

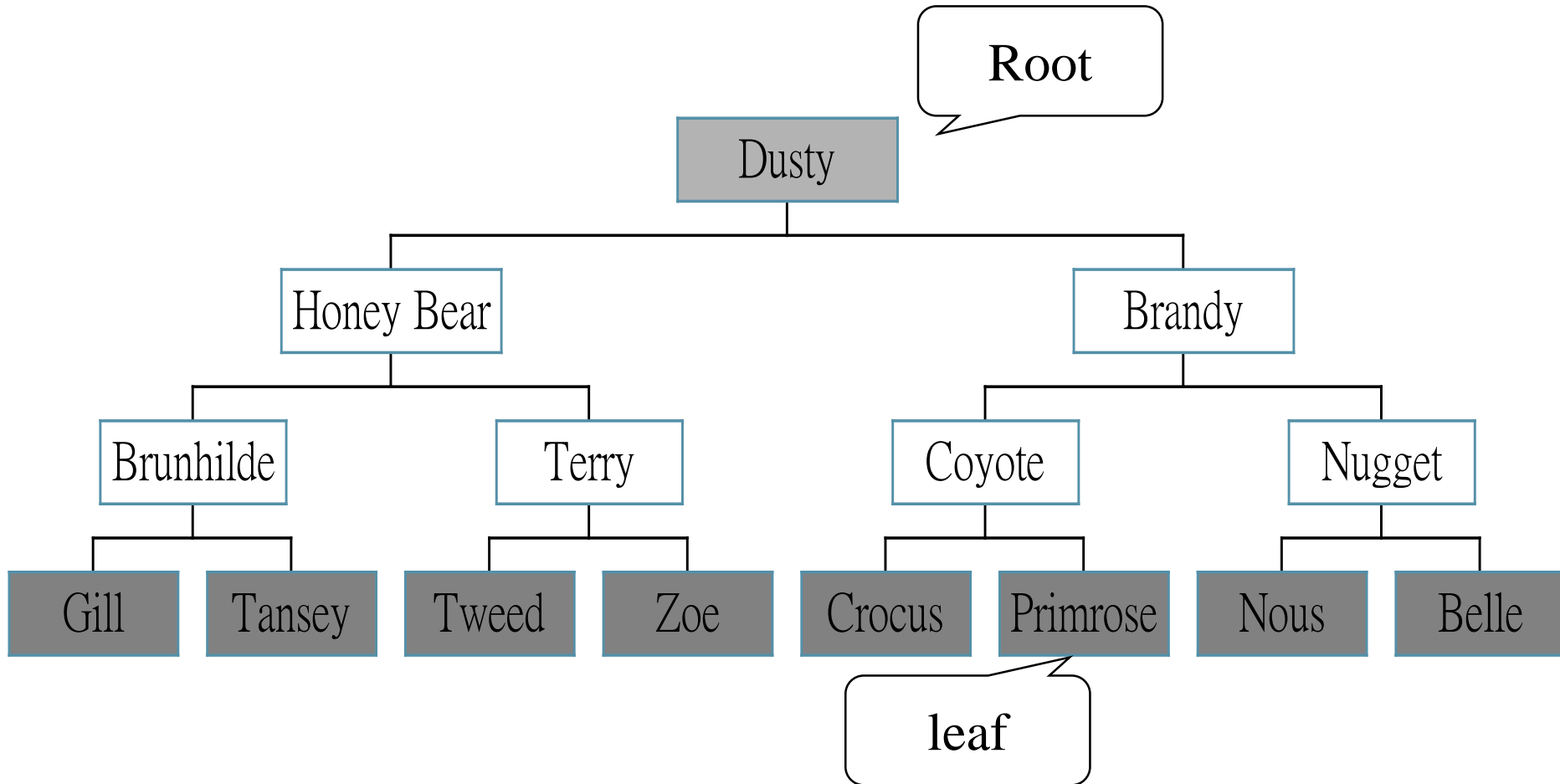
## CHAPTER 5

# Trees

All the programs in this file are selected from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed  
“Fundamentals of Data Structures in C”,

# Trees

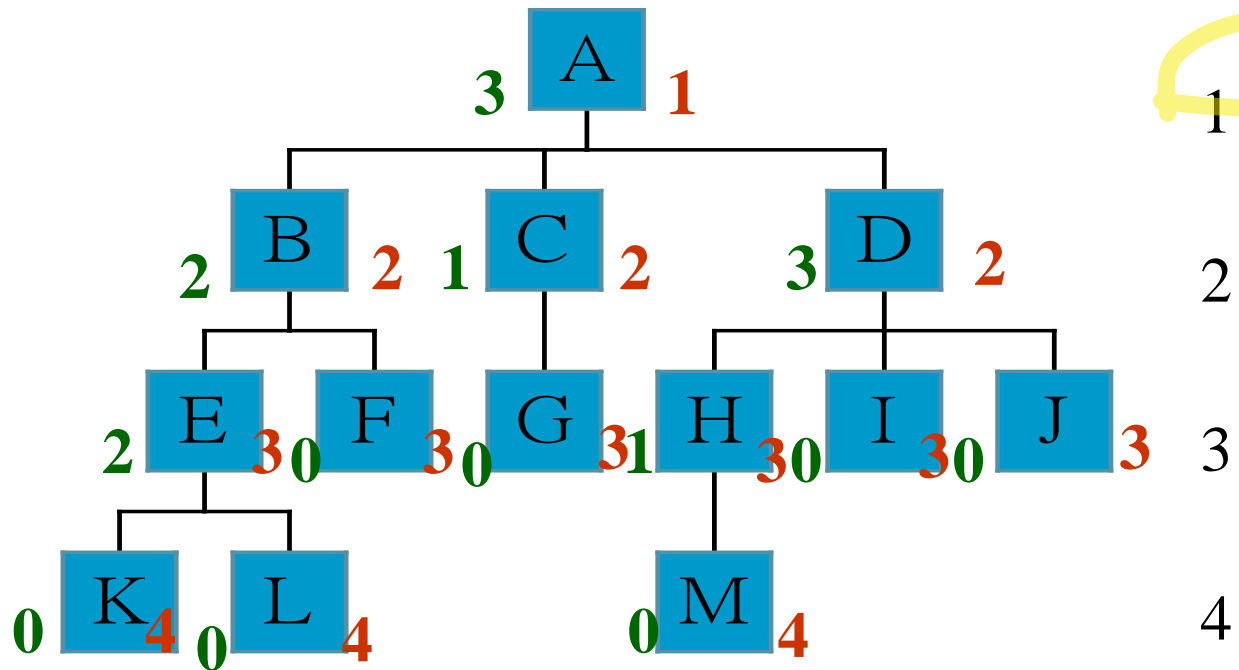


# Definition of Tree

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

# Level and Depth

1. node (13)
2. leaf (terminal)
3. nonterminal
4. parent
5. children
6. sibling
7. degree of a tree (3)
8. ancestor
9. level of a node
10. height of a tree (4)



# Terminology

- The *degree* of a node is the number of subtrees of the node
  - The degree of A is 3; the degree of C is 1.
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the *parent* of the subtrees.
- These subtrees are the *children* of the node.
- Children of the same parent are *siblings*.
- The *ancestors* of a node are all the nodes along the path from the root to the node.

# Representation of Trees

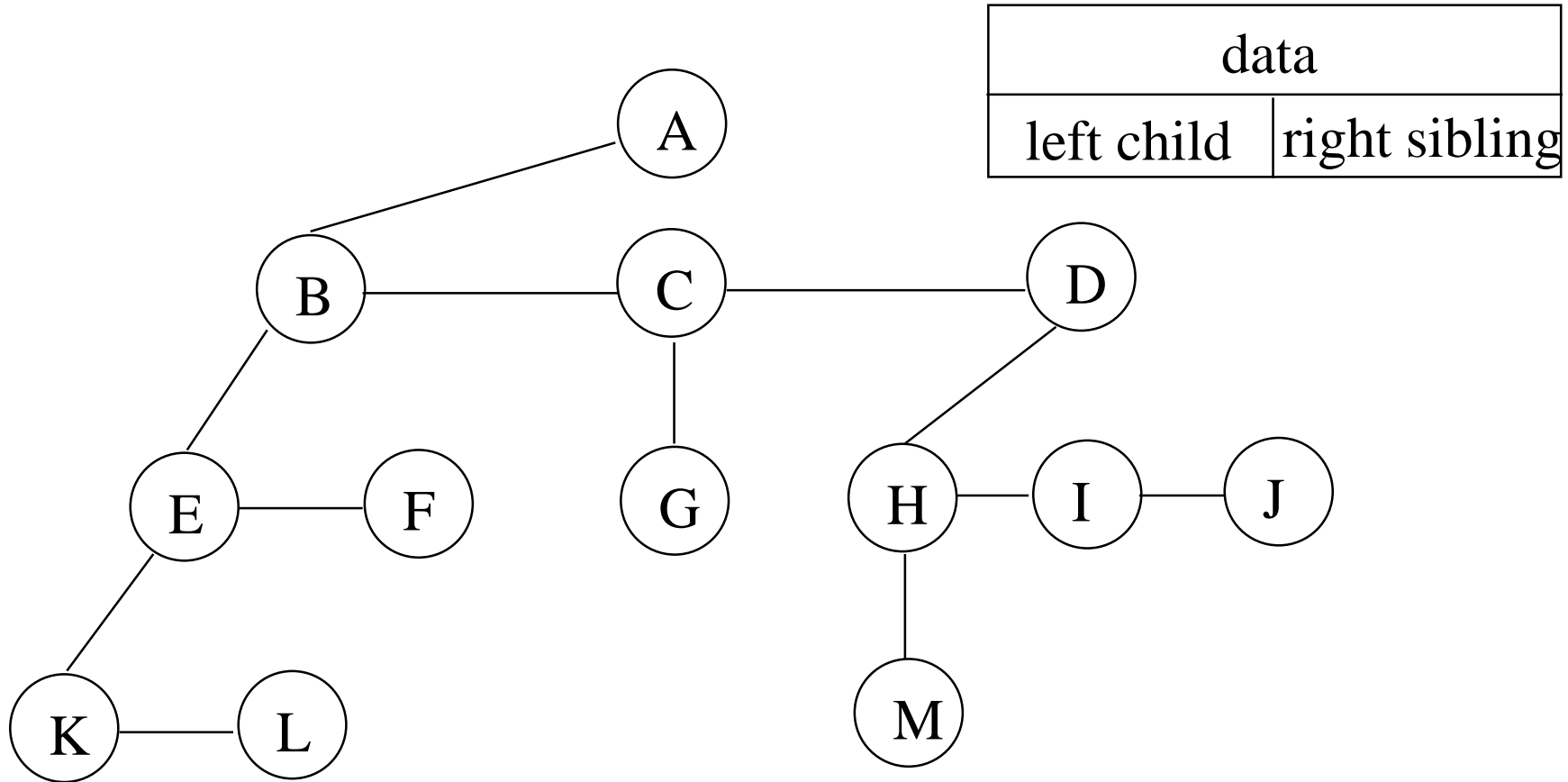
## □ List Representation

- ( A ( B ( E ( K, L ), F ), C ( G ), D ( H ( M ), I, J ) ) )
- The root comes first, followed by a list of sub-trees

data	link 1	link 2	...	link n
------	--------	--------	-----	--------

How many link fields are needed in such a representation?

