ĐẠI HỌC ĐÀ NẴNG

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN

VIETNAM - KOREA UNIVERSITY OF INFORMATION AND COMMUNICATION TECHNOLOGY

한-베정보통신기술대학교

Nhân bản – Phụng sự – Khai phóng

Trees

Data Structures & Algorithms

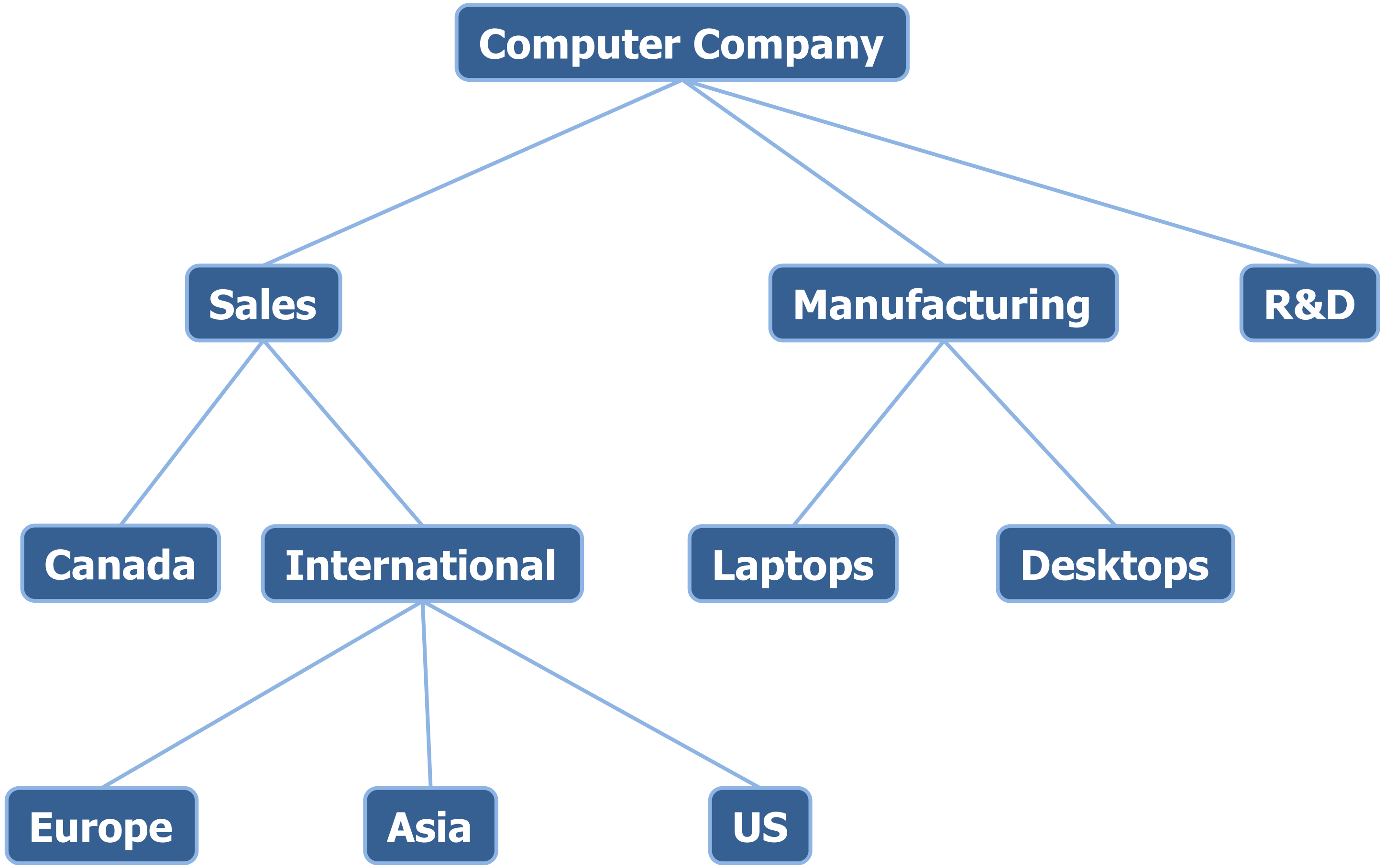
CONTENT

- Introduction
- Binary Trees
- Binary Search Trees
- Forests

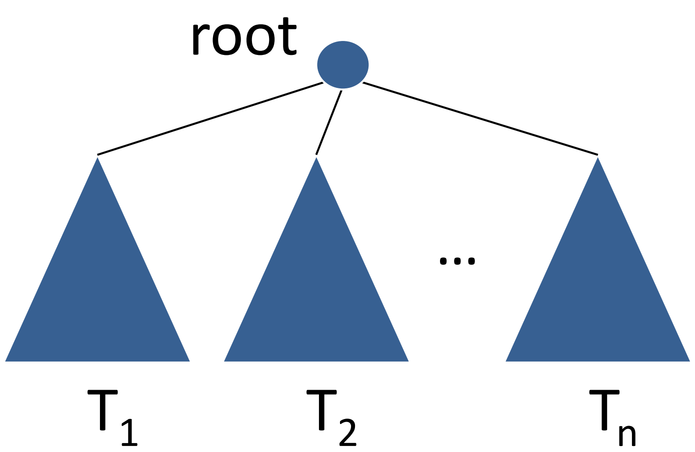
CONTENT

- **Introduction**
- Binary Trees
- Binary Search Trees
- Forests

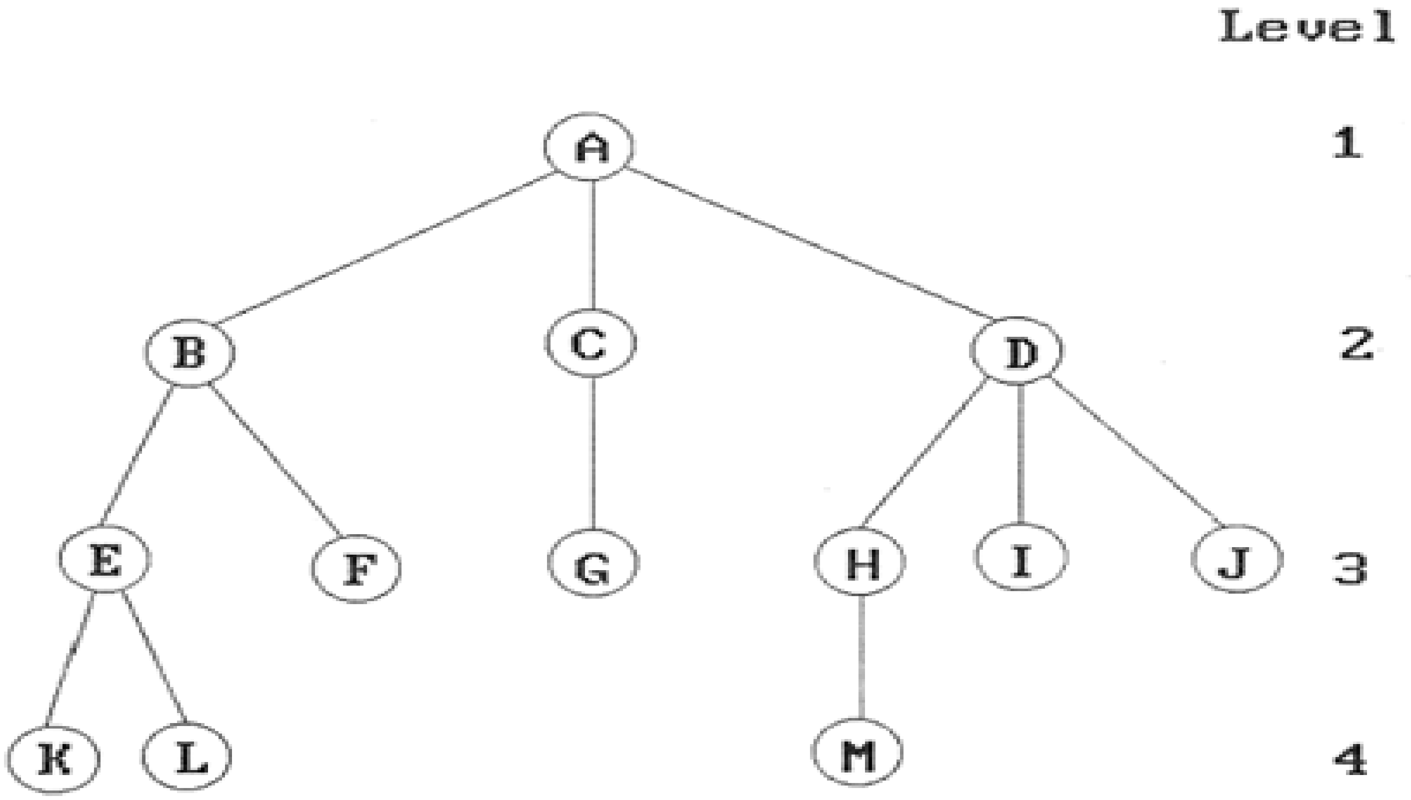
- **Example**


Computer Company
Sales
Manufacturing
R&D
Canada
International
Laptops
Desktops
Europe
Asia
US

- **A *tree* is a finite set of one/more nodes such that**
  - There is a specially designated node called the ***root***.
  - The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree.
    We call $T_1, \ldots, T_n$, the ***sub-trees*** of the root.



