Trees (Chap. 4)

School of Electrical Engineering
Korea University

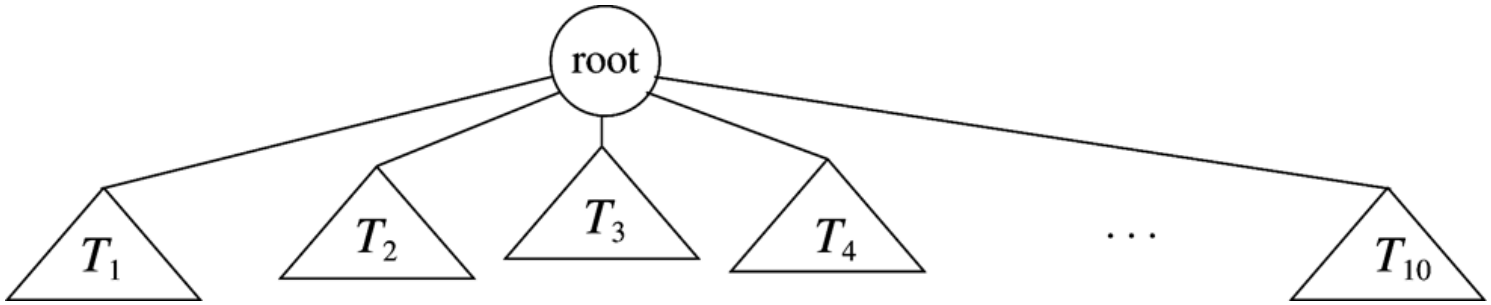
Chap 4: Trees

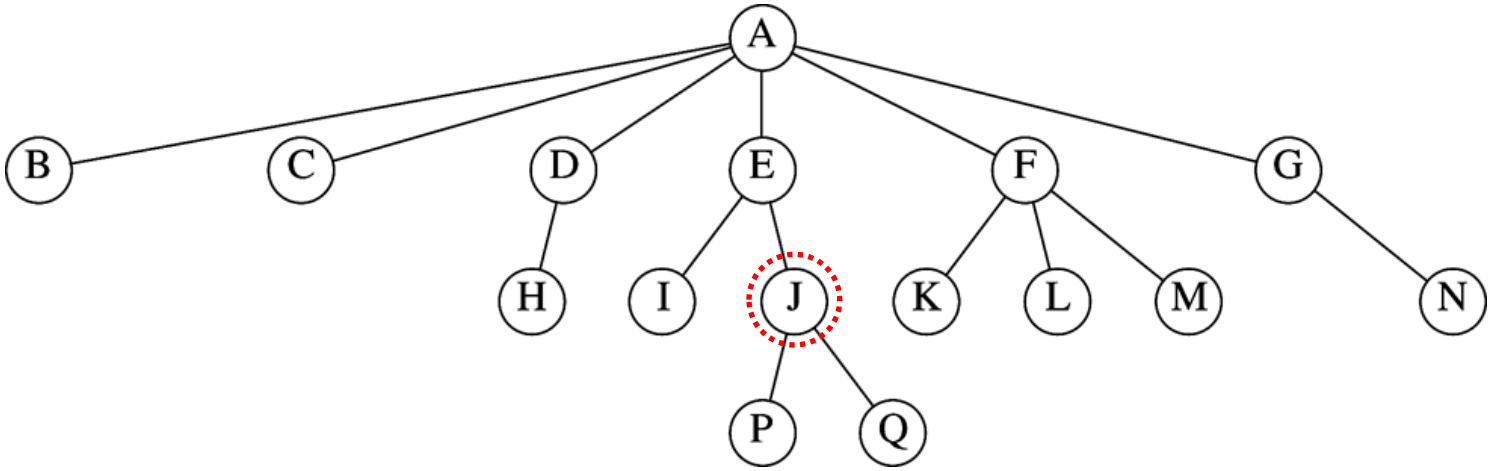
- Access time *O(N)* for linked lists.
- if *N>>1*, faster access is needed.
- *O(log N)* access time for trees.
- Non-linear structure.

Recursive definition

- A tree is a collection of nodes.
- The collection can be empty.
- Otherwise, a tree consists of a distinguished node *r*, called the *root*, and zero or more nonempty (sub)trees T_1, T_2, ⋯, T_k, each of whose roots are connected by a directed *edge* from *r*.
- The root of each subtree is said to be a *child* of r, and r is the *parent* of each subtree root.

Generic tree and example





Terms of Trees

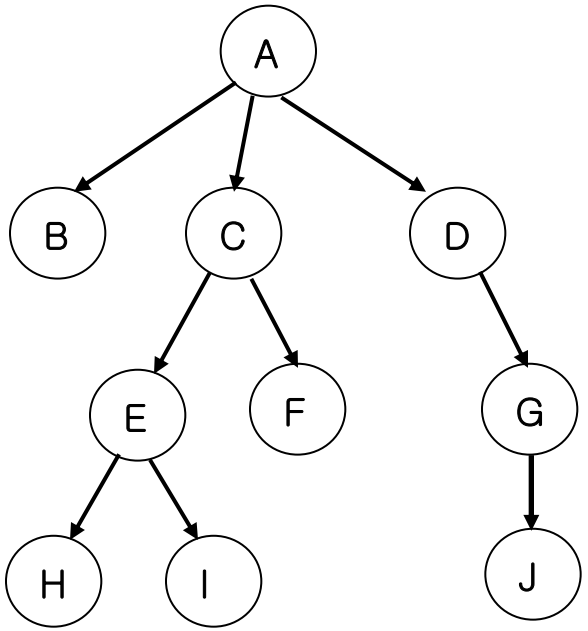
- Root, leaf, parent, children, grand-*, sibling
- Path from n_1 to n_k: a sequence of nodes n_1, n_2, …, n_k such that n_i is the parent of n_{i+1}
- Length of a path: the number of edges on the path
- The depth(level) of a node: the length of the path from the root to the node.
- The height of a node: the length of the longest path from the node to a leaf
- The height of a tree: the height of the root

Properties of Trees

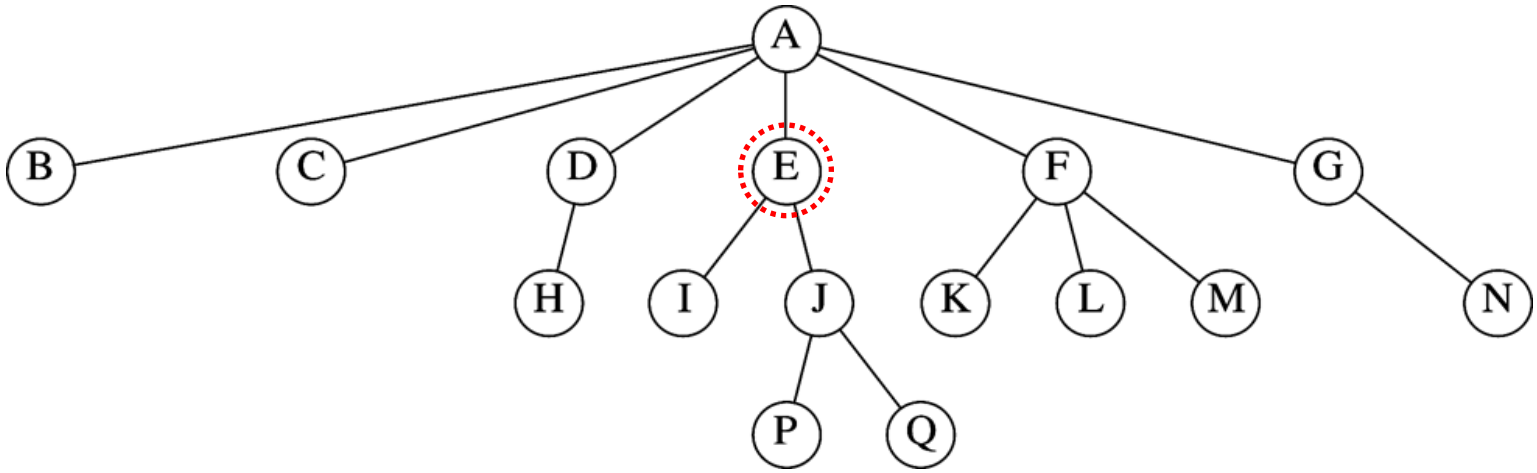
- No loops
- Unique path between two nodes
- Every node except the root has one parent
- *N−1* edges for an *N* node tree. How?

Tree ADT

- Operations
  - Parent(n)
  - Leftmost-child(n)
  - Right-sibling(n)
  - Depth(), Height()
  - makeEmpty()
  - isEmpty()
  - Preorder( )
  - Postorder( )
  - Inorder( )



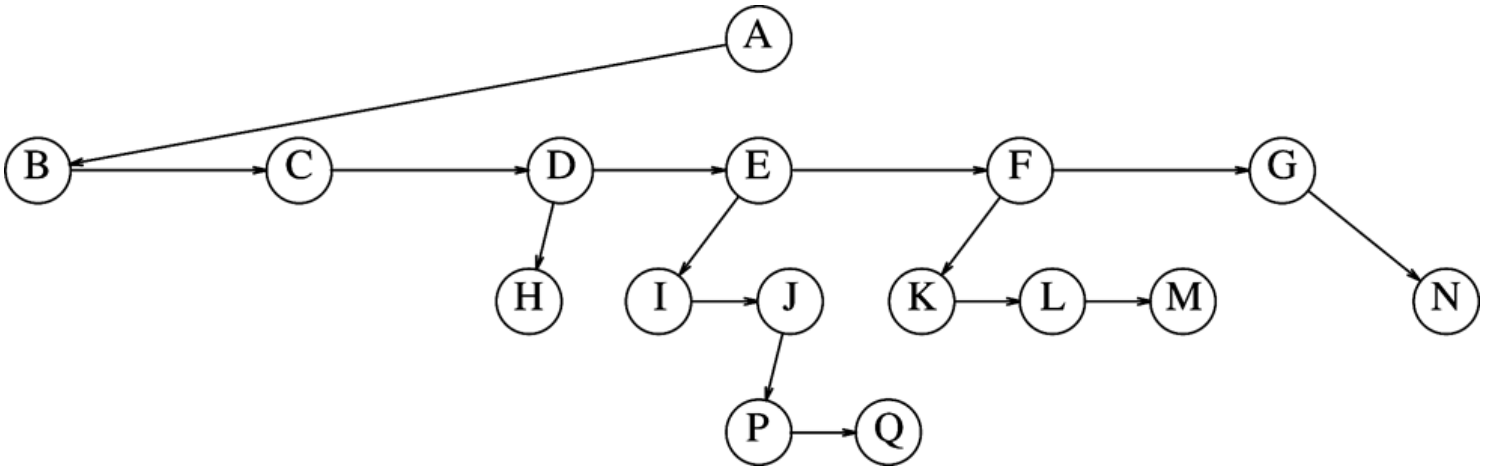
Example



- The length of path AEJQ is 3
- E is at depth 1 and height 2
- F is at depth 1 and height 1
- The height of the tree is 3

Implementation

```
typedef struct TreeNode *PtrToNode;
struct TreeNode
{
      ElementType       Element;
      PtrToNode         FirstChild;
      PtrToNode         NextSibling;
}
```



Tree Traversal Application

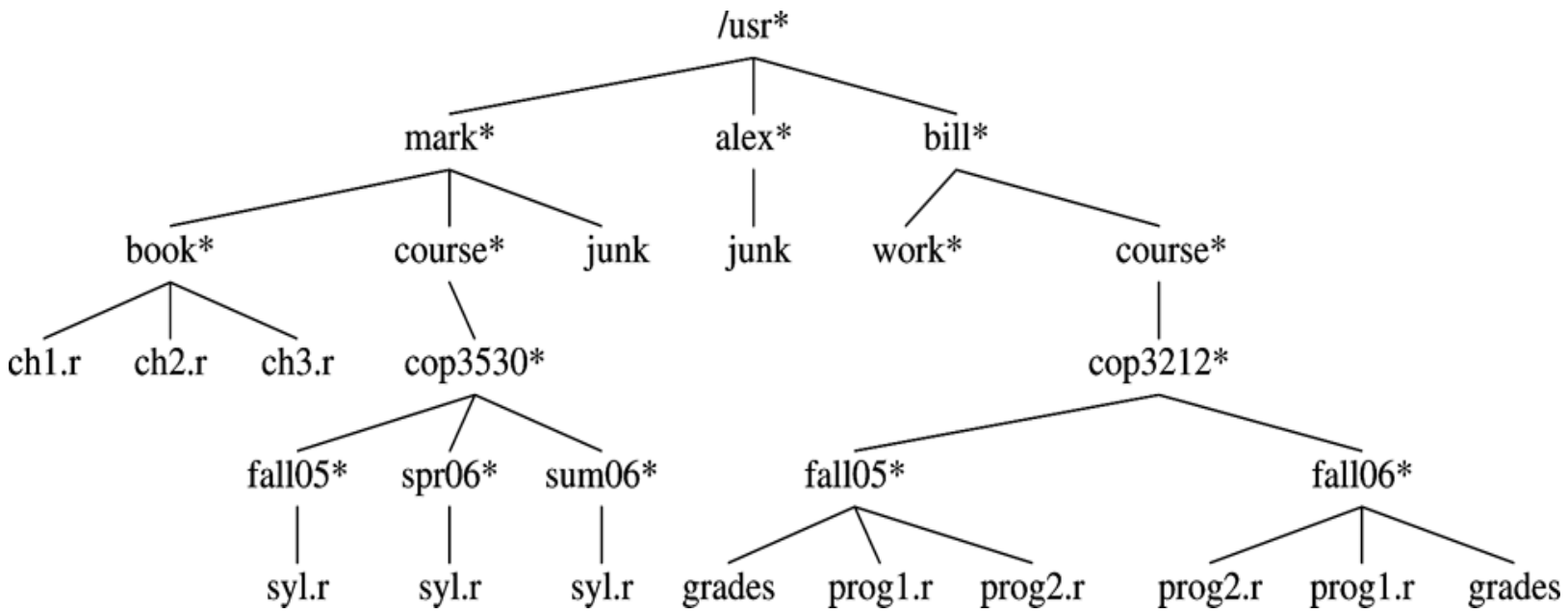


Figure 4.5 Unix directory

Routine to list a directory

```
        static void
        ListDir( DirectoryOrFile D, int Depth )
        {
/* 1*/      if( D is a legitimate entry )
            {
/* 2*/          PrintName( D, Depth );
/* 3*/          if( D is a directory )
/* 4*/              for each child, C, of D
/* 5*/                  ListDir( C, Depth + 1 );
            }
        }

        void
        ListDirectory( DirectoryOrFile D )
        {
            ListDir( D, 0 );
        }
```