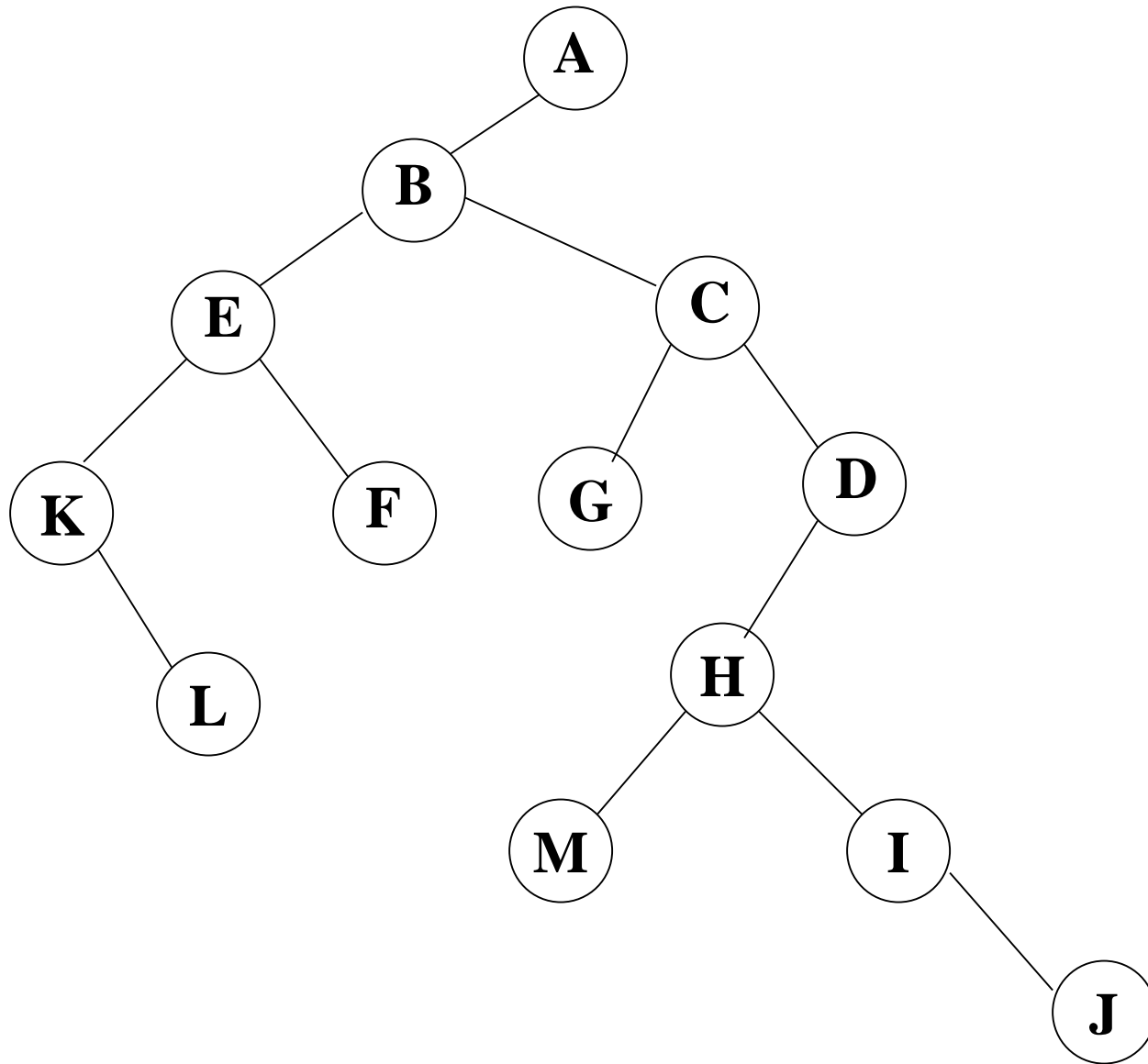
# Left Child - Right Sibling



# Binary Trees

- A 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 subtree* and *the right subtree*.
- Any tree can be transformed into binary tree.
  - by left child-right sibling representation
- The left subtree and the right subtree are distinguished.





\*Figure 5.2 Left child-right child tree representation of a tree

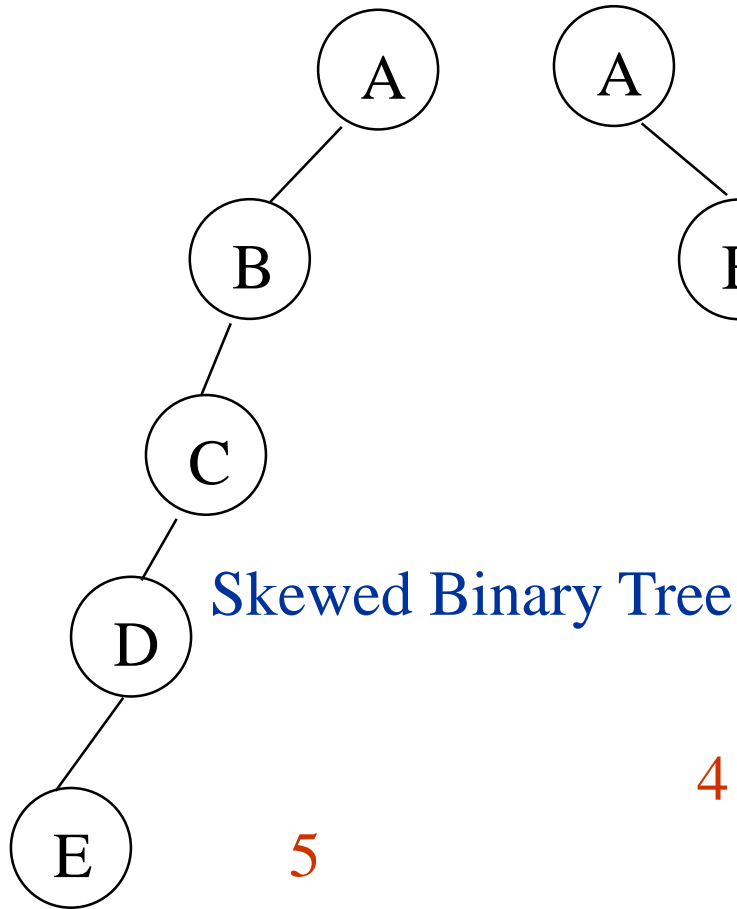
# Abstract Data Type *Binary\_Tree*

- structure *Binary\_Tree* (abbreviated *BinTree*)
- objects: a finite set of nodes either empty or consisting of a root node, *left Binary\_Tree*, and *right Binary\_Tree*.
- functions:
  - for all  $bt, bt1, bt2 \in BinTree, item \in element$
- *Bintree* *Create()*::= creates an empty binary tree
- *Boolean* *IsEmpty(bt)*::= if ( $bt == \text{empty binary tree}$ ) return *TRUE* else return *FALSE*

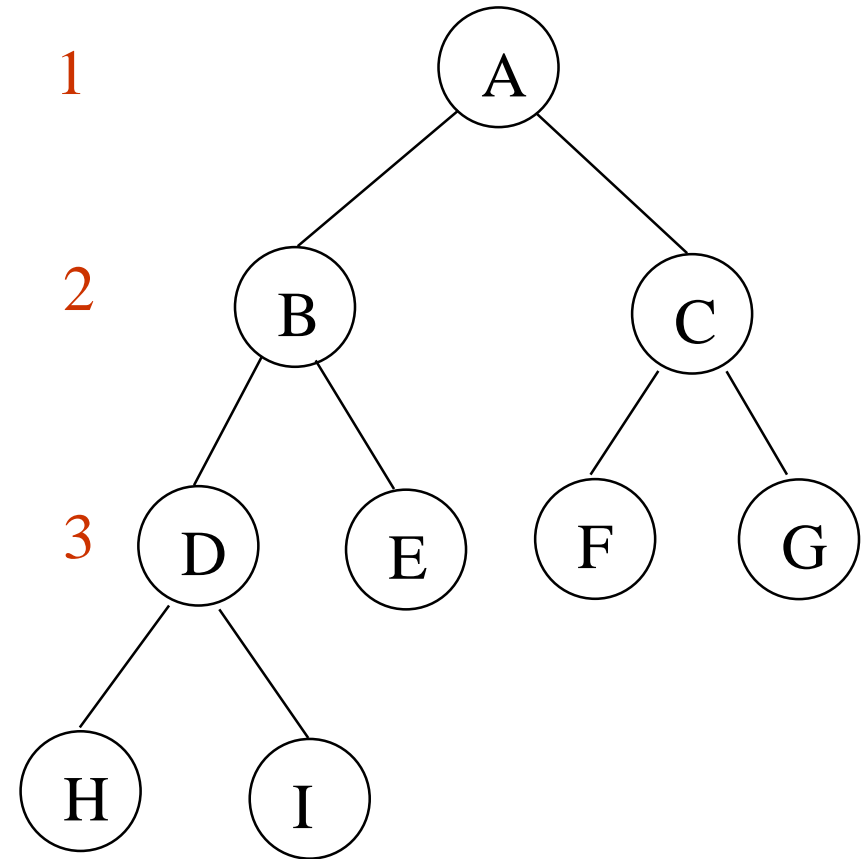
# Abstract Data Type Binary\_Tree

- *BinTree* MakeBT(*bt1*, *item*, *bt2*) ::= return a binary tree whose left subtree is *bt1*, whose right subtree is *bt2*, and whose root node contains the data *item*
- *Bintree* Lchild(*bt*) ::= if (IsEmpty(*bt*)) return error else return the left subtree of *bt*
- *element* Data(*bt*) ::= if (IsEmpty(*bt*)) return error else return the data in the root node of *bt*
- *Bintree* Rchild(*bt*) ::= if (IsEmpty(*bt*)) return error else return the right subtree of *bt*

# Samples of Trees



Complete Binary Tree



# Maximum Number of Nodes in BT

- The maximum number of nodes on level  $i$  of a binary tree is  $2^{i-1}$ ,  $i \geq 1$ .
- The maximum number of nodes in a binary tree of depth  $k$  is  $2^k - 1$ ,  $k \geq 1$ .

**Prove by induction.**

$$\sum_{i=1}^k 2^{i-1} = 2^k - 1$$

pp. 200

# Relations between Number of Leaf Nodes and Nodes of Degree 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  $n_0 = n_2 + 1$

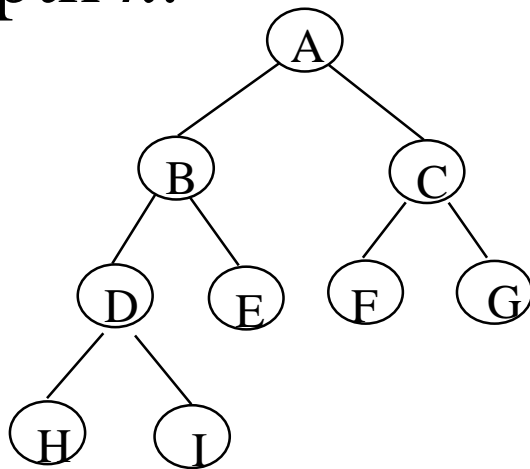
proof:

- Let  $n$  and  $B$  denote the total number of nodes & branches in  $T$ .
- Let  $n_0$ ,  $n_1$ ,  $n_2$  represent the nodes with no children, single child, and two children respectively.

$$n = n_0 + n_1 + n_2, \quad n = B + 1, \quad n = B + 1 = n_1 + 2n_2 + 1, \\ n_1 + 2n_2 + 1 = n_0 + n_1 + n_2 \implies n_0 = n_2 + 1$$

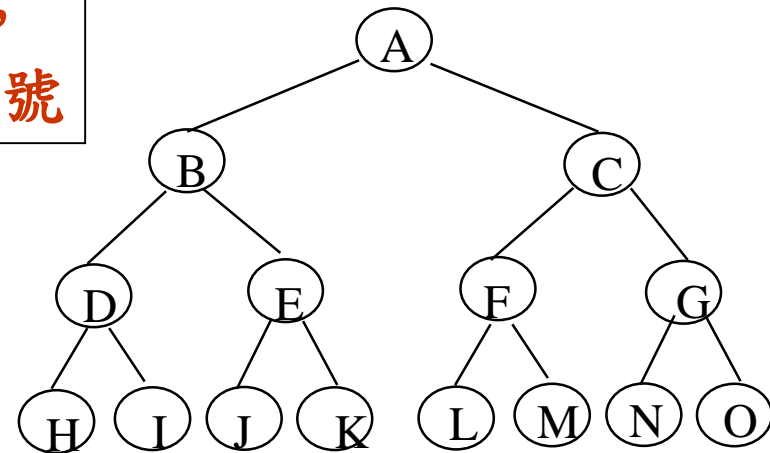
# Full BT VS Complete BT

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



Complete binary tree

由上至下，  
由左至右編號



Full binary tree of depth 4

# Binary Tree Representations

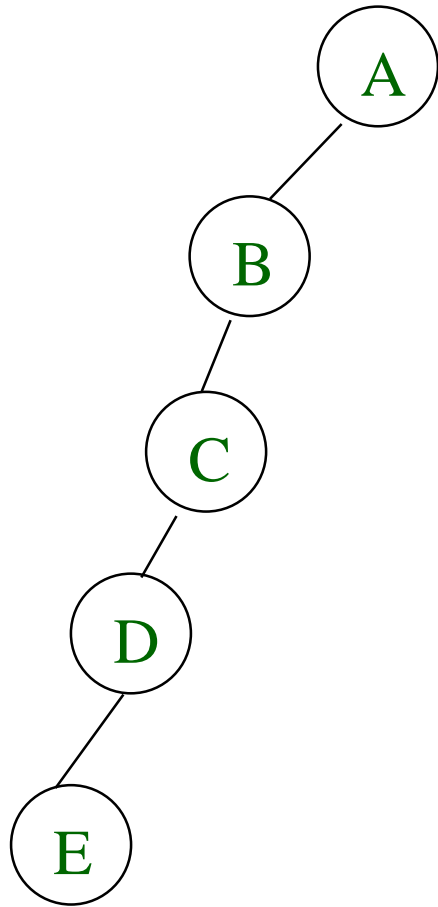
- If a complete binary tree with  $n$  nodes (depth =  $\log n + 1$ ) is represented sequentially, then for any node with index  $i$ ,  $1 \leq i \leq n$ , we have:
- $parent(i)$  is at  $i/2$  if  $i \neq 1$ . If  $i = 1$ ,  $i$  is at the root and has no parent.
  - $left\_child(i)$  is at  $2i$  if  $2i \leq n$ . If  $2i > n$ , then  $i$  has no left child.
  - $right\_child(i)$  is at  $2i + 1$  if  $2i + 1 \leq n$ . If  $2i + 1 > n$ , then  $i$  has no right child.