- **The root of this tree is node *A***
- **Definitions:**
  - Parent (A)
  - Children (E, F)
  - Siblings (C, D)
  - Root (A)
  - Leaf / Leaves
    - K, L, F, G, M, I, J…



…Introduction

- **The *degree* of a node is the number of sub-trees of the node**
- **The *level* of a node**
  - Initially letting the root be at level one
  - For all other nodes, the level is the level of the node's parent plus one.
  - The ***height*** or ***depth*** of a tree is the maximum level of any node in the tree.
- **List Representation**
  - The root comes first, followed by a list of sub-trees

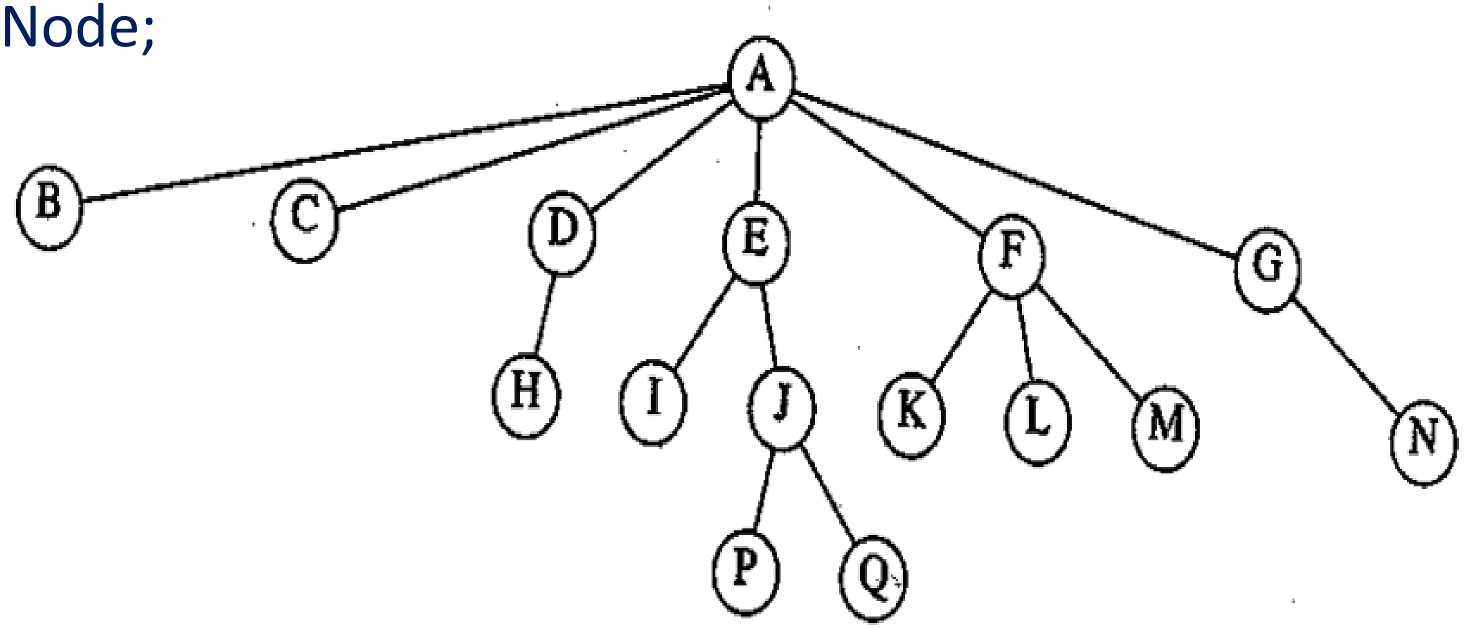
data	link 1	link 2	…	link n

A node must have a varying number of link fields depending on the number of branches

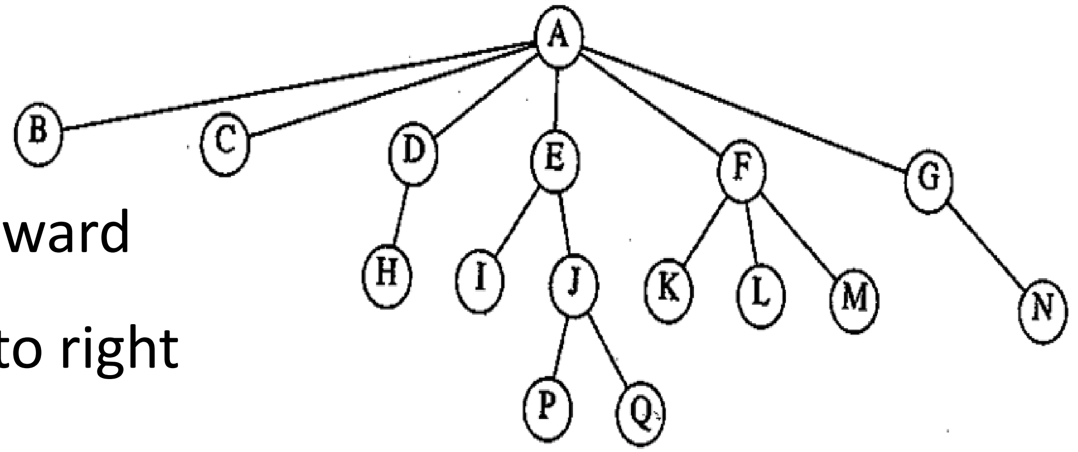
- **Representation of Trees**
  - **Left Child-Right Sibling** Representation

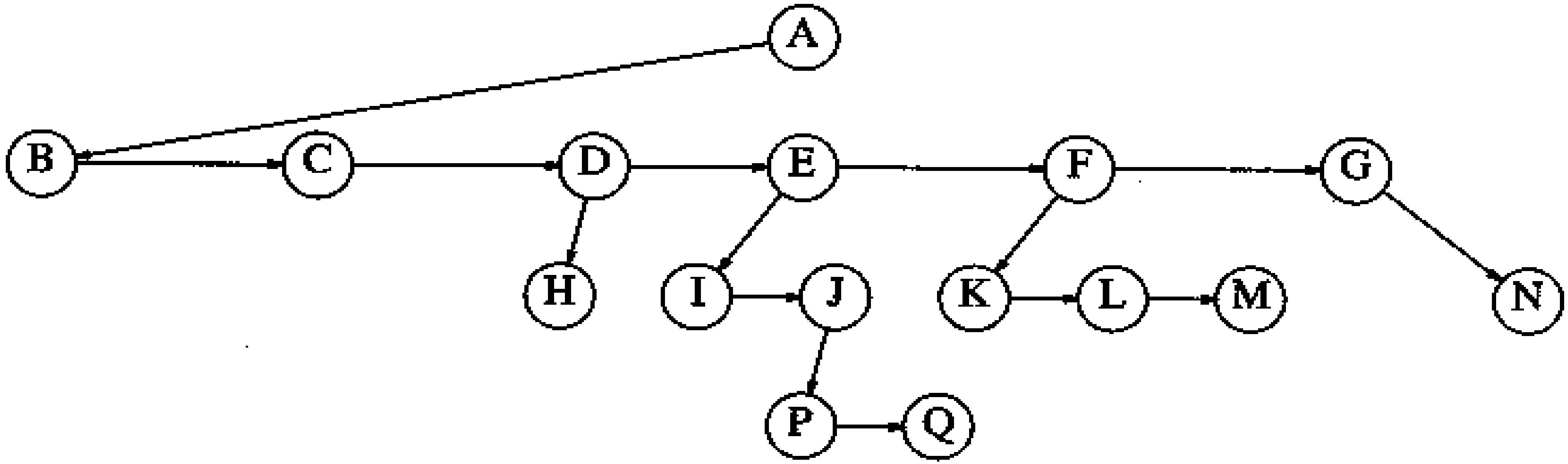
<table>
<tr><td colspan="2">element</td></tr>
<tr><td>left child</td><td>right sibling</td></tr>
</table>

```
typedef struct TreeNode * PtrToNode;
struct TreeNode{
      ElementType element;
      PtrToNode leftChild;
      PtrToNode rightSibling;
};
```



