COMP319 Algorithms 1
Lecture 9
Binary Search Tree

Instructor: Gil-Jin Jang

Slide credits:

홍석원, 명지대학교, Discrete Mathematics, Spring 2013

김한준, 서울시립대학교, 자료구조 및 실습, Fall 2016

J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

기본용어

이진 트리(binary tree)

이진 탐색 트리(binary search tree)

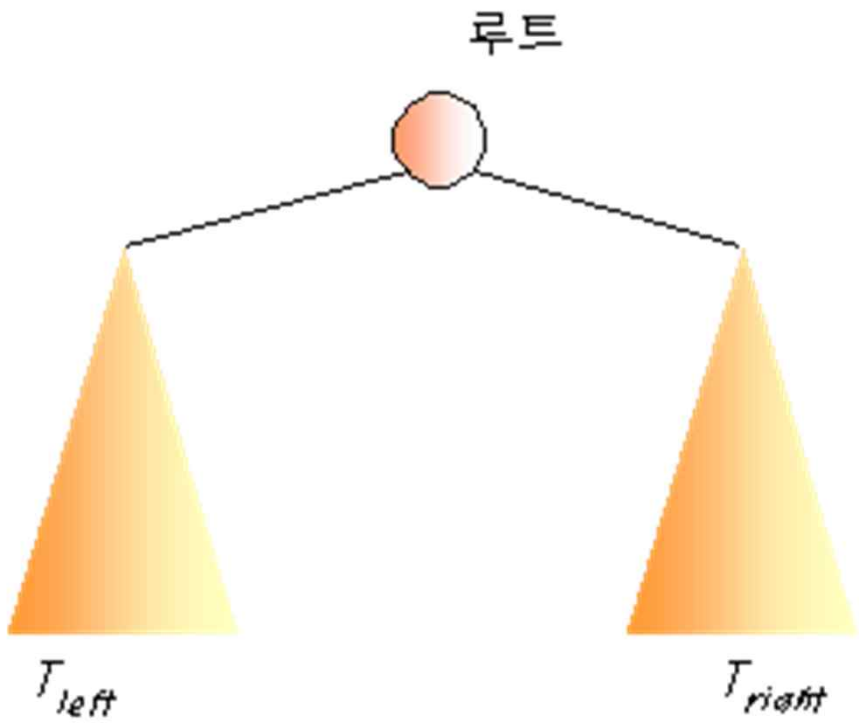
복습: BINARY TREE

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

복습: BINARY TREE

이진 트리 (binary tree)

- 이진 트리(binary tree) : 모든 노드가 2개의 서브 트리를 가지고 있는 트리
  - 서브트리는 공집합일수 있다.
- 이진트리의 노드에는 최대 2개까지의 자식 노드가 존재
- 모든 노드의 차수가 2 이하가 된다
  - 구현하기가 편리함
- 이진 트리에는 서브 트리간의 순서가 존재



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

복습: BINARY TREE

이진 트리의 성질

- 노드의 개수가 *N* 개이면 간선의 개수는 *N*-1 개

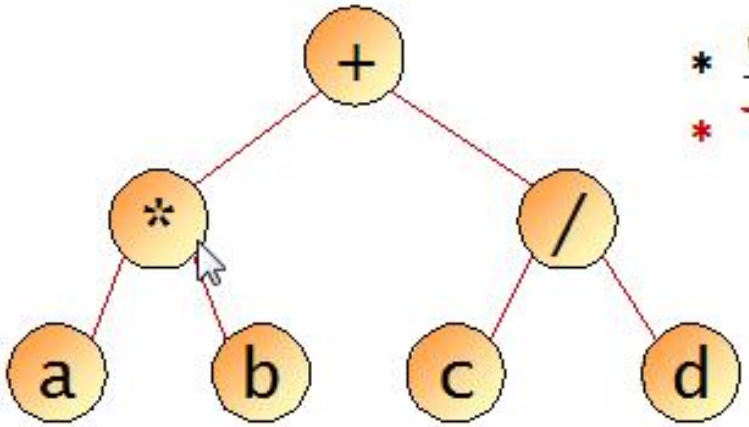


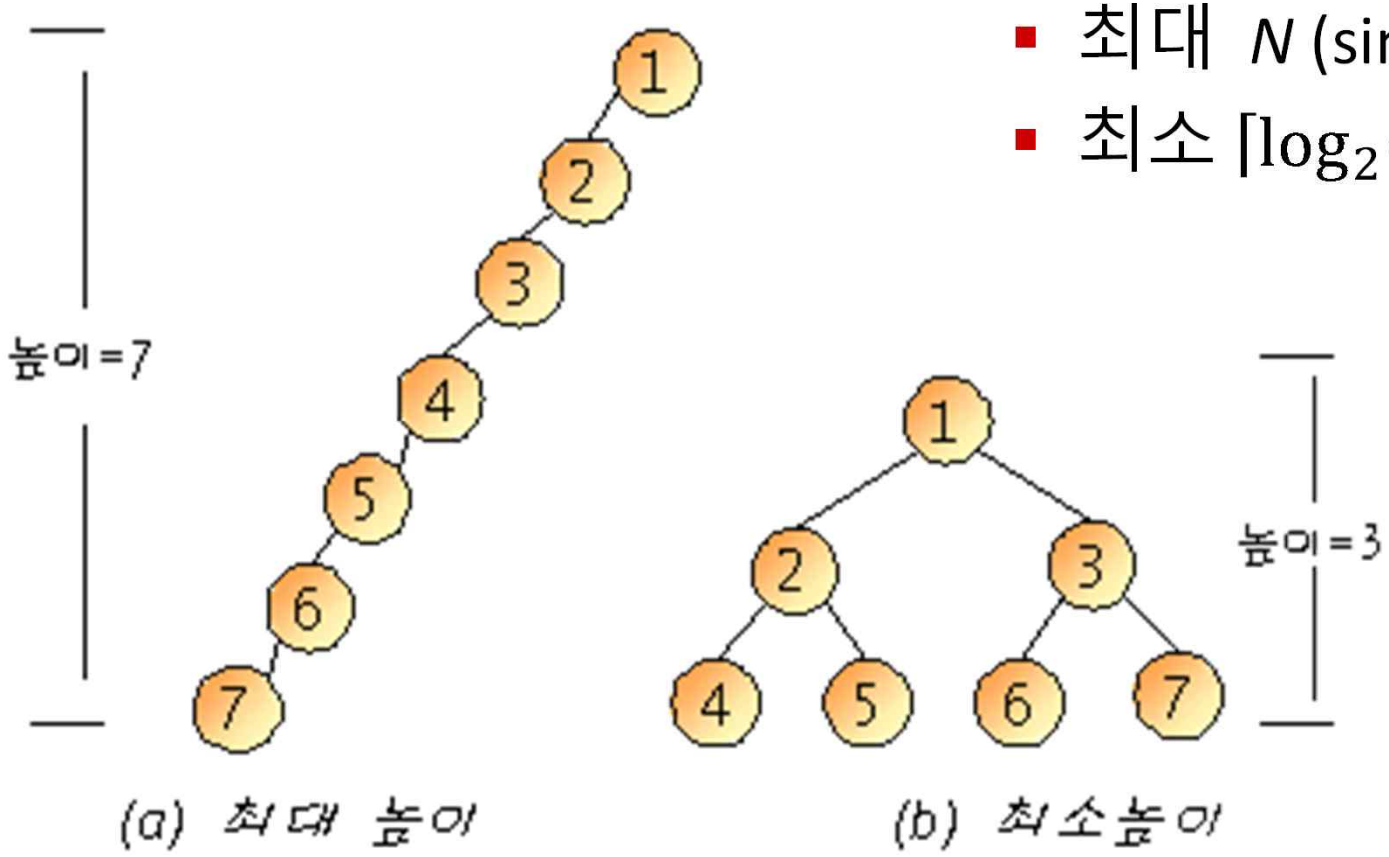
그림 7.10 노드의 개수와 간선의 개수와의 관계

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

복습: BINARY TREE

이진 트리의 성질

- N개의 노드를 가지는 이진트리의 높이
  - 최대 N (single child)
  - 최소 $\lceil \log_2(N+1) \rceil$ (full)

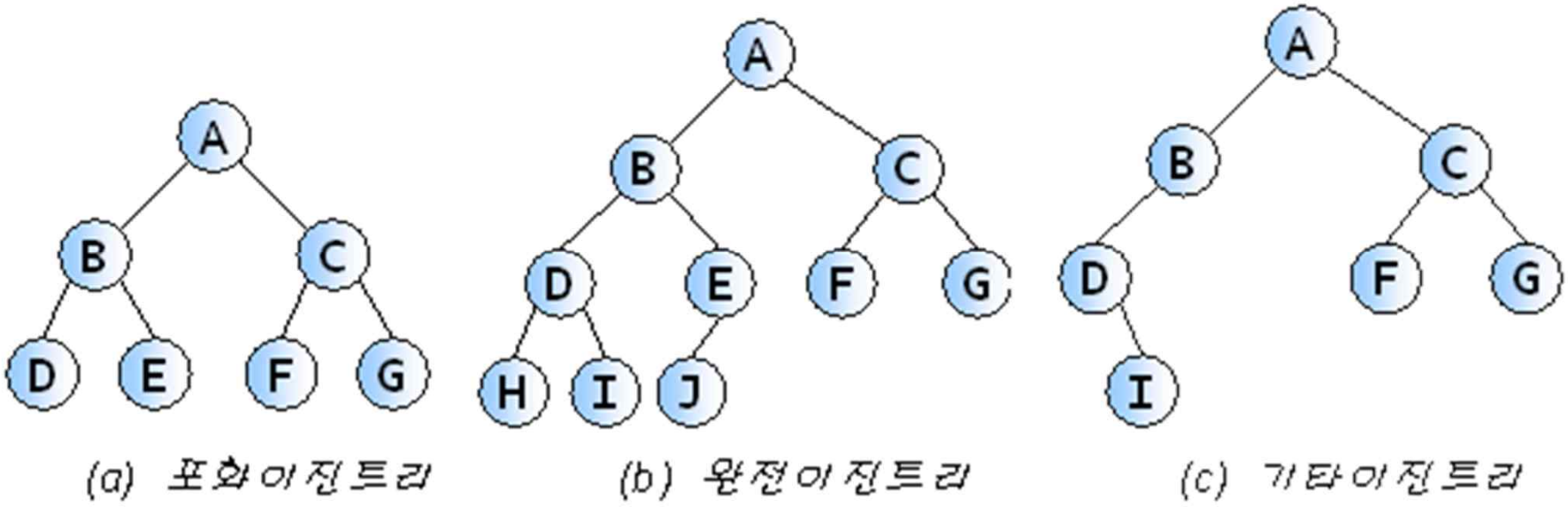


Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

복습: BINARY TREE

이진 트리의 분류

- 포화 이진 트리(full binary tree)
- 완전 이진 트리(complete binary tree)
- 기타 이진 트리

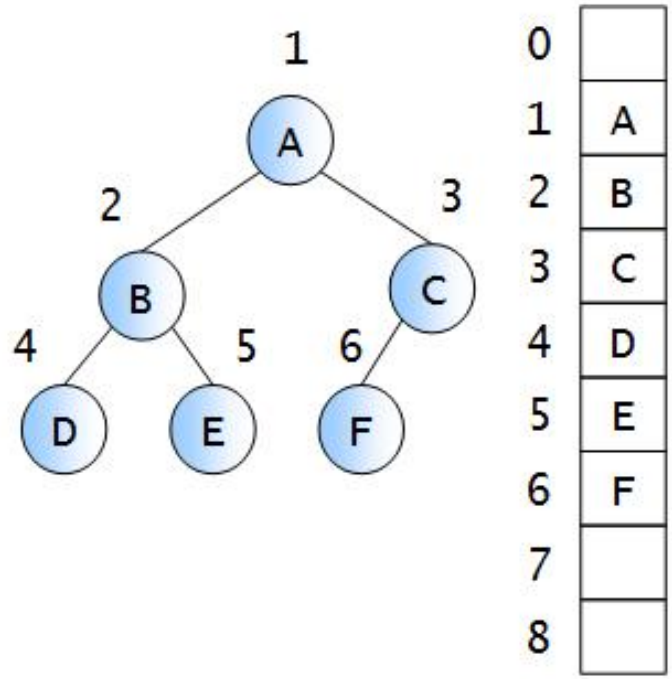


(a) 포화이진트리 (b) 완전이진트리 (c) 기타이진트리

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

부모와 자식 인덱스 관계

- 노드 *i*의 부모 노드 인덱스 = $i/2$
- 노드 *i*의 왼쪽 자식 노드 인덱스 = $2i$
- 노드 *i*의 오른쪽 자식 노드 인덱스 = $2i+1$

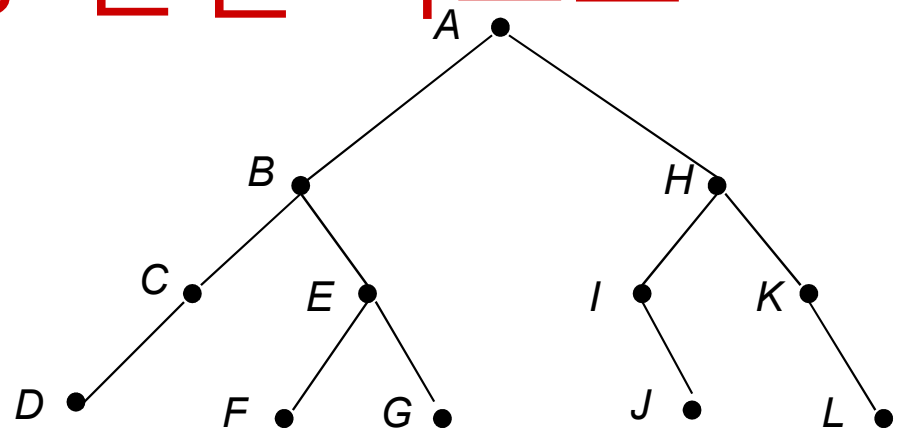


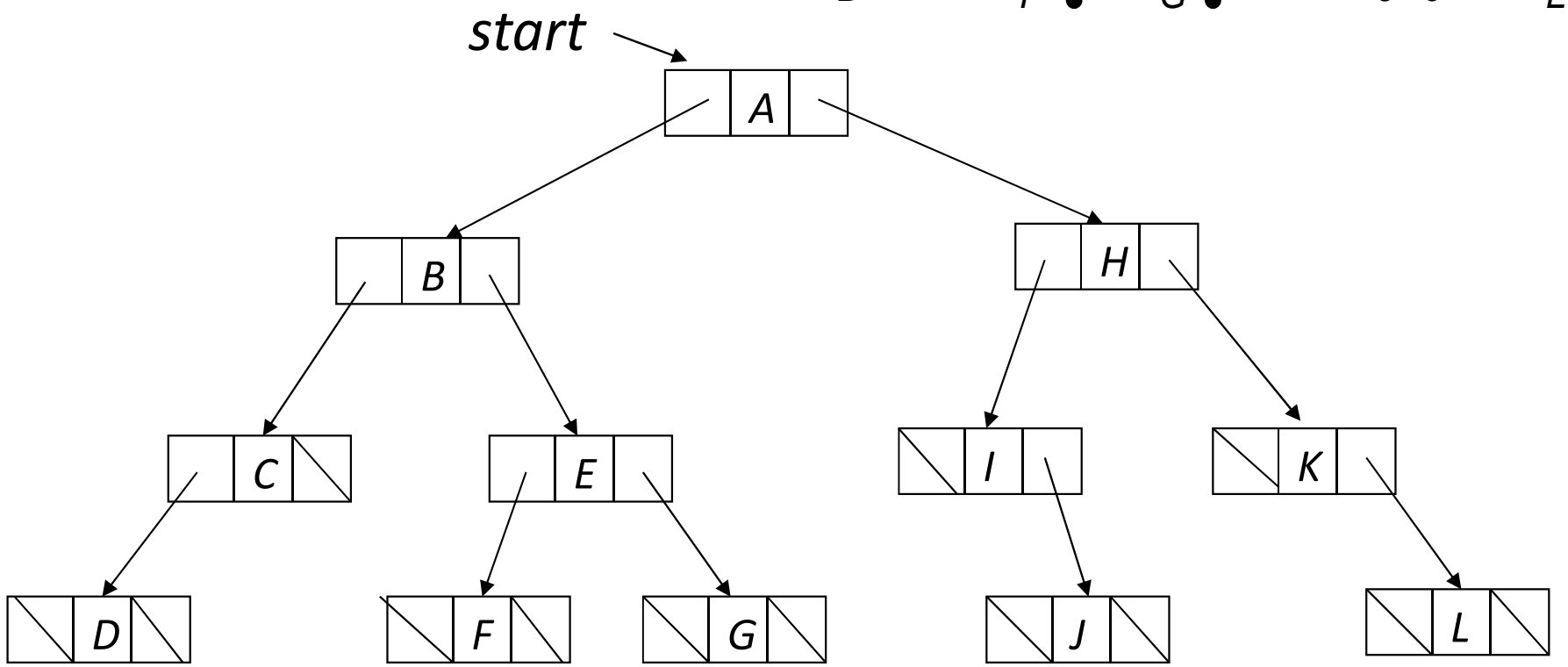
인덱싱의 편이성을 위해 맨 처음 저장공간을 사용하지 않는다.
인덱스 0를 root index로 간주한다.
만약 맨 처음 공간을 사용한다면 루트와 첫번째를 구분하기 위한 다른 방법이 필요하다.
Left(i) = 2i+1
Right(i) = 2(i+1)
Parent(i) = (i+1)/2

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

복습: BINARY TREE

트리구현: 이중 연결 리스트





Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

기본용어

이진 트리(binary tree)

이진 탐색 트리(binary search tree)

BINARY SEARCH TREE

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Review: 순차 검색(sequential search)

리스트의 다음과 같은 키 값들이 존재한다고 하자.

$\{10, 5, 13, 23, 3, 15, 4, 20, 32, 1\}$

이 리스트에서 키 값이 15인 데이터 요소를 찾기 위해서 이 리스트의 첫 번째 키 값부터 차례대로 비교하며 찾아 나가는 것을 순차 검색이라고 한다.

리스트에 N개의 데이터 요소가 존재하면 평균 비교 회수는 N/2이 된다.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Review: 이진 탐색(binary search)

만약 리스트의 키 값이 정렬되어 있다면 순차 검색 보다 빠른 시간 안에 검색을 할 수 있다.

{1, 3, 4, 5, 10, 13, 15, 20, 23, 32}

먼저 찾고자 하는 키 값을 리스트의 중간에 위치한 값, 13과 비교한다. 만약 찾고자 하는 키 값이 13 보다 작으면 이 값은 13보다 왼쪽에 위치하고 있으며, 13보다 크다면 13의 오른쪽에 위치하고 있다. 따라서 다음 단계에서 13보다 왼쪽에 있는 값들, 혹은 오른쪽에 있는 값들을 갖고 같은 절차를 반복한다.