Figure 4.6 Routine to list a directory in a hierarchical file system

Preorder directory listing

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall05
                    syl.r
                spr06
                    syl.r
                sum06
                    syl.r
        junk
    alex
        junk
```

```
    bill
        work
        course
            cop3212
                fall05
                    grades
                    prog1.r
                    prog2.r
                fall06
                    prog2.r
                    prog1.r
                    grades
```

Unix directory with file sizes

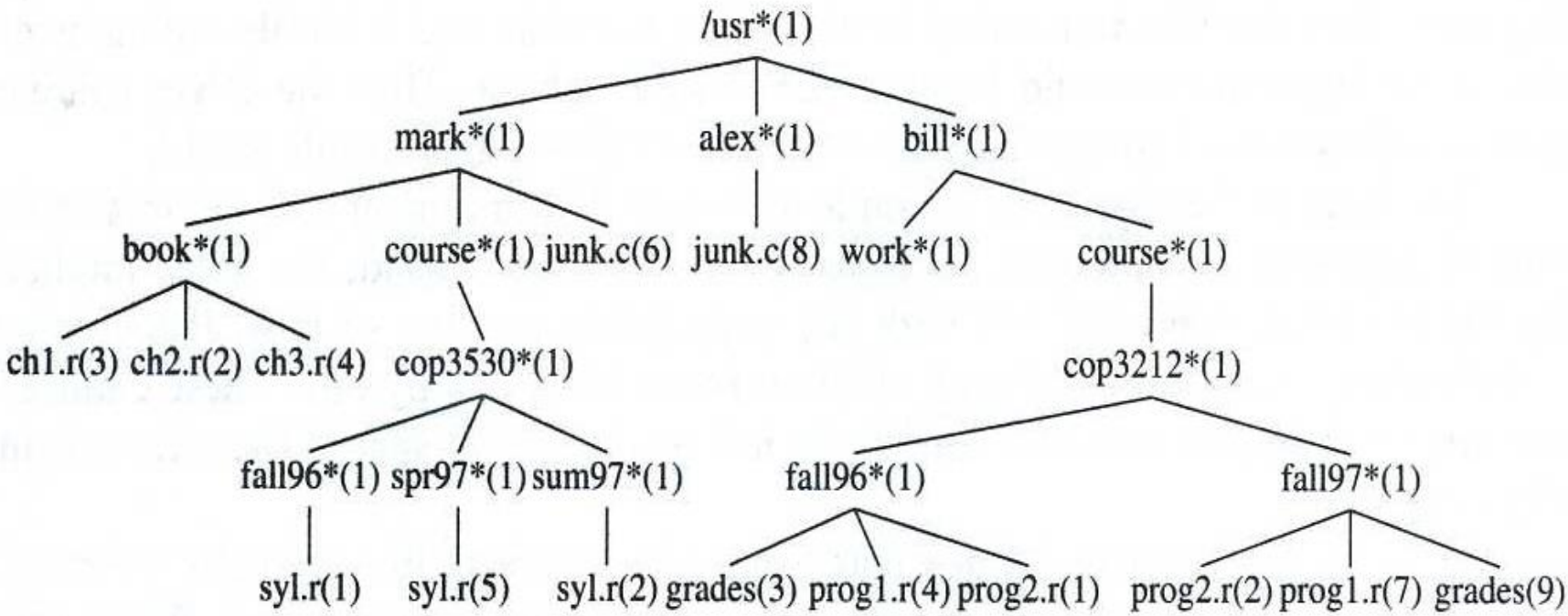


Figure 4.8 Unix directory with file sizes obtained via postorder traversal

Routine to calculate the dir size

Figure 4.9 Routine to calculate the size of a directory

```
        static void
        SizeDirectory( DirectoryOrFile D )
        {
            int TotalSize;

/* 1*/      TotalSize = 0;
/* 2*/      if( D is a legitimate entry )
            {
/* 3*/          TotalSize = FileSize( D );
/* 4*/          if( D is a directory )
/* 5*/              for each child, C, of D
/* 6*/                  TotalSize += SizeDirectory( C );
            }
/* 7*/      return TotalSize;
        }
```

Trace of SizeDirectory

```
        ch1.r            3
        ch2.r            2
        ch3.r            4
    book                10
            syl.r        1
        fall96           2
            syl.r        5
        spr97            6
            syl.r        2
        sum97            3
    cop3530             12
    course              13
    junk.c               6
mark                    30
    junk.c               8
```

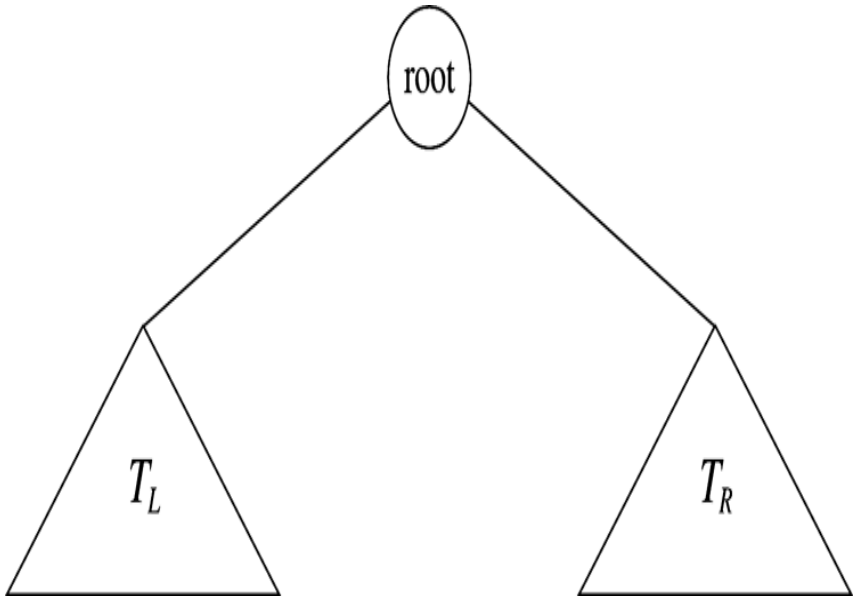
```
alex                     9
    work                 1
            grades       3
            prog1.r      4
            prog2.r      1
        fall96           9
            prog2.r      2
            prog1.r      7
            grades       9
        fall97          19
    cop3212             29
course                  30
bill                    32
/usr                    72
```

Figure 4.10 Trace of the *SizeDirectory* function

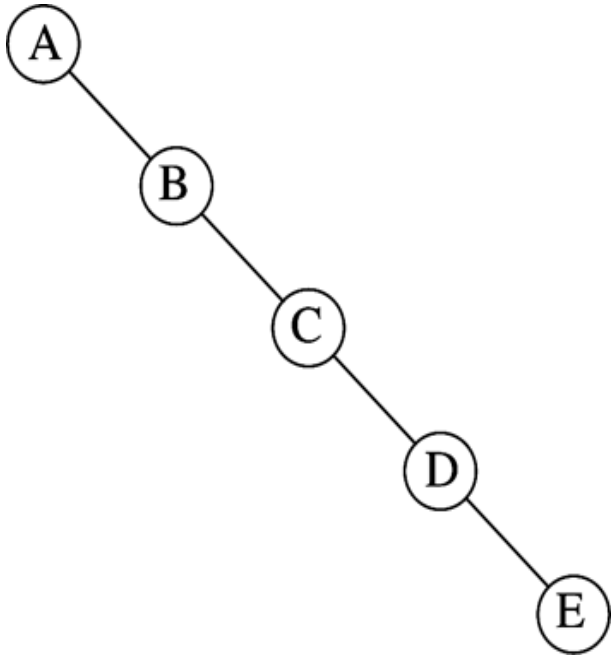
BTs: Binary Trees

- At most two children–left_child, right_child
- Full BT, complete BT
- height = $\lfloor \log N \rfloor$ for *N* node full BT
  min= $\lfloor \log N \rfloor$,
  max= *N–1*

Binary Tree



Generic binary tree



Skewed tree

Properties of Full BTs

- $N = \sum_{k=0}^{h} 2^k = 1 + 2^1 + 2^2 + 2^3 + \ldots + 2^h$ (height=*h*)

$$= \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1$$

- No. of non-leaf nodes = $2^h - 1$

  No. of leaf nodes = 2^h

- $N + 1 = 2^{h+1}$ $\qquad h = \lfloor \log N \rfloor$

BT implementation

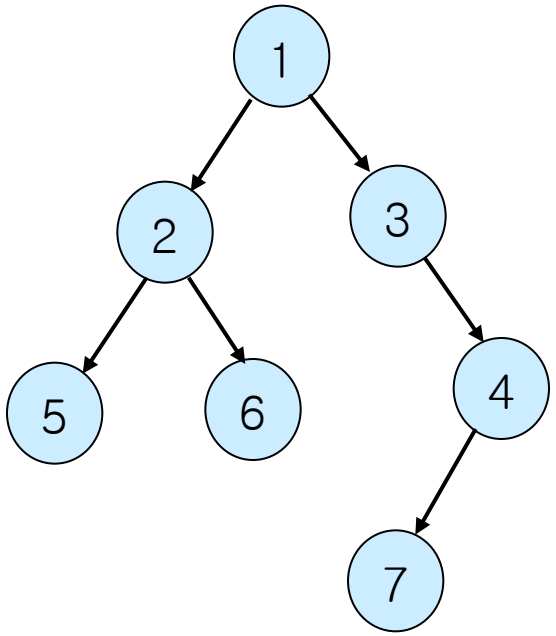
```
typedef struct TreeNode *PtrToNode;
typedef PtrToNode BinaryTree;

struct TreeNode
{ ElementType Element;
  Tree Left;
  Tree Right;
 };
```

Recursions for Binary Trees

```
void PrintTree(BinaryTree T) {    //Inorder traversal
   if (T != NULL) {
         PrintTree(T->Left);
         PrintElement(T->Element);
         PrintTree(T->Right);
   }
 }
int Height(BinaryTree T) {
    if (T == NULL)
         return  -1;
    else
         return 1+ Max(Height(T->Left), Height(T->Right));
   }
```

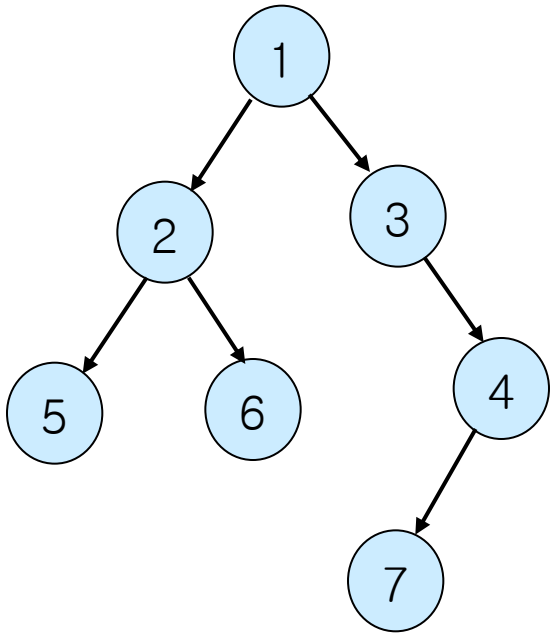
Inorder Binary Tree traversals

```
void INORDER(T)
{
   if(T is null)
      return;

