULL) {
        newNode = (treeNode*)malloc(sizeof(treeNode));
        newNode->key = x;
        newNode->left = NULL;
        newNode->right = NULL;
        return newNode;
    }
    else if (x < p->key) {
        /* x를 p의 왼쪽 서브트리에 삽입 */
        p->left = insertKey(p->left, x);
        return p;
    }
}
```

16



```

else if (x > p->key) { // x를 p의 오른쪽 서브트리에 삽입
    p->right = insertKey(p->right, x);
    return p;
} else { // key 값 x가 이미 이원 탐색 트리에 있음
    printf("Duplicated key value");
    return p;
}
}

void insert(treeNode** root, char x) {
    *root = insertKey(*root, x);
}

treeNode* find(treeNode* root, char x) { // 키 값 x를 가지고 있는 노드의 포인터를 반환
    treeNode* p;
    p = root;
    while (p != NULL) {
        if (x < p->key) {
            p = p->left;
        } else if (x == p->key) { // 키 값 x를 발견
            return p;
        } else p = p->right;
    }
    printf("No such key value is found");
    return p;
}

```



```

void printNode(treeNode* p) { // printTree() 함수에 의해 사용되는 순환 함수
    if (p != NULL) {
        printf("(");
        printNode(p->left); // leftsubtree를 프린트
        printf("%c", p->key);
        printNode(p->right); // rightsubtree를 프린트
        printf(")");
    }
}

void printTree(treeNode* root) { // 서브트리 구조를 표현하는 괄호 형태로 트리를
    // 프린트
    printNode(root);
    printf( "\n");
}

void freeNode(treeNode* p) { // 노드에 할당된 메모리를 반환
    if (p != NULL) {
        freeNode(p->left);
        freeNode(p->right);
        free(p);
    }
}

void freeTree(treeNode* root) { // 트리에 할당된 메모리를 반환
    freeNode(root);
}

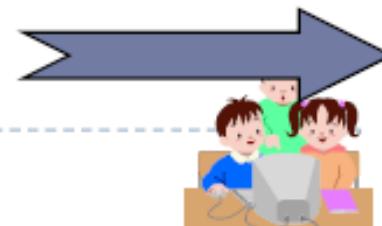
```



```
int main() { // 예제 실행을 위한 main() 함수
    treeNode* root = NULL; // 공백 이원 탐색 트리 root를 선언
    treeNode* N = NULL;
    treeNode* P = NULL;

    /* 그림 7.6의 BST를 구축 */
    insert(&root, 'S');
    insert(&root, 'J');
    insert(&root, 'B');
    insert(&root, 'D');
    insert(&root, 'U');
    insert(&root, 'M');
    insert(&root, 'R');
    insert(&root, 'Q');
    insert(&root, 'A');
    insert(&root, 'G');
    insert(&root, 'E');

    /* 구축된 BST를 프린트 */
    printf("The Tree is ");
    printTree(root);
    printf("\n");
```



```
/* key 값 'R'을 탐색하고 프린트 */
printf("Search For 'R'\n");
N = find(root, 'R');
printf("Key of node found = %c\n", N->key);
printf("\n");

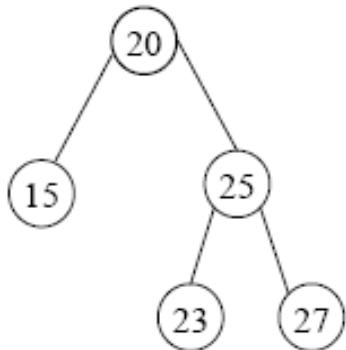
/* key 값 'C'를 탐색하고 프린트 */
printf("Search For 'C'\n");
P = find(root, 'C');
if (P != NULL) {
    printf("Key of node found = %c", P->key);
} else {
    printf("Node that was found = NULL");
}
printf("\n");