# Sequential Representation

(1) waste space

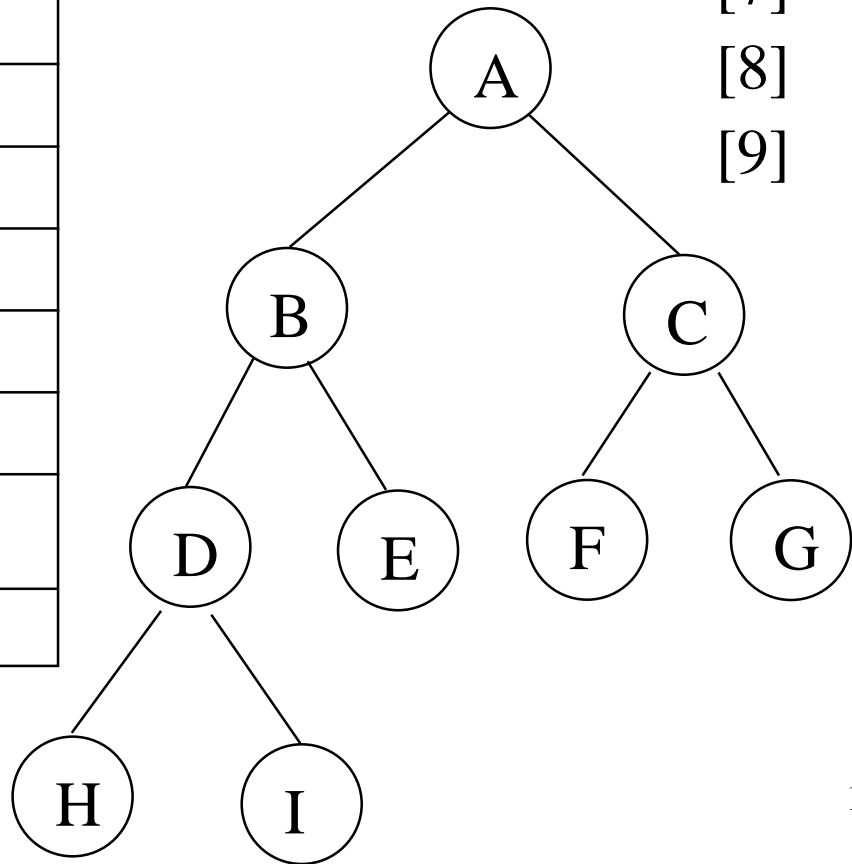
(2) insertion/deletion problem



[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

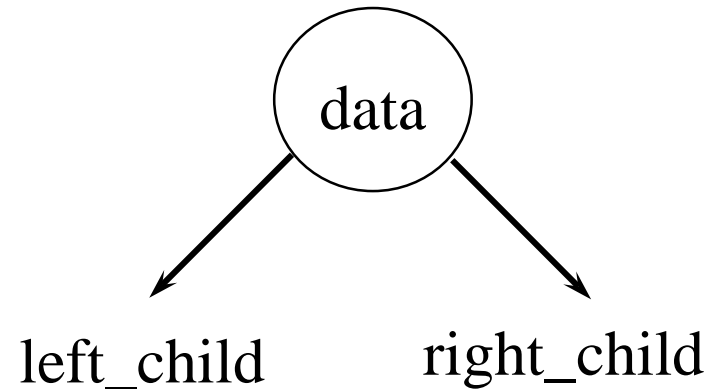
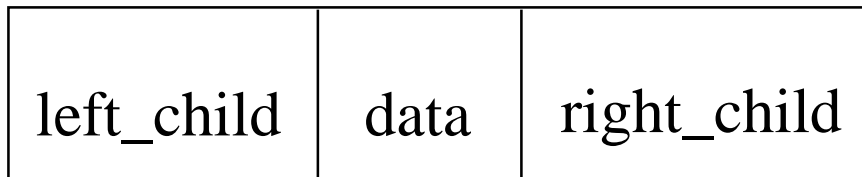
- [1]
- [2]
- [3]
- [4]
- [5]
- [6]
- [7]
- [8]
- [9]

A
B
C
D
E
F
G
H
I



# Linked Representation

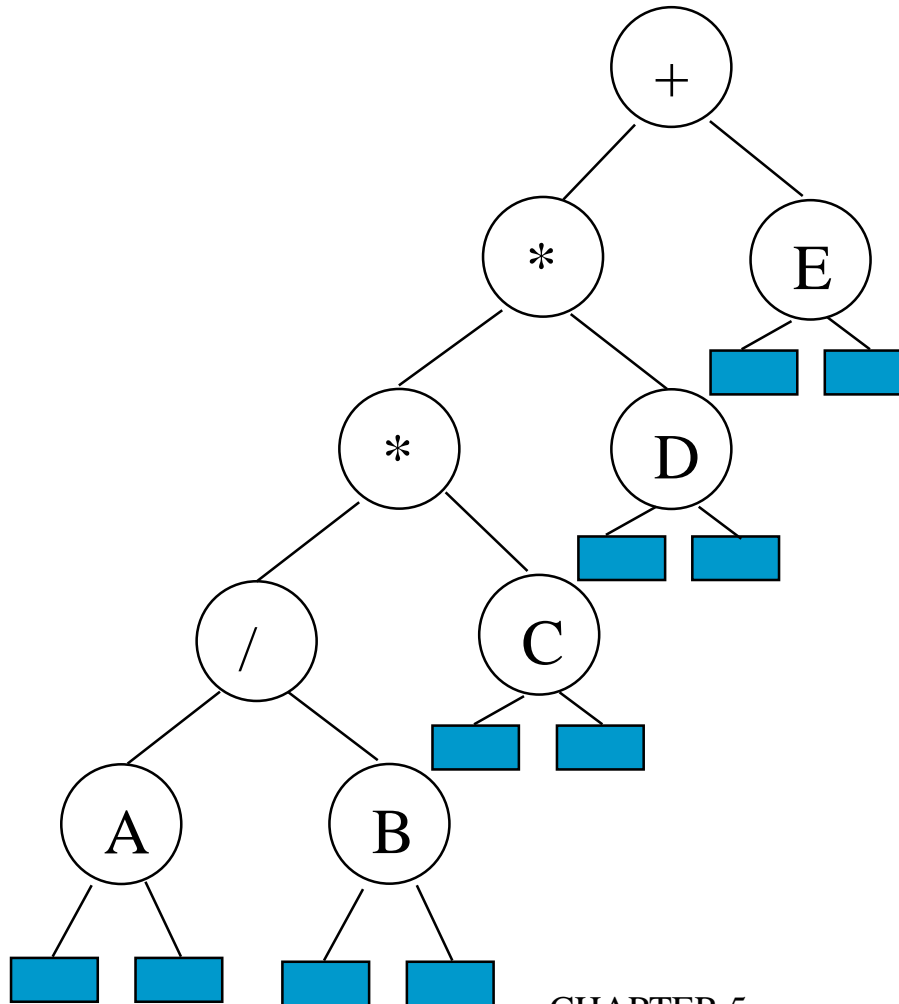
```
typedef struct node *tree_pointer;  
typedef struct node {  
    int data;  
    tree_pointer left_child, right_child;  
};
```



# Binary Tree Traversals

- Let L, V, and R stand for moving left, visiting the node, and moving right.
- There are six possible combinations of traversal
  - LVR, LRV, VLR, VRL, RVL, RLV
- Adopt convention that we traverse left before right, only 3 traversals remain
  - LVR, LRV, VLR
  - inorder, postorder, preorder

# Arithmetic Expression Using BT



inorder traversal

$A / B * C * D + E$

infix expression

preorder traversal

$+ * * / A B C D E$

prefix expression

postorder traversal

$A B / C * D * E +$

postfix expression

level order traversal

$+ * E * D / C A B$

# Inorder Traversal (recursive version)

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

A / B \* C \* D + E

# Preorder Traversal (recursive version)

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

+ \*\* / A B C D E

# Postorder Traversal (recursive version)

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

AB / C \* D \* E +

# Iterative Inorder Traversal

(using stack)

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

**O(n)**



# Trace Operations of Inorder Traversal

Call of inorder	Value in root	Action	Call of inorder	Value in root	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	

# Level Order Traversal

(using queue)

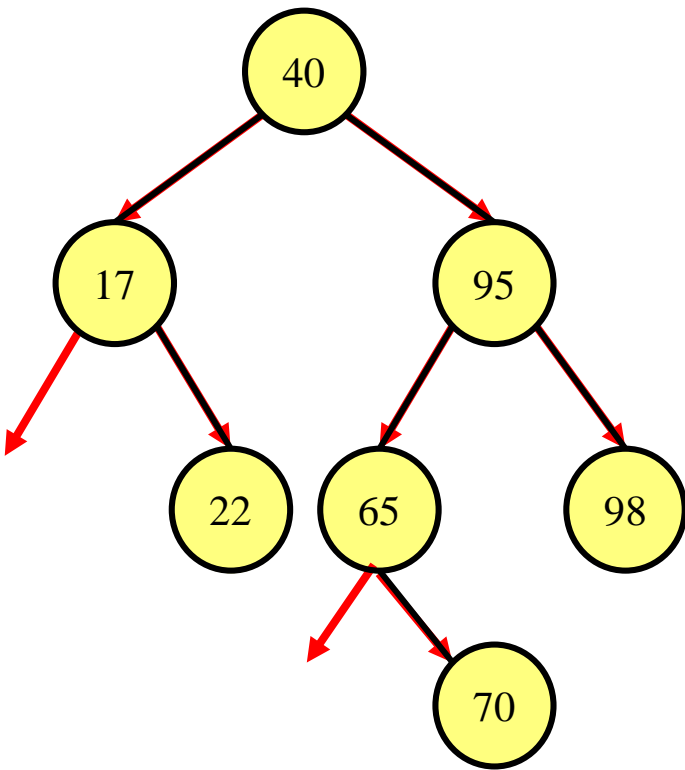
```
void levelOrder(tree_pointer ptr)
/* level order tree traversal */
{
    int front = rear = 0;
    tree_pointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty queue */
    addq(ptr);
    for (;;) {
        ptr = delete();
```

```
if (ptr) {
    printf("%d", ptr->data);
    if (ptr->left_child)
        addq(ptr->left_child);
    if (ptr->right_child)
        addq(ptr->right_child);
}
else break;
}
}
```

+ \* E \* D / C A B

# Traversing a binary tree

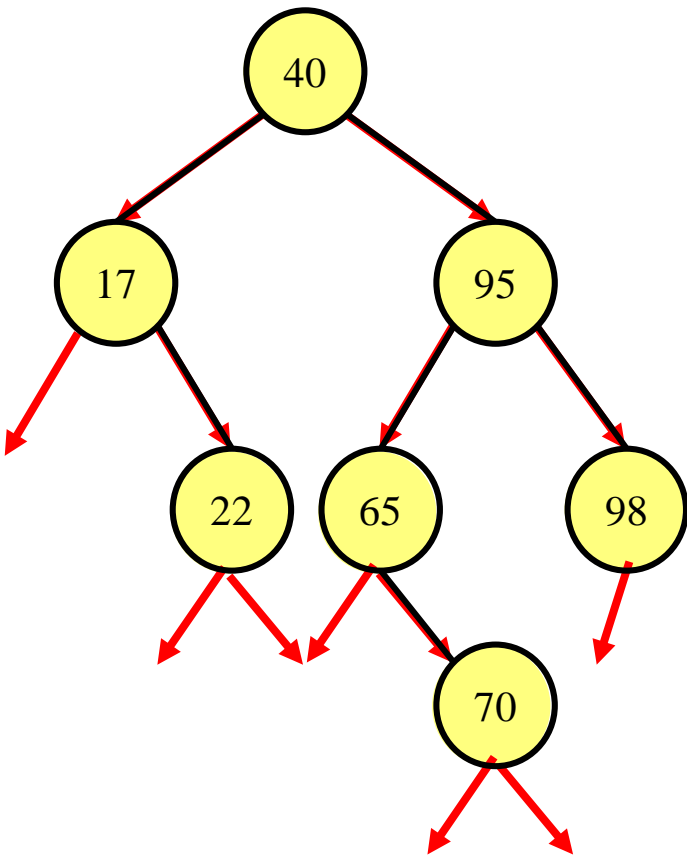
□ 前序走訪(preorder):



```
void preOrder( TreeNodePtr treePtr ){  
    // if tree is not empty, then traverse  
    if ( treePtr != NULL ) {  
        printf( "%3d", treePtr->data );  
        preOrder( treePtr->leftPtr );  
        preOrder( treePtr->rightPtr );  
    }  
}
```