- Representation of Trees
  - **Left Child-Right Sibling** Representation
    - leftChild pointer: arrow that points downward
    - rightSibling pointer: arrow that goes left to right





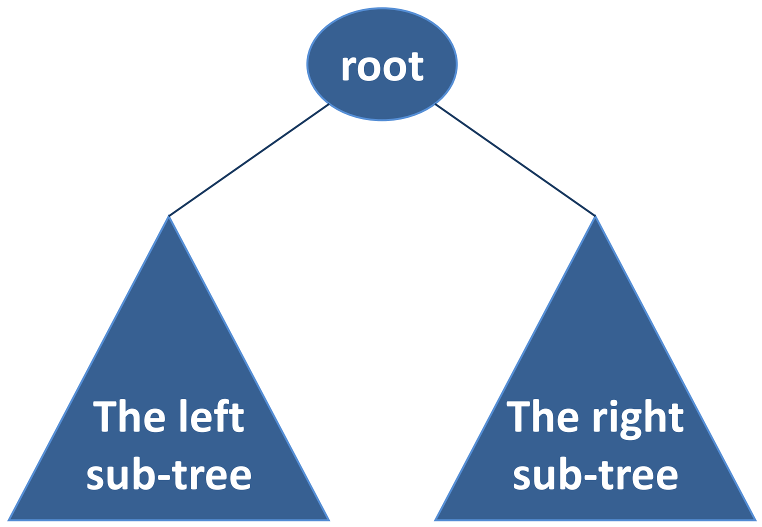
CONTENT

- Introduction
- **Binary Trees**
- Binary Search Trees
- Forests

- **Binary tree**
  - is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called **the left sub-tree** and **the right sub-tree.**

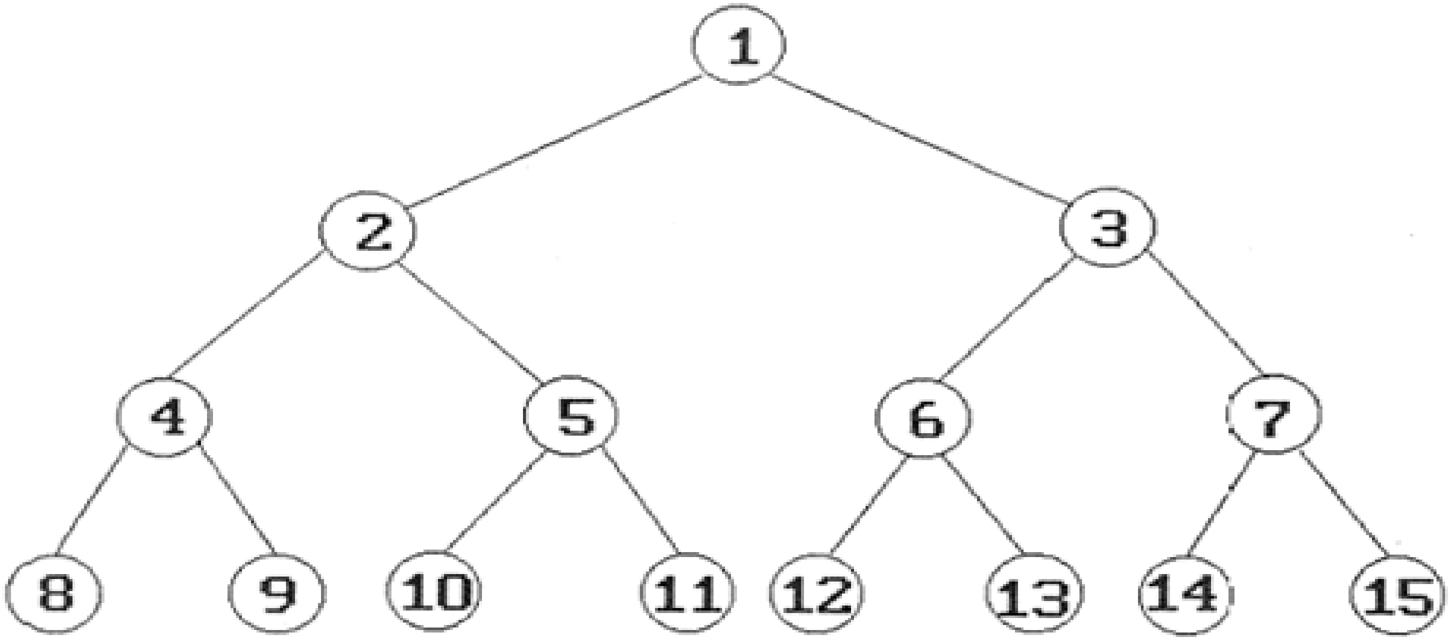
⇨ **Any tree can be transformed into a binary tree**

- By using left child-right sibling representation
- The left and right subtrees are distinguished

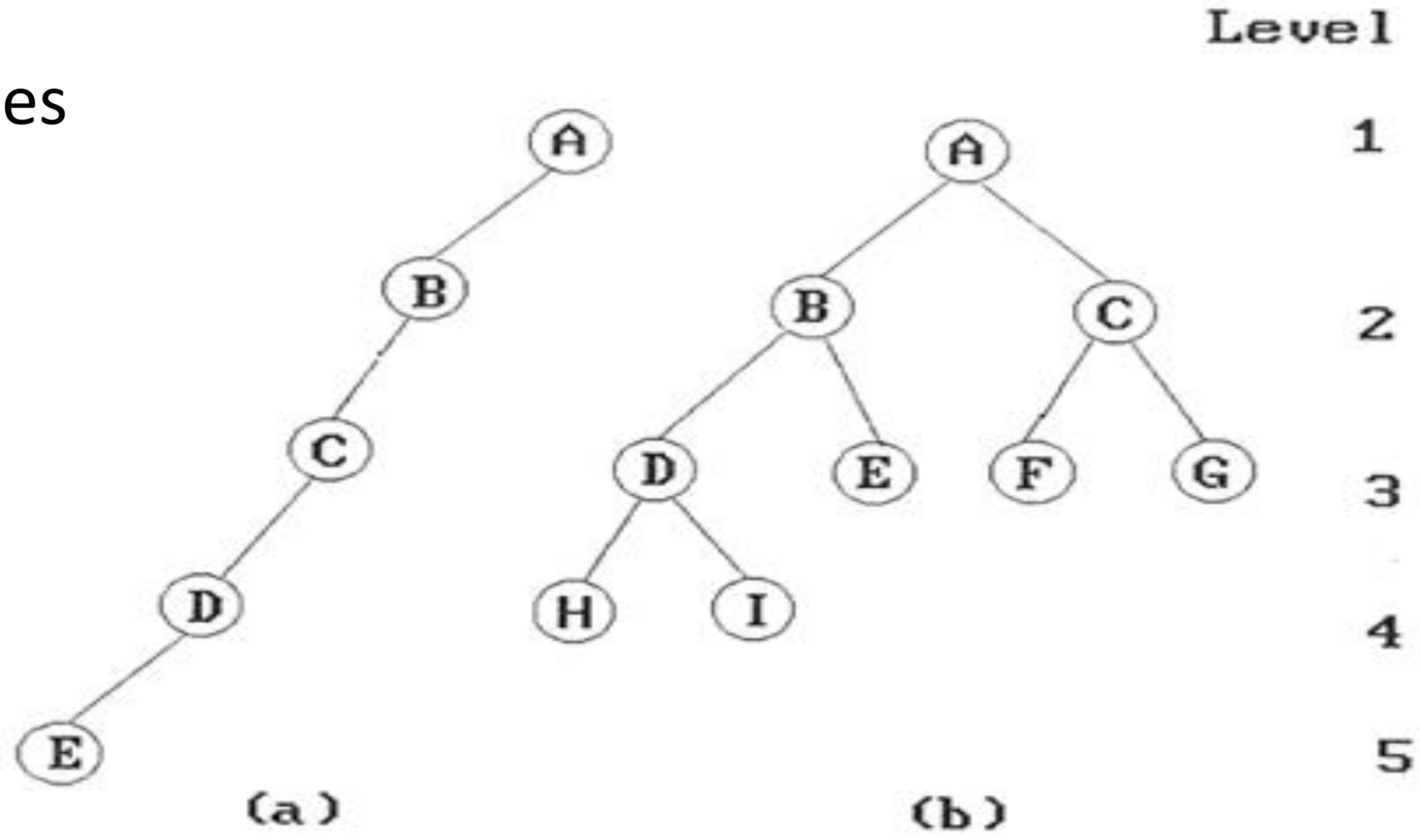


- **Properties of Binary Trees**
  - Lemma 1 [Maximum number of nodes]
    - (1) The maximum number of nodes on level *i* of a binary tree is 2^{i-1}, $i \geq 1$.
    - (2) The maximum number of nodes in a binary tree of depth *k* is $2^k - 1$, $k \geq 1$.
    - The proof is by induction on *i*.
  - Lemma 2
    - For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