/* 트리에 할당된 메모리를 반환 */
freeTree(root);
return 0;
}
```

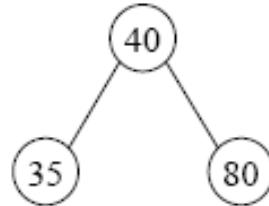
Merge Binary Search Trees

- ▶ 3원 결합 : threeJoin (aBST, x, bBST, cBST)
 - ▶ 이원 탐색 트리 aBST와 bBST에 있는 모든 원소들과 키값 x를 갖는 원소를 루트 노드로 하는 이원 탐색 트리 cBST를 생성
 - ▶ 가정
 - ▶ aBST의 모든 원소 $< x <$ bBST의 모든 원소
 - ▶ 결합이후에 aBST와 bBST는 사용하지 않음
- ▶ 3원 결합의 연산 실행
 - ▶ 새로운 트리 노드 cBST를 생성하여 key값으로 x를 지정
 - ▶ left 링크 필드에는 aBST를 설정
 - ▶ right 링크 필드에는 bBST를 설정

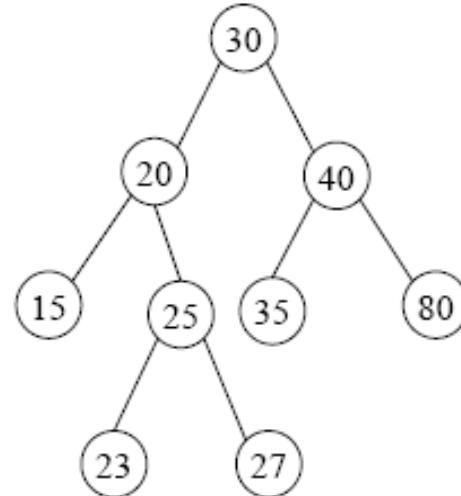
▶ 3원 결합의 예



(a) aBST



(b) bBST



(c) cBST

- ▶ 연산 시간 : $O(1)$
- ▶ 이원 탐색 트리의 높이
 $: \max\{\text{height}(a\text{BST}) , \text{height}(b\text{BST})\} + 1$



▶ 2원 결합 : twoJoin(aBST, bBST, cBST)

- ▶ 두 이원 탐색 트리 aBST와 bBST를 결합하여 aBST와 bBST에 있는 모든 원소들을 포함하는 하나의 이원 탐색 트리 cBST를 생성
- ▶ 가정
 - ▶ aBST의 모든 키값 < bBST의 모든 키값
 - ▶ 연산후 aBST와 bBST는 사용 하지 않음

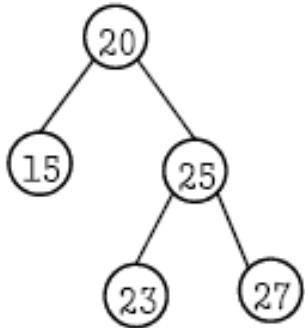
▶ 2원 결합 연산 실행

- ▶ aBST나 bBST가 공백인 경우
 - ▶ cBST : 공백이 아닌 aBST 혹은 bBST