최대 비교 회수는 리스트의 수가 N이라면 $\log_2 N$ *이다.*

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

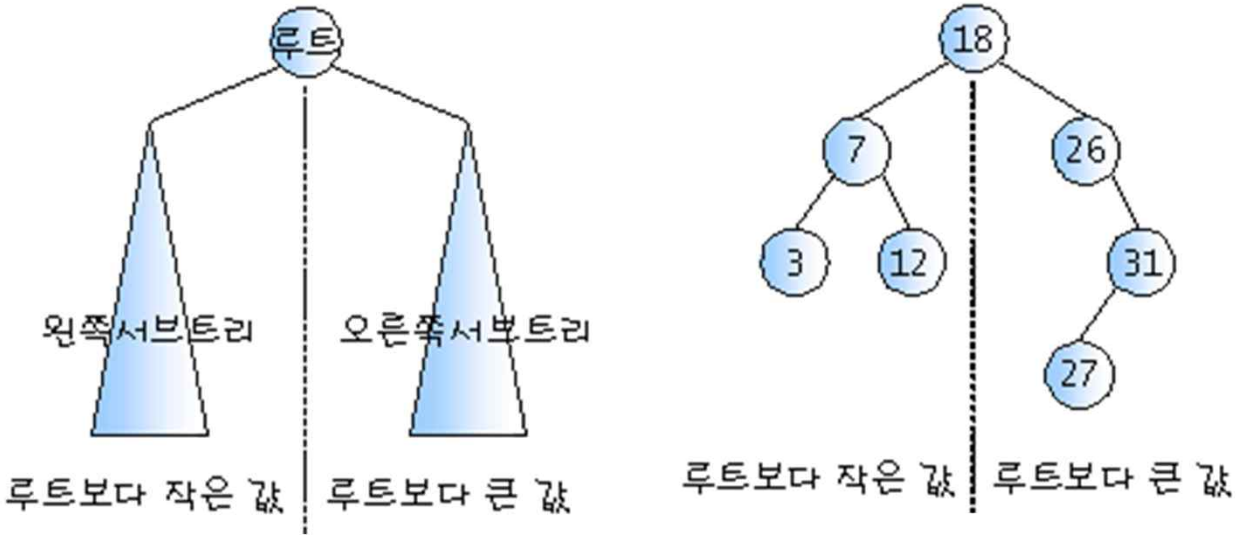
Review: 이진 탐색 알고리즘

[알고리즘] 이진 탐색

```
while (리스트 구간의 크기 > 0)
    구간의 중간값을 구한다.
    if(구간의 중간값 = 키값)
            탐색 종료
    else if(구간의 중간값 > 키값)
            오른쪽 구간 선택
    else
            왼쪽 구간 선택
```

이러한 이진 탐색을 하기 위해서는 먼저 리스트가 순서대로 정렬되어 있어야 한다. 따라서 탐색 전에 순서대로 정렬하기 위한 시간이 요구된다. 그런데 이진 검색을 하기 위해서 이진 탐색 트리를 사용하면 순서대로 정렬하는 것과 이진 탐색을 하는 것이 용이하게 구현될 수 있다.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I



BST: BINARY SEARCH TREE

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Binary Search Tree (BST)

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

- 이진 탐색 트리(binary search tree, BST)의 정의
  - 이진 트리
  - 공백이 아니면 다음 성질을 만족
    - 왼쪽 서브 트리 원소들의 키 < 루트의 키 (*등호도 가능)
    - 오른쪽 서브 트리 원소들의 키 > 루트의 키 (*등호도 가능)
    - 왼쪽 서브 트리와 오른쪽 서브 트리: 이진 탐색 트리
    - 구현방법에 따라 모든 원소는 상이한 키를 가지도록 강제할 수도 있다.

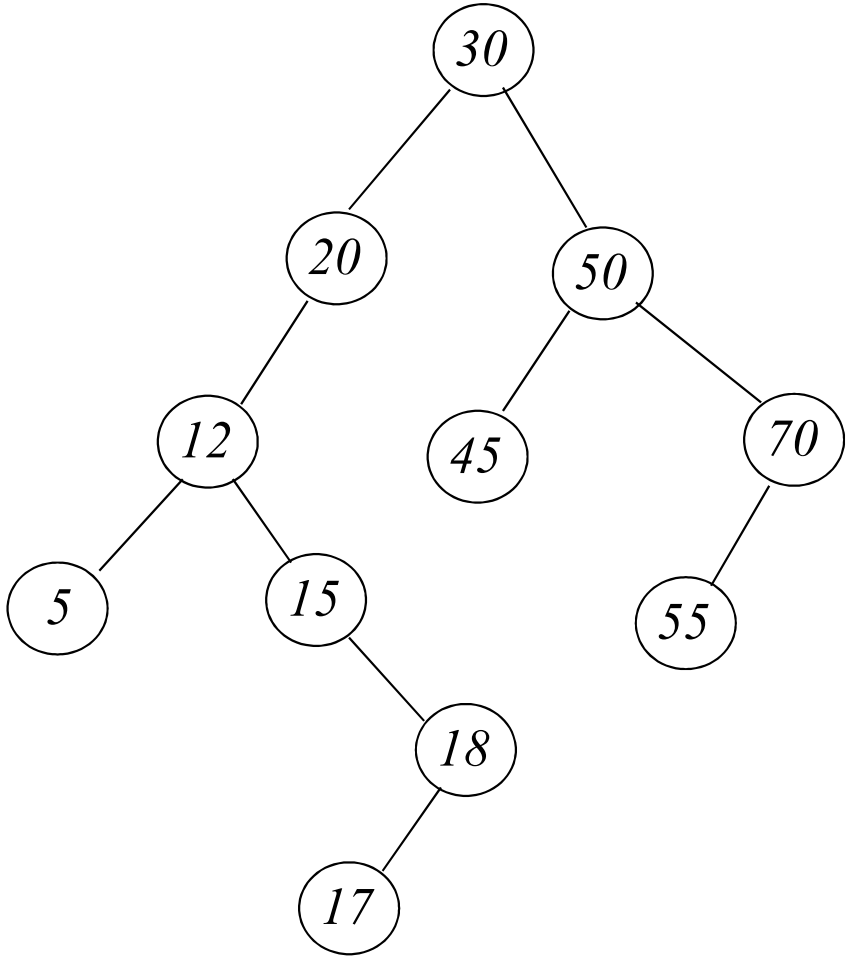
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

이진 탐색 트리의 성질

- 이진 트리 $G=(V,E)$에서 잎(leaf)을 제외한 모든 정점 v와, v의 왼쪽 자식 v_L, 오른쪽 자식 v_R의 키 값이 다음의 관계식을 만족하면 이진 탐색 트리이다.

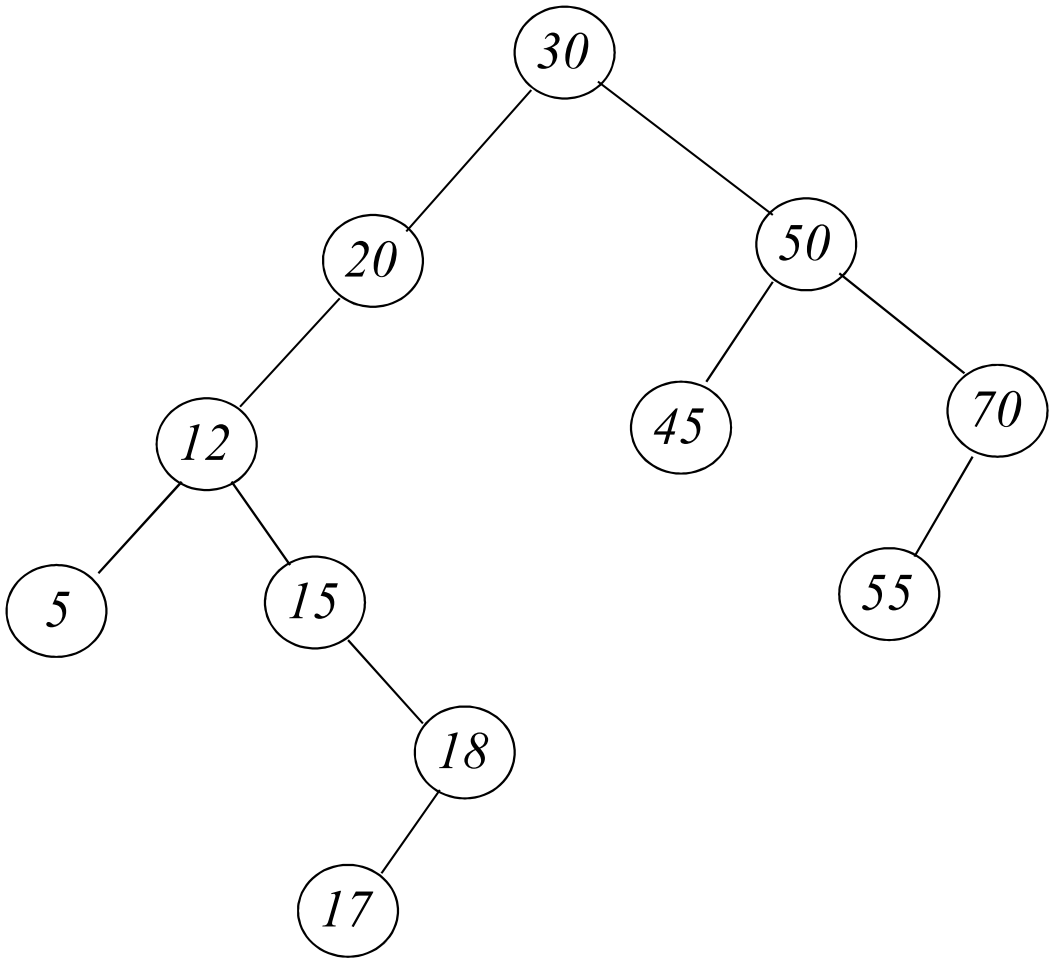
$$key(v_L) \leq key(v) \leq key(v_R)$$

- 이 식의 부등호는 트리 내에 동일한 값을 갖는 정점들이 없는 경우에 해당한다.
- 만약 트리에 동일한 값을 갖는 정점들이 존재하면 그 정점들은 어느 것을 부모 정점 혹은 자식 정점에 놓더라도 상관이 없다.



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

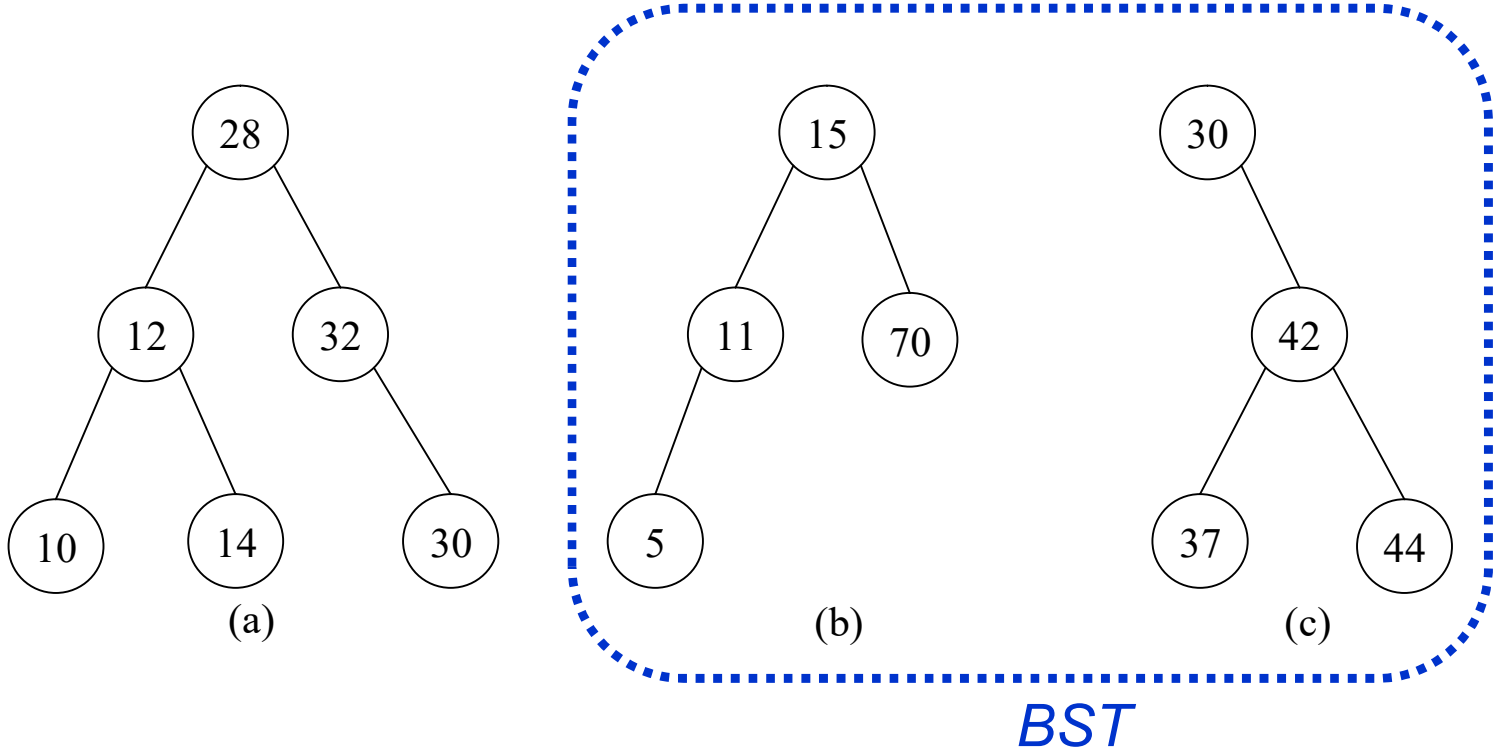
예: 이진 탐색 트리



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST: more examples

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



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Op1 Search

BST Op2 Insertion

BST Op3 Deletion

BINARY SEARCH TREE OPERATIONS

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Operation 1 Search

- 이진 탐색 트리에서의 탐색 (순환적 기술)
  - 키값이 x인 원소를 탐색
  - 시작 : 루트
    - 탐색 트리가 공백이면, 실패로 끝남
    - 루트의 키값 = x 이면, 탐색은 성공하며 종료
    - 키값 x < 루트의 키값이면, 루트의 왼쪽 자식을 기준으로 다시 검색
    - 키값 x > 루트의 키값이면, 루트의 오른쪽 자식을 기준으로 다시 검색
    - 서브트리만 탐색

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Op1 Search: Iterative

[알고리즘] 이진 탐색 트리 검색

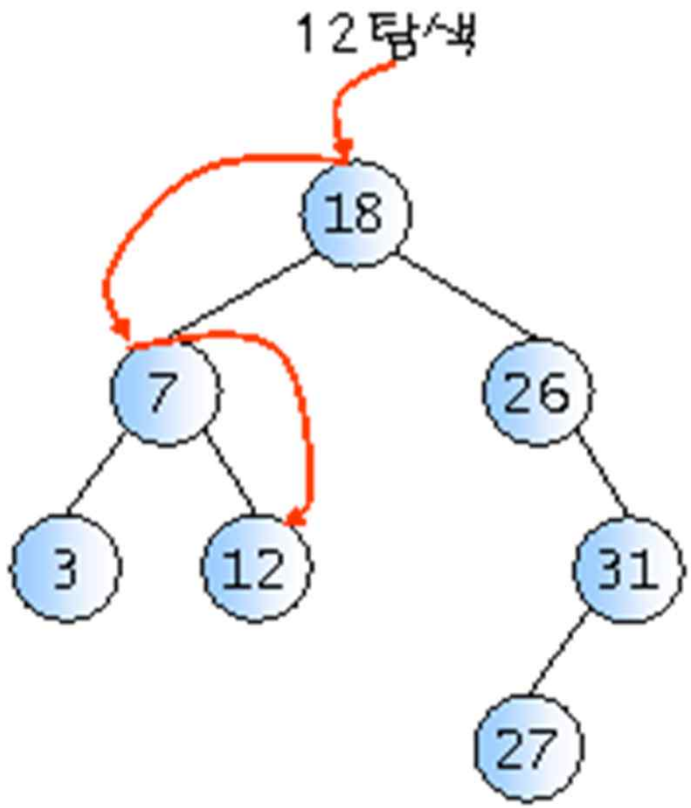
x: 찾으려는 키 값
v_0: 이진 탐색 트리의 뿌리
NULL: 트리의 정점이 아닌 것을 나타냄

$v \leftarrow v_0$
while $(v \neq NULL)$
  if $(x = key(v))$, 탐색 종료
  else if $(x < key(v))$, $v \leftarrow v$의 왼쪽 자식
  else if $(x > key(v))$, $v \leftarrow v$의 오른쪽 자식

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Op1 Search: Recursive

```
/* Pseudo code */
search(x, k)

if x=NULL
      then return NULL;
if k=x->key
      then return x;
else if k<x->key
    then return search(x->left, k);
else
      return search(x->right, k);
```