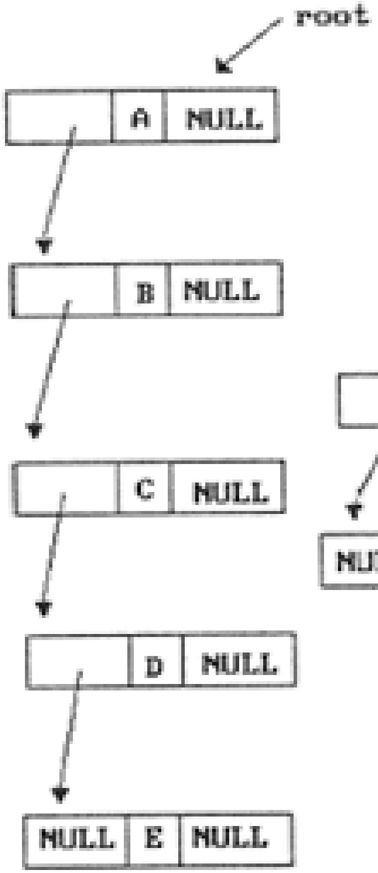
- **Special Binary Trees**
  - A ***full binary tree*** of depth k is a binary tree of depth k having 2^k -1 nodes, $k \geq 0$

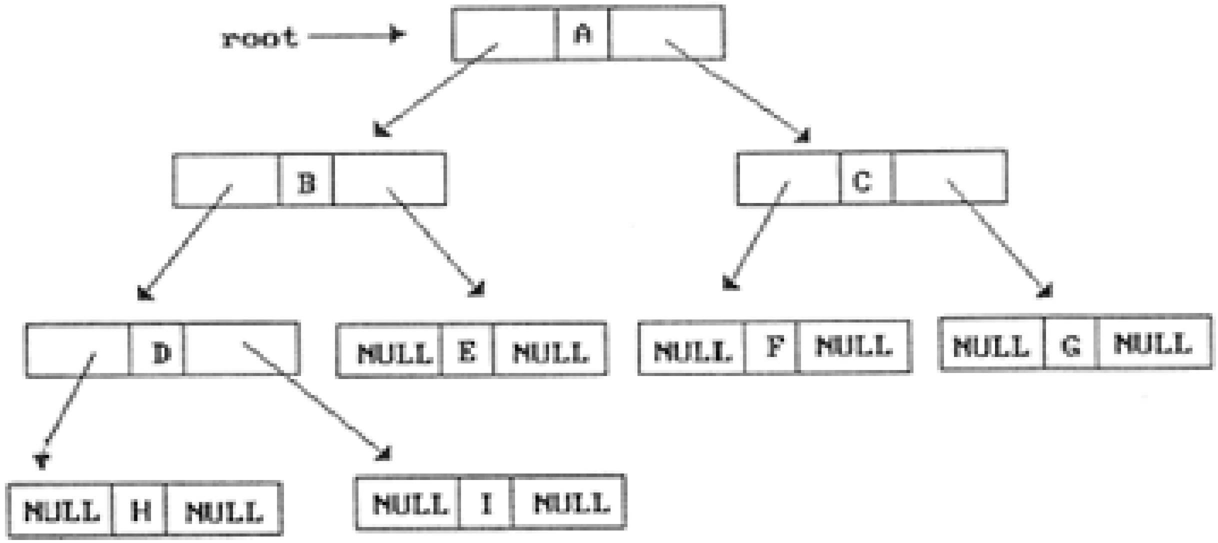


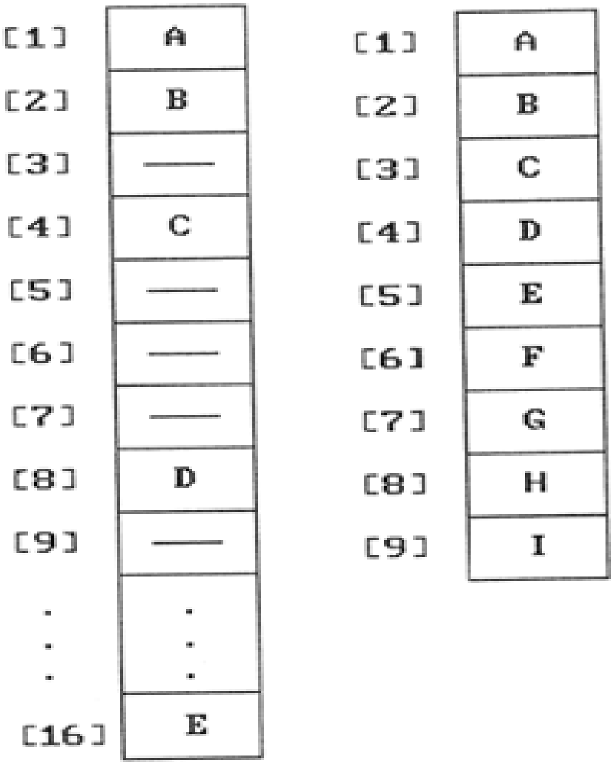
- **Special Binary Trees**
  - A binary tree with *n* nodes and depth *k* is ***complete*** *iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*
  - **Skewed** Binary Trees



- **Binary Tree Representation**
  - Array Representation
  - Linked Representation







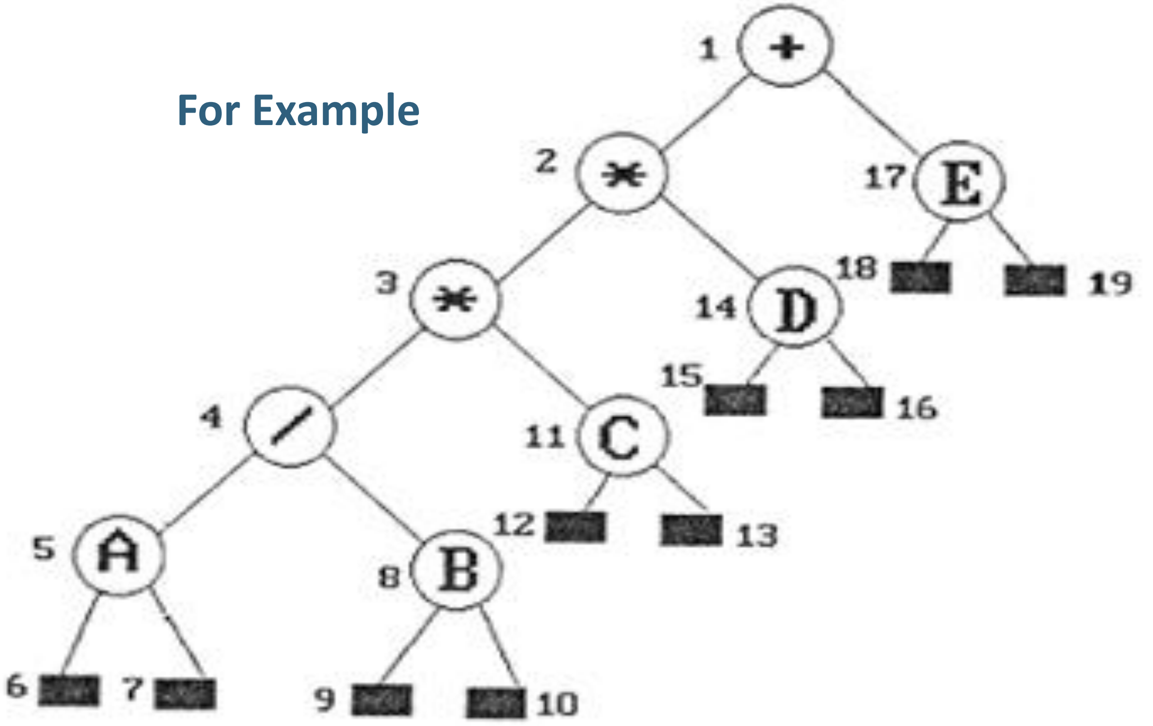
- Binary Tree Representation - **Array Representation**
  - **Lemma 3**: If a complete binary tree with n nodes (depth = $\log_2 n + 1$) is represented sequentially, then for any node with index i,
    $\mathbf{1 \leq i \leq n}$, we have:
    - (1) parent (i) is at **i/2** , $i \neq 1$.
    - (2) left-child (i) is **2i**, if $2i \leq n$.
    - (3) right-child (i) is **2i+1**, if $2i+1 \leq n$.
  - For complete binary trees, this representation is ideal since it wastes no space. However, for the skewed tree, less than half of the array is utilized.