- ▶ aBST와 bBST가 공백이 아닌 경우
 - ▶ 두 이원 탐색 트리 결합 방법은 두 가지로 나뉨

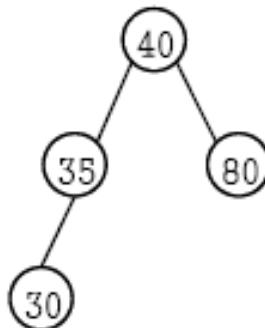


- ▶ aBST에서 키 값이 가장 큰 원소를 삭제

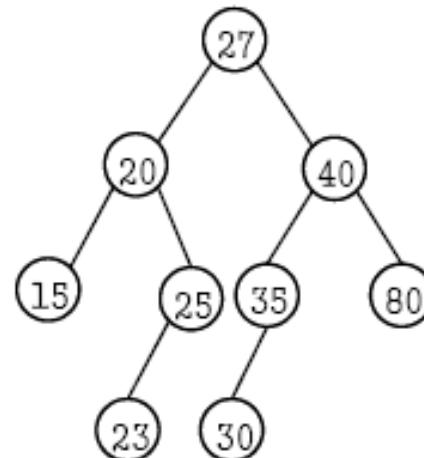
- ▶ 이 결과 이원 탐색트리 : aBST'
- ▶ 삭제한 가장 큰 키값 : max



(a) aBST



(b) bBST

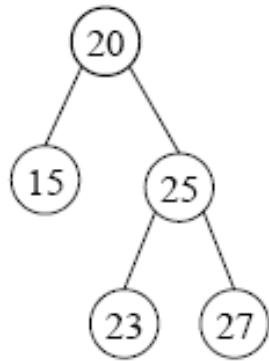


(c) cBST

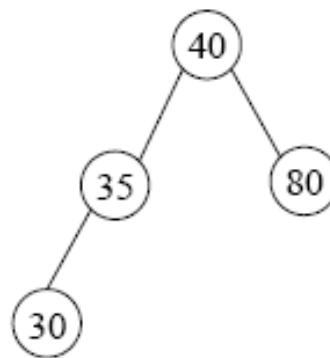
- ▶ 실행 시간 : $O(\text{height } (\text{aBST}))$
- ▶ cBST의 높이 : $\max\{\text{height}(\text{aBST}), \text{height}(\text{bBST})\} + 1$

▶ bBST에서 가장 작은 키값을 가진 원소를 삭제

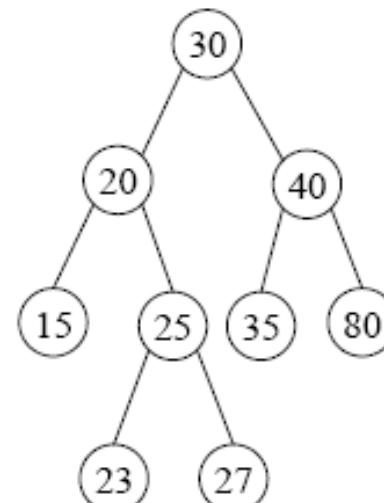
- ▶ 이 결과 이원 탐색 트리 : aBST'
- ▶ 삭제한 가장 작은 키값 : min
- ▶ threeJoin(aBST, min, bBST', cBST)를 실행



(a) aBST



(b) bBST



(c) cBST

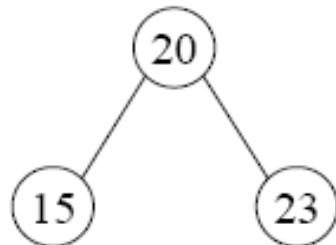
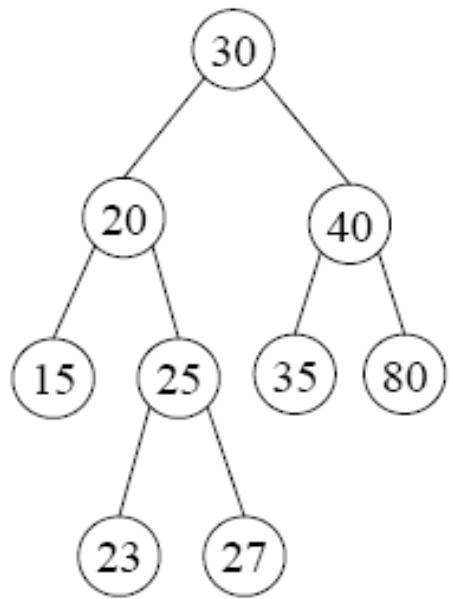
Split in BST

- ▶ 분할 : `split(aBST, x, bBST, cBST)`
 - ▶ aBST 를 주어진 키값 x를 기준으로
두 이원 탐색 트리 bBST와 cBST로 분할
 - ▶ bBST : x보다 작은 키값을 가진 aBST의 모든 원소 포함
 - ▶ cBST : x보다 큰 키값을 가진 aBST의 모든 원소 포함