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Op1 Search: C code

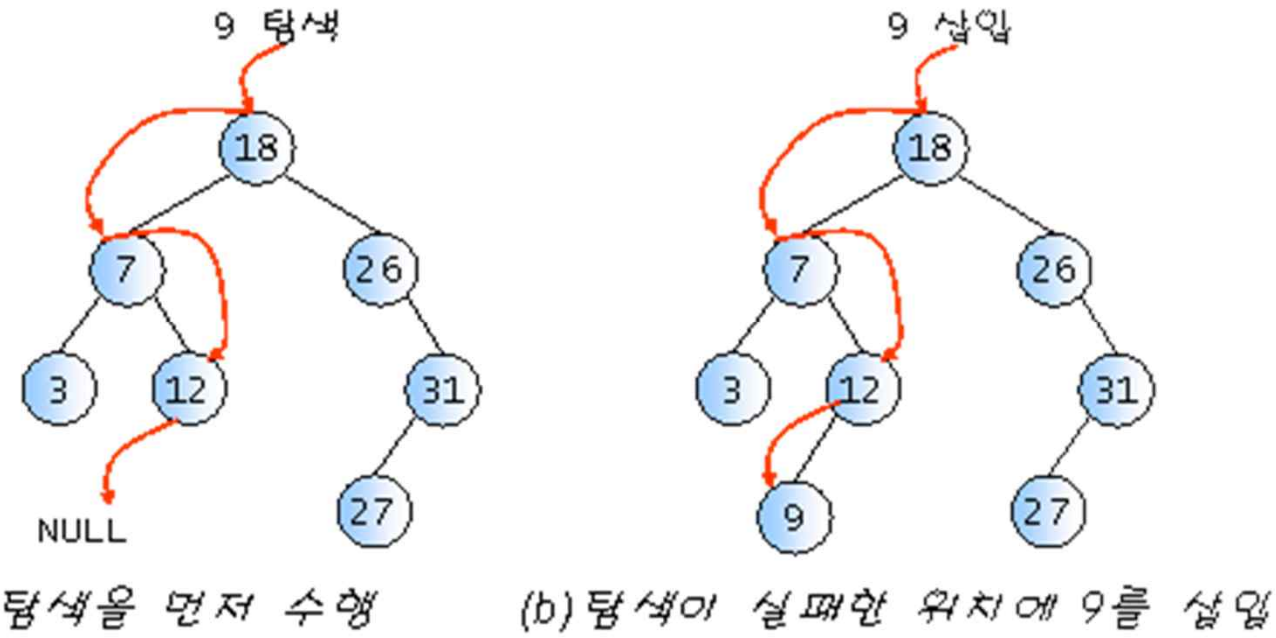
```
// 순환적인 탐색 함수
TreeNode  *search_recursive(
     TreeNode  *node, int  key)
{
  if ( node == NULL )
    return NULL;
  else if ( key == node->key )
    return node;     (1)
  else if ( key < node->key )
     return  search(node->left, key);   (2)
  else
    return  search(node->right, key);  (3)
}
```

```
// 반복적인 탐색 함수
TreeNode  *search_iterative(
     TreeNode  *node, int  key)
{
  while(node != NULL){
    if( key == node->key ) return node;
    else if( key < node->key )
      node = node->left;
    else
      node = node->right;
  }
  return NULL;          // 탐색에
  // 실패했을 경우 실패 반환
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Operation 2 Insertion

- 이진 탐색 트리에 원소를 삽입하기 위해서는 먼저 탐색을 수행
- 탐색에 실패한 위치에 새로운 노드를 삽입

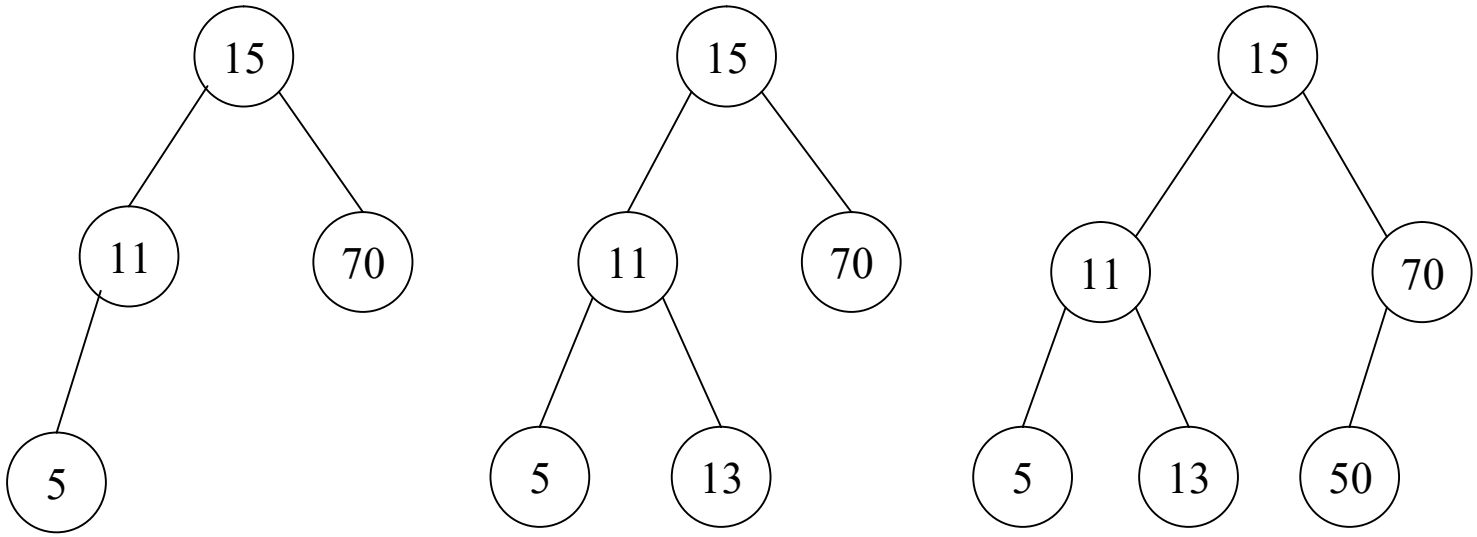


(a) 탐색을 먼저 수행 (b) 탐색이 실패한 위치에 9를 삽입

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Op 2 Insertion: Example

- 예: 키값 13, 50의 삽입 과정
  - x를 키값으로 가진 원소가 있는가를 탐색
  - 탐색이 실패하면, 탐색이 종료된 위치에 원소를 삽입



(a) 이진 탐색 트리 (b) 원소 13을 삽입 (c) 원소 50을 삽입

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Op2 Insertion: Pseudocode

```
insertBST(B, x)
// B는 이진 탐색 트리, x는 삽입할 키값
  p ← B;
  while (p ≠ null) do {
    if (x = p.key) then return B;
       // 키값을 가진 노드가 이미 있음
    else {
      q ← p; // q는 p의 부모 노드를 지시
      if (x < p.key) then p ← p.left;
      else p ← p.right;
    }
  }
  // q는 삽입할 위치의 부모 노드
  // p 는 null node
```

(→ continued)

```
  newNode ← getNode();
  // 삽입할 노드를 만듦
  newNode.key ← x;
  newNode.right ← null;
  newNode.left ← null;

  if (B = null) then B ← newNode;
  // 공백 이진 탐색 트리인 경우
  else if (x < q.key) then
              q.left ← newNode;
  else        q.right ← newNode;

  return B;
end insertBST()
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Op2 Insertion: C code

```
// key를 이진 탐색 트리 root에 삽입
// 이미 있으면 삽입되지 않는다.
void insert_node(
   TreeNode **root, int key)
{
     TreeNode *p;   // 부모노드
     TreeNode *t;   // 현재노드
     TreeNode *n;   // 새로운 노드

     t = *root;  p = NULL;

     // 탐색을 먼저 수행
     while (t != NULL){
        if( key == t->key ) return;
        p = t;
        if( key < t->key ) t = t->left;
        else t = t->right;
     }
     // key가 트리 안에 없으므로
     // 삽입 가능
      n = (TreeNode *)
      malloc(sizeof(TreeNode));

     // 데이터 복사
     n->key = key;
     n->left = n->right = NULL;

     // 부모 노드와 링크 연결
     if( p != NULL )
        if( key < p->key )
              p->left = n;
        else p->right = n;
     else *root = n;
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Operation 3 Deletion

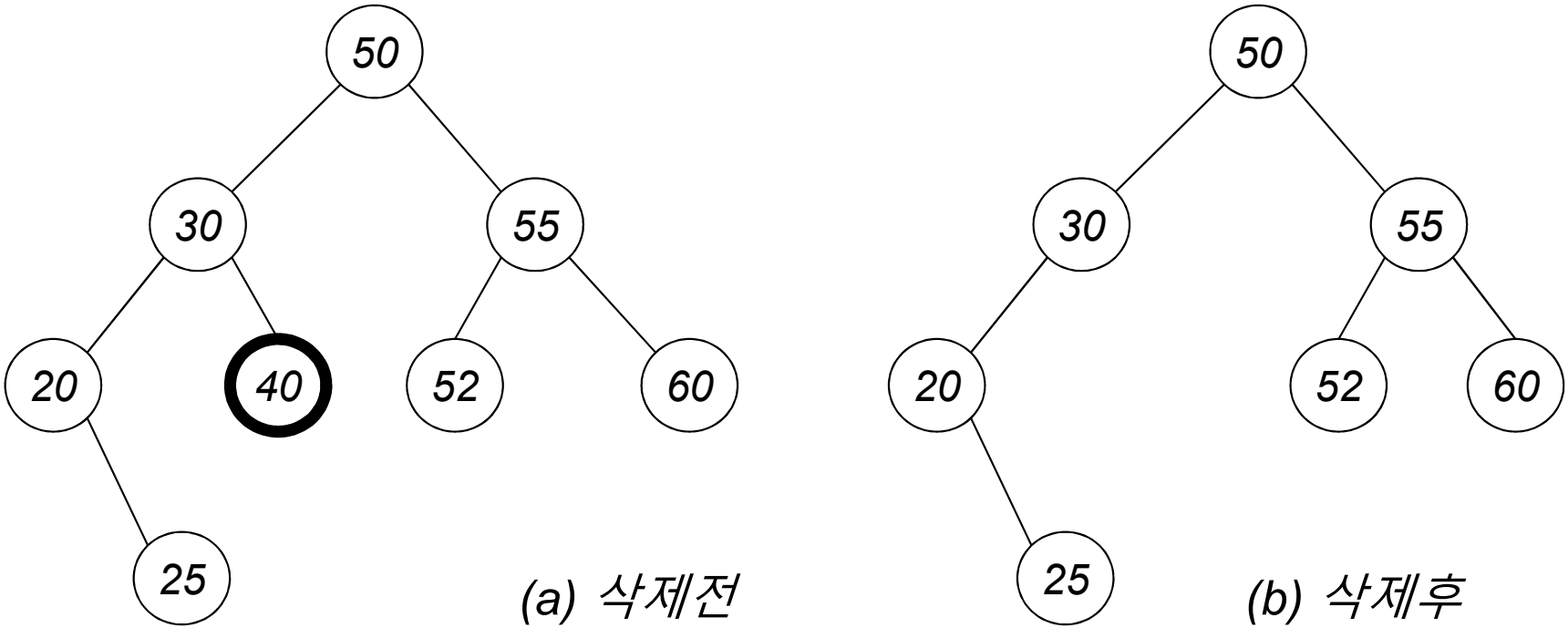
- 삭제하려는 원소의 키값이 주어졌을 때
  - 이 키값을 가진 원소를 탐색
  - 원소를 찾으면 삭제 연산 수행

- 해당 노드의 자식수에 따른 세가지 삭제 연산
  - Deletion of leaf (terminal) node: 자식이 없는 단말 노드의 삭제
  - Deletion of single child node: 자식이 하나인 노드의 삭제
  - Deletion of full children node: 자식이 둘인 노드의 삭제

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Op 3-1: Deleting Leaf Node

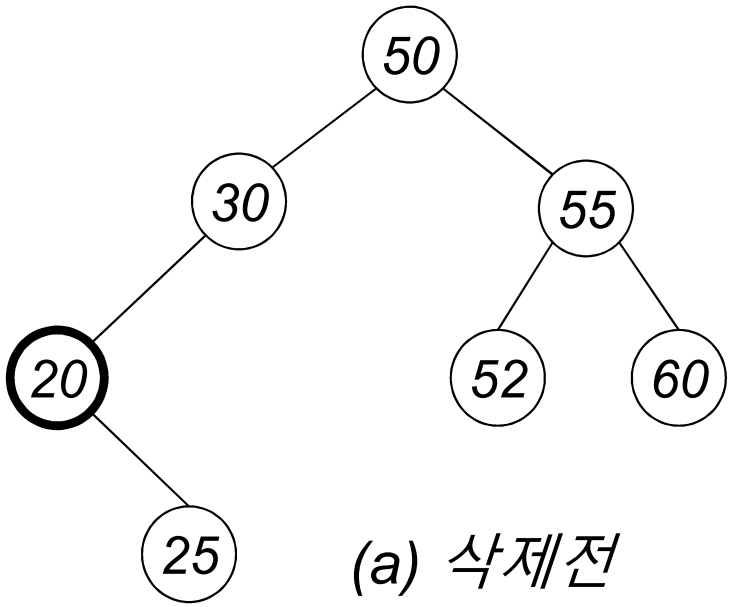
- 자식이 없는 단말 노드의 삭제
  - 단말노드의 부모노드를 찾아서 연결을 끊는다
    - 부모 노드의 링크를 널(null)로 만들고 삭제한 노드 반환
- 예 : 키값 40을 가진 노드의 삭제



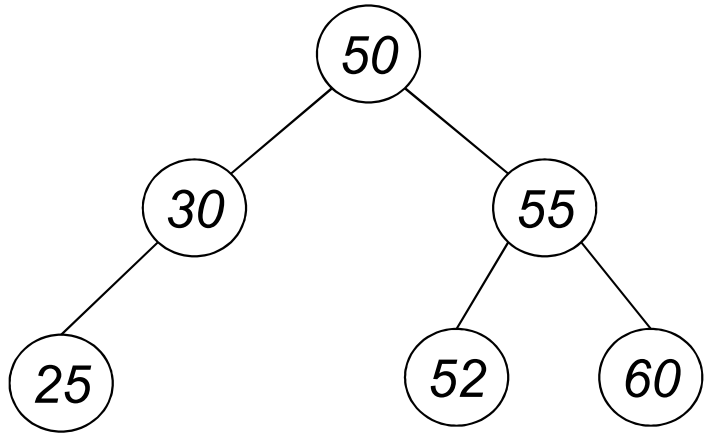
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Op 3-2: Deleting One-Child Node

- 왼쪽이나 오른쪽 서브 트리만 가지고 있을 때
  - 노드는 삭제하고 서브 트리는 부모 노드에 붙여준다
- 예 : 원소 20을 삭제



(a) 삭제전



(b) 삭제후

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

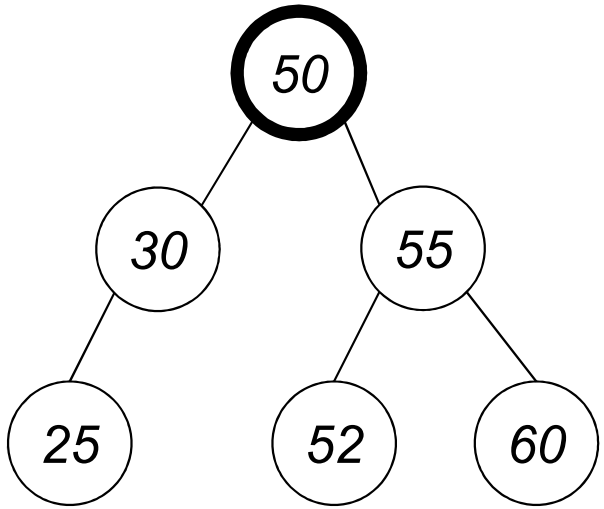
BST Op 3-3: Deleting Full Node

- 자식이 둘인 노드의 삭제
  - 삭제노드와 가장 비슷한 값을 가진 노드를 삭제노드 위치로 가져온다.
- 알고리즘
  - 삭제되는 노드 자리
    - 왼쪽 서브트리에서 제일 큰 원소
    - 또는 오른쪽 서브트리에서 제일 작은 원소로 대체
    - 양쪽 서브트리의 높이와 원소의 개수를 고려하여 결정한다
  - 해당 서브트리에서 대체 원소를 삭제
  - 대체하게 되는 노드의 자식의 개수는 0 이나 1

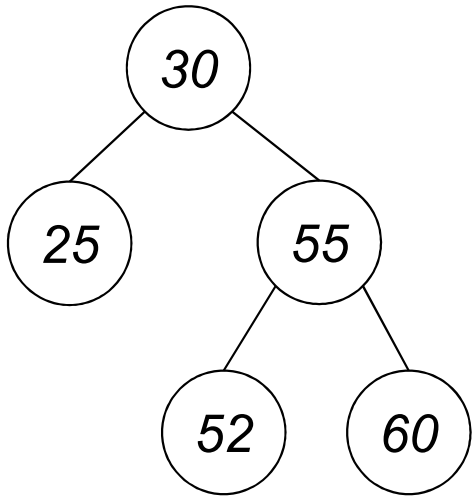
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Op 3-3: Deleting Full Node

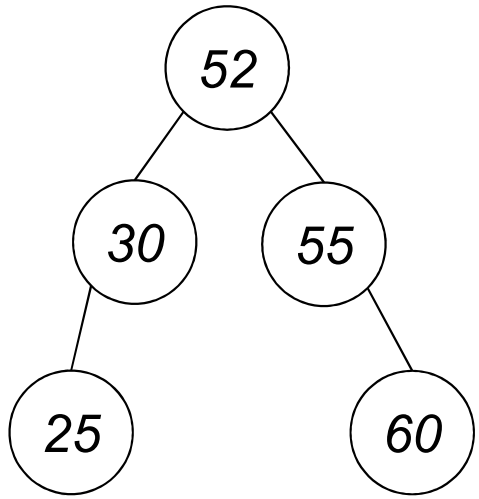
- 예 : 키값이 50인 루트 노드의 삭제시



(a) 삭제전



(b) 왼쪽 서브트리의 최대 원소로 대체



(c) 오른쪽 서브트리의 최소 원소로 대체

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Op 3 Deletion: Pseudocode

```
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
   switch {
        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;
        case 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;
            }
        case p.left ≠ null and p.right ≠ null :   // 삭제할 노드의 차수가 2인 경우
            q ← maxNode(p.left);                  // 왼쪽 서브트리에서 최대 키값을 가진 원소를 탐색
            p.key ← q.key;
            deleteBST(p.left, p.key);
   }
  free(p);
end deleteBST()
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Op 3 Deletion: C code (1)

```
TreeNode *delete_node(TreeNode *root, int key)
{
 TreeNode *p, *child, *succ, *succ_p, *t;

 // p는 t의 부모노드
 p = NULL;
 t = root;

 // key를 갖는 노드 t를 탐색한다.
 while( t != NULL && t->key != key ){
  p = t;
  t = ( key < t->key ) ? t->left : t->right;
 }

 // continue →
```

```
// 탐색이 종료된 시점에 t가 NULL이면
// 트리안에 key가 없음
if( t == NULL ) {
 printf("key is not in the tree");
 return root;
}

// continue →
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Op 3 Deletion: C code (2)

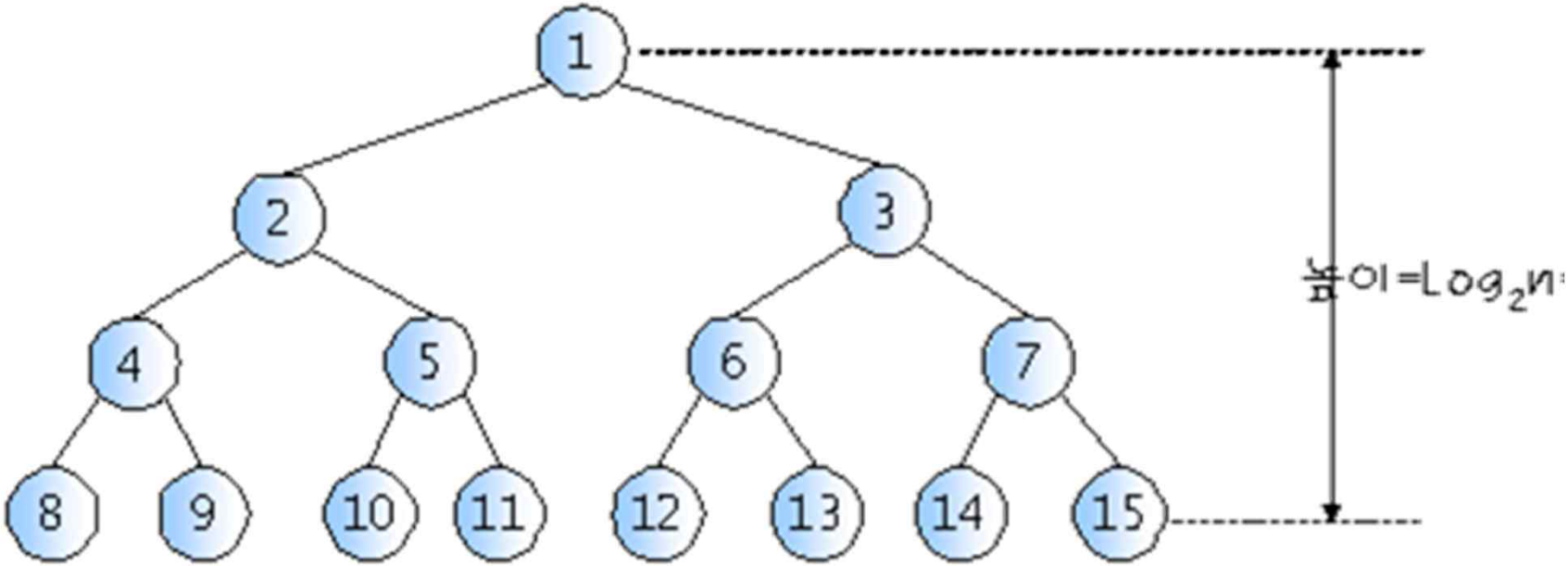
```
// 첫번째 경우: 단말노드인 경우
if( (t->left==NULL) && (t->right==NULL) ){
  if( p != NULL ){
    // 부모노드의 자식필드를 NULL로 만든다.
    if( p->left == t )  p->left = NULL;
    else p->right = NULL;
  }
  else  root = NULL;
}