- Binary Tree Representation - **Linked Representation**

```
typedef struct node *tree_pointer;
typedef struct  node {
            int  data;
            tree_pointer   left_child;
            tree_pointer   right_child;
};
```

- **Binary Tree Traversals**
  - Notations
    - *L* : moving left
    - *V* : visiting the node
    - *R* : moving right
  - Traversing order
    - Inorder Traversal: *LVR*
    - Preorder Traversal: *VLR*
    - Postorder Traversal: *LRV*

For Example

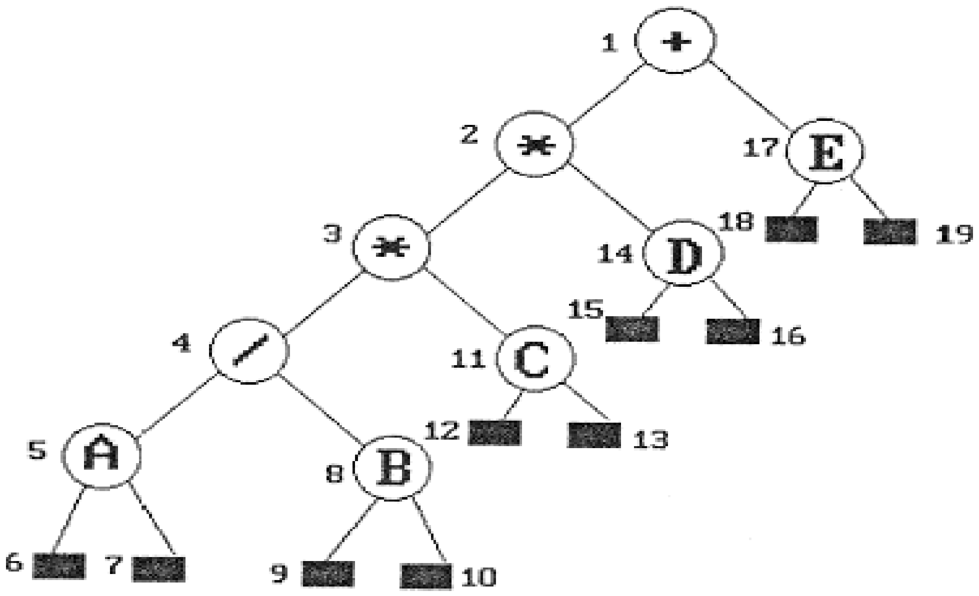


- Inorder Traversal: **A / B * C * D + E**
- Postorder Traversal: **A B / C * D * E +**
- Preorder Traversal: **+ * * / A B C D E**

- Binary Tree Traversals - **Inorder Traversal**
  - A recursive function starting from the root
    - Move left →Visit node →Move right

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
   if (ptr) {
      inorder(ptr->left_child);
      printf("%d",ptr->data);
      inorder(ptr->right_child);
   }
}
```

- Binary Tree Traversals - **Inorder Traversal**



In-order Traversal

A / B * C * D + E

Call of inorder	Value in root	Action	inorder	in root	Value Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	printf
4	/		13	NULL	
5	A		2	*	printf
6	NULL		14	D	
5	A	printf	15	NULL	
7	NULL		14	D	printf
4	/	printf	16	NULL	
8	B		1	+	printf
9	NULL		17	E	
8	B	printf	18	NULL	
10	NULL		17	E	printf
3	*	printf	19	NULL	

- Binary Tree Traversals - **Preorder Traversal**
  - A recursive function starting from the root
    - Visit node →Move left → Move right

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
   if (ptr) {
      printf("%d",ptr->data);
      preorder(ptr->left_child);
      preorder(ptr->right_child);
   }
}
```

- Binary Tree Traversals - **Postorder Traversal**
  - A recursive function starting from the root
    - Move left →Move right →Visit node

```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
   if (ptr) {
      postorder(ptr->left_child);
      postorder(ptr->right_child);
      printf("%d",ptr->data);
   }
}
```

- **Other Traversals**
  - Iterative Inorder Traversal
    - Using a stack to simulate recursion
    - Time Complexity: O(n), n is #num of node.
  - Level Order Traversal
    - Visiting at each new level from the left-most node to the right-most
    - Using Data Structure: Queue

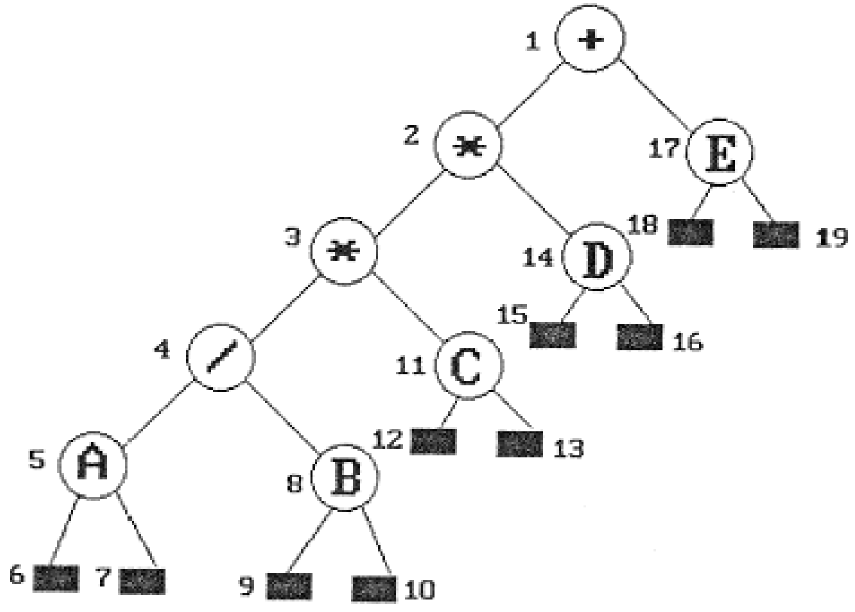
- Other Traversals - **Iterative Inorder Traversal**
  - Using a **stack** to simulate recursion
  - Time Complexity: O(*n*), *n* is #num of node.

```
void iter_inorder(tree_pointer node)
{
   int top = -1; /* initialize stack */
   tree_pointer stack[MAX_STACK_SIZE];
   for (;;) {
      for(; node; node = node->left_child)
         add(&top, node); /* add to stack */
      node = delete(&top); /* delete from stack */
      if (!node) break; /* empty stack */
      printf("%d", node->data);
      node = node->right_child;
   }
}
```

- Other Traversals - **Iterative Inorder Traversal**

Add “+” in stack
Add “*”
Add “*”
Add “/”
Add “A”
Delete “A” & Print
Delete “/” & Print
Add “B”
Delete “B” & Print
Delete “*” & Print
Add “C”

Delete “C” & Print
Delete “*” & Print
Add “D”
Delete “D” & Print
Delete “+” & Print
Add “E”
Delete “E” & Print



In-order Traversal:

A / B * C * D + E

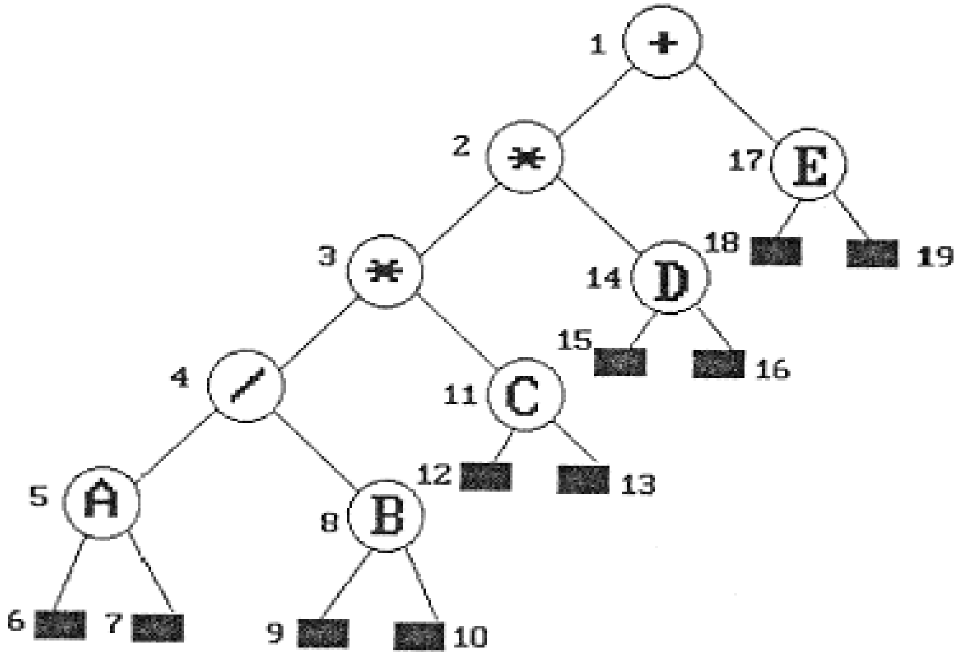
- Other Traversals - **Level Order Traversal**
  - Visiting at each new level from the left-most node to the right-most
  - Using Data Structure: **Queue**

```
void level_order(tree_pointer ptr)
/* level order tree traversal */
{
  int front = rear = 0;
  tree_pointer queue[MAX_QUEUE_SIZE];
  if (!ptr) return; /* empty tree */
  addq(front, &rear, ptr);
  for (;;) {
    ptr = deleteq(&front, rear);
    if (ptr) {
      printf("%d",ptr->data);
      if(ptr->left_child)
        addq(front,&rear,ptr->left_child);
      if (ptr->right_child)
        addq(front,&rear,ptr->right_child);
    }
    else break;
  }
}
```

- Other Traversals - **Level Order Traversal**

Add "+" in Queue
Deleteq "+"
Addq "*"
Addq "E"
Deleteq "*"
Addq "*"
Addq "D"
Deleteq "E"
Deleteq "*"

Addq "/"
Addq "C"
Deleteq "D"
Deleteq "/"
Addq "A"
Addq "B"
Deleteq "C"
Deleteq "A"
Deleteq "B"



Level-order Traversal:

+ * E * D / C A B

- **Additional Binary Tree Operations**
  - Copying Binary Trees
    - Copy a binary tree to another one
  - Testing for Equality of Binary Trees
    - Verify if two binary trees are identical

- **Copying Binary Trees**
  - Modified from postorder traversal program

```
tree_pointer copy(tree_pointer original)
/* this function returns a tree_pointer to an exact copy
of the original tree */
{
   tree_pointer temp;
   if (original) {
      temp = (tree_pointer) malloc(sizeof(node));
      if (IS_FULL(temp)) {
         fprintf(stderr, "The memory is full\n");
         exit(1);
      }
      temp->left_child = copy(original->left_child);
      temp->right_child = copy(original->right_child);
      temp->data = original->data;
      return temp;
   }
   return NULL;
}
```

- **Testing for Equality of Binary Trees**
  - Two binary trees having identical topology and data are said to be equivalent

```
int equal(tree_pointer first, tree_pointer second)
{
/* function returns FALSE if the binary trees first and
second are not equal, Otherwise it returns TRUE */
   return ((!first && !second) || (first && second &&
          (first->data == second->data) &&
          equal(first->left_child,second->left_child) &&
          equal(first->right_child, second->right_child))
}
```

- **Functions to implement**
  - Count the number of nodes in a binary tree
  - Count the number of leaves in a binary tree
  - Search a node in a binary tree
  - Find a sub-tree in a binary tree

CONTENT

- Introduction
- Binary Trees
- **Binary Search Trees**
- Forests

- **Binary Search Trees** (BST)
  - BST is a binary tree, that may be satisfies the following properties:
    - (1) Every element has a unique key
    - (2) The keys in a nonempty left sub-tree must be smaller than the key in the root of the sub-tree
    - (3) The keys in a nonempty right sub-tree must be larger than the key in the root of the sub-tree
    - (4) The left and right sub-trees are also binary search trees
- **Declrations:**

```
typedef struct tree_node *tree_ptr;
struct tree_node{
        element_type element;
        tree_ptr  left;
        tree_ptr  right;
};
typedef tree_ptr SEARCH_TREE;
```

- **Search in a BST**
  - **Recursive search**

```
tree_ptr  recSearch(element_type x, SEARCH_TREE T ) {
          if( T == NULL )  return NULL;
          if( x < T->element )              return(recSearch( x, T->left ) );
          else  if( x > T->element )        return(recSearch( x, T->right ) );
          else                              return T;
}
```

- Search in a BST
  - **Iterative search**

```
tree_ptr  iteSearch(element_type x, SEARCH_TREE T ) {
        while (T){
                if (x == T->element) return T;
                if (x < T->element)       T = T-> left;
                else                      T = T-> right;
        }
        return NULL;
}
```

- Time Complexity
  - recSearch: O(h), h is the height of BST.
  - iteSearch: O(h)

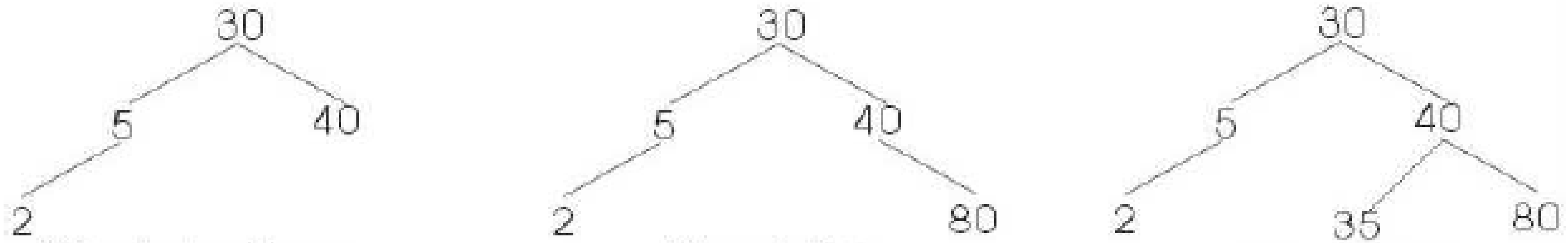
- **Search the smallest element**

```
tree_ptr  searchMin( SEARCH_TREE T ){
        if( T == NULL )  return NULL;
        else if( T->left == NULL )        return( T );
        else                              return(searchMin ( T->left ) );
}
```

- **Search the largest element**

```
tree_ptr  searchMax( SEARCH_TREE T ){
        if( T != NULL )
        while( T->right != NULL )       T = T->right;
        return T;
}
```

- **Inserting an element to a BST**
  - First search key in the tree. Note search always terminates at a null sub-tree
  - Add the key at the null sub-tree where search terminates



(A) original tree (B) add 80. (C) add 35.