   INORDER(T->Left);
   print(T->Element);
   INORDER(T->Right);
}
```



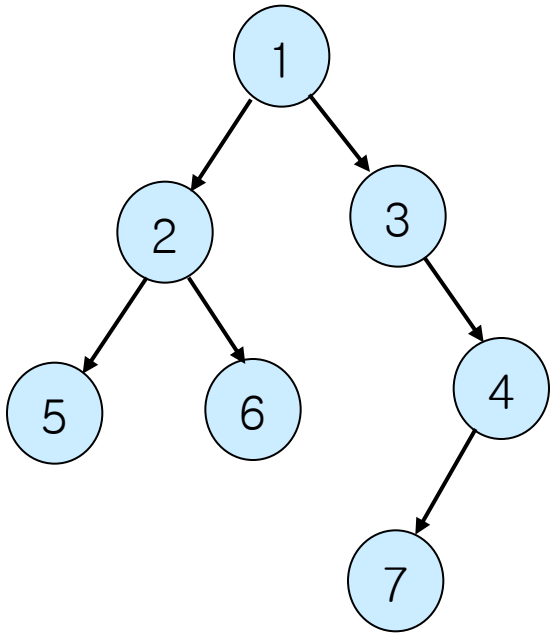
Binary Tree traversals

```
void PREORDER(T)
{
  if(T is null)
     return;

  print(T->Element);
  PREORDER(T->Left);
  PREORDER(T->Right);
}
```

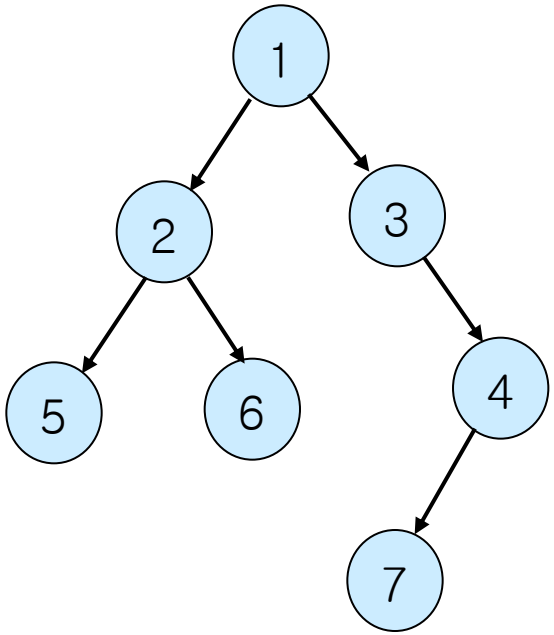


Binary Tree traversals

```
void POSTORDER(T)
{
    if(T is null)
      return;

    POSTORDER(T->Left);
    POSTORDER(T->Right);
    print(T->Element);
}
```



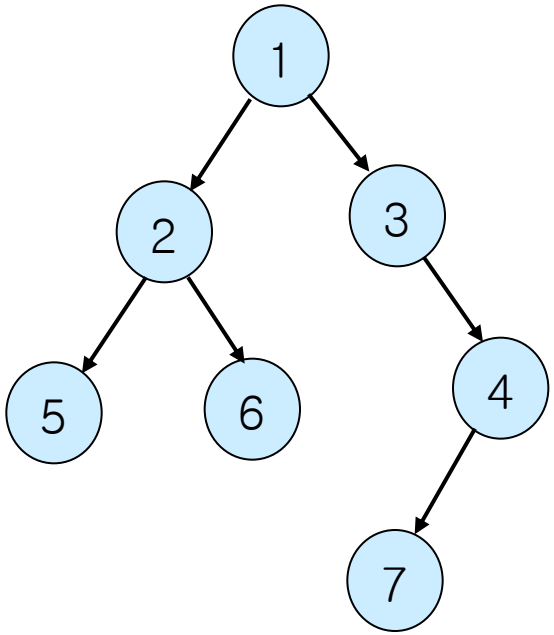
Tree Traversal



Preorder traversal

```
print  1
Pre(2)
        print  2
        Pre(5)
                print  5
        Pre(6)
                print  6
Pre(3)
        print  3
        Pre(4)
                print 4
                Pre(7)
                        print  7
```

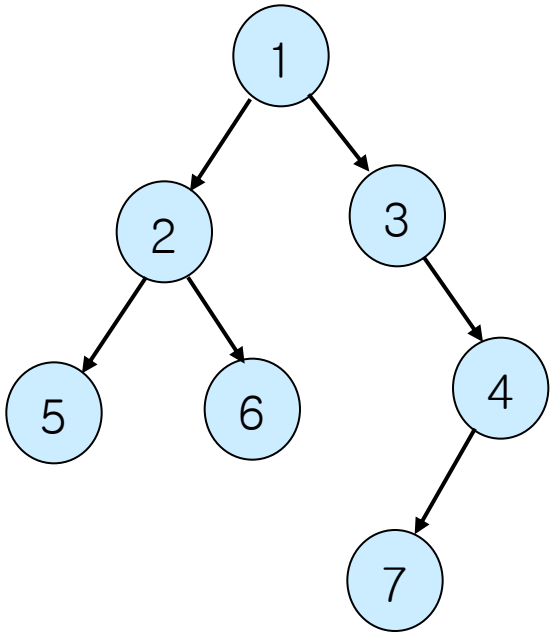
Tree Traversal



Postorder traversal

```
Post(2)
        Post(5)
                print  5
        Post(6)
                print  6
        print  2
Post(3)
        Post(4)
                Post(7)
                        print  7
                print  4
        print  3
print  1
```

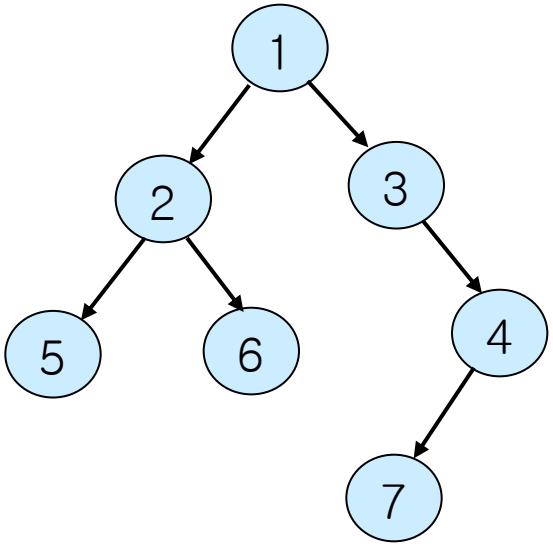
Tree Traversal



Postorder traversal

```
Post(2)
        Post(5)
                print  5
        Post(6)
                print  6
        print  2
Post(3)
        Post(4)
                Post(7)
                        print  7
                print  4
        print  3
print  1
```

Tree Traversal




```
void POST(T){
    if(T is null)
        return;
    POSTORDER(T->Left);
    POSTORDER(T->Right);
    print(T->Element);
}
```

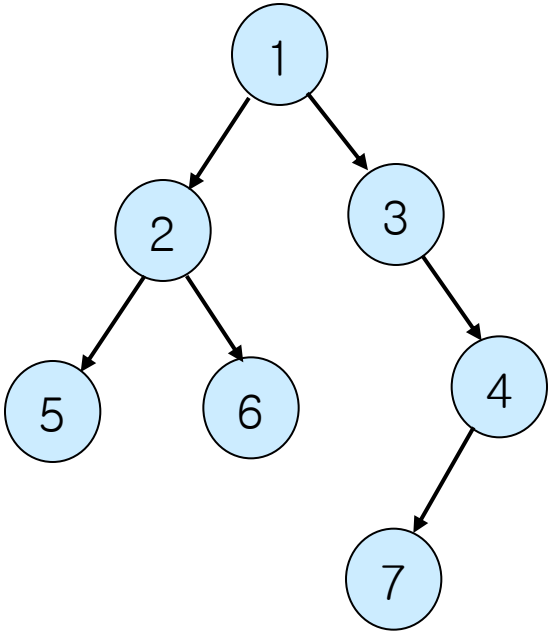
Postorder traversal

```
start
POST(2)
POST(5)
print 5
POST(6)
print 6
print 2
POST(3)
POST(4)
POST(7)
print 7
print 4
print 3
print 1
end
```

indent

```
start
POST (2)
    POST (5)
        print 5
    POST (6)
        print 6
    print 2
POST (3)
    POST (4)
        POST (7)
            print 7
        print 4
    print 3
print 1
end
```

Tree Traversal



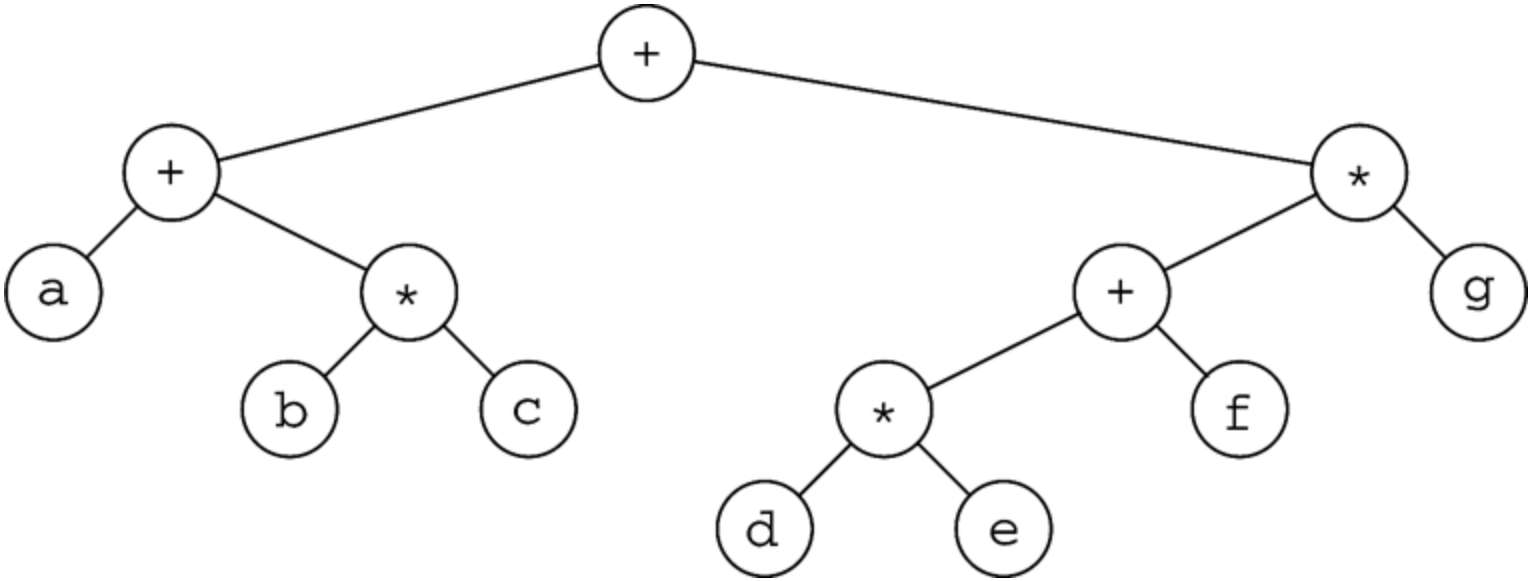
Inorder traversal => 5 2 6 1 3 7 4

Preorder traversal => 1 2 5 6 3 4 7

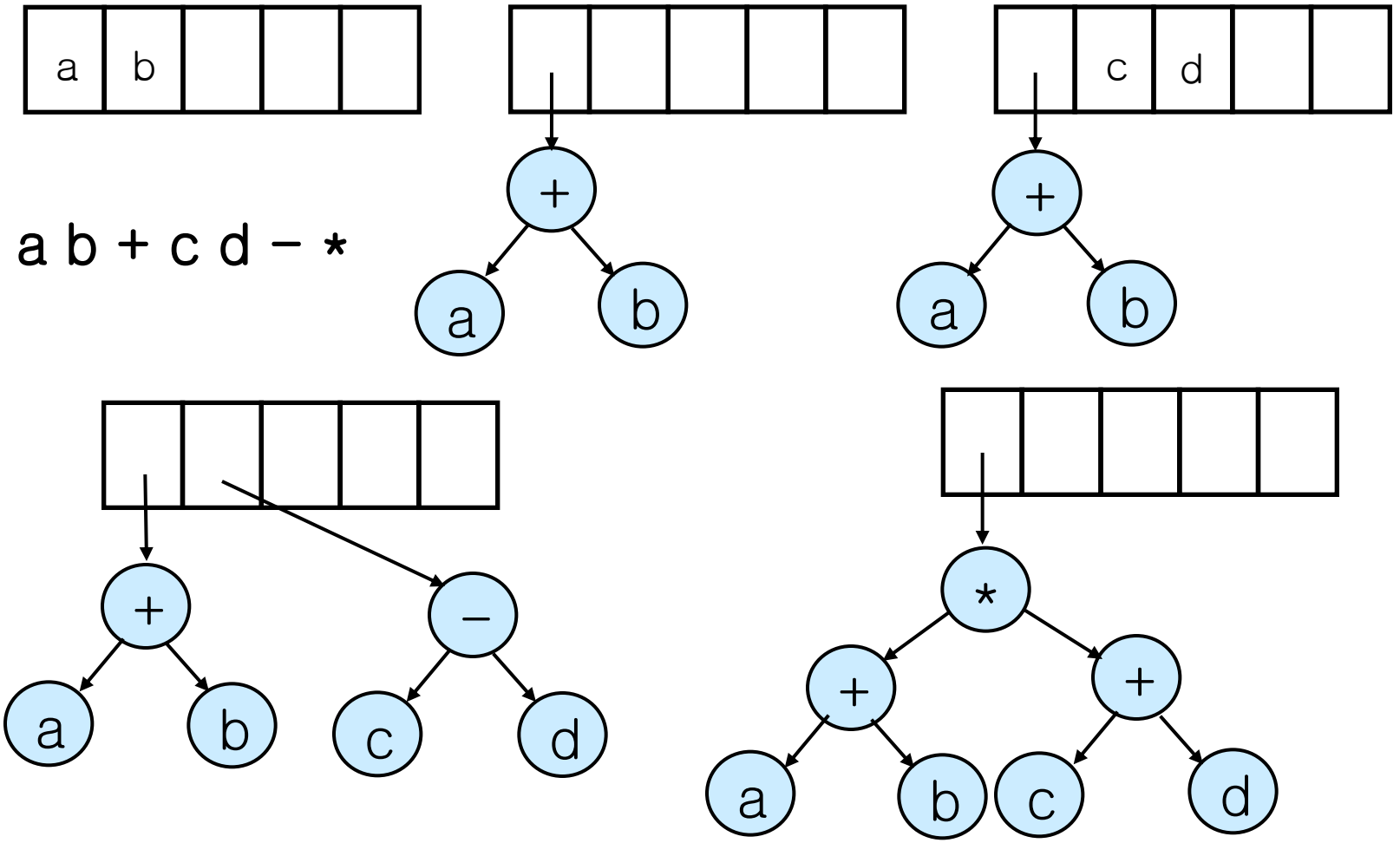
Postorder traversal => 5 6 2 7 4 3 1

Expression Trees

Exp. tree for (a + b * c) + ((d * e + f) * g)



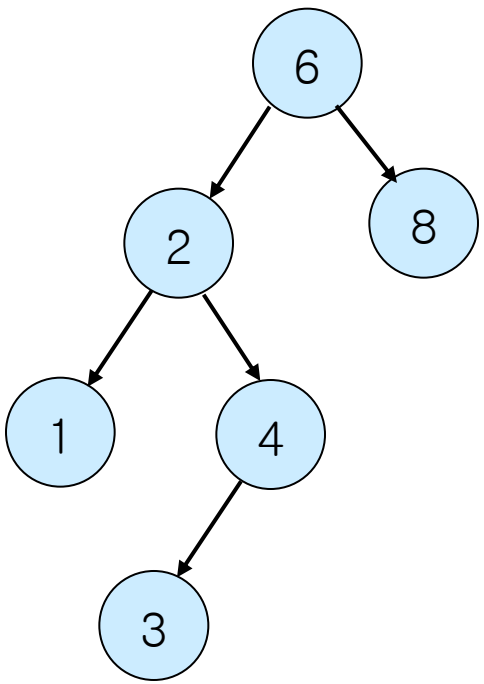
Postfix to expression tree

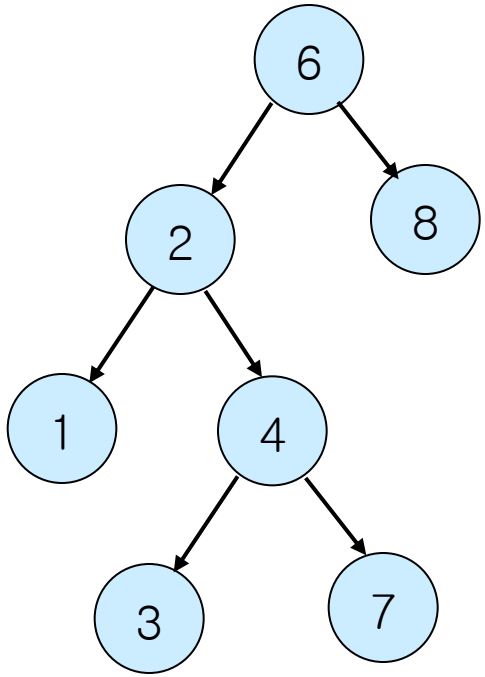


Binary Search Trees

- Keys – stored data in the node
- Key order(larger, smaller)
- For any node v T,
  Values in the left subtree – smaller
  Values in the right subtree – larger
- Used to store and retrieve data efficiently & dynamically (sort, matching, ...)
- Need fast access? Make it *balanced!*

Two binary trees – which one BST?





Binary search tree declaration