# Traversing a binary tree

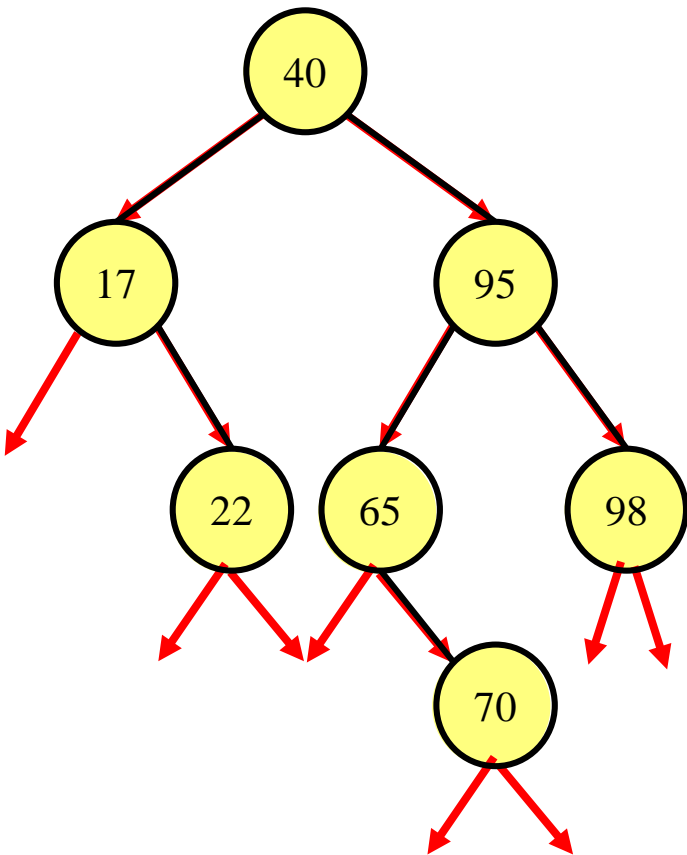
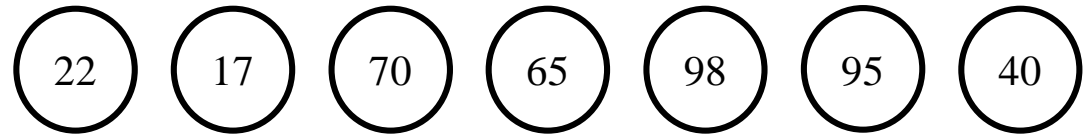
□ 中序走訪(inorder):



```
void inOrder( TreeNodePtr treePtr ){  
    // if tree is not empty, then traverse  
    if ( treePtr != NULL ) {  
        ● inOrder( treePtr->leftPtr );  
        ● printf( "%3d", treePtr->data );  
        ● inOrder( treePtr->rightPtr );  
    }  
}
```

# Traversing a binary tree

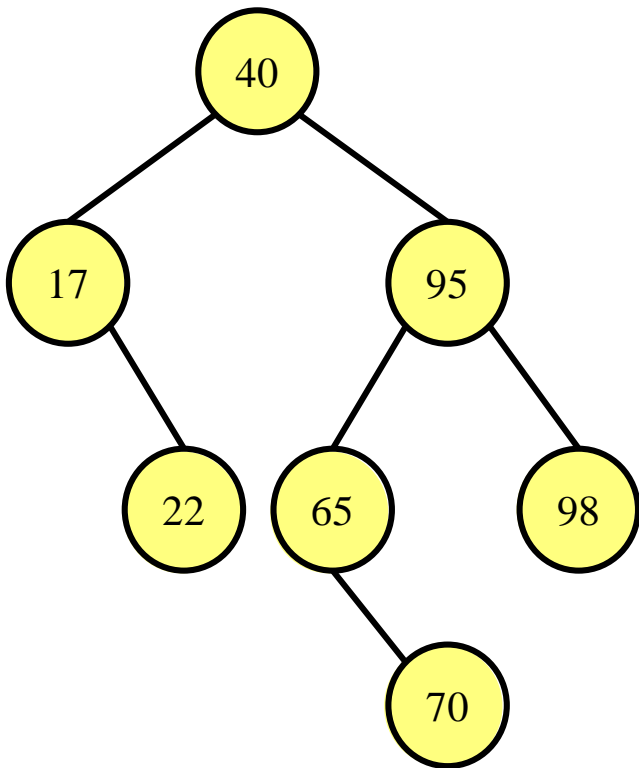
□ 後序走訪(postorder):



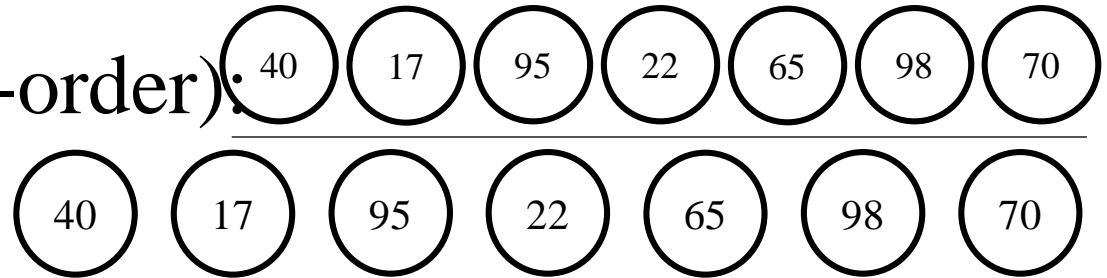
```
void postOrder( TreeNodePtr treePtr ){  
    // if tree is not empty, then traverse  
    if ( treePtr != NULL ) {  
        ● postOrder( treePtr->leftPtr );  
        ● postOrder( treePtr->rightPtr );  
        ● printf( "%3d", treePtr->data );  
    }  
}
```

# Traversing a binary tree

□ 階層走訪(level-order):



queue



```
void levelOrder(TreeNodePtr treePtr){  
    if(!treePtr) return;  
    addq(treePtr);  
    for(;;){  
        treePtr = deleteq();  
        if(treePtr){  
            printf("%3d", treePtr->data);  
            if(treePtr->leftPtr)  
                addq(treePtr->leftPtr);  
            if(treePtr->rightPtr)  
                addq(treePtr->rightPtr);  
        }  
        else  
            break;  
    }  
}
```

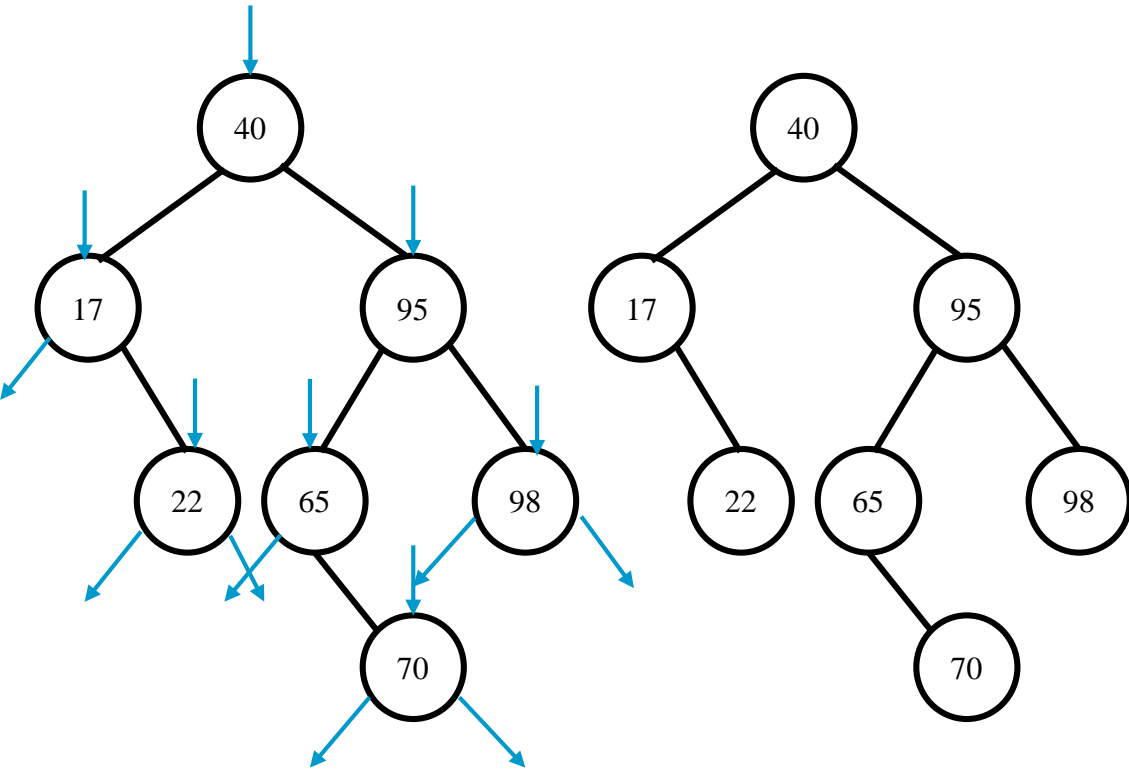
# Copying Binary Trees

```
tree_pointer copy(tree_pointer original)
{
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;
}
```

postorder



# Copying Binary Trees



```
● tree_pointer copy(tree_pointer original) {  
    tree_pointer temp;  
    if (original) {  
        temp=(tree_pointer)  
        malloc(sizeof(node));  
        temp->left_child=copy(original-  
>left_child);  
        temp->right_child=copy(original-  
>right_child);  
        temp->data=original->data;  
        return temp;  
    } //end-of-if  
    return NULL;  
} //end-of-copy()
```

# Equality of Binary Trees

the same topology and data

```
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)))
}
```

# Propositional Calculus Expression

- A variable is an expression.
- If  $x$  and  $y$  are expressions, then  $\neg x$ ,  $x \wedge y$ ,  $x \vee y$  are expressions.
- Parentheses can be used to alter the normal order of evaluation ( $\neg > \wedge > \vee$ ).
- Example:  $x_1 \vee (x_2 \wedge \neg x_3)$
- satisfiability problem: Is there an assignment to make an expression true?

$$(X_1 \wedge \neg X_2) \vee (\neg X_1 \wedge X_3) \vee \neg X_3$$

(t,t,t)

(t,t,f)

(t,f,t)

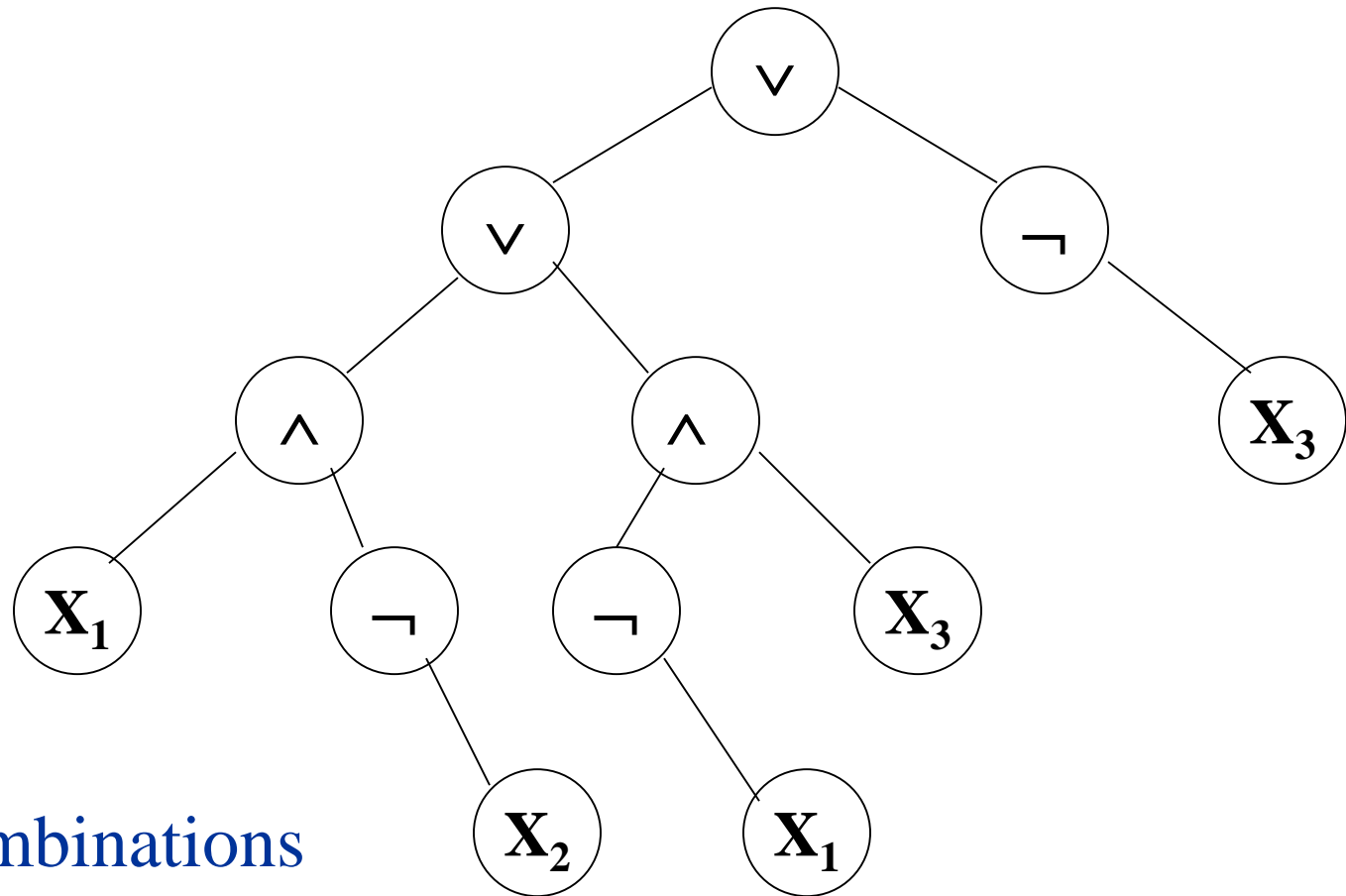
(t,f,f)