- **Inserting an element to a BST**

```
tree_ptr insert( element_type x, SEARCH_TREE T ){
    if( T == NULL ) {     /* Create and return a one-node tree */
        T = (SEARCH_TREE) malloc ( sizeof (struct tree_node) );
        if( T == NULL )  fprintf(stderr, “the memory is full.\n”); exit(1);
        else {          T->element = x;
                        T->left = T->right = NULL;
        }
    }
    else  if( x < T->element )   T->left = insert( x, T->left );
    else  if( x > T->element )   T->right = insert( x, T->right );
          /* else x is in the tree already. We'll do nothing */
    return T;
}
```

- **Delete a node from a BST**

if *the tree is empty* **return** *false*
else attempt to locate the node containing the target using the binary search algorithm

if the target is not found **return** false

else the target is found, so remove its node as follows:

Case 1: **if** the node has 2 empty sub-trees

then replace the link in the parent with null

Case 2: **if** the node has no left child

then link the parent of the node to the right (non-empty) sub-tree

- **Delete a node from a BST**

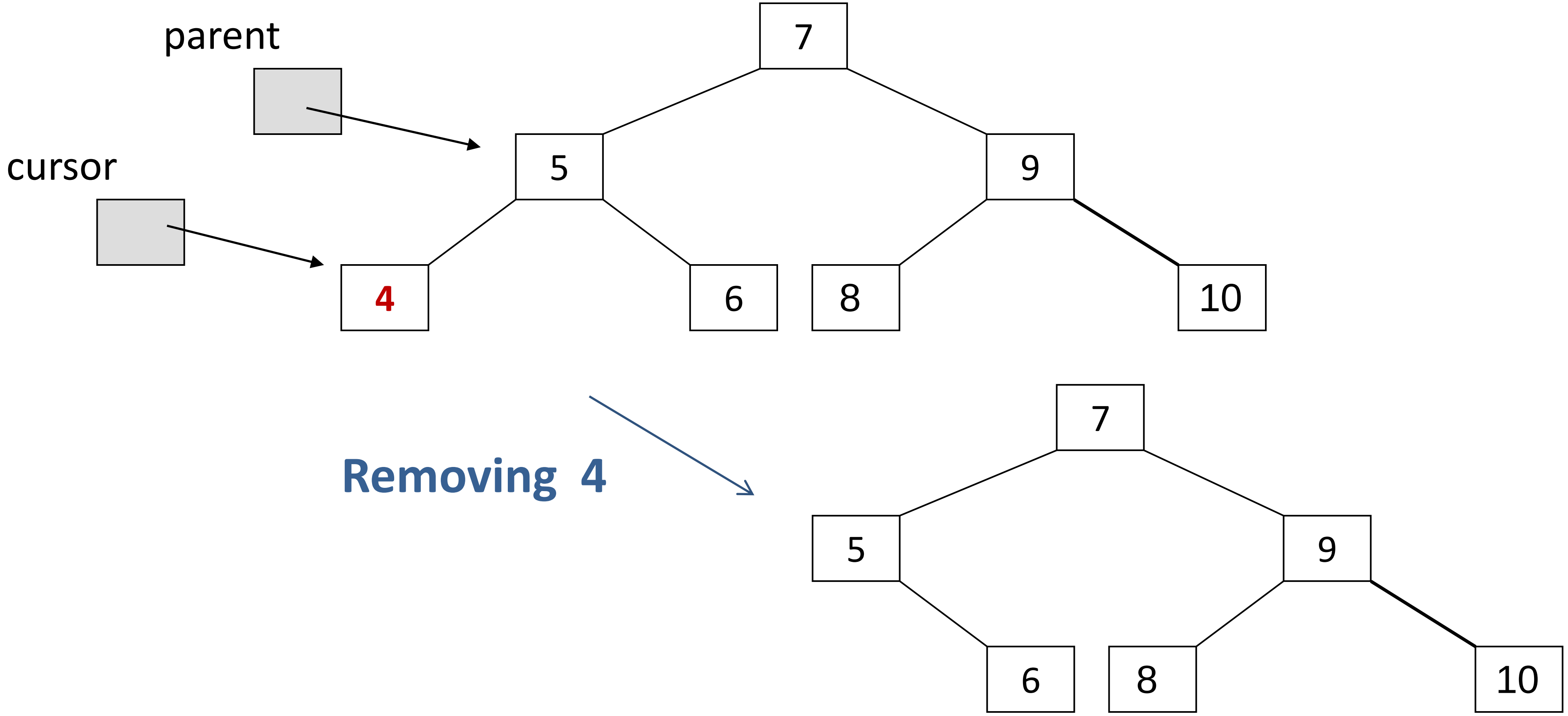
Case 3: **if** the node has no right child

then link the parent of the node to the left (non-empty) sub-tree

Case 4: **if** the node has a left sub-tree and a right sub-tree,
then Replace the node with the largest element in the left sub-tree
or the smallest element from the right sub-tree.

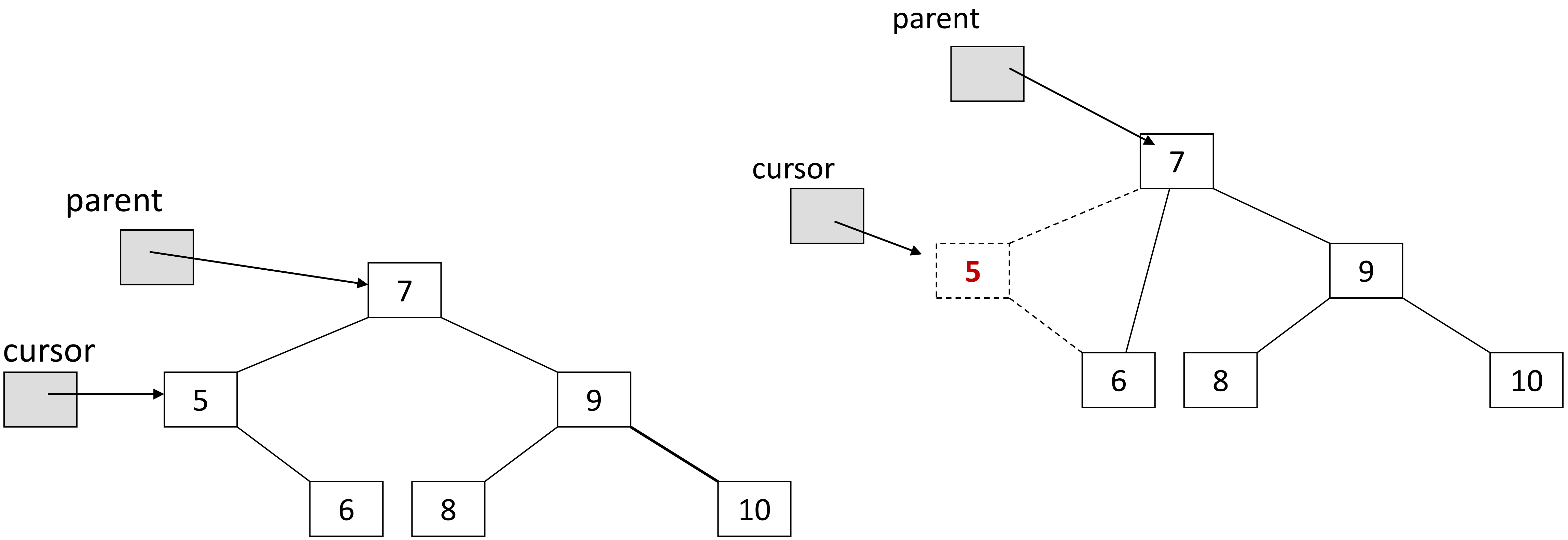
Delete the largest (or smallest, respectively) element in
the respective sub-tree.

- **Delete a node from a BST**
  - **Case 1**: removing the node has 2 empty sub-trees



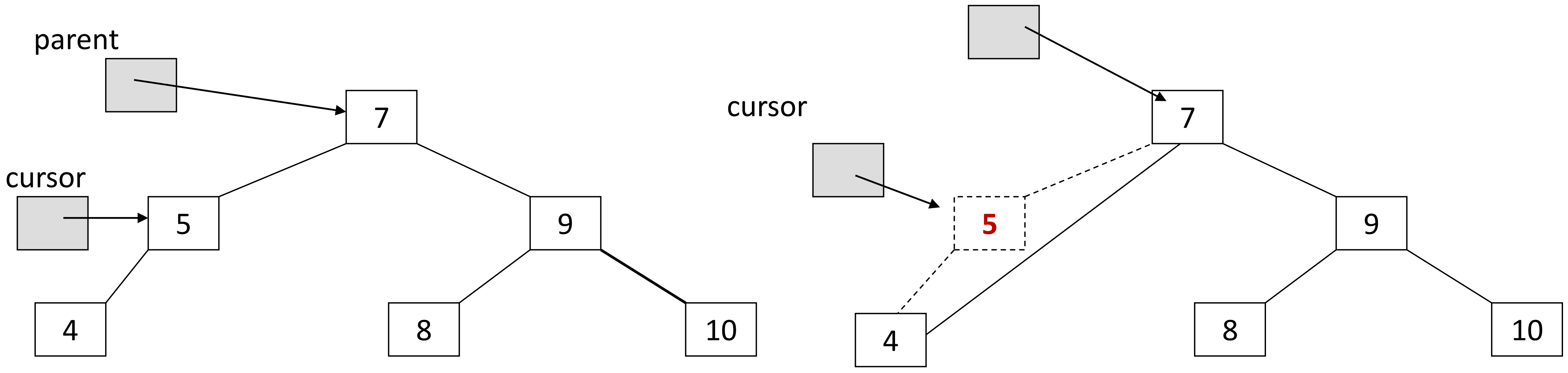
Removing 4

- **Delete a node from a BST**
  - **Case 2**: removing the node has no left child