```
typedef int ElementType;

struct TreeNode;
typedef struct TreeNode *Position;
typedef struct TreeNode *SearchTree;

SearchTree MakeEmpty( SearchTree T );
Position Find( ElementType X, SearchTree T );
Position FindMin( SearchTree T );
Position FindMax( SearchTree T );
SearchTree Insert( ElementType X, SearchTree T );
SearchTree Delete( ElementType X, SearchTree T );
ElementType Retrieve( Position P );
```

To make an empty tree

```
struct TreeNode  {
      ElementType Element;
      SearchTree  Left;
      SearchTree  Right;
};
SearchTree MakeEmpty(SearchTree T) {
    if(T != NULL) {
         MakeEmpty(T->Left);
         MakeEmpty(T->Right);
         free( T );
    }
    return NULL;
}
```

Functions in BSTs

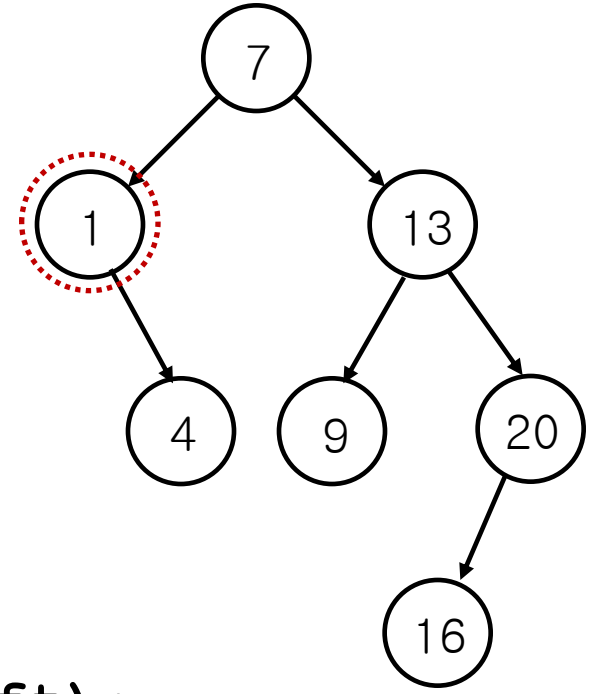
- *Find (X, T)* – return the pointer of node that has X.
- *Insert(X, T)* – if X is already there, do nothing; else add X at the right place.
- *FindMin/Max(T)* – return the minimum/ maximum in the BST *T*. (leftmost/rightmost)
- *Delete(X, T)* –Delete X and reconstruct the tree if necessary *(hard).*

Find on binary search tree

```
Position Find(ElementType X, SearchTree T)
{
    if(T == NULL)
        return NULL;
    if(X < T->Element)
        return Find(X, T->Left);
    else
        if(X > T->Element)
            return Find(X, T->Right);
        else
            return T;
}
```

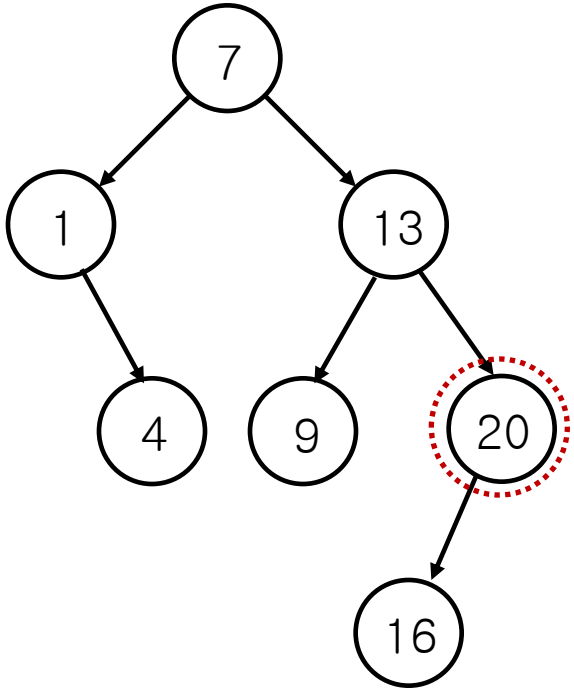
FindMin

```
Position
FindMin(SearchTree T)
{
  if(T == NULL)
     return NULL;
  else
     if(T-> Left == NULL)
         return T;
     else
         return FindMin(T->Left);
}
```



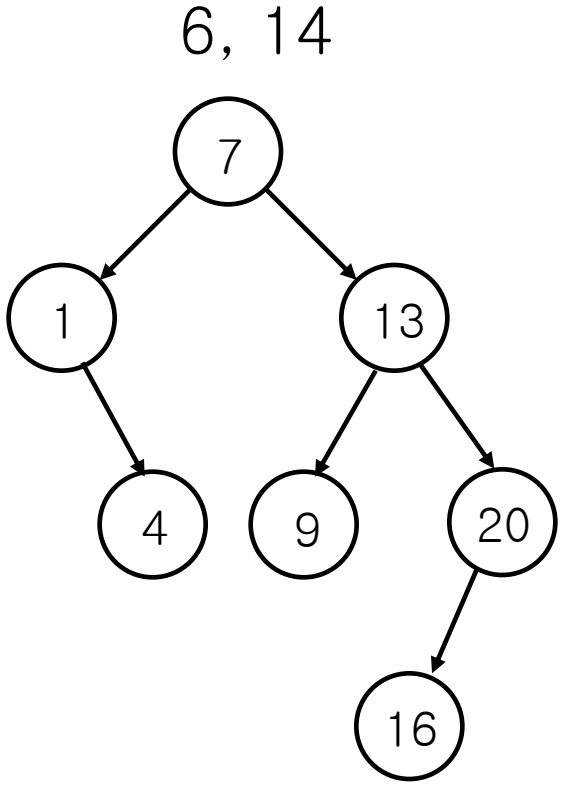
FindMax (nonrecursive)

```
Position
FindMax(SearchTree T)
{
    if(T!=NULL)
        while(T->Right!=NULL)
            T=T->Right;