(f,t,t)

(f,t,f)

(f,f,t)

(f,f,f)



$2^n$  possible combinations  
for  $n$  variables

postorder traversal (postfix evaluation)

# Node Structure

<i>left_child</i>	<i>data</i>	<i>value</i>	<i>right_child</i>
-------------------	-------------	--------------	--------------------

```
typedef enum { not, and, or, true, false } logical;
typedef struct node *tree_pointer;
typedef struct node {
    tree_pointer left_child;
    logical      data;
    short int    value;
    tree_pointer right_child;
} ;
```

# First version of satisfiability algorithm

```
for (all  $2^n$  possible combinations) {  
    generate the next combination;  
    replace the variables by their values;  
    evaluate root by traversing it in postorder;  
    if (root->value) {  
        printf(<combination>);  
        return;  
    }  
}  
printf("No satisfiable combination \n");
```

# Post-order-eval function

```
void postOrderEval(tree_pointer node)
{
/* modified post order traversal to evaluate a propositional
calculus tree */
  if (node) {
    post_order_eval(node->left_child);
    post_order_eval(node->right_child);
    switch(node->data) {
      case not: node->value =
                !node->right_child->value;
                break;
```

```
case and:    node->value =
            node->right_child->value &&
            node->left_child->value;
            break;
```

```
case or:     node->value =
            node->right_child->value ||
            node->left_child->value;
            break;
```

```
case true:  node->value = TRUE;
            break;
```

```
case false: node->value = FALSE;
```

```
}
```

```
}
```

```
}
```



# Threaded Binary Trees

- Many null pointers in current representation of binary trees

n: number of nodes;

total links:  $2n$

number of non-null links:  $n-1$

**null links:  $2n-(n-1) \Rightarrow n+1$**

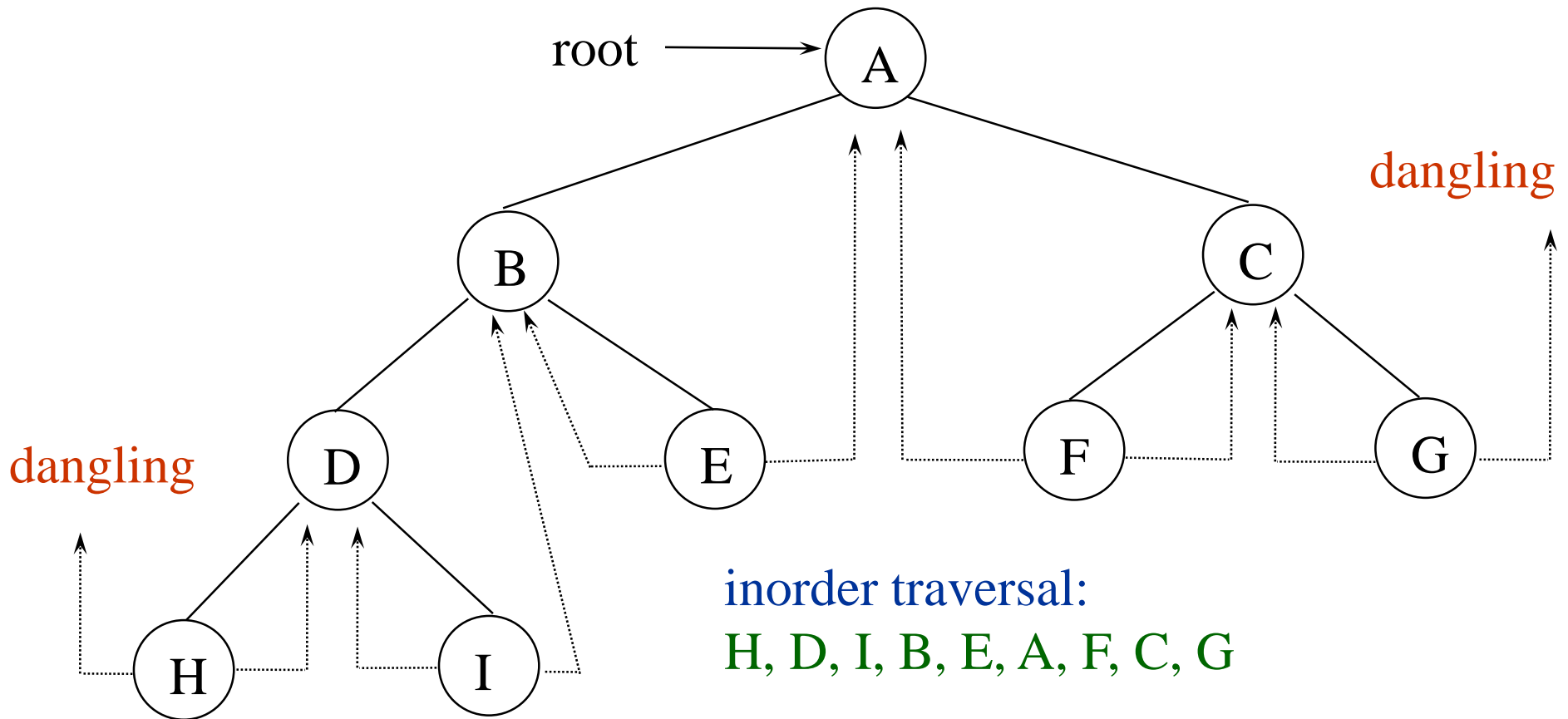
- Replace these null pointers with some useful “threads”.

## Threaded Binary Trees *(Continued)*

If `ptr->left_child` is null,  
replace it with a pointer to the node that would be  
visited *before ptr in an inorder traversal*

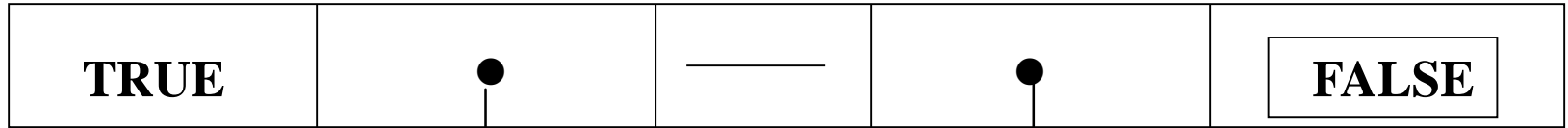
If `ptr->right_child` is null,  
replace it with a pointer to the node that would be  
visited *after ptr in an inorder traversal*

# A Threaded Binary Tree



# Data Structures for Threaded BT

left\_thread   left\_child   data   right\_child   right\_thread



**TRUE: thread**

**FALSE: child**

```
typedef struct threaded_tree
```

```
  *threaded_pointer;
```

```
typedef struct threaded_tree {
```

```
  short int left_thread;
```

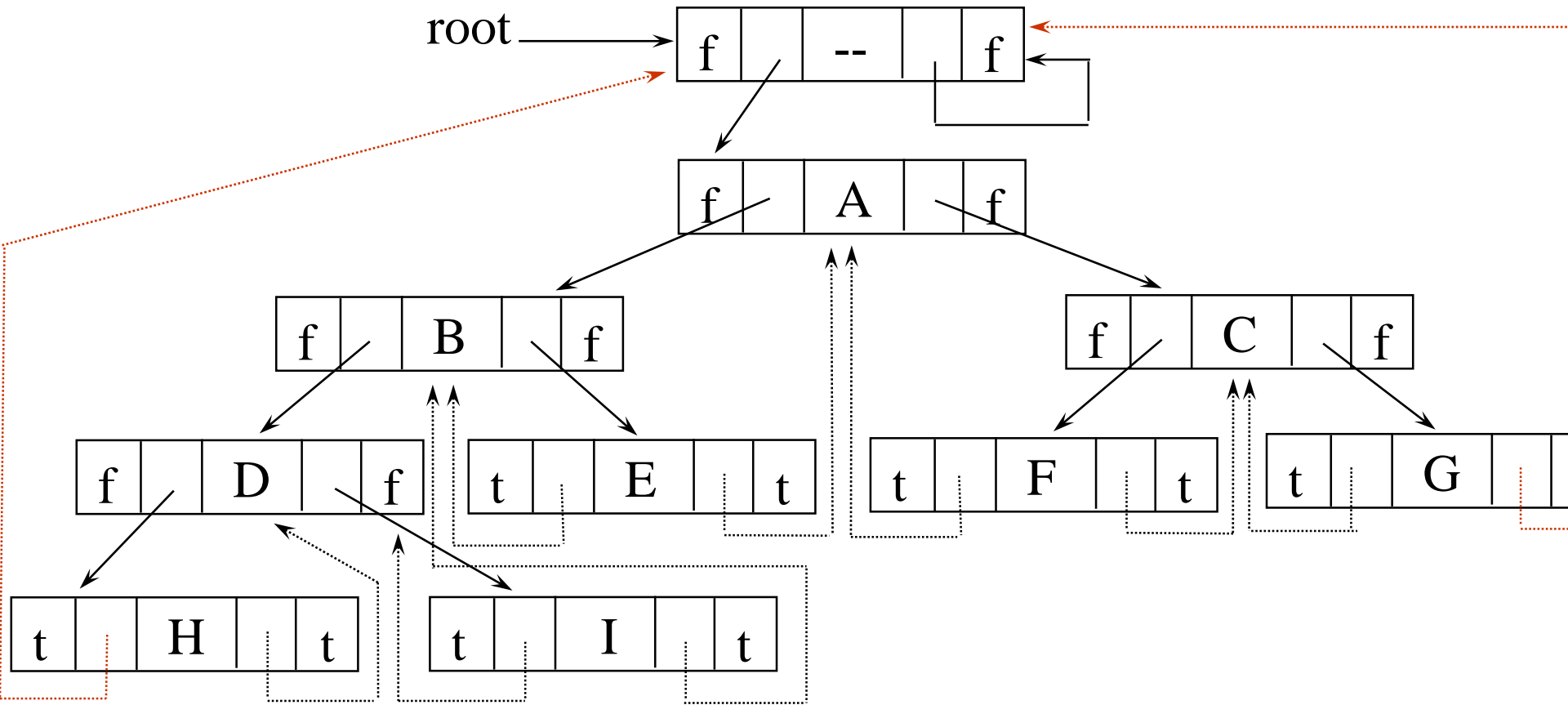
```
  threaded_pointer left_child;
```

```
  char data;
```

```
  threaded_pointer right_child;
```

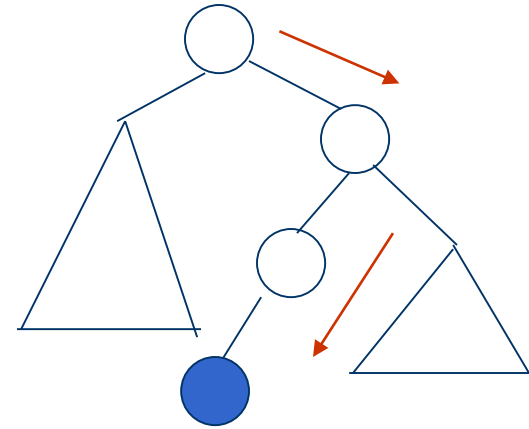
```
  short int right_thread; };
```

# Memory Representation of A Threaded Binary Tree



# Next Node in Threaded BT

```
threaded_pointer insucc(threaded_pointer
    tree)
{
    threaded_pointer temp;
    temp = tree->right_child;
    if (!tree->right_thread)
        while (!temp->left_thread)
            temp = temp->left_child;
    return temp;
}
```