 - ▶ bBST와 cBST는 각각 이원 탐색 트리 성질을 만족
 - ▶ 키값 x가 aBST에 있으면 true 반환 , 아니면 false 반환

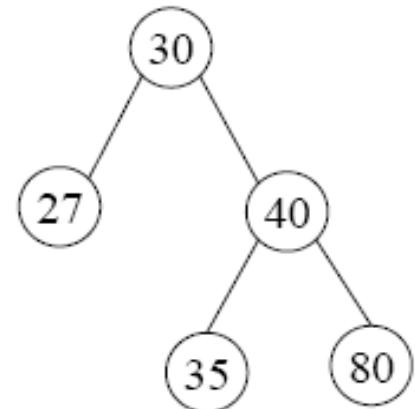
▶ 분할 연산 실행

- ▶ aBST의 루트 노드가 키값 x 를 가질 때
 - ▶ 왼쪽 서브트리 : bBST
 - ▶ 오른쪽 서브트리 : cBST
 - ▶ True 반환
- ▶ $x <$ 루트 노드의 키값
 - ▶ 루트와 그 오른쪽 서브트리는 cBST에 속한다.
- ▶ $x >$ 루트 노드의 키값
 - ▶ 루트와 그 왼쪽 서브트리는 bBST에 속한다.
- ▶ 키값 x 를 가진 원소를 탐색하면서 aBST를 아래로 이동

▶ Split(aBST , 25 , bBST , cBST)



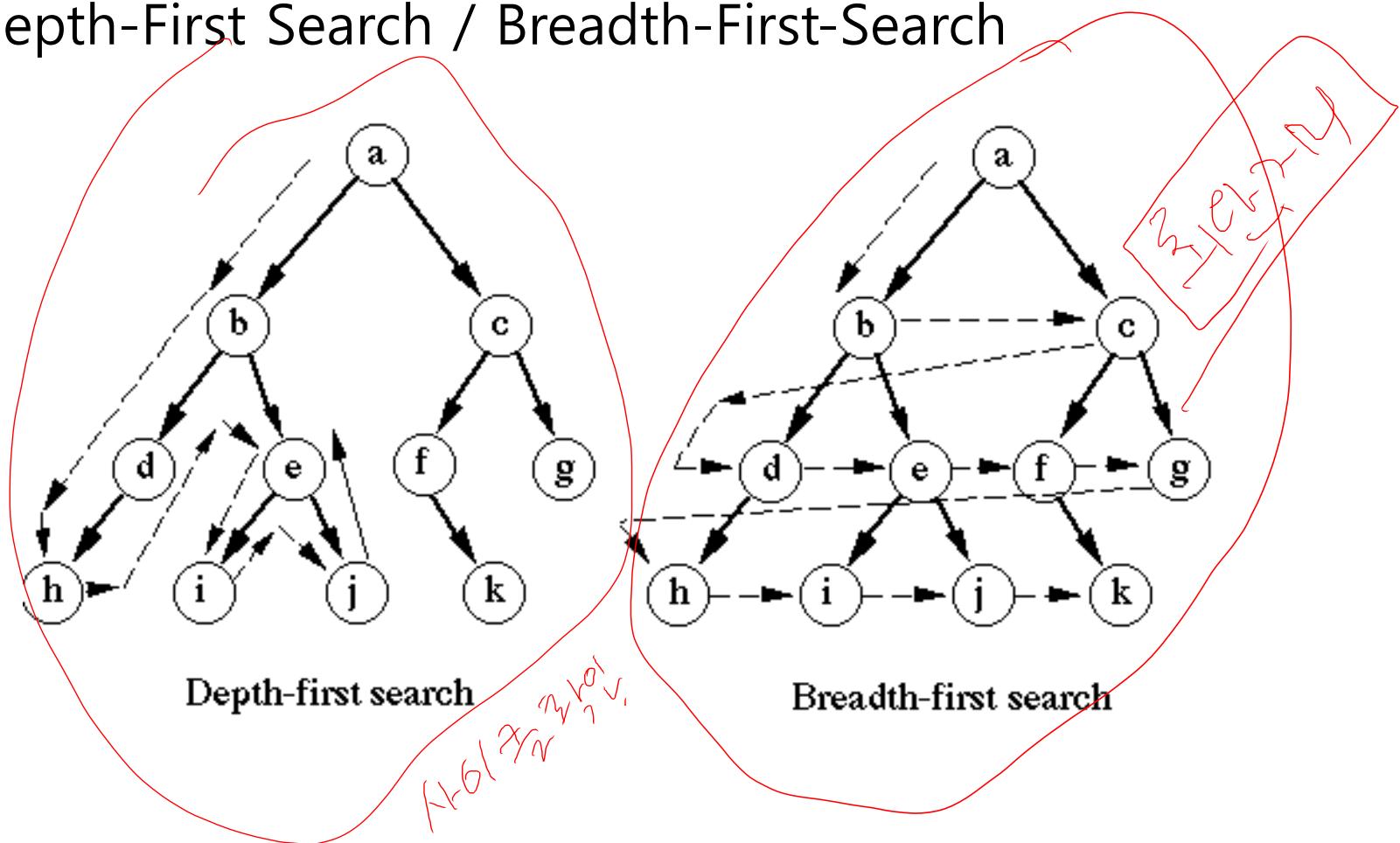
(b) bBST



(c) cBST

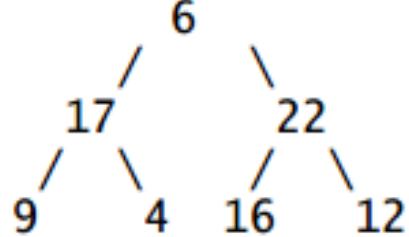
Tree Search

- Depth-First Search / Breadth-First-Search



Quiz. <http://www.fas.harvard.edu/~cscie119/>

2. If the binary tree below is printed by a preorder traversal, what will the result be?



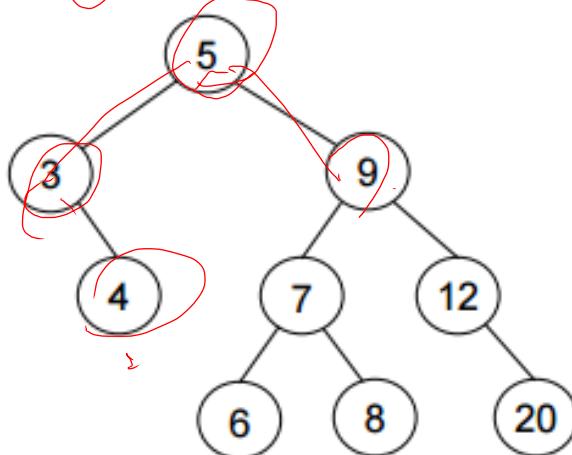
Handwritten notes in red:

- 6 (circled)
- 17 (circled)
- 9 (circled)
- 4 (circled)
- 16 (circled)
- 22 (circled)
- 12 (circled)

- A. 9 4 17 16 12 11 6
- B. 9 17 6 4 16 22 12
- C. 6 9 17 4 16 22 12
- D. 6 17 22 9 4 16 12