    return T;
}
```

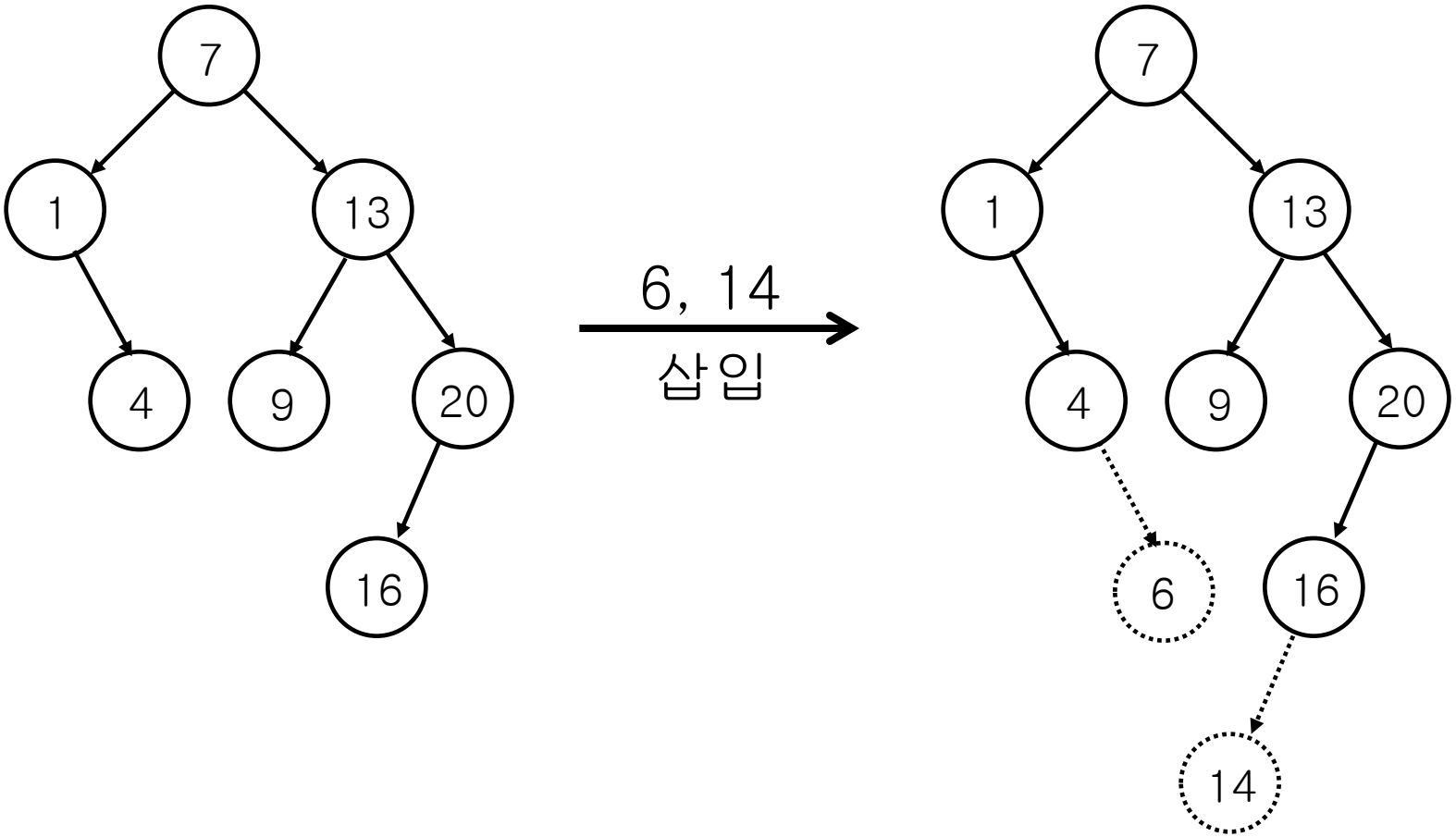


Insert in BST

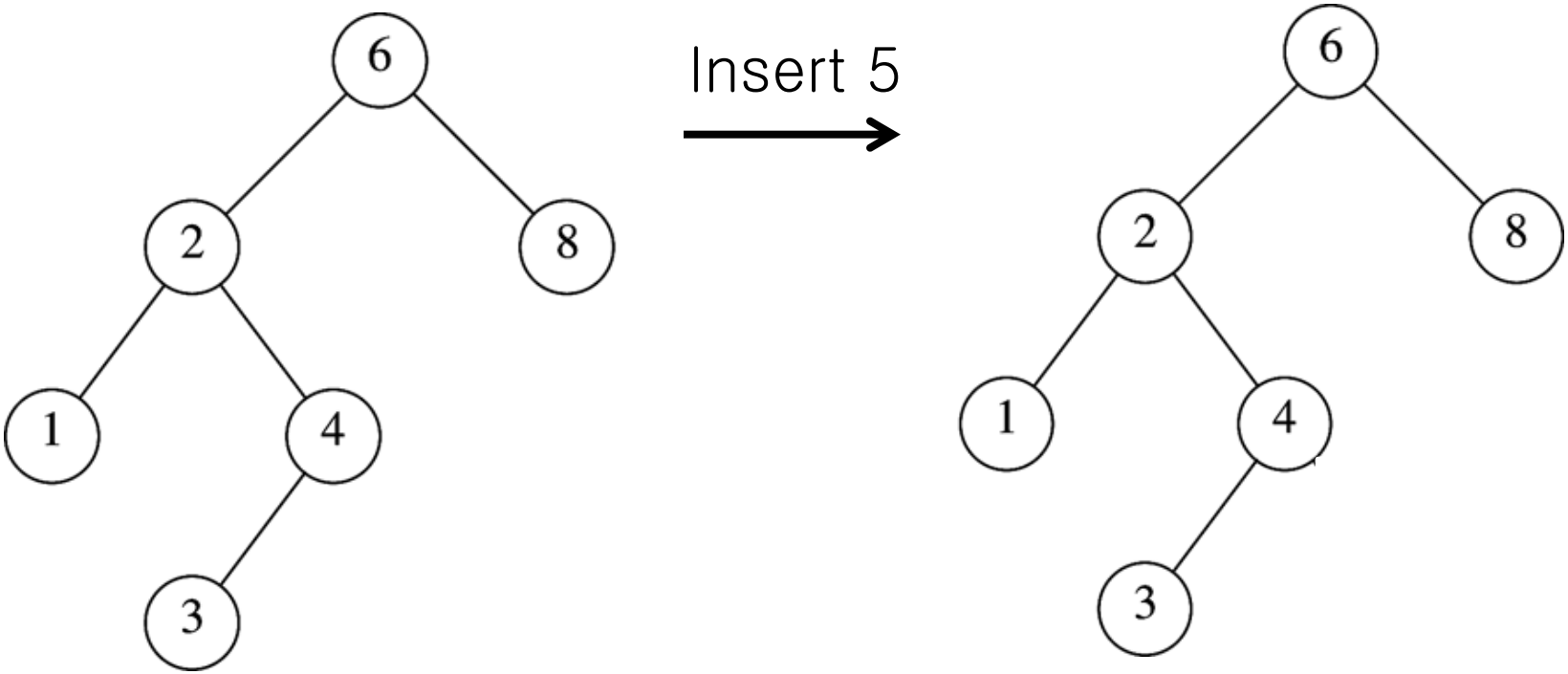
- Insert(X,T) – Find(X,T)를 실행했을 때 X가 자리잡고 있어야 할 지점을 찾은 후 그 위치에 새로 노드를 생성해 삽입함.



Insert in BST



More example

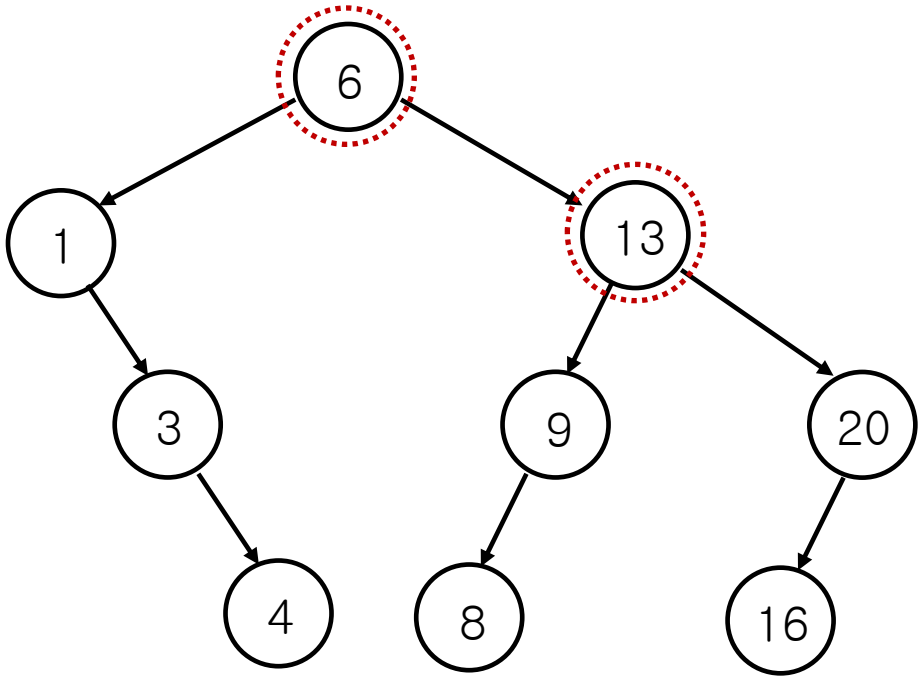


```
        SearchTree Insert(ElementType X, SearchTree T) {
/* 1*/      if(T == NULL) {
                /* Create and return a one-node tree */
/* 2*/          T = malloc(sizeof(struct TreeNode));
/* 3*/          if(T == NULL)
/* 4*/              FatalError("Out of space!!!");
                else {
/* 5*/              T->Element = X;
/* 6*/              T->Left = T->Right = NULL; }
            } else
/* 7*/      if(X < T->Element)
/* 8*/          T->Left = Insert(X, T->Left);
            else
/* 9*/      if(X > T->Element)
/*10*/          T->Right = Insert(X, T->Right);
            /* Else X is in the tree already; we'll do nothing */
/*11*/      return T;  /* Do not forget this line!! */
        }
```

Delete in BST

- *Delete(X, T)* : 재귀함수(recursion)를 써서 구현함.

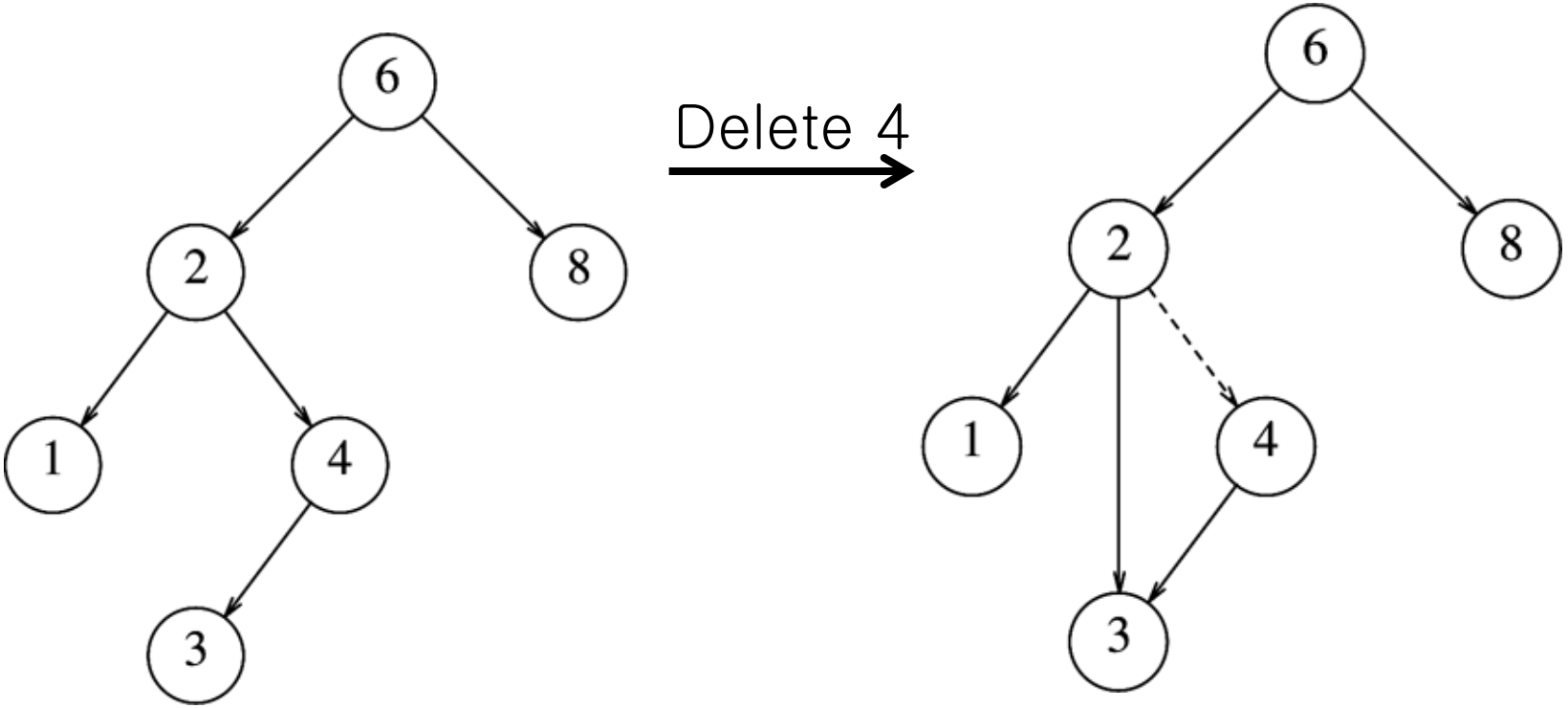
  /* recursion 결과를 assign 하는 경우 매번 return 되는 값이 무엇인지 정확히 추적하여야 함 */



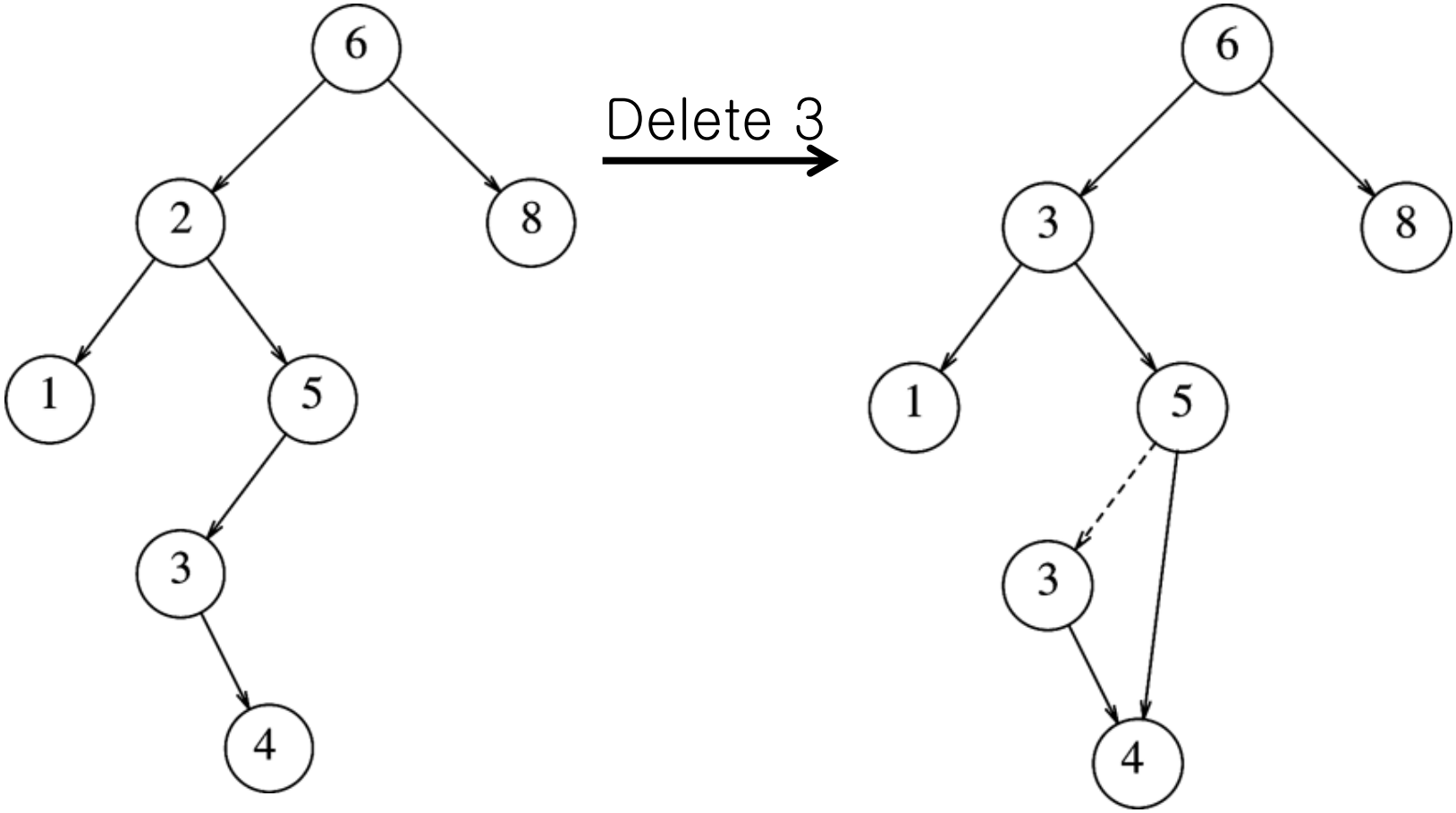
Delete in BST