// 두번째 경우: 하나의 자식만 가지는 경우
else if((t->left==NULL)||(t->right==NULL)){
  child = (t->left != NULL) ? t->left : t->right;
  if( p != NULL ){
    if( p->left == t ) p->left = child;
    else p->right = child;
  }
  else root = child;
}
 // continue →
```

```
// 세번째 경우: 두개의 자식을 가지는 경우
else{
  // 오른쪽 서브트리에서 후계자를 찾는다.
  succ_p = t;
  succ = t->right;
  // 후계자를 찾아서 계속 왼쪽으로 이동한다.
  while(succ->left != NULL){
    succ_p = succ; succ = succ->left;
  }
  // 후속자의 부모와 자식을 연결
  if( succ_p->left == succ )
    succ_p->left = succ->right;
  else succ_p->right = succ->right;
  // 후속자가 가진 키값을 현재 노드에 복사
  t->key = succ->key;
  // 원래의 후속자 삭제
  t = succ;
}
free(t);          // 메모리 반환은 꼭 해주어야 함
return root;  // 새롭게 변경된 트리를 반환
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

이진탐색트리의 성능분석

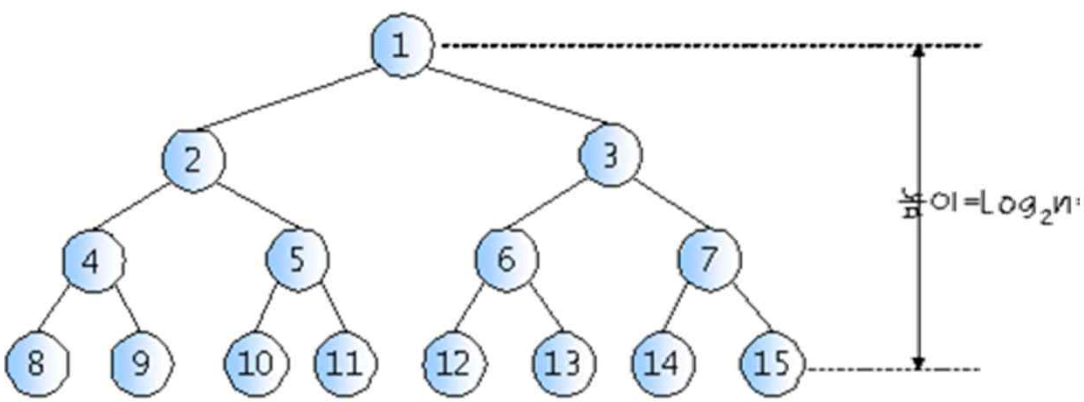
- 이진 탐색 트리에서의 **탐색**, **삽입**, **삭제** 연산의 시간 복잡도는 트리의 높이를 h라고 했을때 h에 비례한다

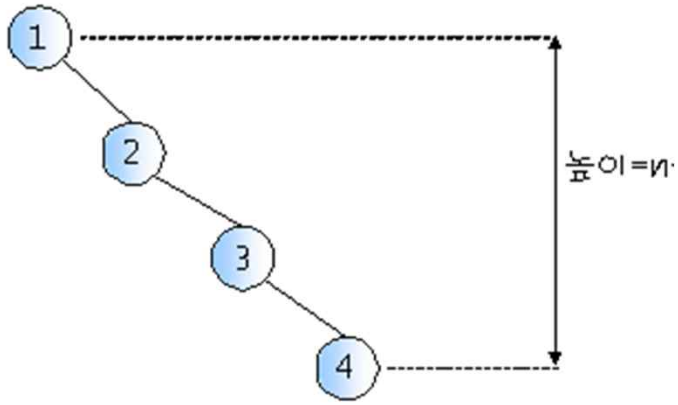


Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

이진탐색트리의 성능분석

- 최선의 경우
  - 이진 트리가 균형적으로 생성되어 있는 경우
  - $h=\log_2 N$
- 최악의 경우
  - 한쪽으로 치우친 경사이진트리의 경우
  - 순차탐색과 시간복잡도가 같다: $h=N$





Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

입력 배열로부터 이진 탐색 트리 구성

BINARY SEARCH TREE CONSTRUCTION

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Construction

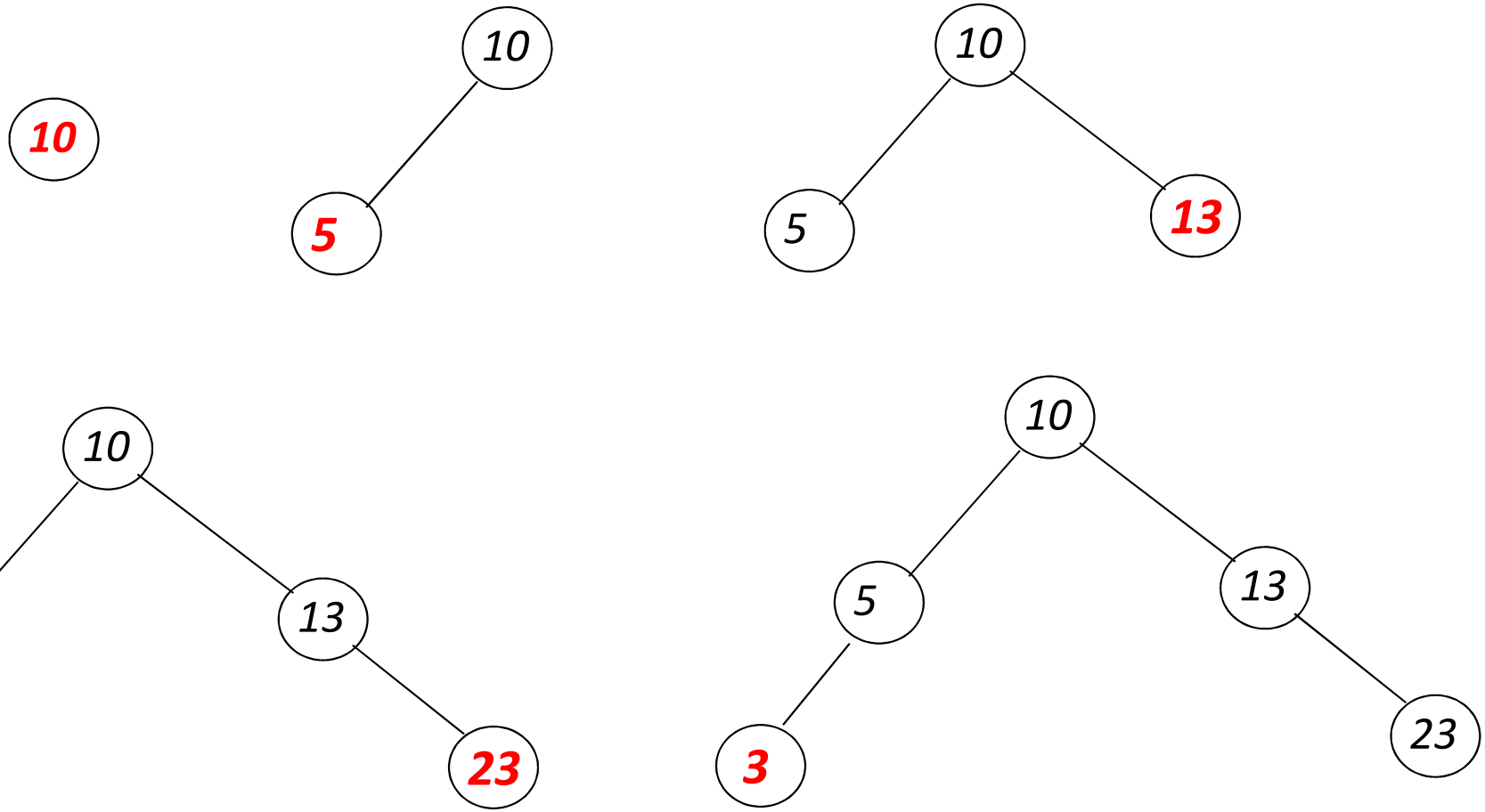
- 임의의 순서로 주어진 array, linked list 를 binary search tree 로 만든다
  - Empty tree 로 시작한다
  - 말단 노드의 아래에 하나씩 추가한다

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

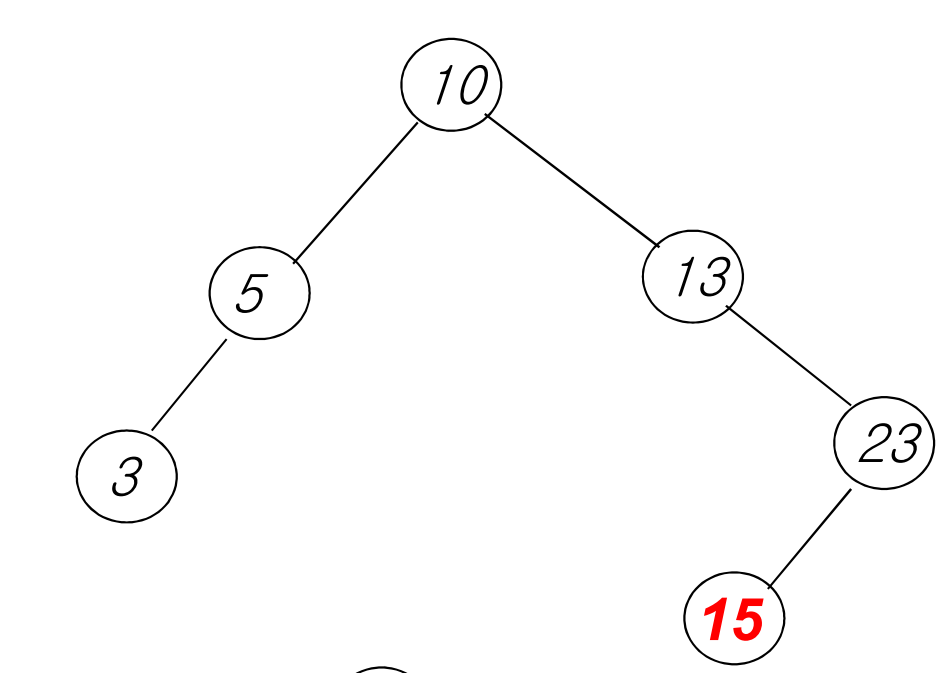
이진 탐색 트리를 만드는 절차

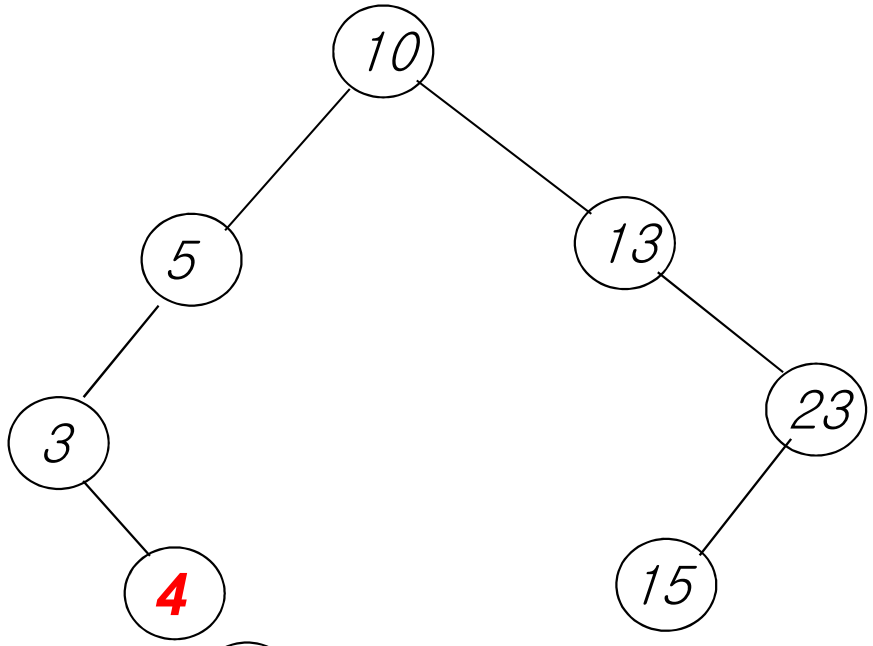
리스트에 {10, 5, 13, 23, 3, 15, 4, 20, 32, 1}의 순서로 입력될 때