# Inorder Traversal of Threaded BT

```
void tinorder(threaded_pointer tree)
{
/* traverse the threaded binary tree
inorder */
    threaded_pointer temp = tree;
    for (;;) {
        temp = insucc(temp);
        O(n) if (temp==tree) break;
        printf("%3c", temp->data);
    }
}
```

# Inserting Nodes into Threaded BTs

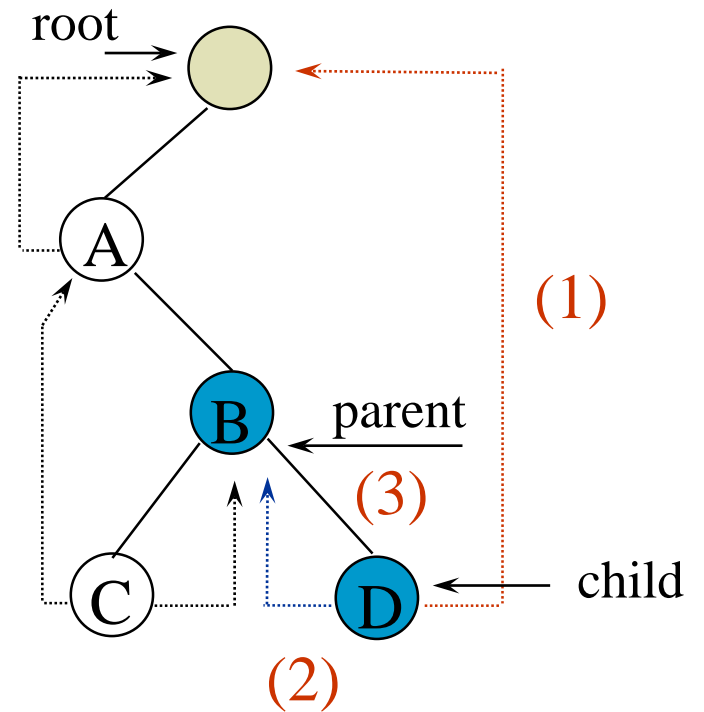
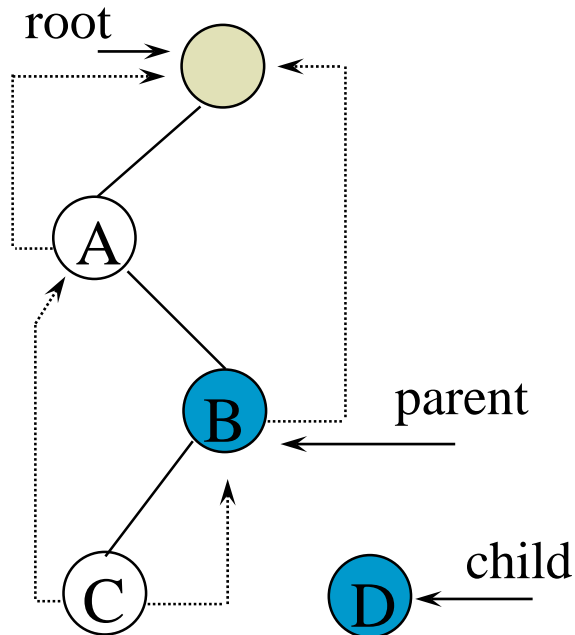
- Insert `child` as the right child of node (`parent`)
  - change `parent->right_thread` to `FALSE`
  - set `child->left_thread` and `child->right_thread` to `TRUE`
    1. set `child->right_child` to `parent->right_child`
    2. set `child->left_child` to point to `parent`
    3. change `parent->right_child` to point to `child`



# Examples

Insert a node D as a right child of B.

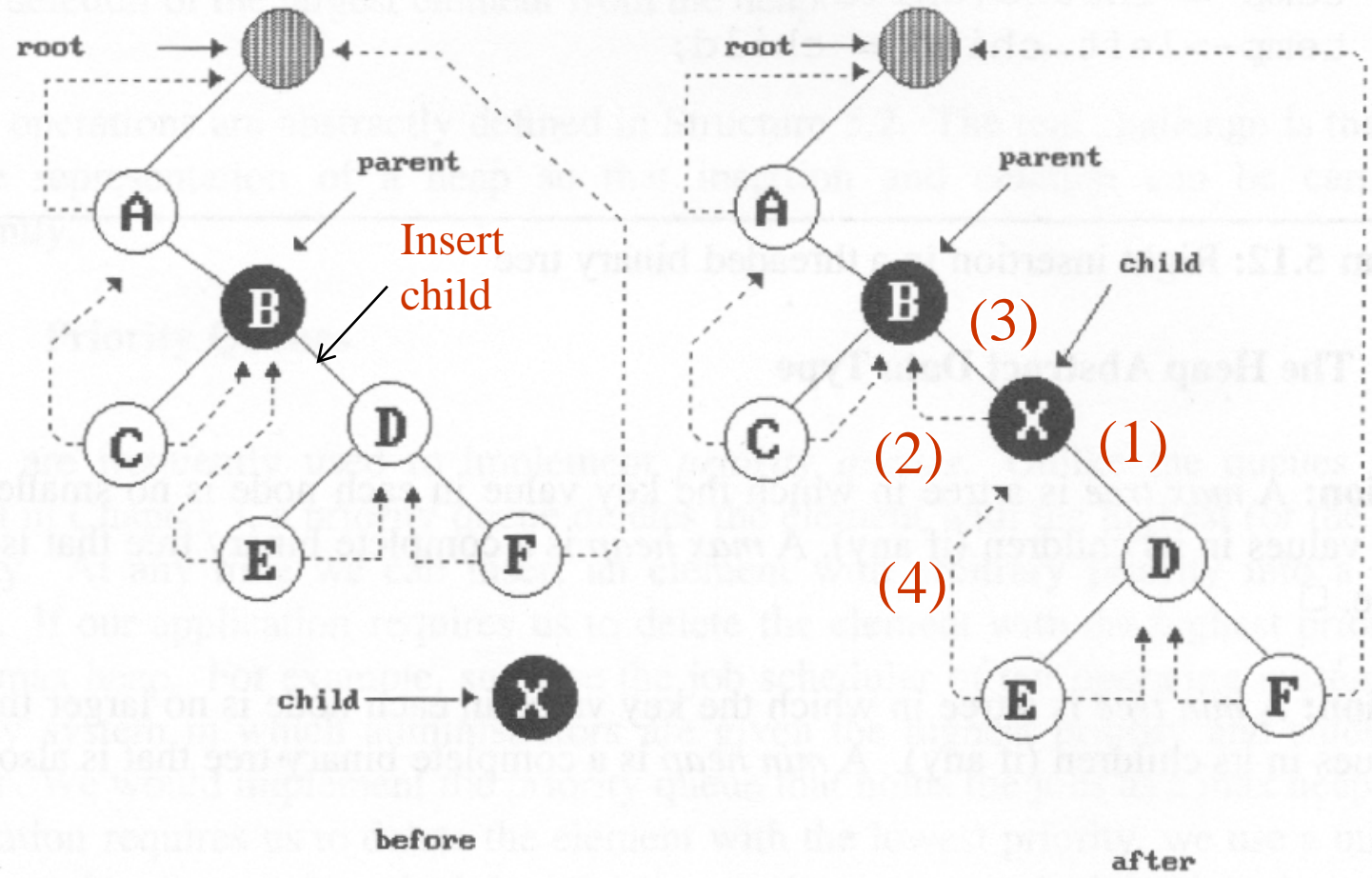
empty



(a)

**\*Figure 5.24:** Insertion of child as a right child of parent in a threaded binary tree nonempty

nonempty



(b)

# Right Insertion in Threaded BTs

```
void insertRight(threaded_pointer parent,  
                threaded_pointer child)
```

```
{
```

```
    threaded_pointer temp;
```

```
(1) child->right_child = parent->right_child;  
    child->right_thread = parent->right_thread;
```

```
(2) child->left_child = parent;           case (a)  
    child->left_thread = TRUE;
```

```
(3) parent->right_child = child;  
    parent->right_thread = FALSE;  
    if (!child->right_thread) {           case (b)
```

```
(4)     temp = insucc(child);  
        temp->left_child = child;
```

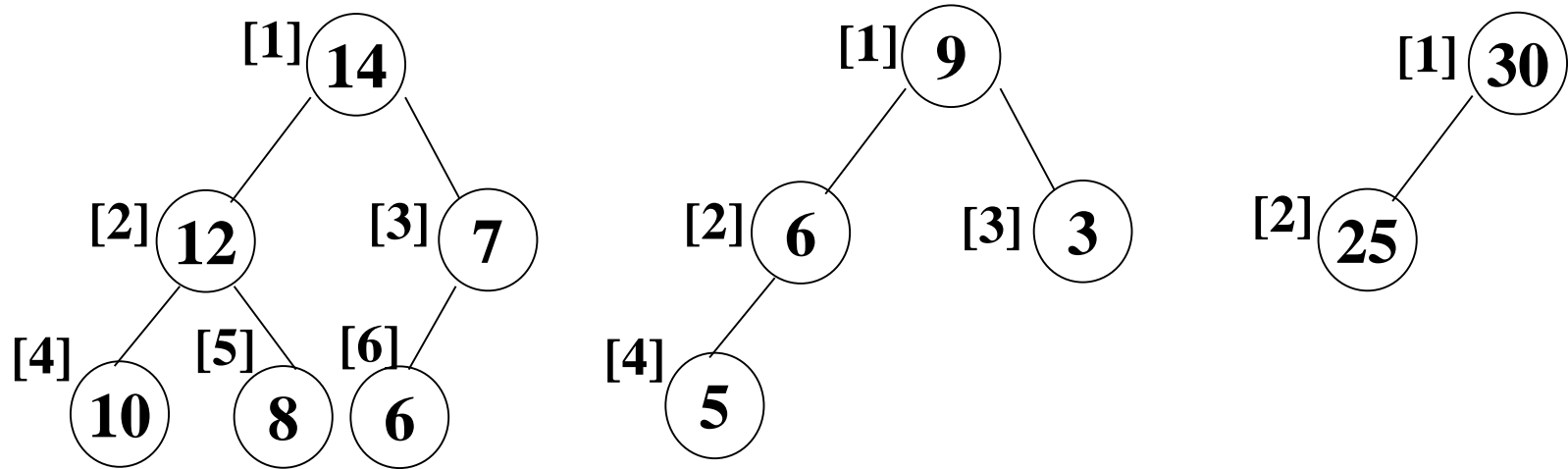
```
    }
```

```
}
```

# Heap

- A *max tree* is a tree in which the key value in each node is *no smaller than* the key values in its children.
  - A *max heap* is a *complete binary tree* that is also a max tree.
- A *min tree* is a tree in which the key value in each node is *no larger than* the key values in its children.
  - A *min heap* is a *complete binary tree* that is also a min tree.
- Operations on heaps
  - creation of an empty heap
  - insertion of a new element into the heap
  - deletion of the largest element from the heap

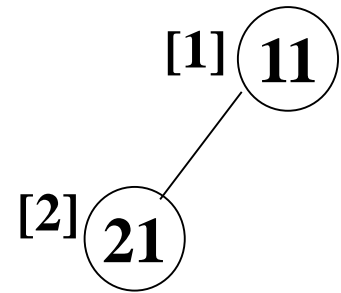
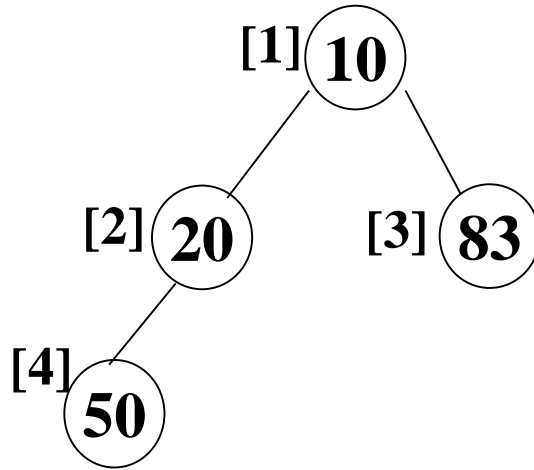
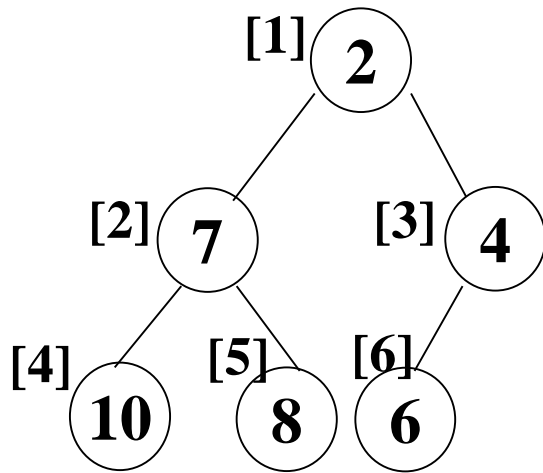
\*Figure 5.25: Max heaps



Property:

The root of max heap (min heap) contains the largest (smallest).