```
SearchTree Delete(ElementType X, SearchTree T) {
    if ( T == NULL)  Error( "Element not found" );
    else if ( X < T→Element )
            T->Left = Delete( X, T→Left);
    else if (X > T→Element )
            T->Right= Delete( X, T→Right);
    else if ( T→ Left && T→Right )  /* Two children */
    {       T→Element = findMin( T→Right )→Element;
            T→ Right = Delete(T→Element, T→Right ); }
    else
    {       old = T;
            T = (T→left != NULL)? T→Left : T→Right;
            free(old);
    }
    return T; }
```

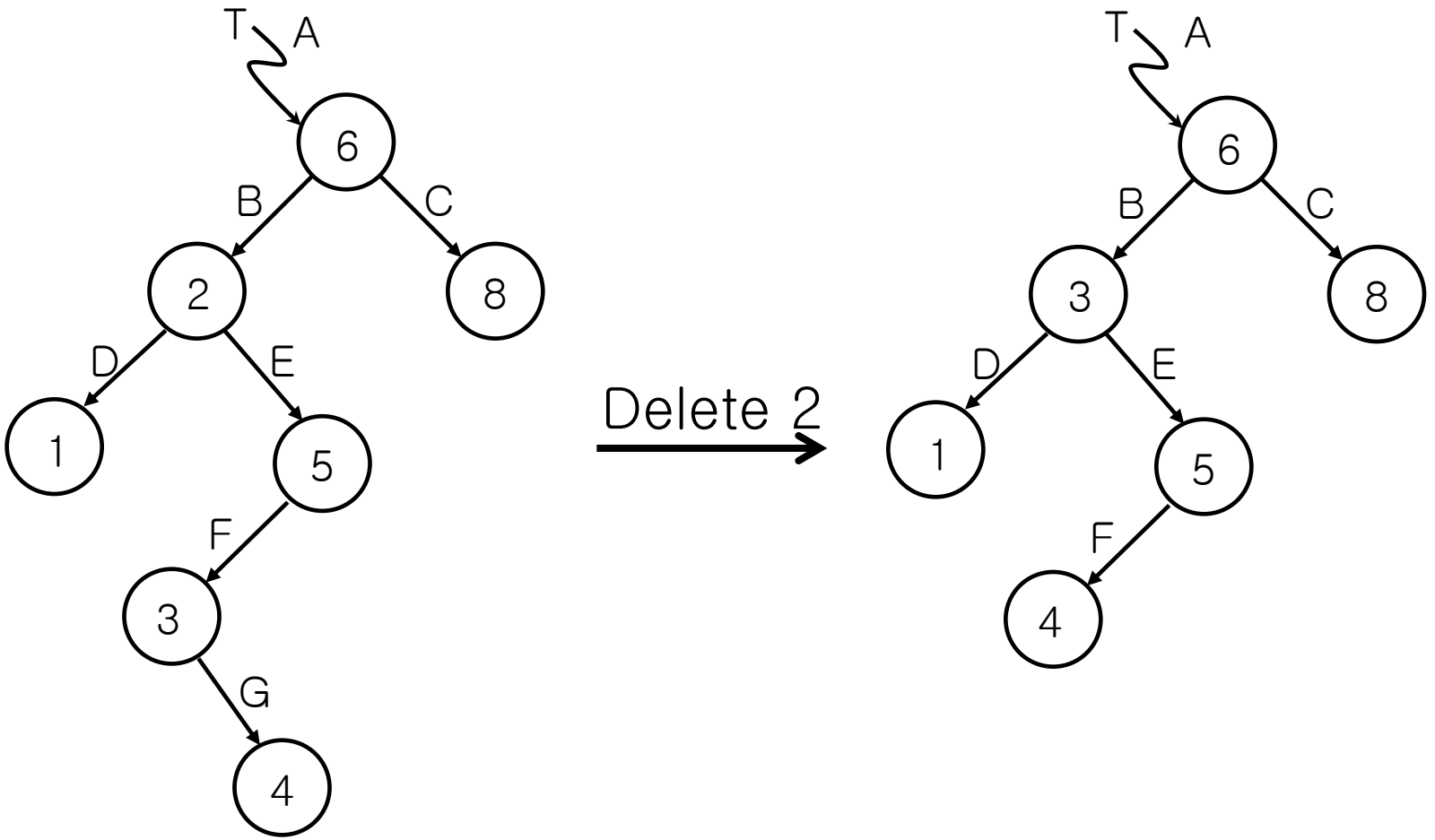
Delete Example



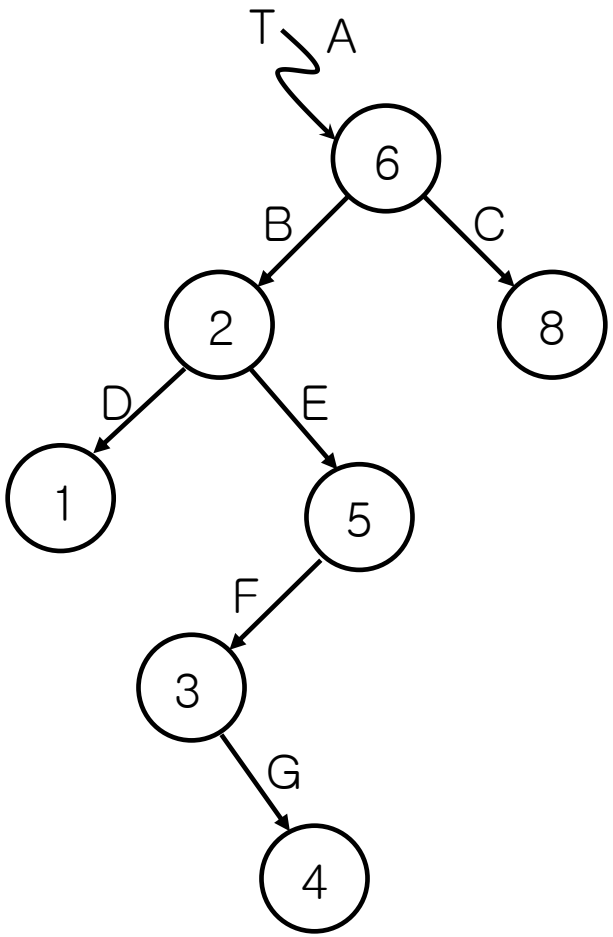
More Delete Example



More Delete Example



Delete Tracing