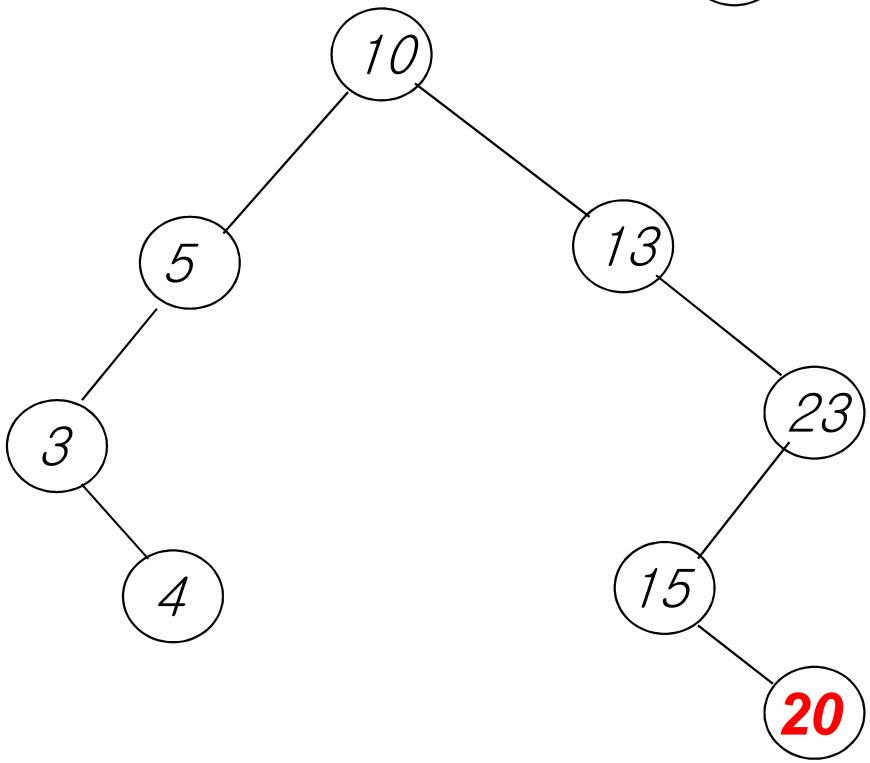
*말단(leaf node)에만 삽입된다

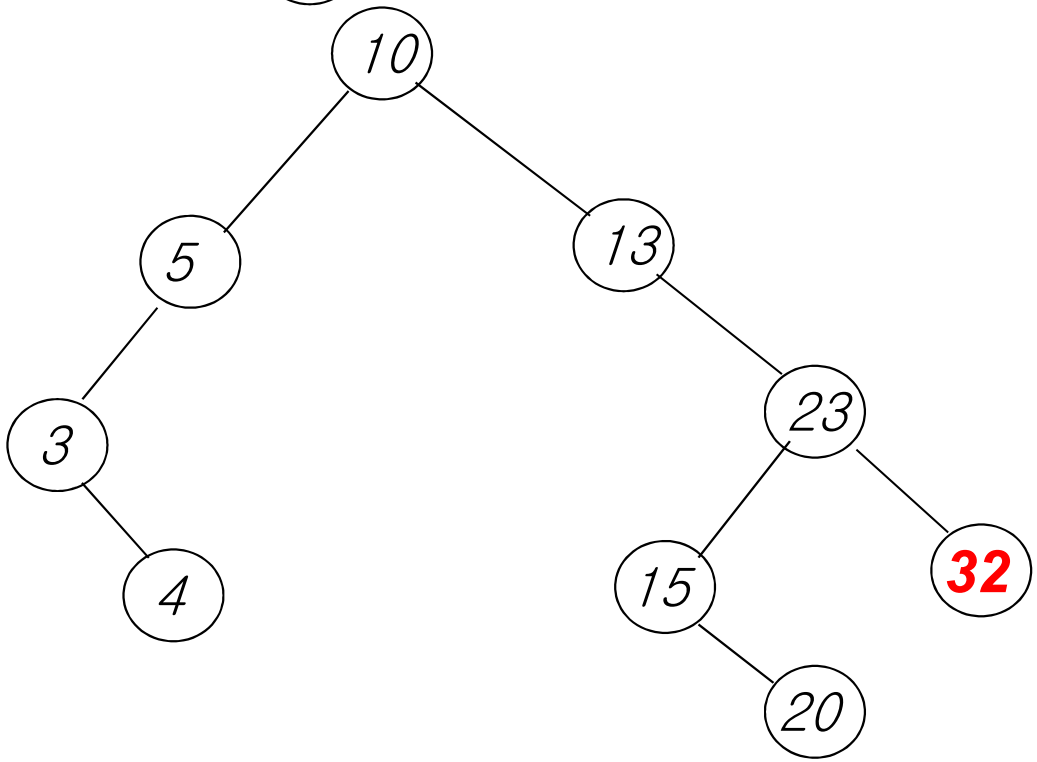


Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

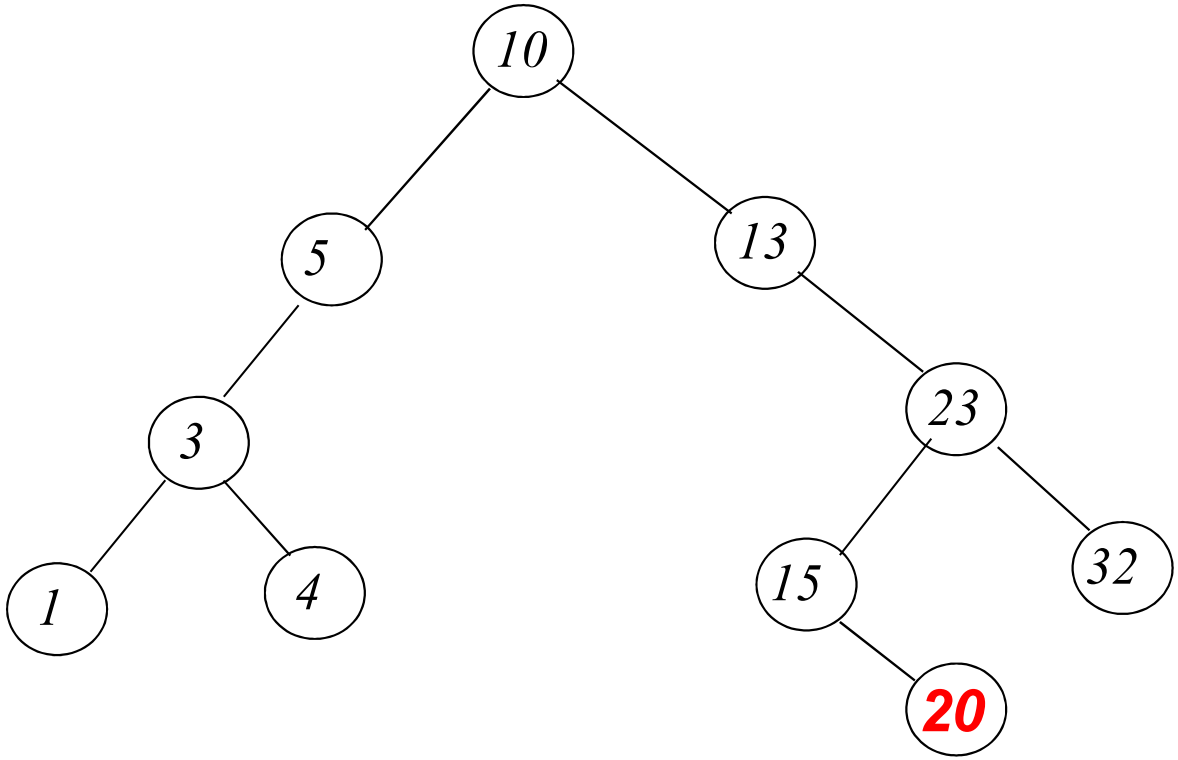








Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST Construction Analysis

탐색을 할 때 비교하는 최대 회수

균형잡힌 이진 탐색 트리(balanced binary search tree)의 경우 완전 이진 트리에 접근하므로 근사적으로 $\lfloor \log_2 N \rfloor$이다.

이진 탐색 트리를 구성할 때 비교하는 최대 회수

비교 회수는 이진 트리의 깊이와 같다. 따라서 리스트의 수가 N이라면 트리의 높이는 $\lfloor \log_2 N \rfloor \sim N$ 이며, 최대 비교 회수는

- (Best) balanced BST: $\lfloor \log_2 N \rfloor$
- (Worst) all single child nodes BST: N

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

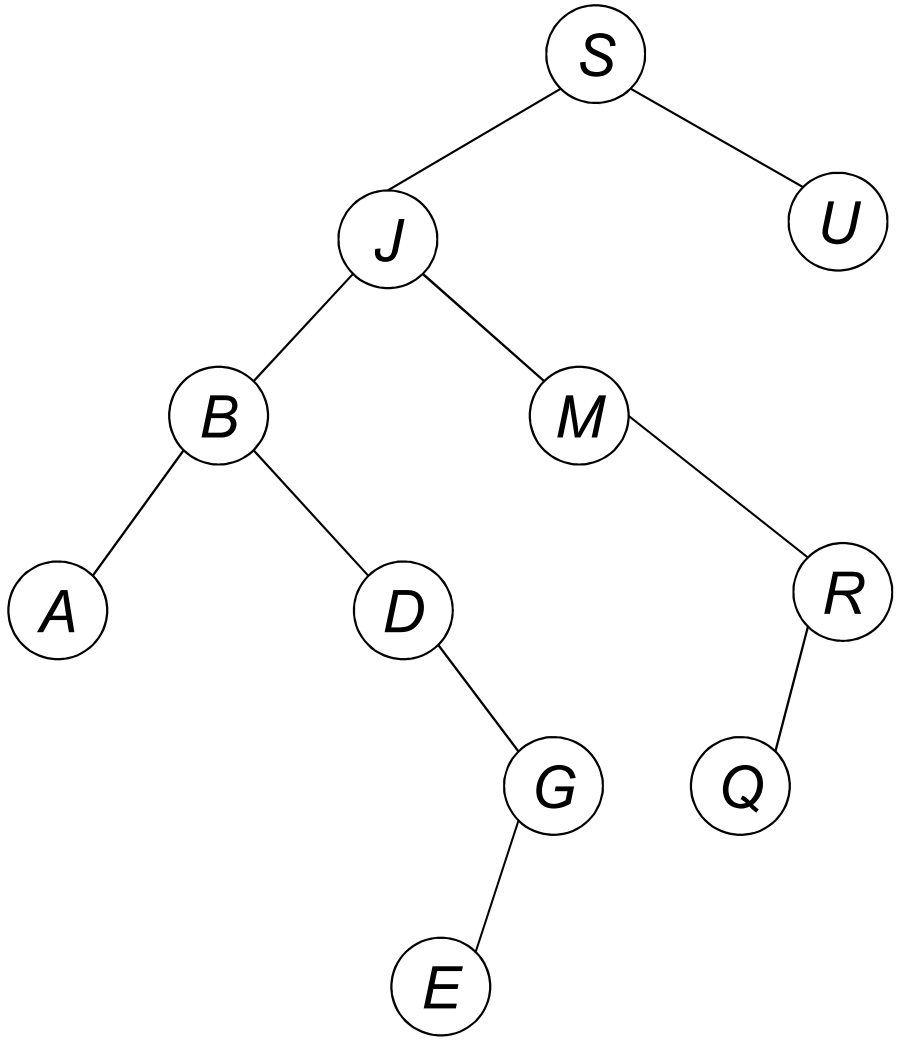
Case study:

BINARY SEARCH TREE IMPLEMENTATION

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST C Implementation Example

- 예제: 스트링 타입의 키값을 가진 이진 탐색 트리 구축
  - char "R"을 가지고 있는 노드 탐색
  - 트리에 없는 char "C"를 탐색



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>   // malloc

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;
   }
```

```
    else if (x > p->key) {        // x를 p의 오른쪽 서브트리에 삽입
        p->right = insertKey(p->right, x);
        return p;
    } else {                      // key 값 x가 이미 이진 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) {  return p;  } // 키 값 x를 발견
        else p = p->right;
    }
    printf("No such key value is found");
    return p;
}
```

```
void printNode(treeNode* p) {        // printTree() 함수에 의해 사용
    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 형태 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);
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

```
int main() { // 예제 실행을 위한 main() 함수
    treeNode* root = NULL;   // 공백 이진 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");
```

The Tree is ((((A)B(D((E)G)))J(M((Q)R)))S(U))

Search For 'R'
Key of node found = R

Search For 'C'
No such key value is found
Node that was found = NULL

```
        /* 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;
    }
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Binary Search Tree: Summary

- 이진 탐색 트리의 높이
  - 이진 탐색 트리의 높이가 크면
    - 원소의 검색,삽입,삭제 연산 수행 시간이 길어진다.
  - N개의 노드를 가진 이진 탐색 트리의 최대 높이: $N - 1$
  - 평균적인 이진 탐색 트리의 높이: $O(\log N)$

- 균형 탐색 트리 (balanced search tree)
  - 탐색 트리의 높이가 최악의 경우 $O(\log N)$이 되는 경우
  - 검색, 삽입, 삭제 연산을 $O(\text{height} = \log N)$시간에 수행
  - 예) AVL , 2-3, 2-3-4 , red-black, B-tree

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Review: Quicksort on Arrays

Quicksort on (doubly) linked lists

Using quicksort to build a binary search tree

QUICKSORT AND BINARY SEARCH TREE

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Review: Array Quicksort

```
Quicksort(A, p, r)  {
  if (p < r)  {
    q = Partition(A, p, r);
    Quicksort(A, p, q);
    Quicksort(A, q+1, r);
  }
}
```

```
Partition(A, p, r)
  x = A[p]; i = p - 1; j = r + 1;
  while (TRUE)
    repeat j--;
    until A[j] ≤ x;
    repeat i++;
    until A[i] ≥ x;
    if (i < j)
      SWAP(A, i, j);
    else
      return j;
```

*Array Quicksort requires **SWAPPING** two elements, which may take extensive copy overhead for large objects*

- *Use **LINKED LISTS***

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Quicksort on Linked Lists

```
NODE { key: number;
  heavy_content: mass_type;
  next: NODE*; }
Quicksort(H: NODE*)  {
  if (!empty(H) AND !empty(H->next) )  {
    [q,L,R] = Partition(H); // q is pivot, a single node
    L = Quicksort(L);
    R = Quicksort(R);
    H = Concatenate(L,q,R);
  }
  return H;
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Partition on L-Lists: Split by Pivot

```
Partition(H: NODE*)  {
   [q,H] = ExtractHead(H);
   // first item as pivot

   L = null; R = null;
   while ( !empty(H) )  {
      [x,H] = ExtractHead(H);
      // examine one by one
      if ( x->key ≤ q->key )
         L = AddToTail(L,x);
      else R = AddToTail(R,x);
   }
   return (q,L,R);
}
```

```
ExtractHead(H: NODE*)  {
   x = H;
   Next = H->next;
   x->next = null;
   // isolate
   return (x,Next);
}
```

*Why **AddToTail**?

- For **stable** sorting
- 뒤에 붙이면 원소들의 key값이 같을 때 원래의 순서가 유지된다
- Singly linked list로 구현이 복잡하다

(효율적인 삽입을 위해 tail pointer와 head pointer 둘다 유지 필요)

https://en.wikipedia.org/wiki/Sorting_algorithm#Stability
https://stackoverflow.com/questions/44594627/does-qsort-preserve-the-original-order-when-sorting-using-a-second-field

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Partition on L-Lists: Revised

```
Partition(H: NODE*)  {
  [q,H] = ExtractHead(H);
  // first item as pivot

  LH = null; RH = null;
  LT = null; RT = null;
  while ( !empty(H) )  {
    [x,H] = ExtractHead(H);
    // examine one by one
     if ( x->key ≤ q->key )
      (LH,LT) =
        AddToTailHT(LH,LT,x);
     else (RH,RT) =
        AddToTailHT(RH,RT,x);
   }
  return (q,LH,RH);
}
```

```
ExtractHead(H: NODE*)  {
  x = H;
  Next = H->next;
  x->next = null;
  // isolate
  return (x,Next);
}
```

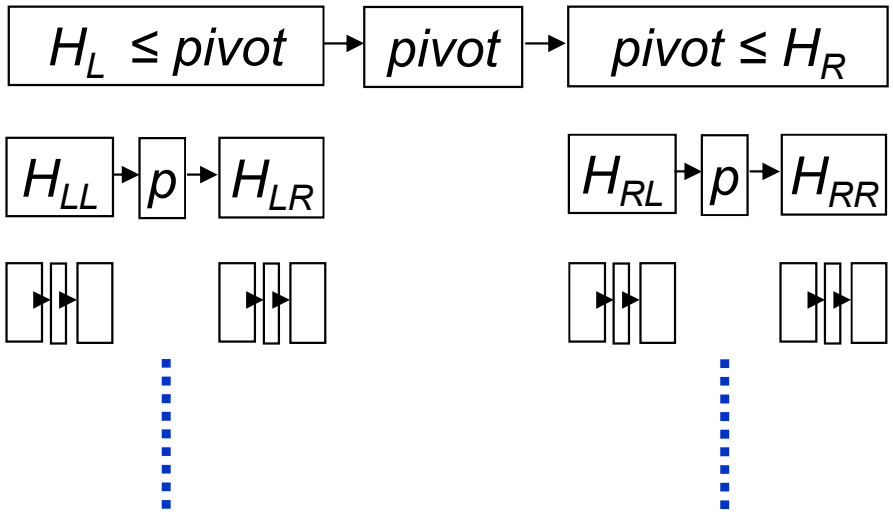
```
AddToTailHT(
      H: NODE*, T: NODE*,
      x: NODE*)  {
   if empty(H)
     H = x;
   else
     T->next = x;
   T = x;
   return (H,T);
}
```

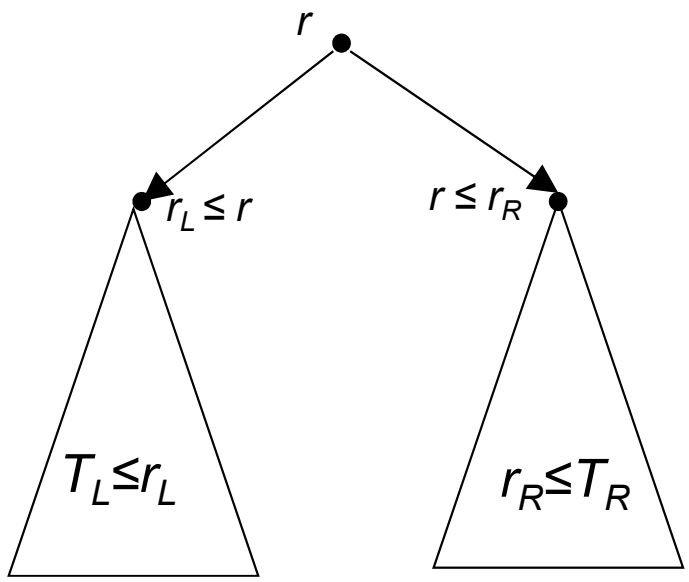
C에서 둘 이상의 return 값을 보내려면 pointer 변수를 추가로 써야 한다
따로 함수로 구현하지 않고 Partition 내부에 집어 넣을 수도 있음

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Comparing Partitioning and BST

- Partitioning in quicksort:
  - left partition < pivot < right partition
- Binary search tree:
  - Left subtree < root < right subtree
- We can use quicksort to build a binary search tree!



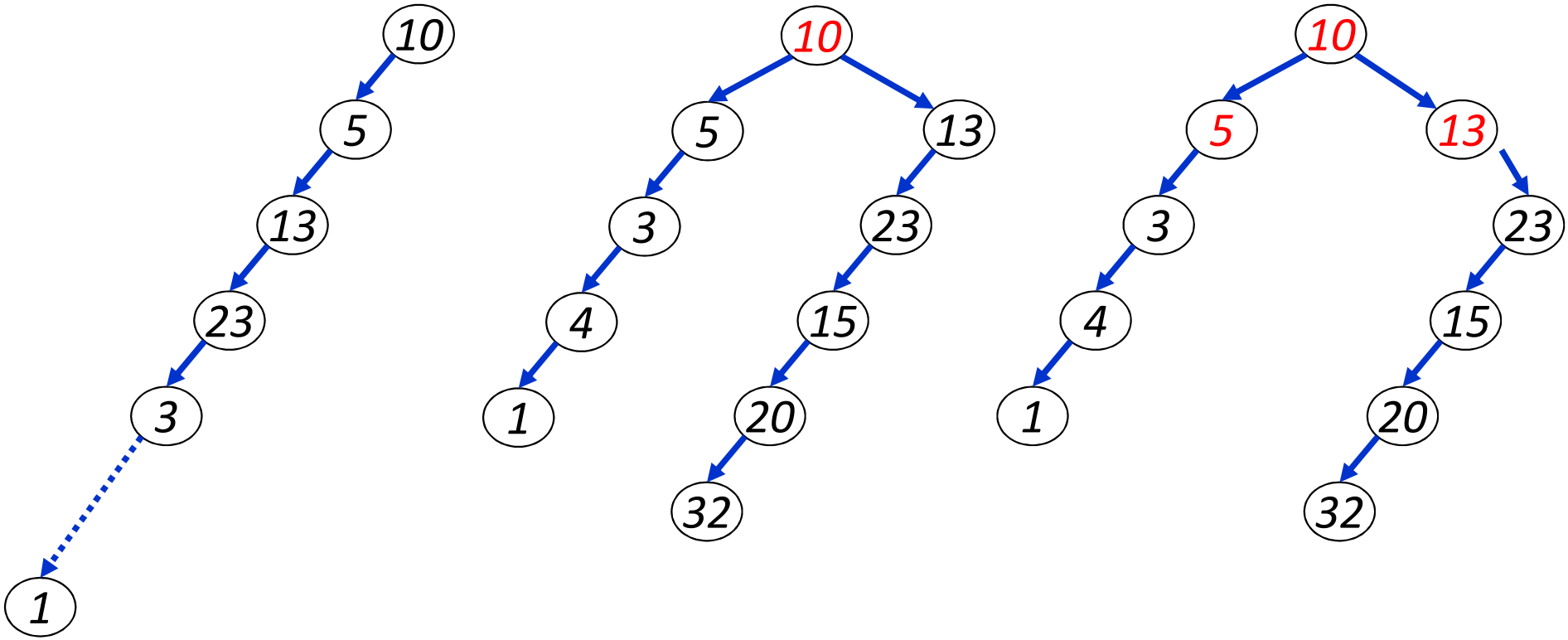


Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Using quicksort to build a BST

1. *left-skewed binary tree 를 입력으로*
2. *맨 처음 node 를 pivot 으로 선택*
3. *Partition into BT-L ≤ pivot and pivot < BT-R*
4. *Repeat until leaf nodes*