- E. 6 17 9 4 22 16 12

(C) circled

10. The binary search tree shown below was constructed by inserting a sequence of items into an empty tree.



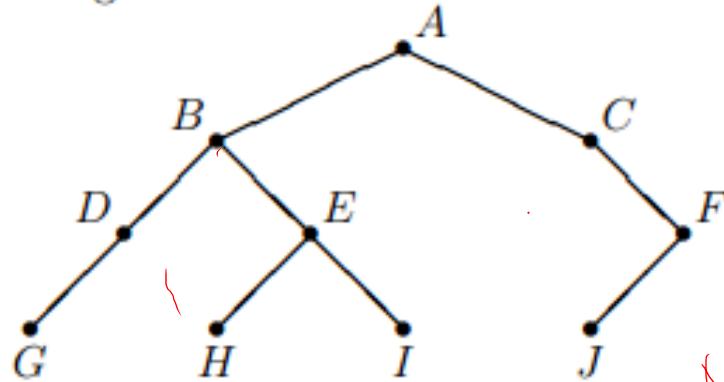
Which of the following input sequences will *not* produce this binary search tree?

- A. 5 3 4 9 12 7 8 6 20
- B. 5 9 3 7 6 8 4 12 20
- C. 5 9 7 8 6 12 20 3 4
- D. 5 9 7 3 8 12 6 4 20
- E. 5 9 3 6 7 8 4 12 20

Quiz.

<http://www.cs.cornell.edu/courses/cs2110/2011fa/>

9. (6 points) List the sequence of nodes visited by preorder, inorder, and postorder traversals of the following tree:

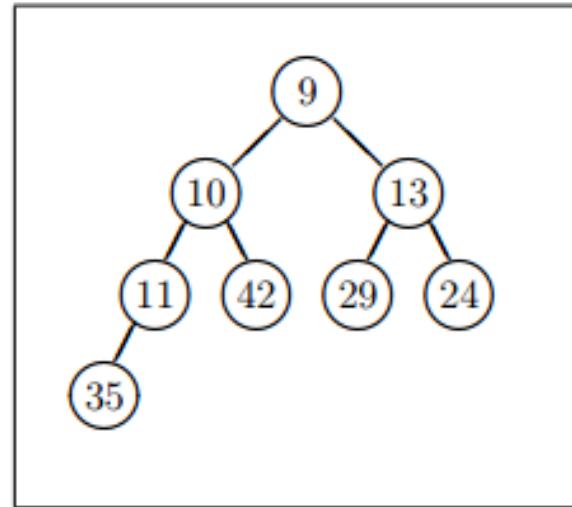
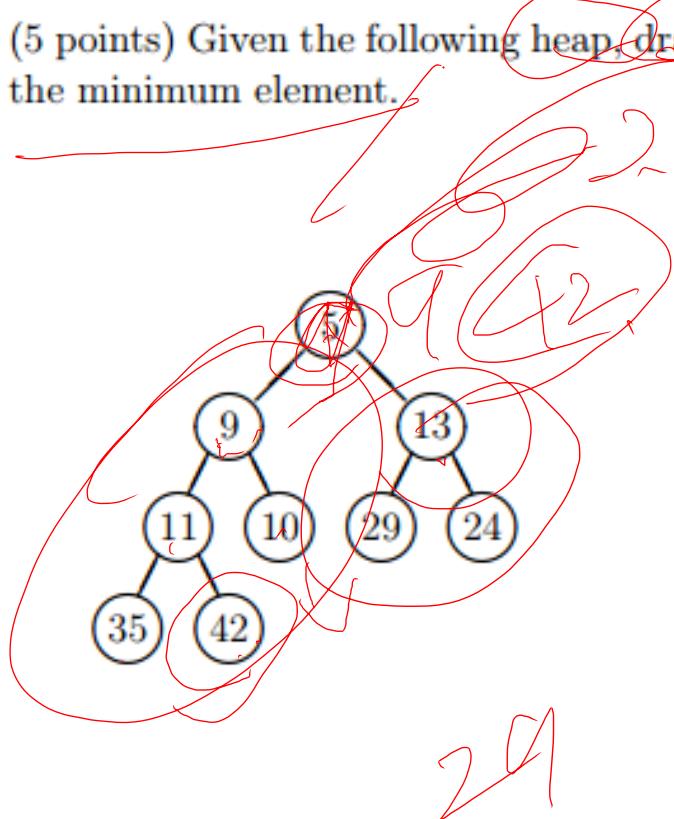


- (a) preorder: **ABDGEHICFJ**
(b) inorder: **GDBHEIACJF**
(c) postorder: **GDHIEBJFCA**

Quiz.

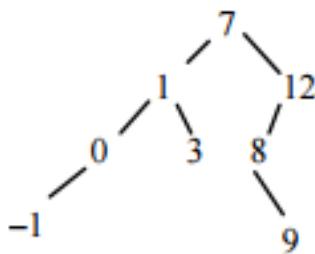
<http://www.cs.cornell.edu/courses/cs2110/2011fa/>

11. (5 points) Given the following heap, draw the heap that would result after deleting the minimum element.