```
DL(2,A)
    A->Left = DL(2,B)
        Tmp=FindMin(C)  // F
        B->Element = Tmp->Element
        B->Right = DL(3,E)
            E->Left = DL(3,F)
                Tmp = F;
                F = G;
                free(Tmp);
                return G;
            return E;
        return B
    return A
```

Dynamic Properties

- 데이터(random)를 여러 차례 insert/delete 한 후 전체 트리의 형태 – unbalanced
  - 바람직한 속성: depth = O(log N)

- Average-case analysis:

  Average depth for N-node BST with random-data insertion & deletion = O(log N)

Balanced Trees

- 어느 위치에서도 좌우의 균형을 항상 유지하는 경우
  - depth = O(log N)

- 강제적인 균형유지 구조
  - AVL(Adelson Valskii Landis) tree
  - Splay tree: self adjusting tree

B-Trees

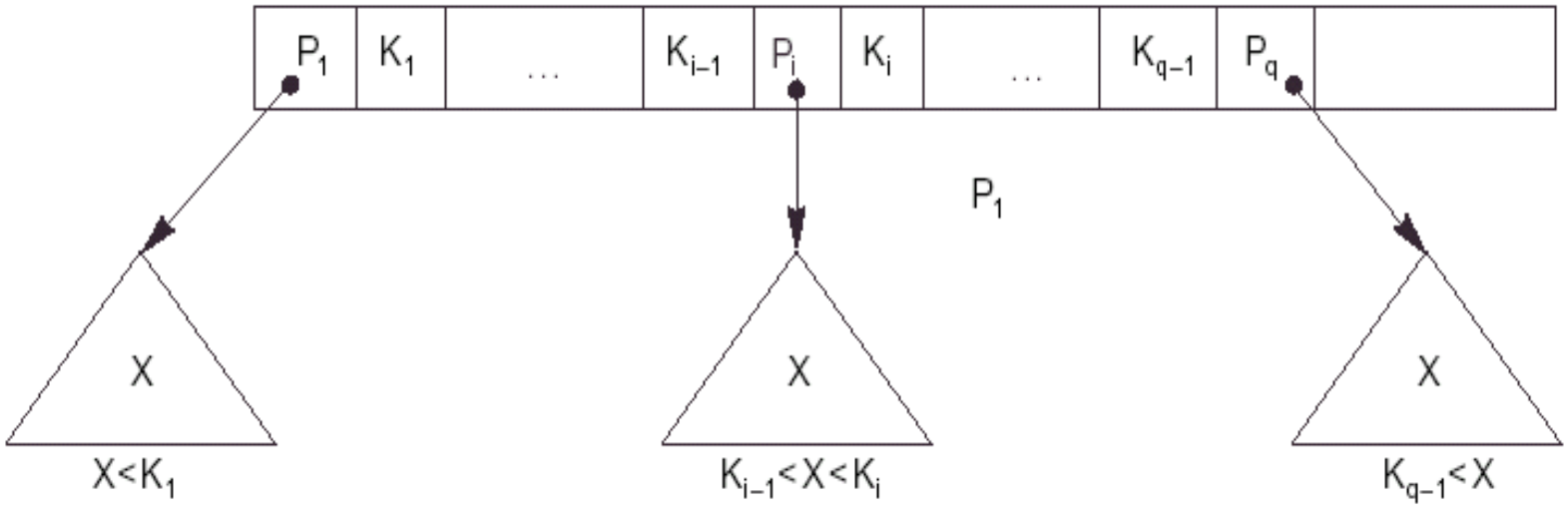
- Popular search tree which is not binary

- B tree of order M is a tree with properties:
  - The root is either a leaf or has 2~*M* children,
  - All nonleaf nodes have [*M*/2] ~ *M* children,
  - All leaves are at the same depth.

Properties of B-Trees

- All data are stored at the leaves
- Each node contains