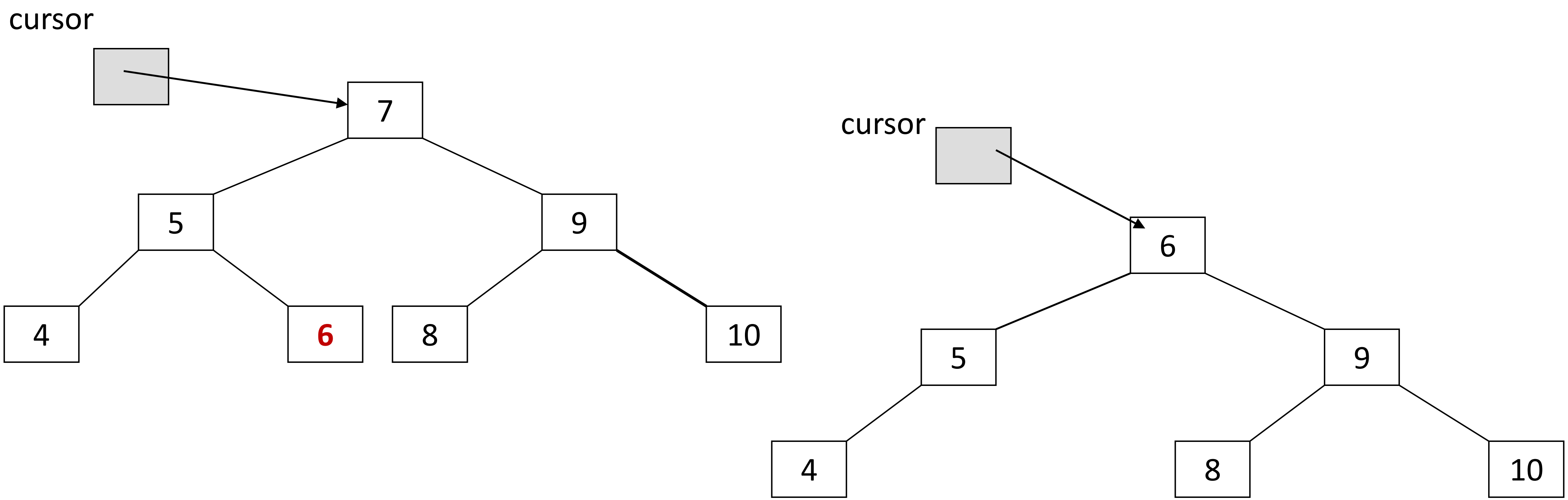
Removing 5

- **Delete a node from a BST**
  - **<u>Case 3</u>**: removing the node has no right child



Removing 5

- **Delete a node from a BST**
  - **<u>Case 4</u>**: removing the node has 2 sub-trees



Removing 7

- **Delete a node from a BST**

```
tree_ptr delete( element_type x, SEARCH_TREE T ){
        tree_ptr tmp_cell;
        if( T == NULL ) {
                fprintf(stderr, “Element not found”);
                exit(1);
        }
        else if( x < T->element )          T->left = delete( x, T->left );      /* Go left */
        else if( x > T->element )          T->right = delete( x, T->right );   /* Go right */
        else    /* Found element to be deleted */
```

```
        if( T->left && T->right ){                /* Two children : case 4 */
                /* Replace with smallest in right sub-tree */
                tmp_cell = search_min( T->right );
                T->element = tmp_cell->element;
                T->right = delete( T->element, T->right );
        }
        else {                                    /* One child & 0 child : case 1, 2, 3 */
                tmp_cell = T;
                if( T->left == NULL )    T = T->right;        /* a right child, also handles 0 child*/
                else if( T->right == NULL )   T = T->left;     /* Only a left child */
                free( tmp_cell );
        }
    return T;
}
```

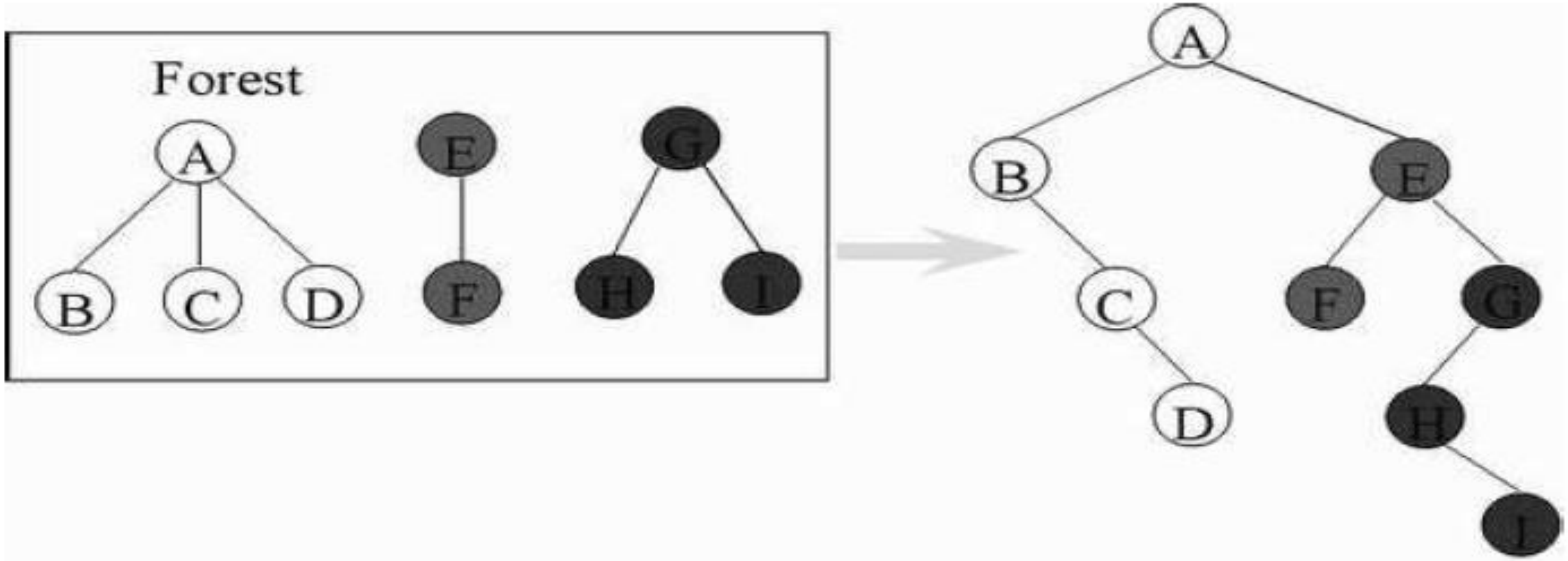
- **Make a BST empty**

```
SEARCH_TREE  make_empty( SEARCH_TREE T ){
        if( T != NULL ){
                make_empty( T->left);
                make_empty( T->right);
                free (T);
        }
        return NULL;
}
```

CONTENT

- Introduction
- Binary Trees
- Binary Search Trees
- **Forests**

- **Forests**
  - **forest** is an ordered set of $n \geq 0$ disjoint trees
  - $T_1, \ldots, T_n$ is a forest of trees
  - Transforming a forest into a binary tree
    - Transform each tree into a binary tree by using left-child right-sibling
    - Connect the binary trees into a single tree



- **Forest Traversals**

Pre-order:

(1) If F is empty, then return.
(2) Visit the root of the first tree of F.
(3) Traverse the subtrees of the first tree in tree preorder.
(4) Traverse the remaining trees of F in preorder.

In-order:

(1) If F is empty, then return.
(2) Traverse the subtrees of the first tree in tree inorder.
(3) Visit the root of the first tree.
(4) Traverse the remaining trees in tree inorder.

Post-order:

(1) If F is empty, then return.
(2) Traverse the subtrees of the first tree of F in tree postorder.
(3) Traverse the remaining trees of F in tree postorder.
(4) Visit the root of the first tree of F.

SUMMARY

- Introduction
- Binary Trees
- Binary Search Trees
- Forests

Enjoy the Course…!