{10, 5, 13, 23, 3, 15, 4, 20, 32, 1}의 리스트가 주어질 때

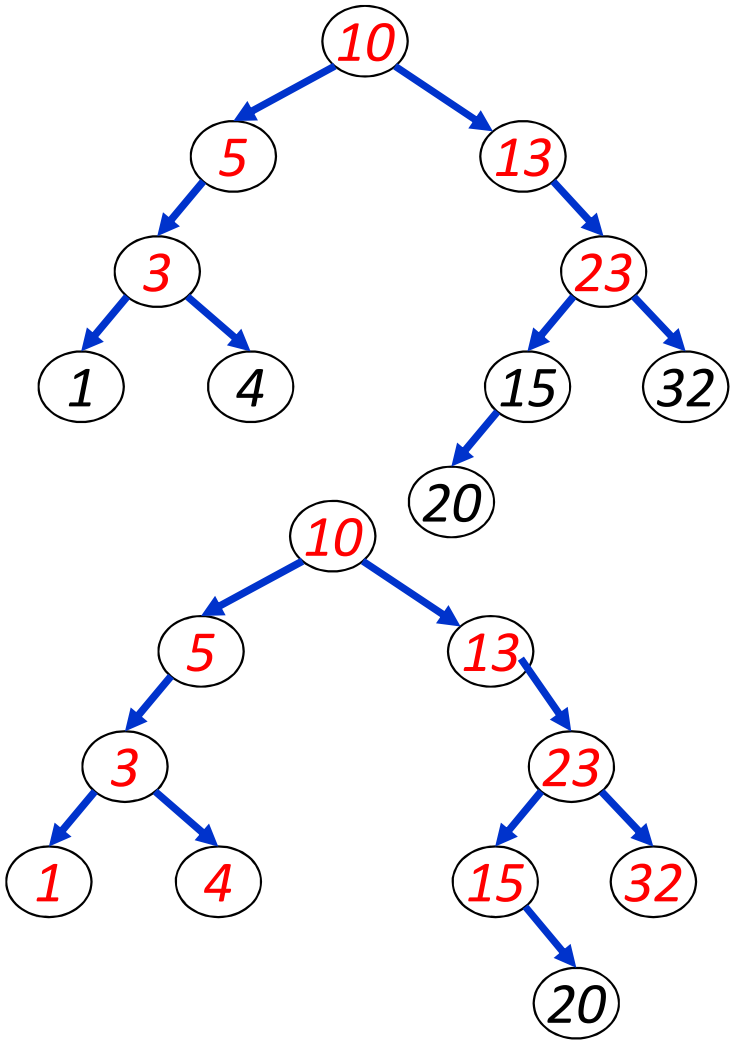


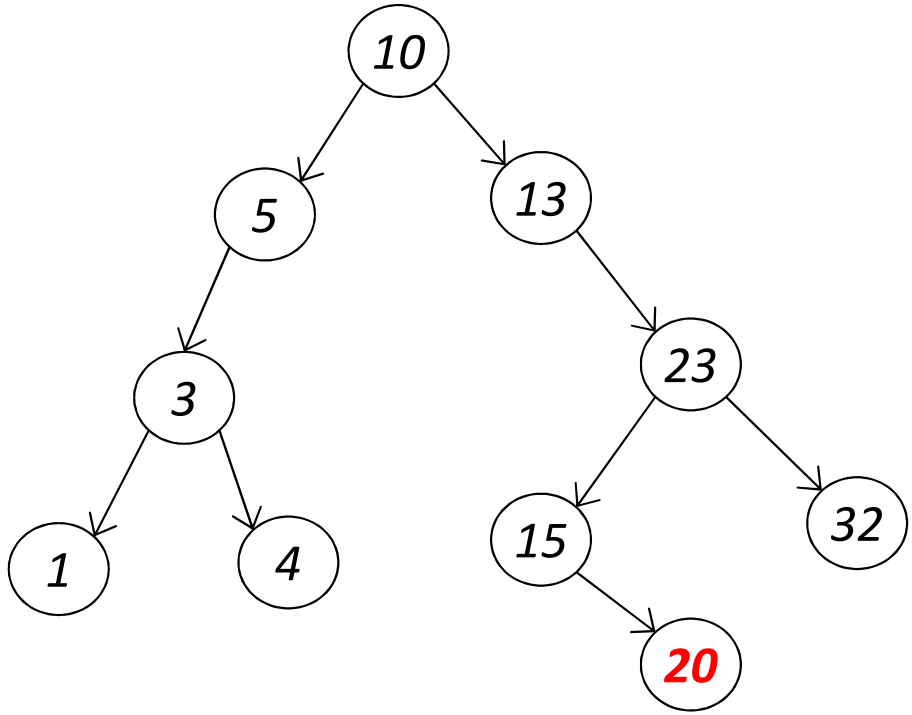
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Using quicksort to build a BST

1. *left-skewed binary tree 를 입력으로*
2. *맨 처음 node 를 pivot 으로 선택*
3. *Partition into BT-L ≤ pivot and pivot < BT-R*
4. *Repeat until leaf nodes*

{10, 5, 13, 23, 3, 15, 4, 20, 32, 1}
의 리스트가 주어질 때





BST insertion 으로
만들어진 BST

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BST by Quicksort

```
BTNODE { key: number;
  heavy_content: mass_type;
  left, right: NODE*; }
QuicksortBST(SkewedBT: BTNODE*)  {
  if (!empty(SkewedBT) AND !empty(SkewedBT->left) )  {
    [q,L,R] = QBSTPartition(SkewedBT); // q is pivot, a single node
    q->left = QuicksortBST(L);
    q->right = QuicksortBST(R);
    return q;
  }
  return SkewedBT;
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Quicksort BST Partition

```
QBSTPartition(B: BTNODE*)   {
  [q,B] = ExtractRoot(B);
  // root of the skewed as pivot

  LB = null; RB = null;
  LT = null; RT = null;
  while ( !empty(B) )  {
    [x,B] = ExtractRoot(B);
    // examine one by one
   if ( x->key ≤ q->key )
        (LH,LT) =
         AddToTailBT(LB,LT,x);
    else (RH,RT) =
         AddToTailBT(RB,RT,x);
   }
  return (q,LH,RH);
}
```

```
ExtractRoot(B: BTNODE*)  {
  if ( B->right != NULL) ERROR;
  x = B;
  Next = B->left;
  x->left = null;  // isolate
  return (x,Next);
}
```

```
AddToTailBT(
     B, T, x: BTNODE*)   {
   if empty(B)
     B = x;
   else
     T->left = x;
     T->right = null;
   T = x;
   return (B,T);
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analyzing QuicksortBST

- QuicksortBST = ConstructBST, when
  - Pivot 을 첫번째 element로
  - Stable partition: keep original order by AddToTail
- Analyzing QuicksortBST
  - QuicksortBST = QBSTPartition + QBST(L) + QBST(R)
  - QBSTPartition = ExtractRoot + *N* * (ExtractRoot + AddToTail) = O(1) + *N* * (O(1)+O(1)) = O(*N*)
  - Same as Quicksort on Arrays
    - Worst O(*N*^2), Best O(*N* lg *N*), Average O(*N* lg *N*)
- Analyzing ConstructBST
  - (lg1 + lg2 + … + lg *N*) < *N* lg *N* ➔ O(*N* lg *N*)
    - * lg = log2

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Preorder traversal

Inorder traversal

Postorder traversal

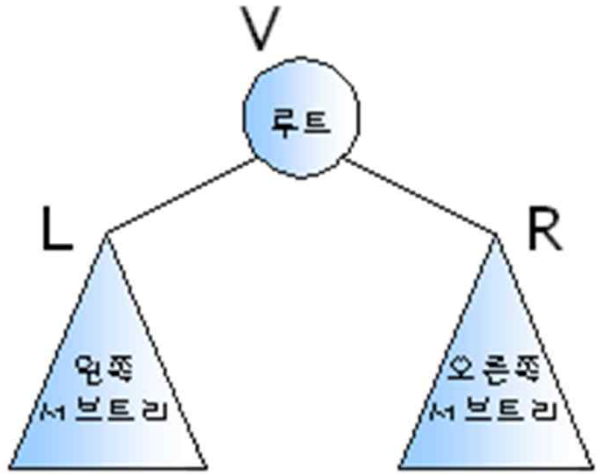
Levelorder traversal

BINARY TREE TRAVERSAL

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

이진 트리의 순회

- 순회(traversal): 트리의 노드들을 체계적으로 방문하는 것
- 3가지의 기본적인 순회방법
  - 전위순회(preorder traversal): VLR
    - 자손노드보다 루트노드를 먼저 방문한다.
  - 중위순회(inorder traversal): LVR
    - 왼쪽 자손, 루트, 오른쪽 자손 순으로 방문한다.
  - 후위순회(postorder traversal): LRV
    - 루트노드보다 자손을 먼저 방문한다.

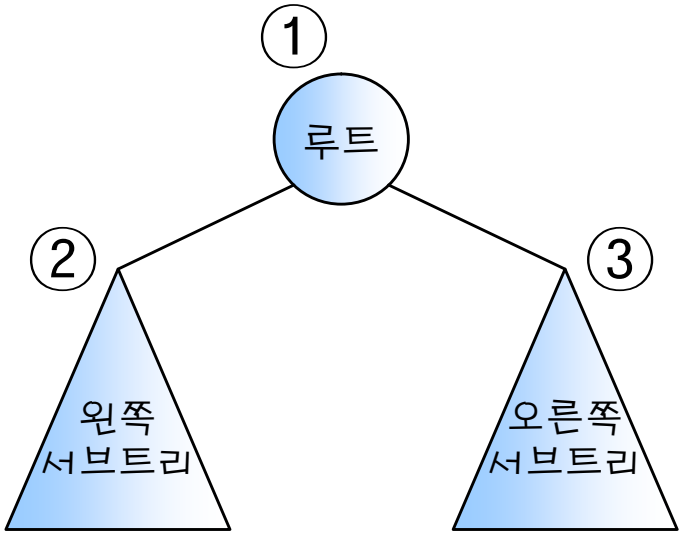


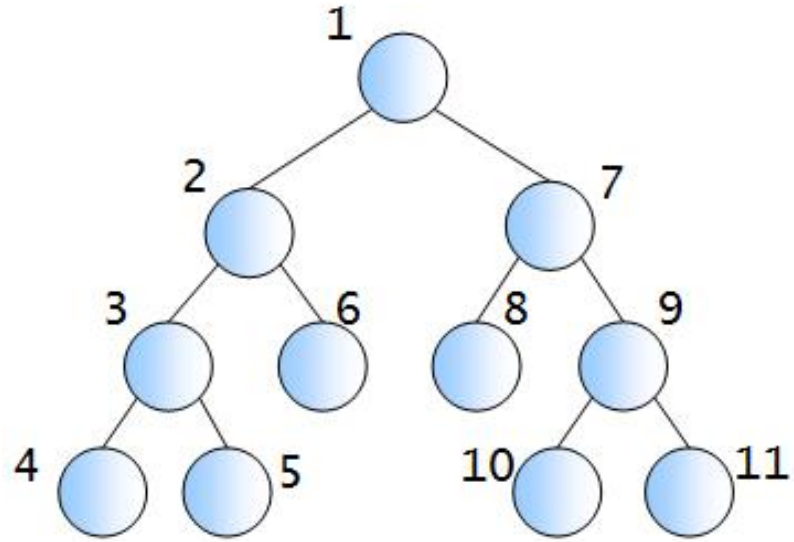
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

전위 순회(Preorder Traversal)

1. 루트 노드를 방문한다
2. 왼쪽 서브트리를 방문한다
3. 오른쪽 서브트리를 방문한다

```
preorder(x)
if x≠NULL
    then print DATA(x);
         preorder(LEFT(x));
         preorder(RIGHT(x));
```





Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Preorder 알고리즘

입력: r(이진 트리의 뿌리)
출력: 각 노드의 값

```
preorder(r){
  if(r == null) return
  display r
  lchild = r의 왼쪽 자식
  preorder(lchild)
  rchild = r의 오른쪽 자식
  preorder(rchild)
}
```

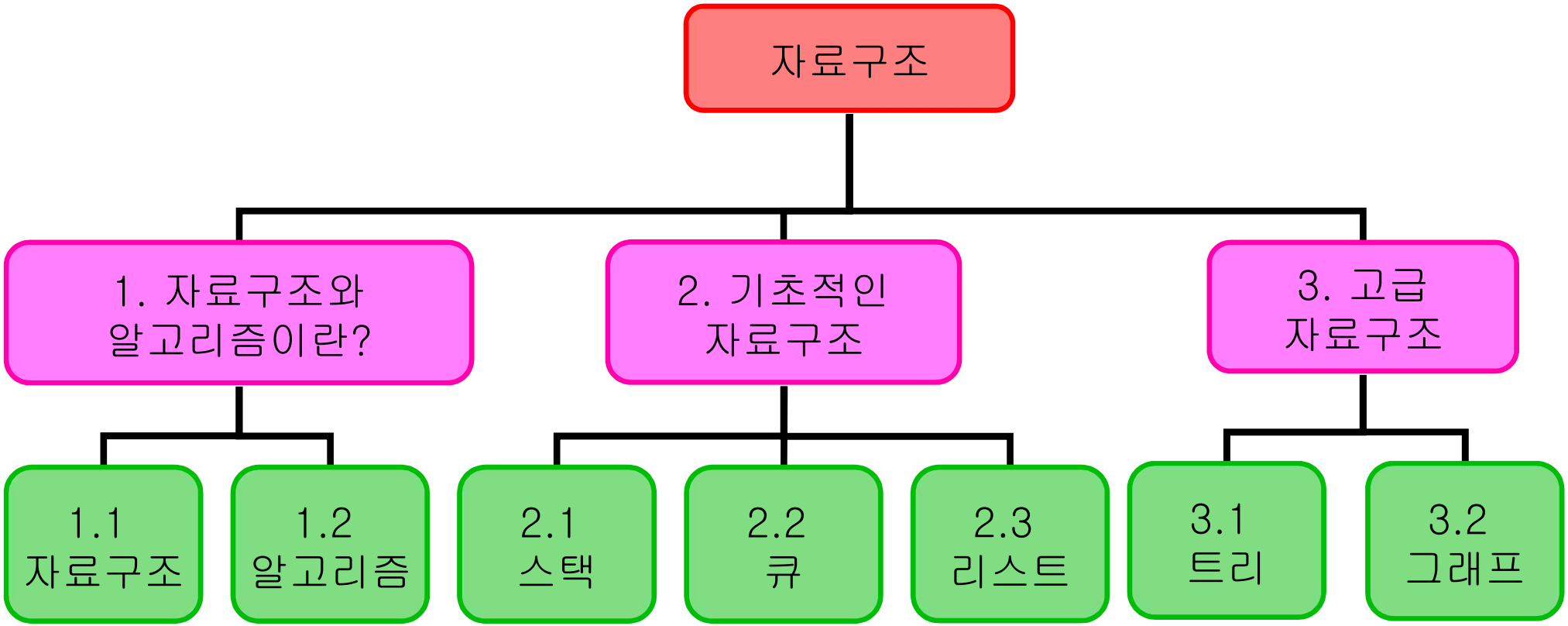
입력: r(이진 트리의 뿌리)
출력: 각 노드의 값

```
preorder_stack(r){
  PUSH(S, r)
  while (queue ≠ empty) {
    x= POP(S)
    display x
    if(x has a right child)
            PUSH(S, x의 right child)
    if(x has a left child)
            PUSH(S, x의 left child)
  }
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

전위 순회 응용

- (예) 구조화된 문서출력
  - 책을 만들 수 있다

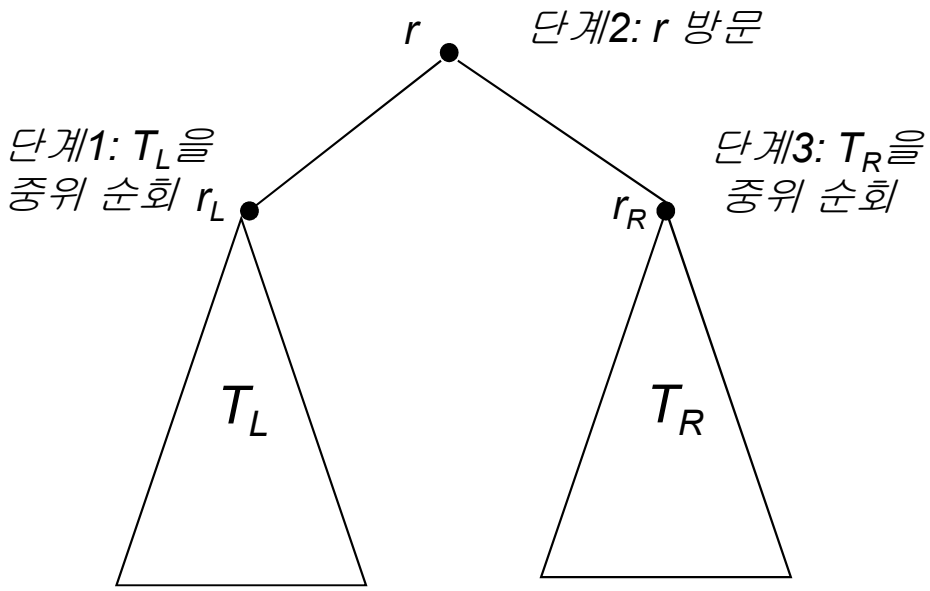


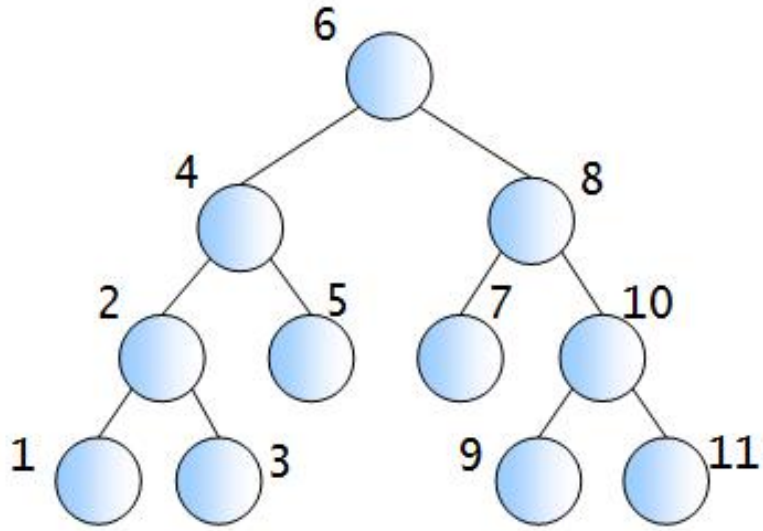
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

중위 순회(Inorder Traversal)

1. 왼쪽 서브트리를 방문한다
2. 루트 노드를 방문한다
3. 오른쪽 서브트리를 방문한다
4. 더 이상 방문할 노드가 없으면 종료된다.

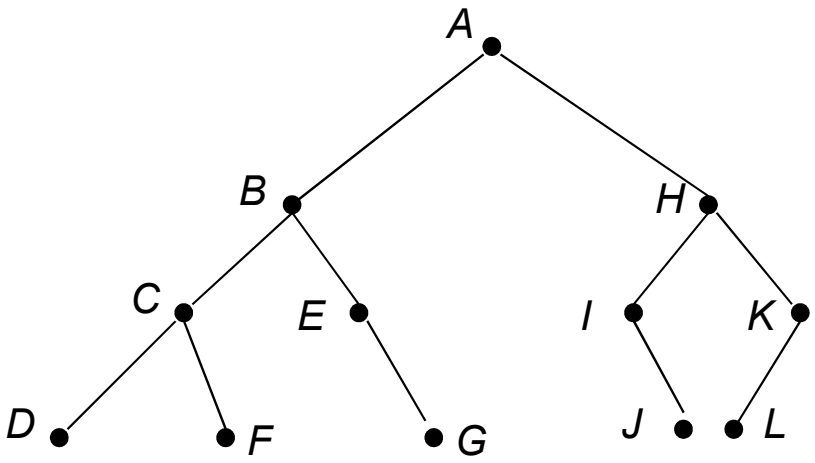
```
inorder(x)
if x≠NULL
    then inorder(LEFT(x));
         print DATA(x);
         inorder(RIGHT(x));
```



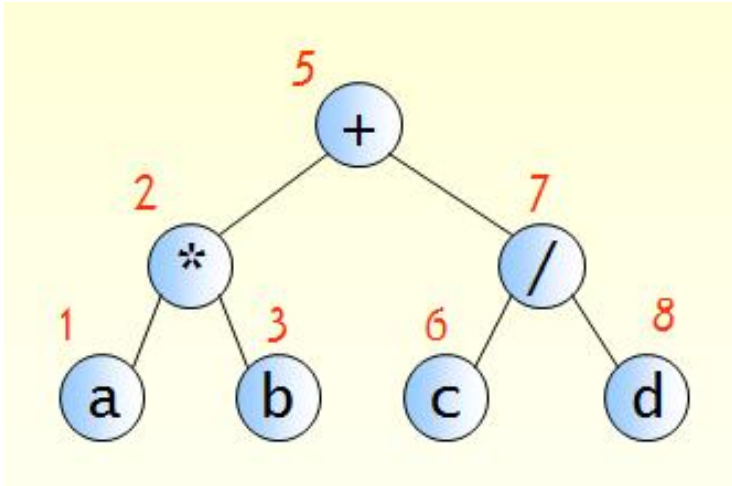


Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

중위 순회 응용에



DCFBEGAIJHLK



수식 트리

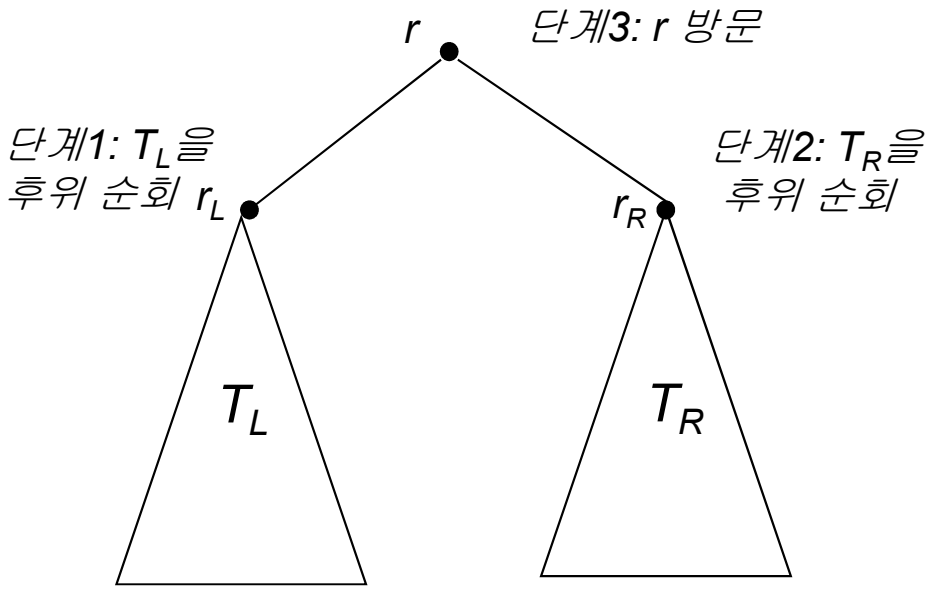
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

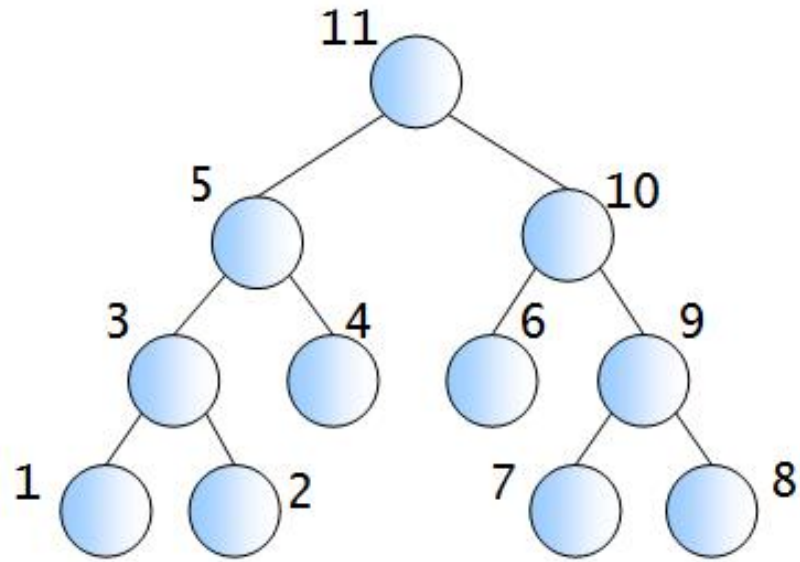
후위 순회(postorder traversal)

1. 왼쪽 서브트리를 방문한다
2. 오른쪽 서브트리를 방문한다
3. 루트 노드를 방문한다

*depth-first traversal 이라고도 불림