  - M pointers to the children $P_1, P_2, \cdots, P_M$, and
  - $M-1$ values, $k_1, k_2, \cdots k_{M-1}$ representing the smallest key found in the subtrees P_2, $P_3, \cdots, P_M$, respectively.
- For every node, all the keys in subtree P_1 are smaller than the keys in subtree P_2, and so on.

Node Structure



B-Tree of order 4

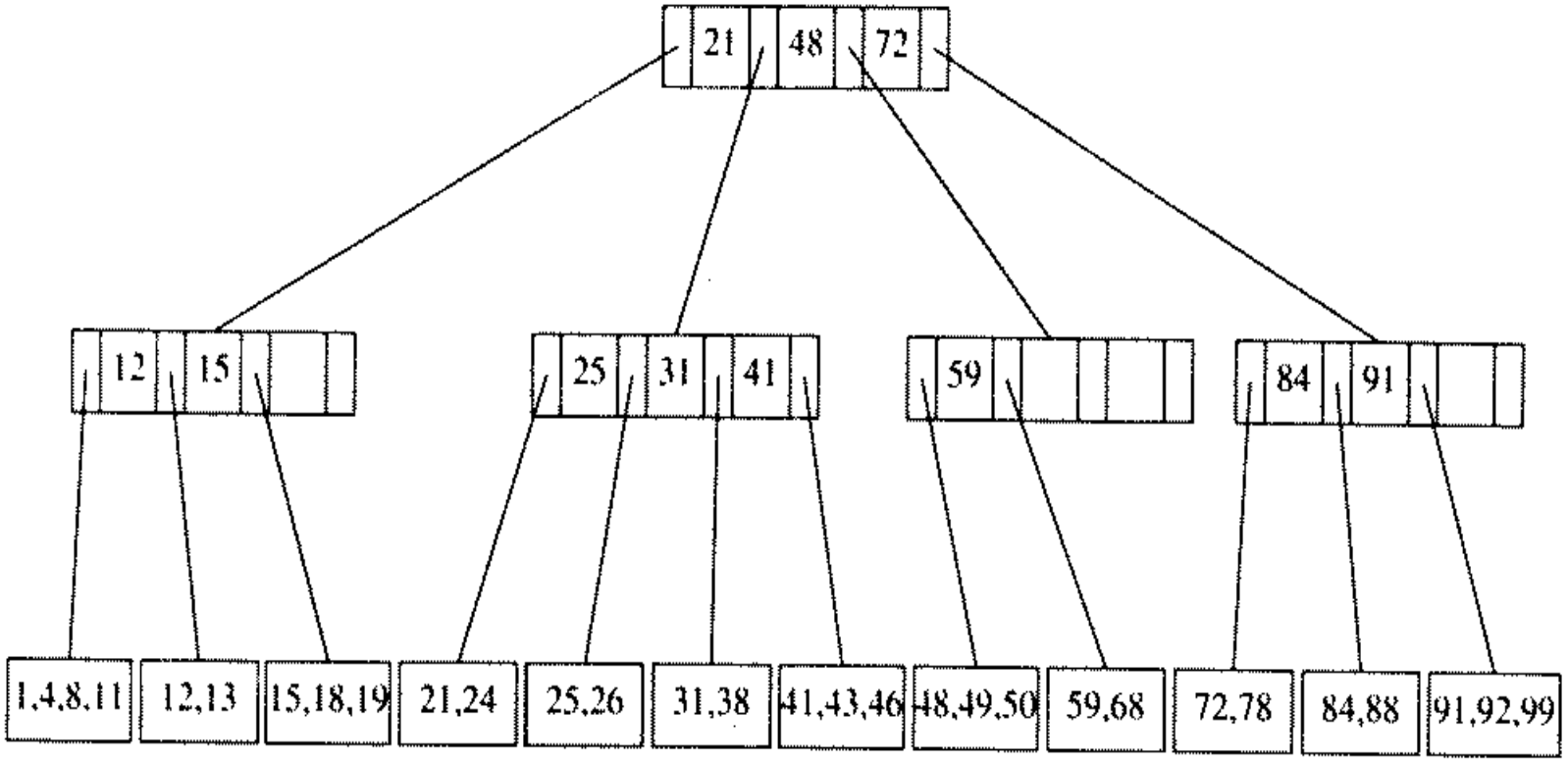
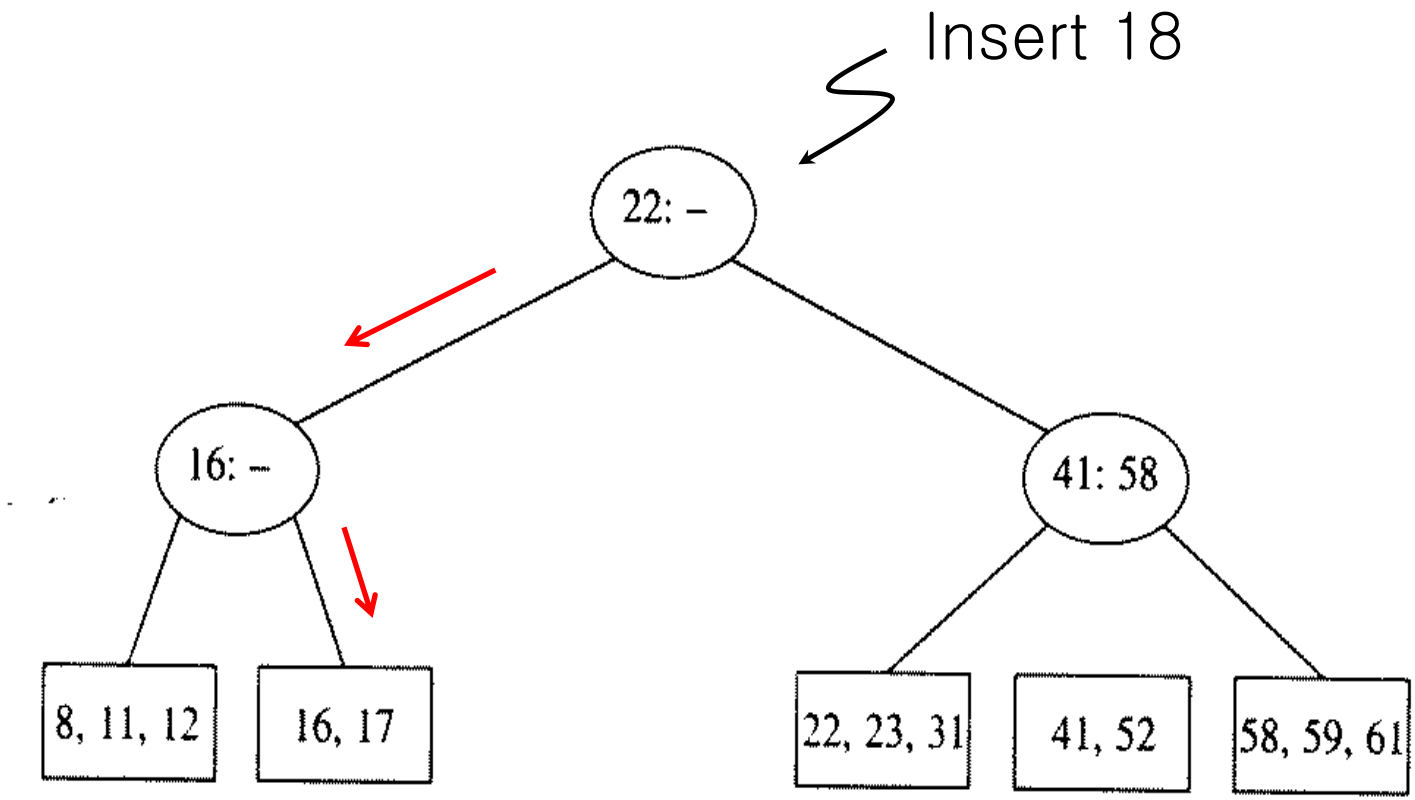


Figure 4.58 B-tree of order 4

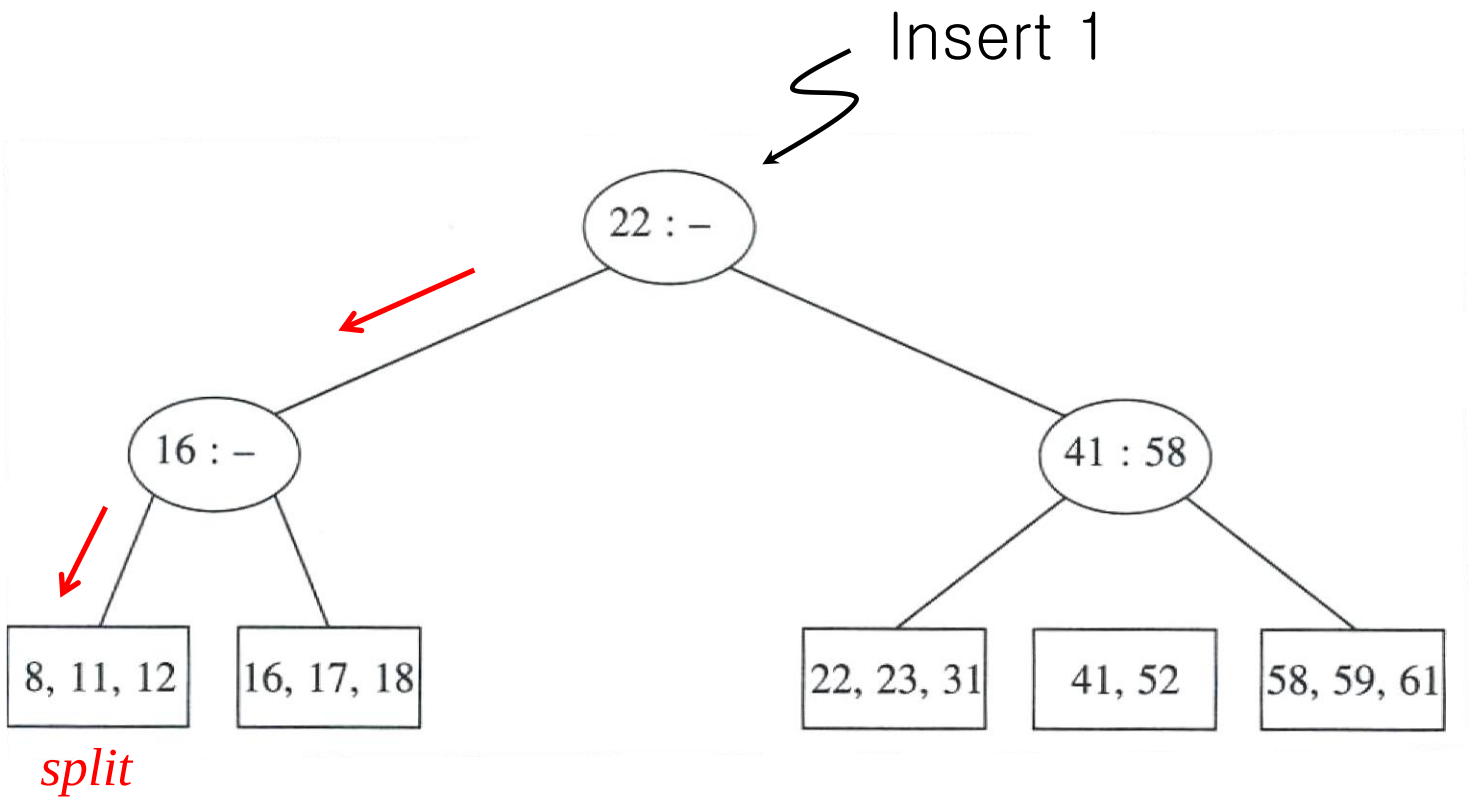
Properties of B-Trees

- A B-tree of order 4 is known as a 2-3-4 tree
- A B-tree of order 3 is known as a 2-3 tree
- B-Tree Operation
  - Insert: If the inserted node is full, split it.
  - Delete: If the deleted node is sparse, merge it with neighbor

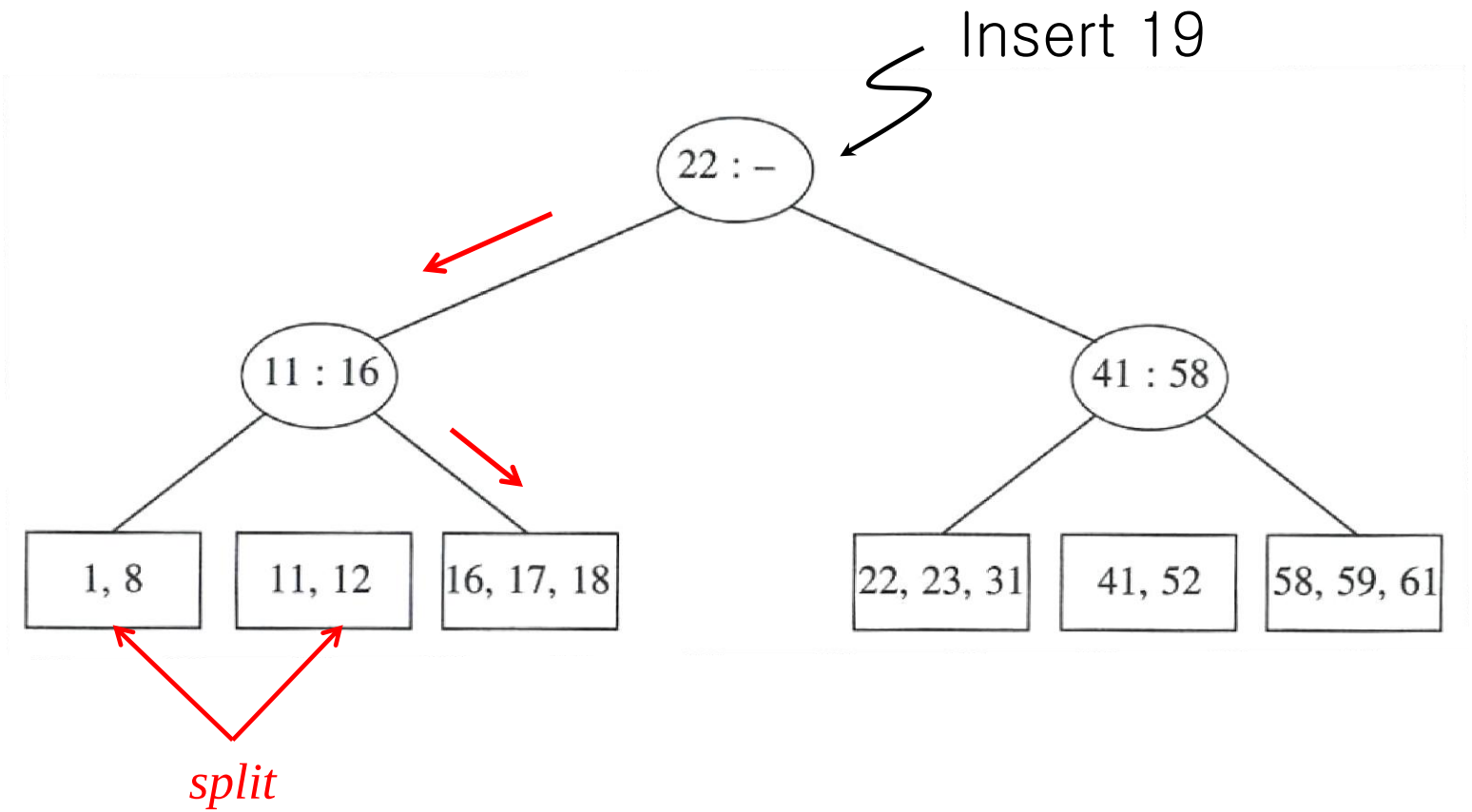
Insert operation



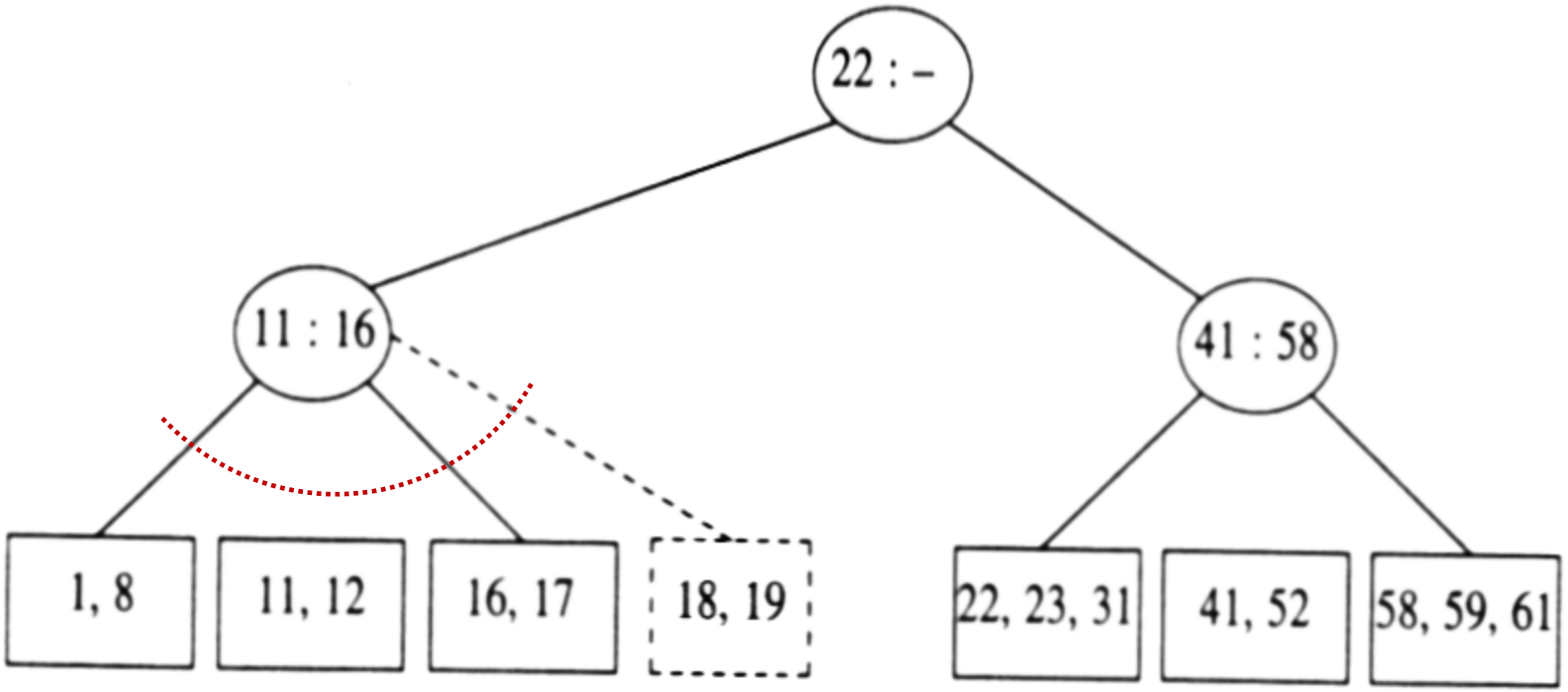
Insert operation(Cont.)



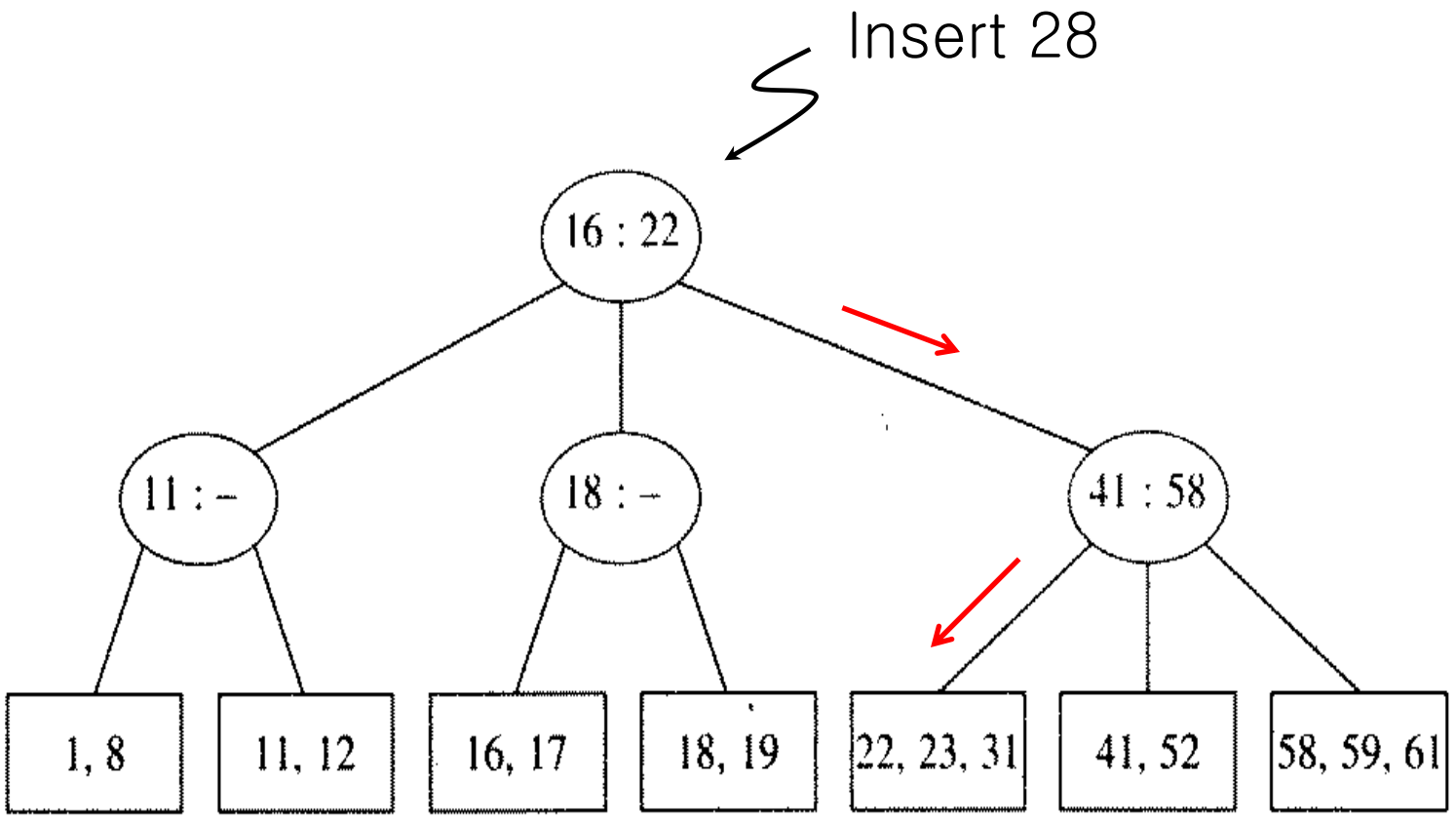
Insert operation(Cont.)



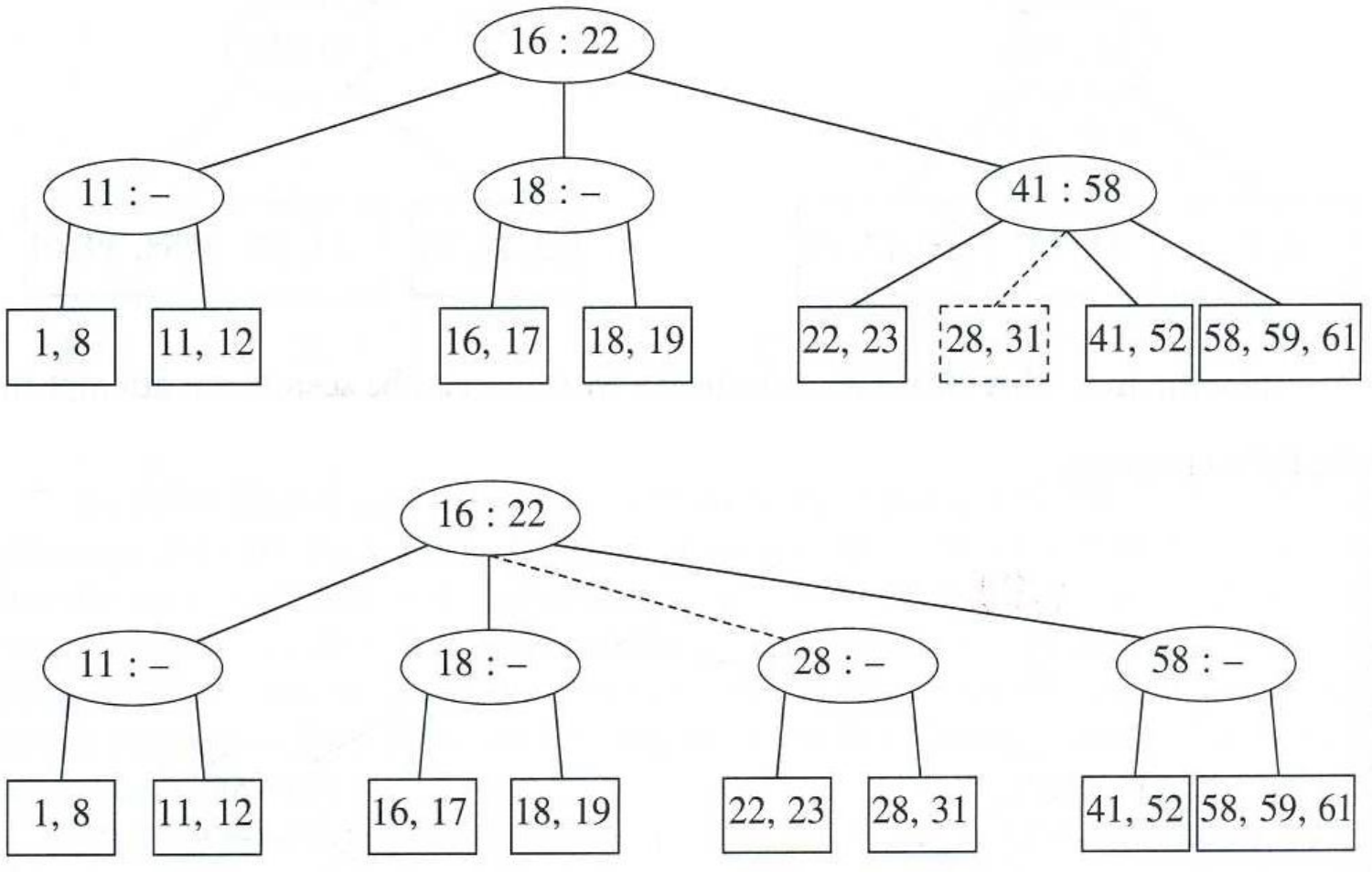
Insert operation(Cont.)



Insert operation(Cont.)



Insert operation(Cont.)



Insert operation(Cont.)