\*Figure 5.26: Min heaps



# ADT for Max Heap

structure MaxHeap

□ objects: a complete binary tree of  $n > 0$  elements organized so that the value in each node is at least as large as those in its children

functions:

for all *heap* belong to *MaxHeap*, *item* belong to *Element*,  $n$ ,  
*max\_size* belong to integer

- MaxHeap Create(max\_size)::= create an empty heap that can hold a maximum of max\_size elements
- Boolean HeapFull(heap, n)::= if (n==max\_size) return TRUE  
else return FALSE
- MaxHeap Insert(heap, item, n)::= if (!HeapFull(heap,n)) insert item into heap and return the resulting heap  
else return error
- Boolean HeapEmpty(heap, n)::= if (n>0) return FALSE  
else return TRUE
- Element Delete(heap,n)::= if (!HeapEmpty(heap,n)) return one instance of the **largest** element in the heap and remove it from the heap  
else return error

# Application: priority queue

- Machine service (Example 5.1)
  - amount of time (min heap)
  - amount of payment (max heap)
- Factory (Example 5.2)
  - time tag (min heap)



## ADT MaxPriorityQueue是

物件： $n$ 個元素形成的集合( $n > 0$ )，每個元素有一個鍵值

函式：對所有的 $q \in \text{MaxPriorityQueue}$ ， $item \in \text{Element}$ ， $n$ 是整數

<b>MaxPriorityQueue</b> <b>create(max_size)</b>	::= 建立一個空的優先權佇列
<b>Boolean isEmpty(q,n)</b>	::= <b>if</b> ( $n > 0$ ) <b>return</b> <b>FALSE</b> <b>else return</b> <b>TRUE</b>
<b>Element top(q,n)</b>	::= <b>if</b> ( <b>!isEmpty</b> ( $q,n$ )) <b>return</b> $q$ 內 最大的元素 <b>else return</b> 錯誤
<b>Element pop(q,n)</b>	::= <b>if</b> ( <b>!isEmpty</b> ( $q,n$ )) <b>return</b> $q$ 內 最大的元素並把它從堆積中 <u>移除</u> <b>else return</b> 錯誤
<b>MaxPriorityQueue</b> <b>push(q,item,n)</b>	::= 把 $item$ 插入 $q$ 中並 <u>回傳優先</u> <u>權佇列的結果</u>

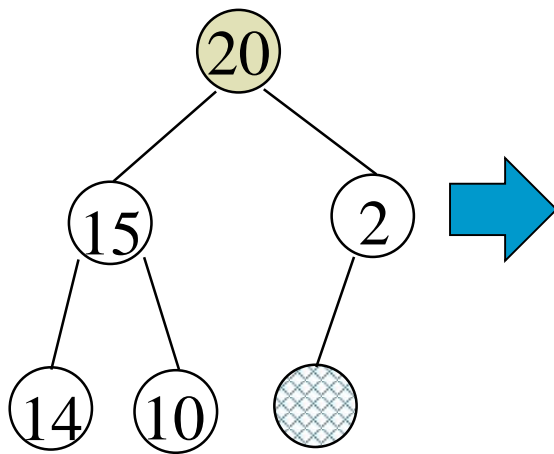
# Data Structures

- unordered linked list
- unordered array
- sorted linked list
- sorted array
- heap

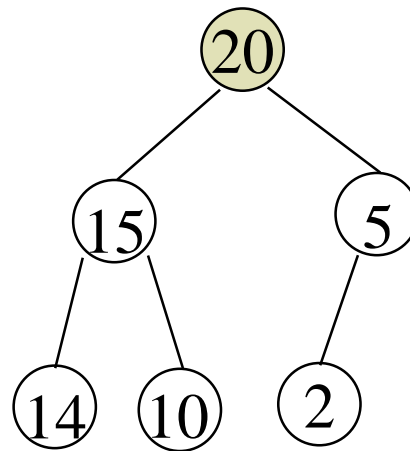
**\*Figure 5.27: Priority queue representations**

Representation	Insertion	Deletion
Unordered array	$\Theta(1)$	$\Theta(n)$
Unordered linked list	$\Theta(1)$	$\Theta(n)$
Sorted array	$O(n)$	$\Theta(1)$
Sorted linked list	$O(n)$	$\Theta(1)$
Max heap	$O(\log_2 n)$	$O(\log_2 n)$

# Example of Insertion to Max Heap

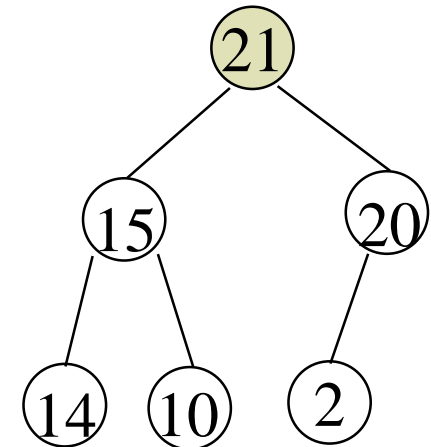


initial location of new node



insert 5 into heap

Case 1



insert 21 into heap

Case 2

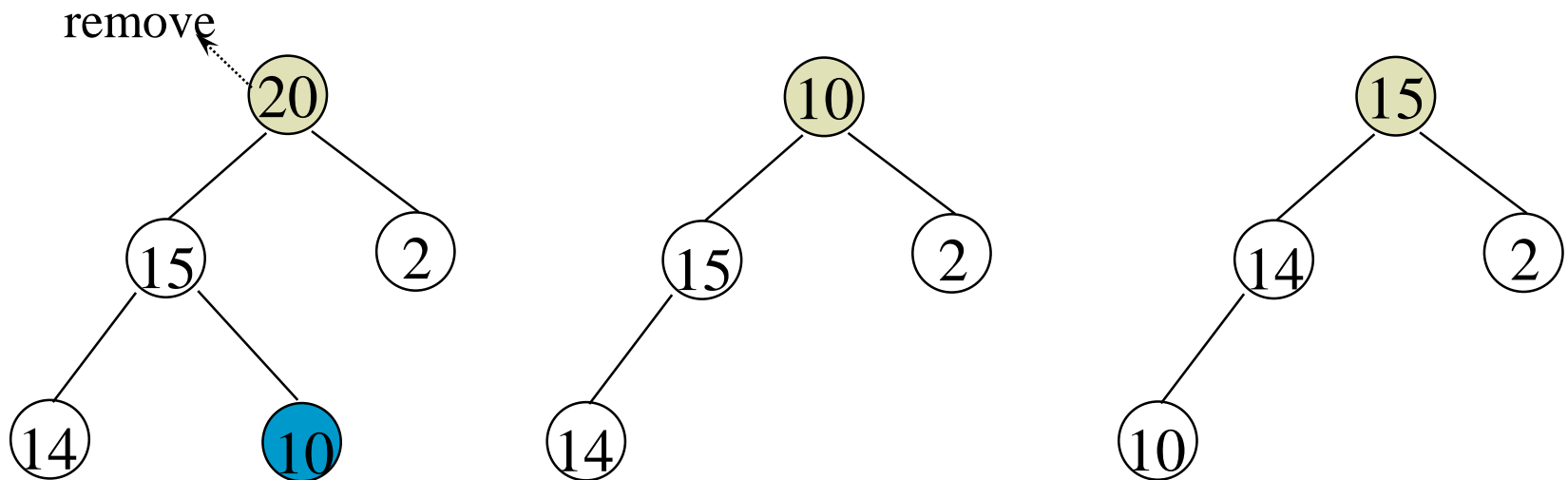
# Insertion into a Max Heap

```
void push(element item, int *n)
{ /* 把項目加入目前大小是n的最大堆積 */
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "the heap is full.\n");
        exit(1);
    }
    i = ++(*n);
    while ((i!=1)&&(item.key>heap[i/2].key)) {
        heap[i] = heap[i/2]; // moving up to root
        i /= 2;
    }
    heap[i]= item;
}
```

$O(\log_2 n)$

$$2^k - 1 = n \implies k = \lceil \log_2(n+1) \rceil$$

# Example of Deletion from Max Heap



# Deletion from a Max Heap

```
element pop(int *n)
{ /* 從堆積中刪除鍵最高的元素 */
    int parent, child;
    element item, temp;
    if (HEAP_EMPTY(*n)) {
        fprintf(stderr, "The heap is empty\n");
        exit(1);
    }
    /* save value of the element with the
       highest key */
    item = heap[1];
    /* use last element in heap to adjust heap */
    temp = heap[(*n)--];
    parent = 1;
    child = 2;
```

```

while (child <= *n) {
    /* find the larger child of the current
       parent */
    if ((child < *n)&&
        (heap[child].key < heap[child+1].key))
        child++;
    if (temp.key >= heap[child].key) break;
    /* move to the next lower level */
    heap[parent] = heap[child];
    head = child;
    child *= 2;
}
heap[parent] = temp;
return item;
}

```



# Binary Search Tree

**ADT Dictionary**是

物件： $n$ 個資料對形成的集合( $n > 0$ )，每個資料對有一個鍵值和搭配的項目

函式：

對於所有的 $d \in \text{Dictionary}$ ， $item \in \text{Item}$ ， $k \in \text{Key}$ ， $n$ 是整數

**Dictionary Create(max\_size)** ::= 建立一個空的字典

**Boolean IsEmpty(d,n)** ::= if( $n > 0$ ) return FALSE  
else return TRUE

**Element Search(d,k)** ::= return 鍵值為 $k$ 的項目  
return NULL 如果沒有此元素

**Element Delete(d,k)** ::= 刪除並回傳(如果有)鍵值為 $k$ 的項目

**void Insert(d,item,k)** ::= 把鍵值為 $k$ 的 $item$ 插入 $d$ 中

# Binary Search Tree

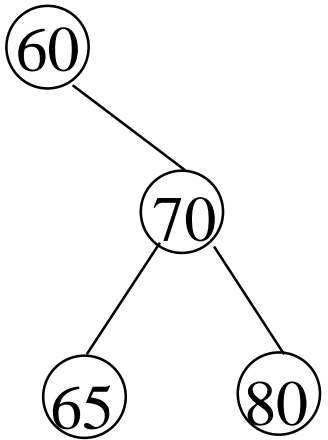
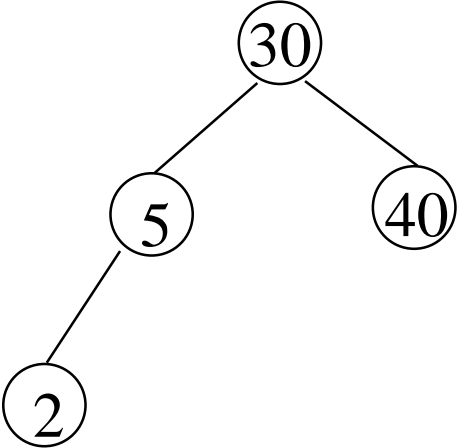
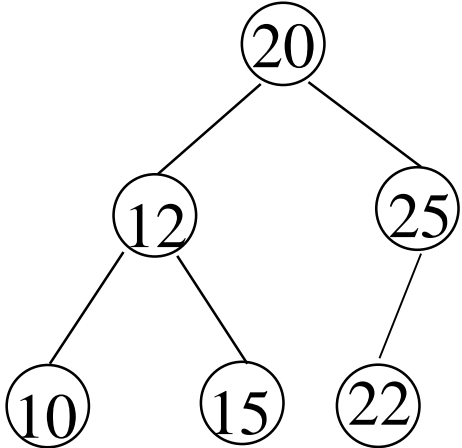
## □ Heap

- a min (max) element is deleted.  $O(\log_2 n)$
- deletion of an arbitrary element  $O(n)$
- search for an arbitrary element  $O(n)$

## □ Binary search tree

- Every element has a unique key.
- The keys in a nonempty **left subtree** (**right subtree**) are **smaller** (**larger**) than the key in the root of subtree.
- The left and right subtrees are also binary search trees.