```
postorder(x)
if x≠NULL then
    postorder(LEFT(x));
    postorder(RIGHT(x));
    print DATA(x);
```





Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

BT Traversal: C code

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

//                        15
//            4                       20
//          1                       16 25
TreeNode n1={1, NULL, NULL};
TreeNode n2={4, &n1, NULL};
TreeNode n3={16, NULL, NULL};
TreeNode n4={25, NULL, NULL};
TreeNode n5={20, &n3, &n4};
TreeNode n6={15, &n2, &n5};
TreeNode *root= &n6;
```

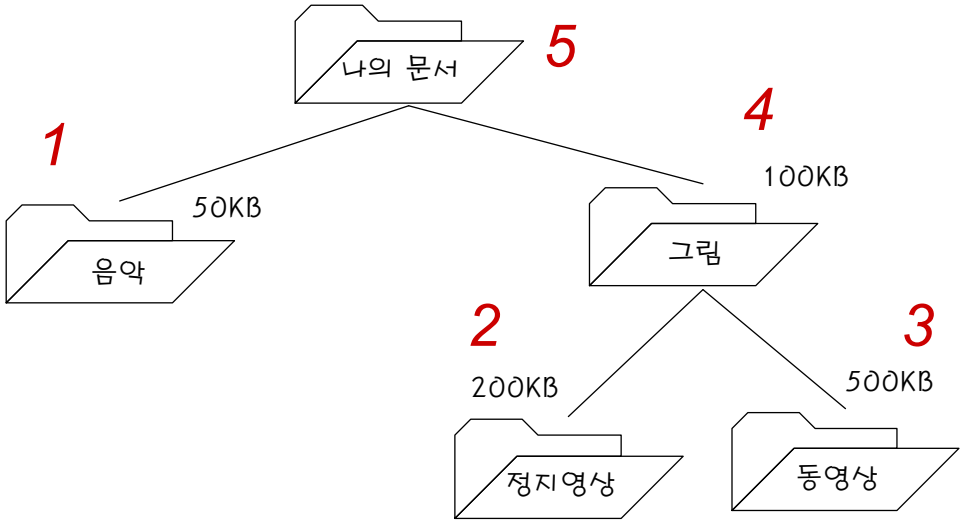
```
// 중위 순회
inorder( TreeNode *root ){
  if ( root ){
    inorder( root->left );
    printf("%d", root->data );
    inorder( root->right );
  }
}
// 전위 순회
preorder( TreeNode *root ){
  if ( root ){
    printf("%d", root->data );
    preorder( root->left );
    preorder( root->right );
  }
}
// 후위 순회
postorder( TreeNode *root ){
  if ( root ){
    postorder( root->left );
    postorder( root->right );
    printf("%d", root->data );
  }
}
```

```
void main()
{
  inorder(root);
  preorder(root);
  postorder(root);
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

후위 순회 응용:디렉토리 용량 계산

- 디렉토리의 용량을 계산하는데 후위 트리 순회 사용




```
int calc_direc_size(TreeNode *root)
{
  int left_dir, right_dir;
  if ( root ){
    left_size = calc_size( root->left );
    right_size = calc_size(root->right );
    return (root->data+left_size+right_size);
  }
}
void main()
{
  TreeNode n4={500, NULL, NULL};
  TreeNode n5={200, NULL, NULL};
  TreeNode n3={100, &n4, &n5};
  TreeNode n2={50, NULL, NULL};
  TreeNode n1={0, &n2, &n3};
  printf("디렉토리의 크기=%d\n",
    calc_direc_size(&n1));
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Arithmetic Expression

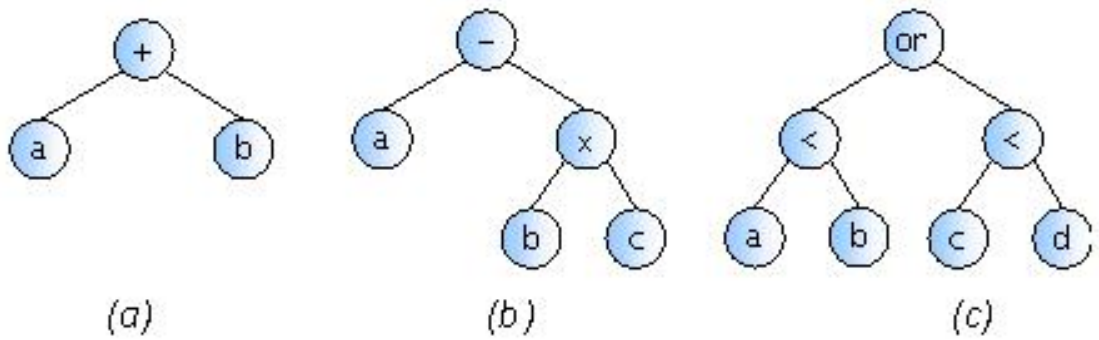
- 수식은 연산자(operator)와 피연산자(operand)로 이루어져 있다.
  - 연산자와 피연산자를 결합하여 수식을 표현한다.
- 수식을 표현하는 방법
  - 전위 표기법(prefix notation): * + A / + B C D - E F
  - 중위 표기법(infix notation): (A + ((B+C) /D)) * (E-F)
  - 후위 표기법(postfix notation): A B C +D / + E F - *

- 전위 표기법과 후위 표기법은 연산자의 우선 순위를 위해서 괄호를 사용할 필요가 없다는 장점이 있다.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

수식 트리

- 수식을 이진 트리로 표현한 것
  - 비단말노드(non-terminal node): 연산자(operator)
  - 단말노드(terminal node): 피연산자(operand)



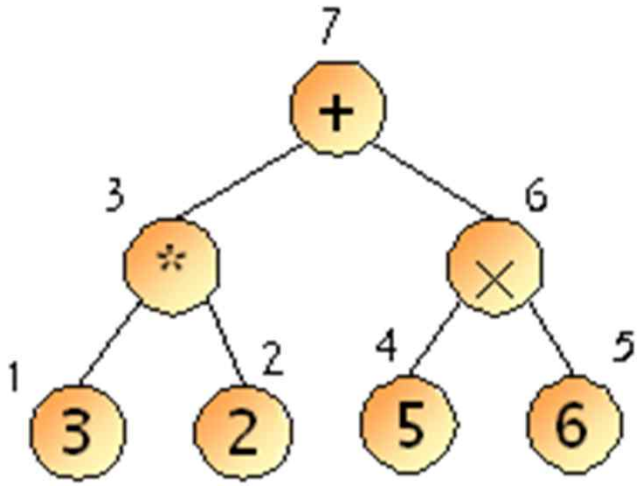
수식	a + b	a - (b × c)	(a < b) or (c < d)
전위순회	+ a b	- a × b c	or < a b < c d
중위순회	a + b	a - b × c	a < b or c < d
후위순회	a b +	a b c × -	a b < c d < or

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

수식 트리 계산(Evaluation)

- 후위순회를 사용
  - 서브트리의 값을 순환호출로 계산
- Non-terminal node를 방문
  - 양쪽 서브트리의 값을 먼저 계산
  - 노드에 저장된 연산자를 이용하여 계산한다

```
evaluate(exp)
1. if exp = NULL
2.    then return 0;
3.    else x←evaluate(exp->left);
4.         y←evaluate(exp->right);
5.         op←exp->data;
6.         return (x op y);
```



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Expression Tree Evaluation: C code

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

//                    +
//          *                   +
//         1 4                16 25
TreeNode n1={1, NULL, NULL};
TreeNode n2={4, NULL, NULL};
TreeNode n3={'*', &n1, &n2};
TreeNode n4={16, NULL, NULL};
TreeNode n5={25, NULL, NULL};
TreeNode n6={'+', &n4, &n5};
TreeNode n7={'+', &n3, &n6};
TreeNode *exp= &n7;
```

```
int evaluate(TreeNode *root) {
  if( root == NULL)  return 0;
  if( root->left == NULL
              && root->right == NULL)
    return root->data;
  else {
    int op1 = evaluate(root->left);
    int op2 = evaluate(root->right);
    switch(root->data){
      case '+':       return op1+op2;
      case '-':       return op1-op2;
      case '*':       return op1*op2;
      case '/':       return op1/op2;
    }
  }
  return 0;
}
```

```
void main()
{
  printf("%d", evaluate(exp));
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

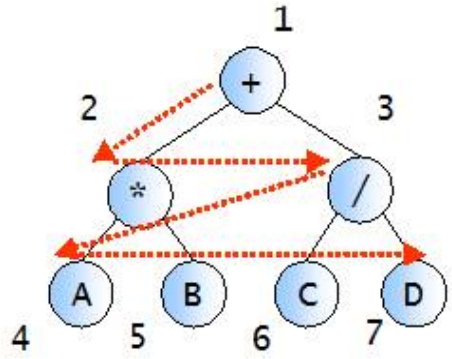
층별 순회(Level Order Traversal)

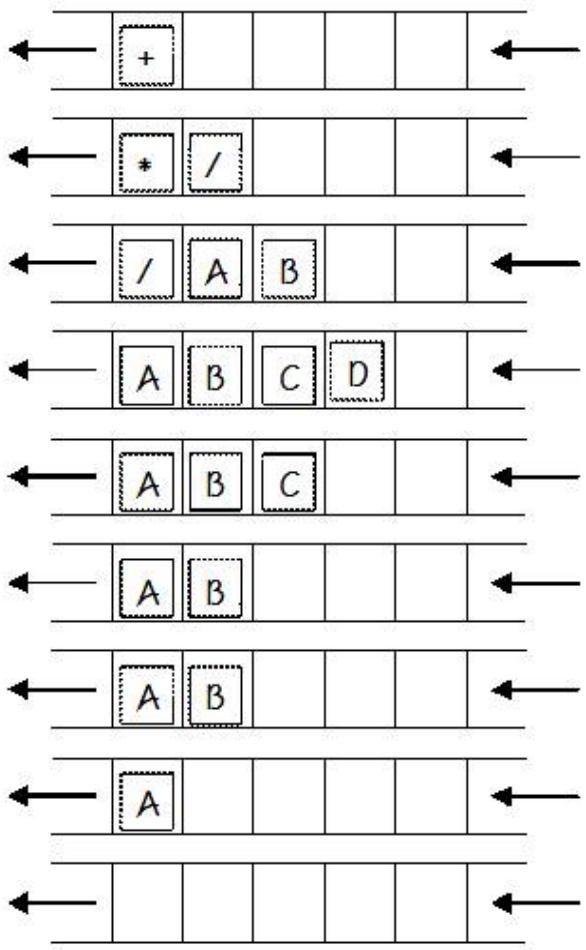
- *지금까지의 순회 법이 스택을 사용했던 것에 비해 레벨 순회는 큐를 사용하는 순회 법이다.*

단계1: 트리 T의 뿌리 r을 방문한다.

단계2: 트리 T의 level 2의 sibling 노드들을 방문한다.

단계3: level i(i=3,4,...)의 sibling 노드들을 방문한다.





Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

레벨 순회 알고리즘

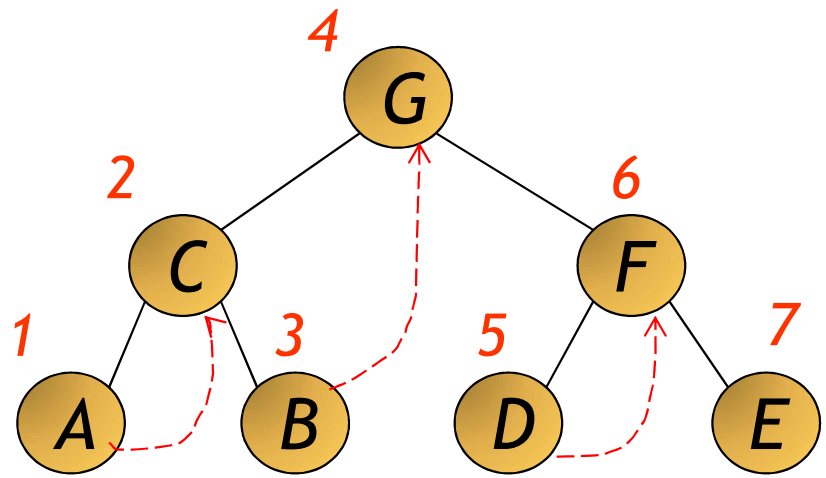
- **level_order(root)**

```
1. initialize queue;
2. enqueue(queue, root);
3. while is_empty(queue)≠TRUE do
4.     x← dequeue(queue);
5.     if( x≠NULL) then
6.         print DATA(x);
7.     enqueue(queue, LEFT(x));
8.     enqueue(queue, RIGHT(x));
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Threaded Binary Tree

- 이진트리의 NULL 링크를 이용하여 순환 호출 없이도 트리의 노드들을 순회
- NULL 링크에 중위 순회시에 후속 노드인 중위 후속자(**inorder successor**)를 저장시켜 놓은 트리가 스레드 이진 트리(threaded binary tree)



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Threaded Binary Tree: C code

```
// 중위 후속자를 찾는 함수
TreeNode *find_successor(TreeNode *p) {
  TreeNode *q = p->right; // q는 p의 오른쪽 포인터
  // 만약 오른쪽 포인터가 NULL 또는 스레드 --> 오른쪽 반환
  if( q==NULL || p->is_thread == TRUE) return q;
  // 만약 오른쪽 자식이면 다시 가장 왼쪽 노드로 이동
  while( q->left != NULL ) q = q->left;
  return q;
}
// 스레드 버전 중위 순회 함수
void thread_inorder(TreeNode *t)  {
  TreeNode *q;
  if ( q == NULL ) return;
  for (q=t; q->left; q = q->left);  // 가장 왼쪽 노드로 간다.
  do {
    printf("%c ", q->data);  // 데이터 출력
    q = find_successor(q); // 후속자 함수 호출
  } while(q);              // NULL이 아니면
}
```

```
// 단말노드와 비단말노드의 구분을 위하여 is_thread 필드 필요
typedef struct TreeNode {
  int data;
  struct TreeNode *left, *right;
  int is_thread;
  //만약 오른쪽 링크가 스레드이면 TRUE
} TreeNode;
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

ThreeJoin

TwoJoin

Split

JOINING AND SPLITTING BINARY SEARCH TREE

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

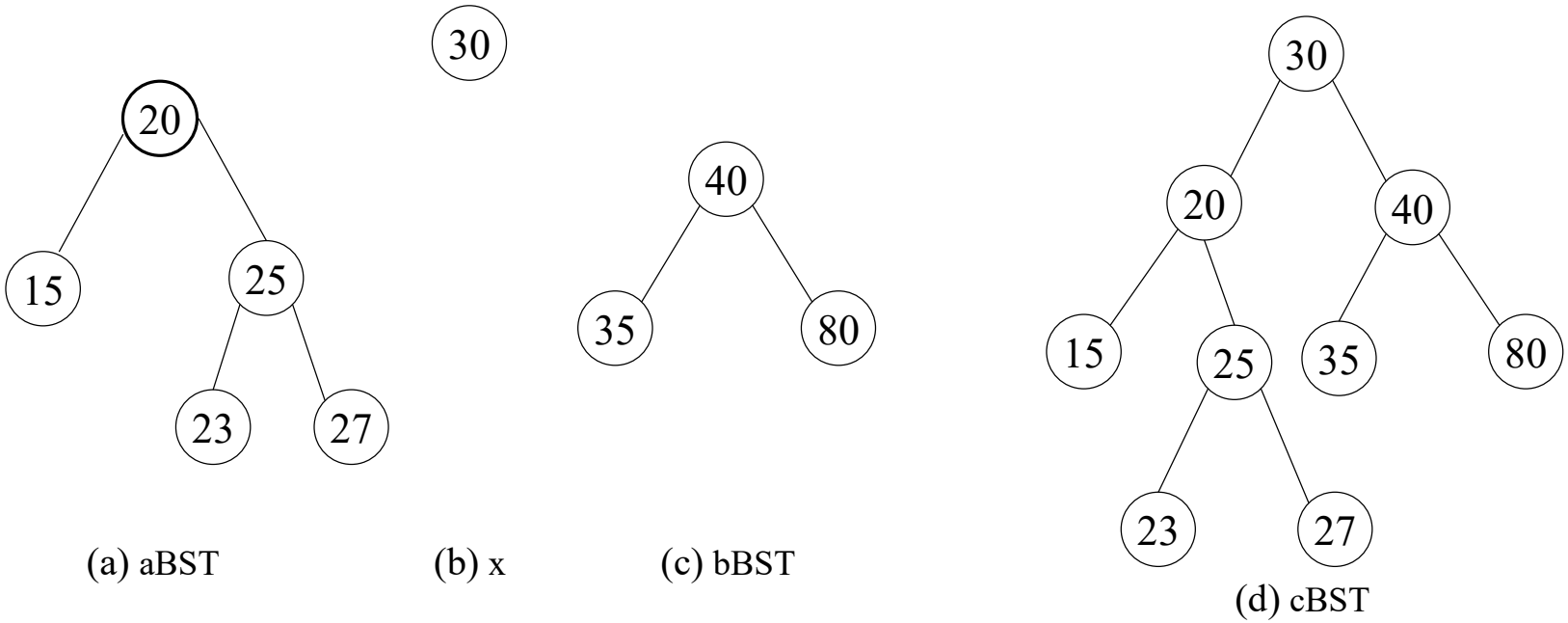
이진 탐색 트리의 결합과 분할

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

3원 결합의 예

- 연산 시간 : O(1)
- 이진 탐색 트리의 높이
  : max{height(aBST) , height(bBST)} + 1



(a) aBST (b) x (c) bBST (d) cBST

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

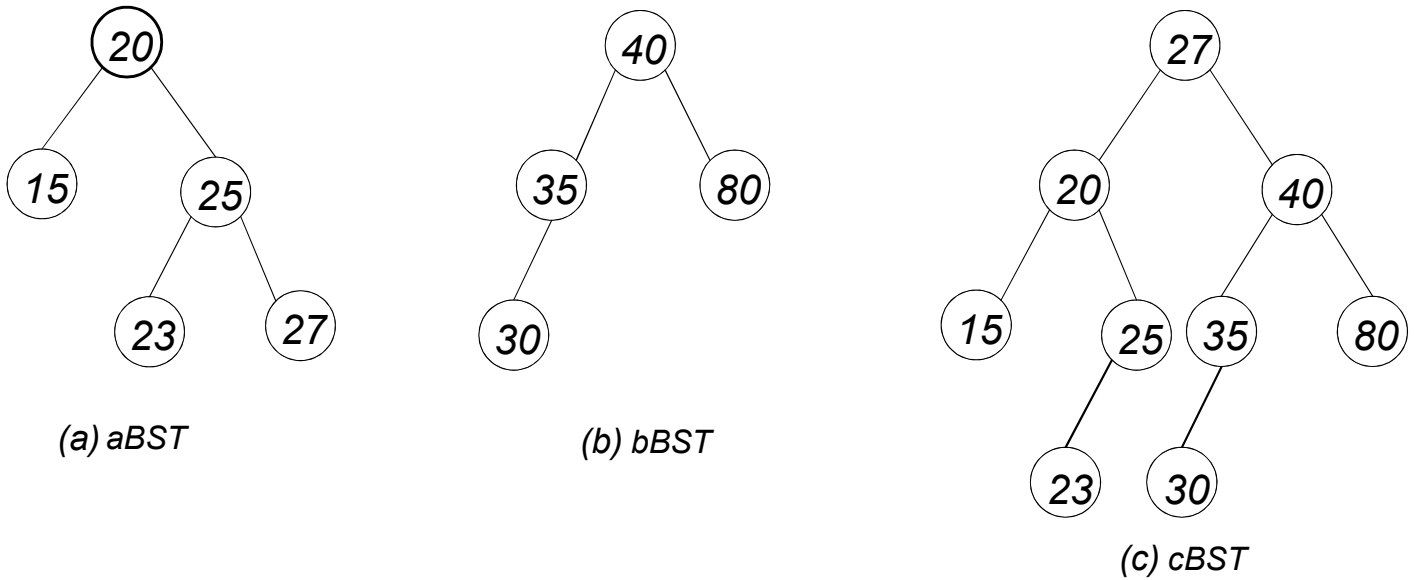
2원 결합

- twoJoin(aBST, bBST, cBST)
  - 두 이원 탐색 트리 aBST와 bBST를 결합하여 aBST와 bBST에 있는 모든 원소들을 포함하는 하나의 이원 탐색 트리 cBST를 생성
  - 가정
    - aBST와 bBST는 이진탐색트리
    - aBST의 모든 키값 < bBST의 모든 키값
    - 연산후 aBST와 bBST는 사용 하지 않으며 cBST는 이진탐색트리
- 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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

이원 결합 1

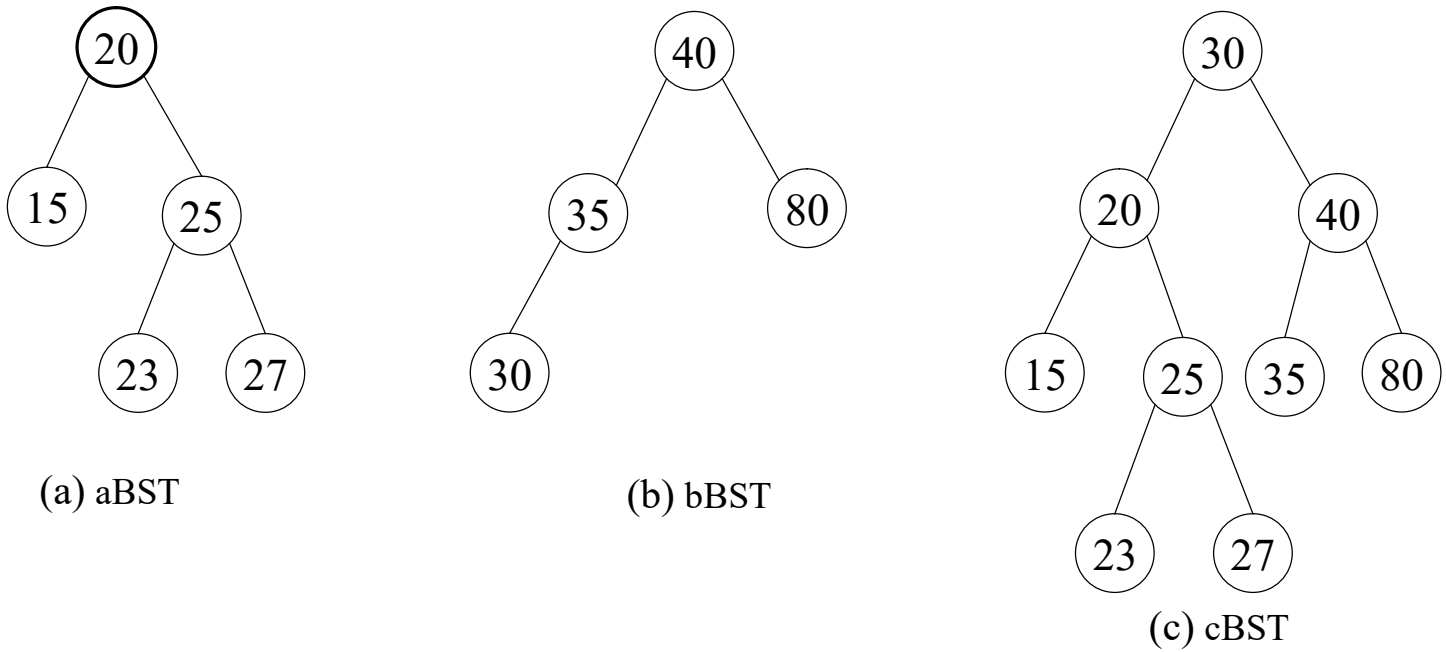
- aBST에서 키 값이 가장 큰 원소를 삭제
  - 이 결과 이원 탐색트리 : aBST'
  - 삭제한 가장 큰 키값 : max
  - threeJoin(aBST', max, bBST, cBST) 실행
- 실행 시간 : O(height (aBST))
- cBST의 높이 : max{height(aBST'), height(bBST)} + 1



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

이원 결합 2

- bBST에서 가장 작은 키값을 가진 원소를 삭제
  - 이 결과 이원 탐색 트리 : bBST’
  - 삭제한 가장 작은 키값 : min
  - threeJoin(aBST, min, bBST', cBST)를 실행



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

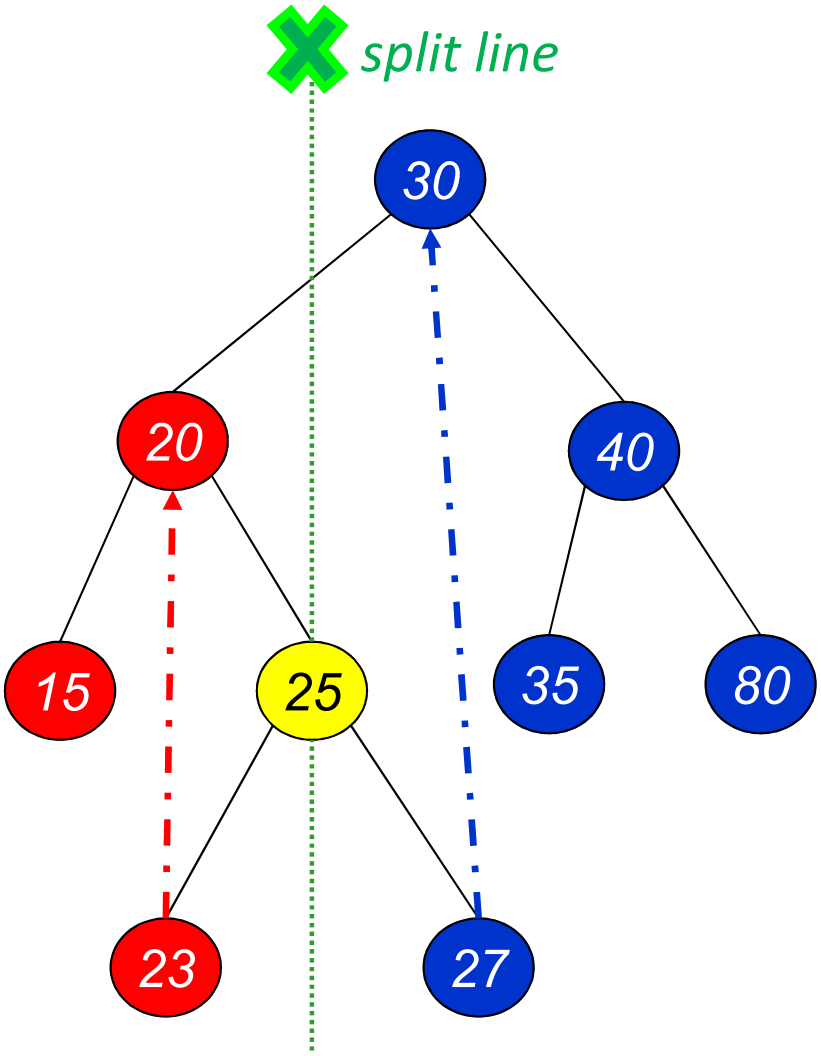
Splitting BST

- 분할 : split(aBST, x, bBST, cBST)
  - aBST 를 주어진 키값 x를 기준으로
    두 이진 탐색 트리 bBST와 cBST로 분할

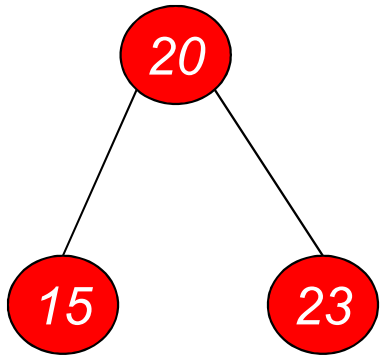
  - bBST : x보다 작은 키값을 가진 aBST의 모든 원소 포함
  - cBST : x보다 큰 키값을 가진 aBST의 모든 원소 포함
  - bBST와 cBST는 각각 이진 탐색 트리 성질을 만족
  - 키값 x가 aBST에 있으면 true 반환 , 아니면 false 반환

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

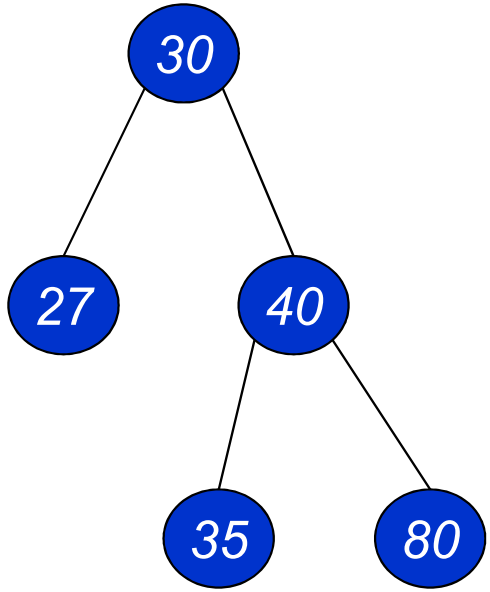
Split(aBST , 25 , bBST , cBST)



(a) aBST



(b) bBST



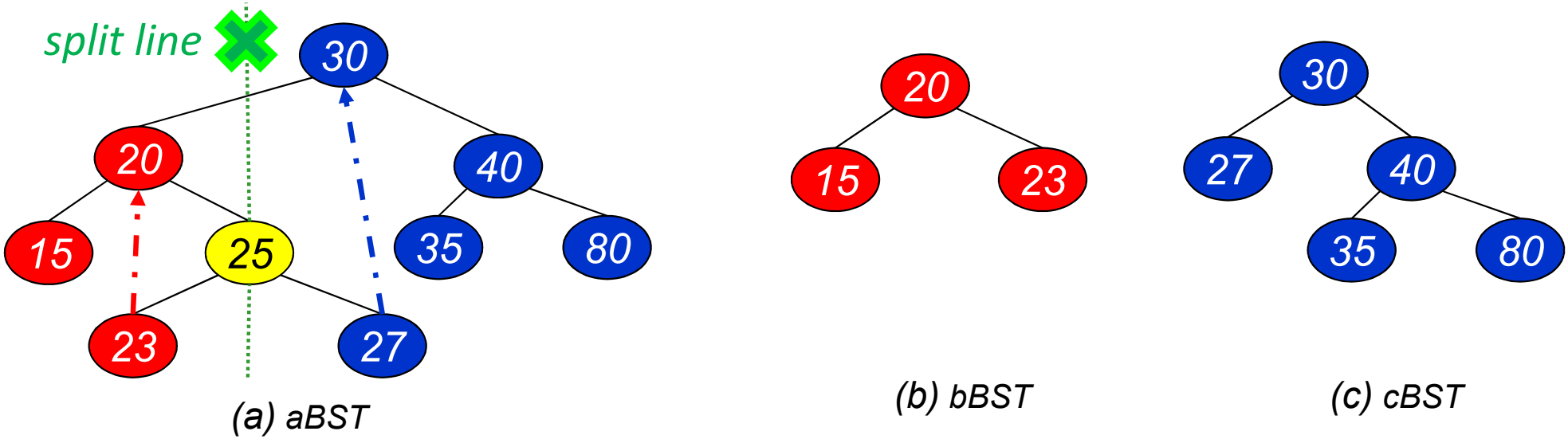
(c) cBST

https://www.learnbay.io/split-bst/

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

분할 연산 실행

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



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

분할 연산 알고리즘

```
def splitBST(root, V):
    if root == NULL: return NULL, NULL
    elif root.val == V: % V is root, remove
        return root.left, root.right
    elif root.val < V:  % V in right subtree
        a, b = splitBST(root.right, V)
        root.right = a
        return root, b
    else:    % V in left subtree
        a, b = splitBST(root.left, V)
        root.left = b
        return a, root
```

https://www.learnbay.io/split-bst/

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

END OF LECTURE 9

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I