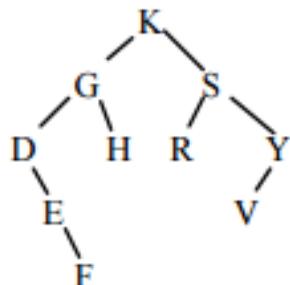


Quiz. <http://www.cs.toronto.edu/~hojjat/148s07>

- (i) The integers 7, 1, 12, 8, 3, 0, -1, 9 are inserted in that order into an initially empty binary search tree. Draw the tree after the last insertion. (No explanation is required.)



- (j) Here is a binary tree:



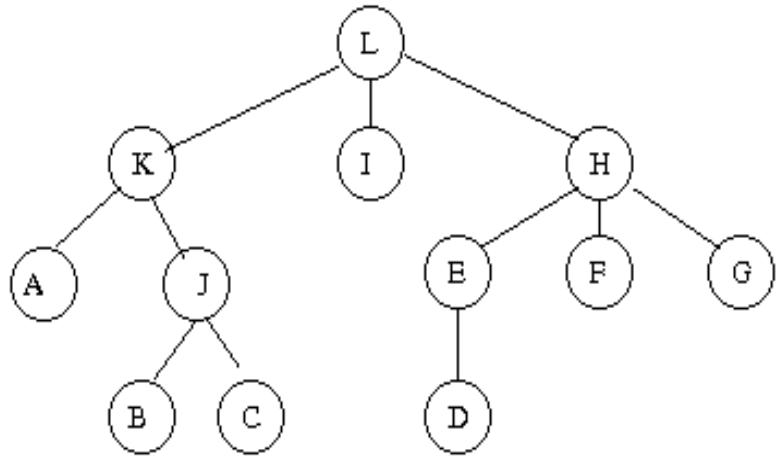
Write the node labels in the order they would be printed in an in-order traversal of the tree. (No explanation is required.)

D E F G H K R S V Y

Quiz. <http://cs.nyu.edu/courses/fall11/>

Question 1

List the nodes of the tree below in preorder, postorder, and breadth-first order.



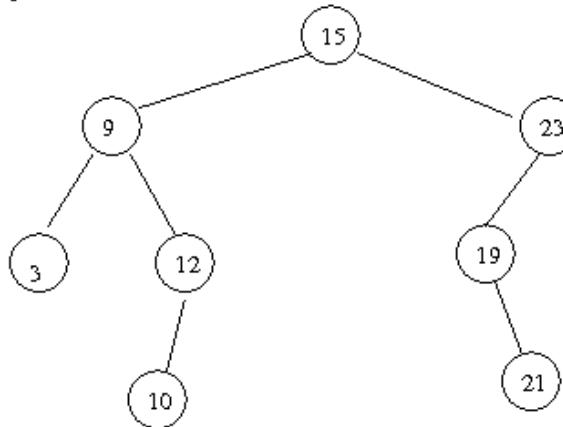
Answer: Preorder: L,K,A,J,B,C,I,H,E,D,F,G

Postorder: A,B,C,J,K,I,D,E,F,G,H,L

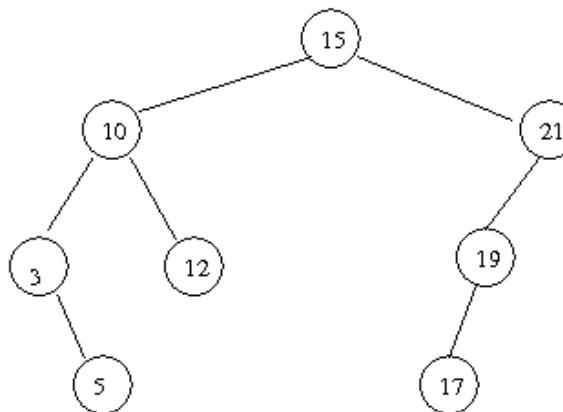
Breadth-first: L,K,I,H,A,J,E,F,G,B,C,D

Question 2

In the binary search tree below, carry out the following operations in sequence: Add 5, add 17, delete 23, delete 9.



Answer: There is more than one way to do the deletes, so the final answer is not unique, but here is one:

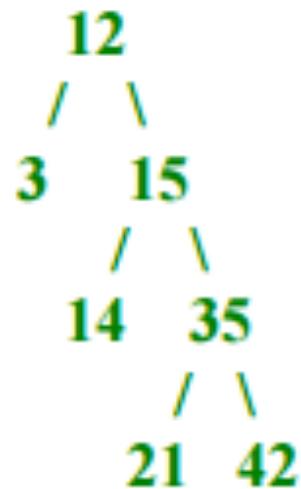


Quiz.

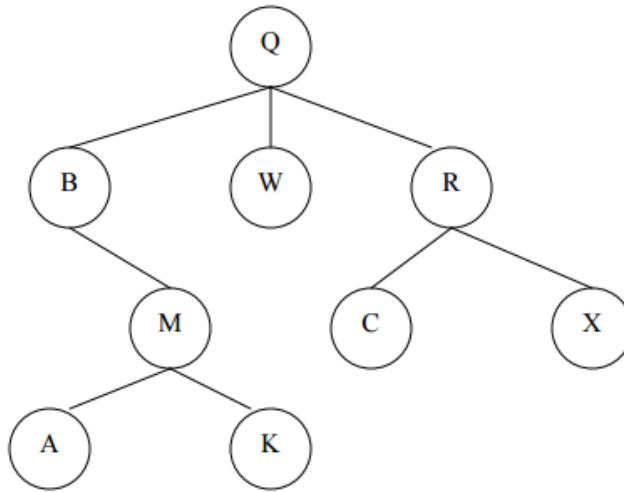
<http://courses.cs.washington.edu/courses/cse143/>

Question 1. (8 points) (a) Draw the binary search tree that is created if the following numbers are inserted in the tree in the given order.

12 15 3 35 21 42 14



Question 9. (7 points) Consider the following tree, which is not a binary tree.



(a) Which node(s) is(are) the roots of this tree?

Q

(b) Which node(s) is(are) the leaves of this tree?

A K W C X

(c) Write down the nodes in the order they are reached if we perform a *postorder* traversal of this tree starting with node Q.

A K M B W C X R Q

Quiz. chara.cs.illinois.edu/cs225/

A post-order traversal of the following tree visits the nodes in which order?