# Examples of Binary Search Trees



# Searching a Binary Search Tree

```
tree_pointer search(tree_pointer root,
                    int key)
{
    /* return a pointer to the node that
    contains key. If there is no such
    node, return NULL */

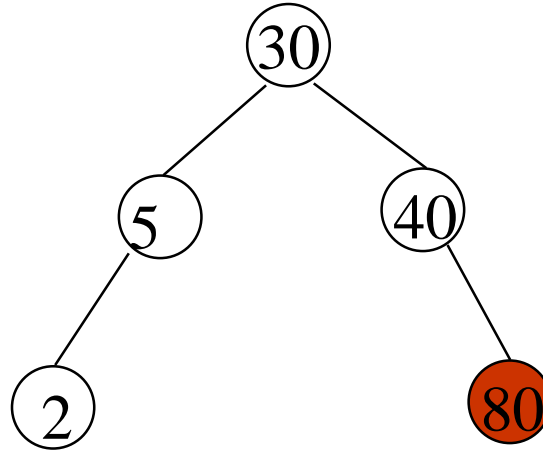
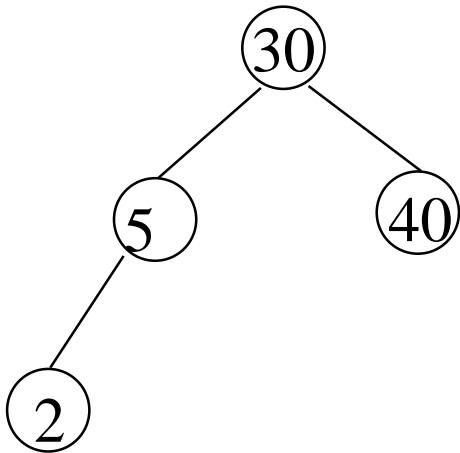
    if (!root) return NULL;
    if (key == root->data) return root;
    if (key < root->data)
        return search(root->left_child,
                       key);
    return search(root->right_child, key);
}
```

# Another Searching Algorithm

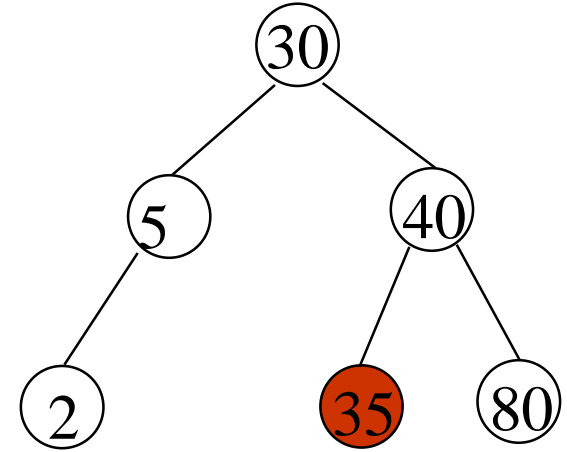
```
tree_pointer iterSearch(tree_pointer
    tree, int key)
{
    while (tree) {
        if (key == tree->data) return tree;
        if (key < tree->data)
            tree = tree->left_child;
        else tree = tree->right_child;
    }
    return NULL;
}
```

**O(h)**

# Insert Node in Binary Search Tree



Insert 80



Insert 35

# Insertion into a Binary Search Tree

```
void insert(tree_pointer *node, int k, itemType
    theItem)
{tree_pointer ptr,
    temp = modified_search(*node, k);
if (temp || !(*node)) {/* k不在樹中*/
    ptr = (tree_pointer) malloc(sizeof(node));
    if (IS_FULL(ptr)) {
        fprintf(stderr, "The memory is full\n");
        exit(1);
    }
    ptr->data.key = k; ptr->data.item = theItem;
    ptr->left_child = ptr->right_child = NULL;
    if (*node)
        if (k < temp->data) temp->left_child=ptr;
        else temp->right_child = ptr;
    else *node = ptr;
}
```

# Binary search tree insert and search

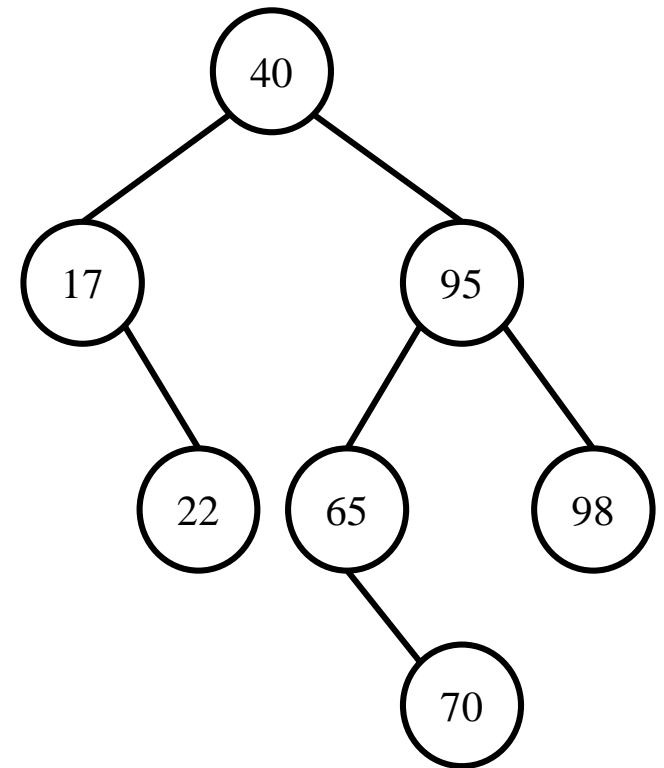
- 用list的方式實作binary search tree的插入和搜尋
- 實作步驟
  1. 建立樹的node結構(struct)
  2. 建立一棵binary search tree
  3. 主要function
    - 插入: insertNode()
    - 搜尋: searchNode()



# Binary search tree insert and search

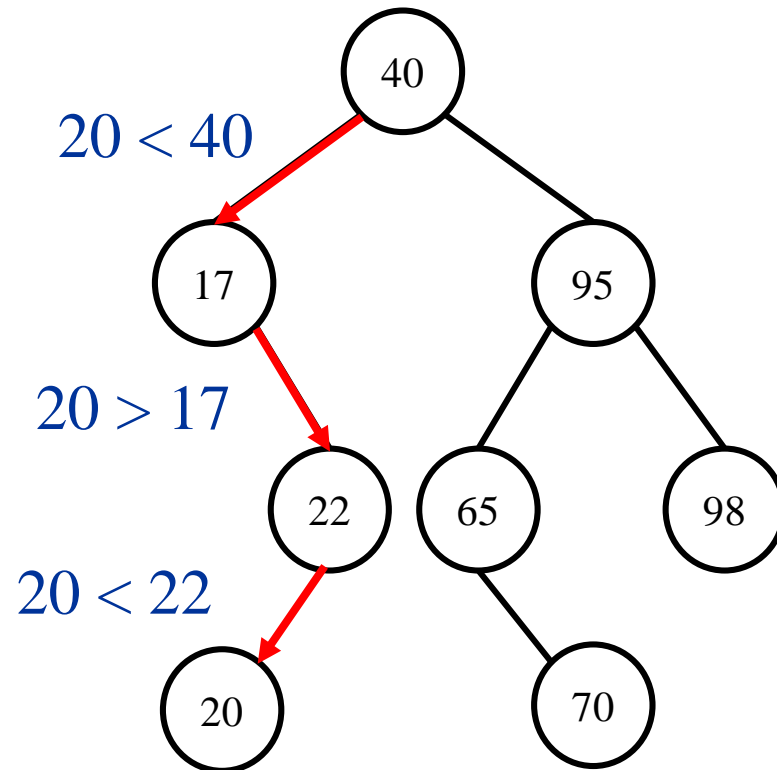
□ 建立一棵binary search tree

40, 17, 95, 22, 65, 70, 98



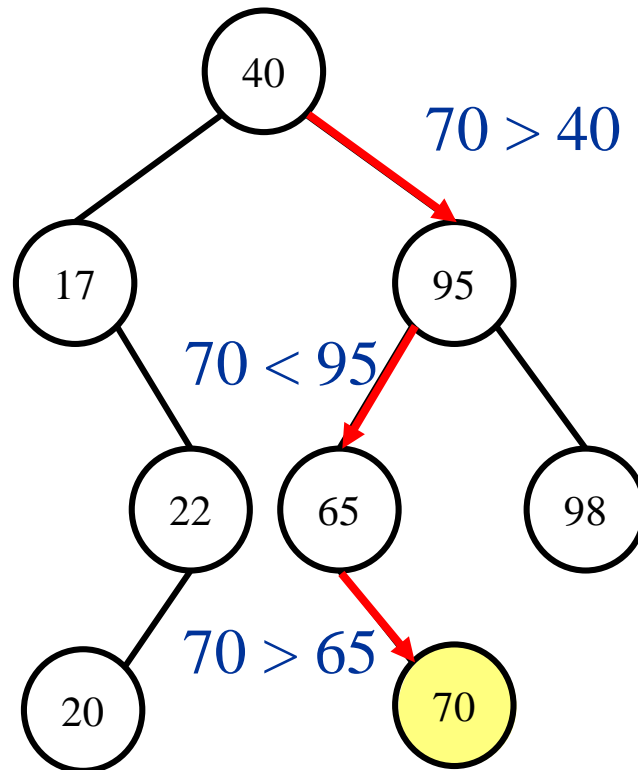
# Binary search tree insert and search

□ 插入 20



# Binary search tree insert and search

□ 搜尋 70



# Binary search tree insert and search

## □ 建立樹的node結構(struct)

```
struct treeNode {
    struct treeNode *leftPtr; // pointer to left subtree
    int data; // node value
    struct treeNode *rightPtr; // pointer to right subtree
};
typedef struct treeNode TreeNode; // synonym for struct treeNode
typedef TreeNode *TreeNodePtr; // synonym for TreeNode*
```

# Binary search tree insert and search

## □ 插入: insertNode()

```
void insertNode( TreeNodePtr *treePtr, int value ){
    // if tree is empty
    if ( *treePtr == NULL ) {
        *treePtr = malloc( sizeof( TreeNode ) );
        // if memory was allocated, then assign data
        if ( *treePtr != NULL ) {
            ( *treePtr )->data = value;
            ( *treePtr )->leftPtr = NULL;
            ( *treePtr )->rightPtr = NULL;
        }
        else {
            printf( "%d not inserted. No memory available.\n", value );
        }
    }
}
```

# Binary search tree insert and search

## □ 插入: insertNode()

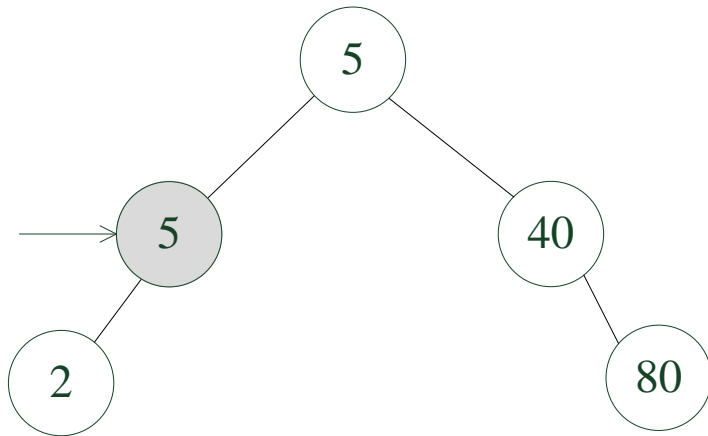
```
else { // tree is not empty
    // data to insert is less than data in current node
    if ( value < ( *treePtr )->data ) {
        insertNode( &( ( *treePtr )->leftPtr ), value );
    }
    // data to insert is greater than data in current node
    else if ( value > ( *treePtr )->data ) {
        insertNode( &( ( *treePtr )->rightPtr ), value );
    }
    else { // duplicate data value ignored
        printf( "%s", "dup" );
    }
}
}
```

# Binary search tree insert and search

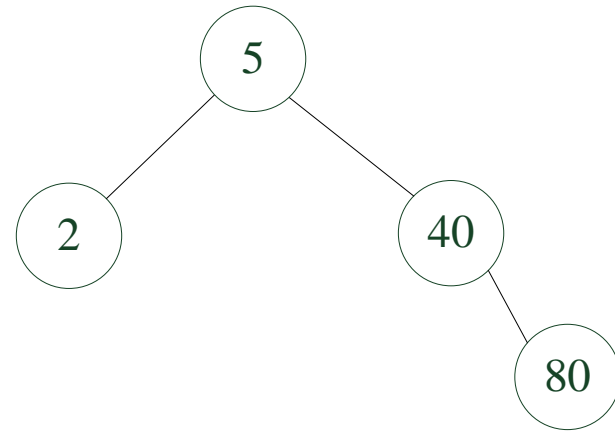
## □ 搜尋: searchNode()

```
TreeNodePtr searchNode(TreeNodePtr treePtr, int key){
    if(!treePtr){
        printf("Not in tree!!\n");
        return NULL;
    }
    else if(key == treePtr->data){ // key in tree
        printf("treePtr->data = %d\n", treePtr->data);
        return treePtr;
    }
    else if(key < treePtr->data) // search next key
        return searchNode(treePtr->leftPtr, key);
    return searchNode(treePtr->rightPtr, key);
}
```

# Deletion for a Binary Search Tree



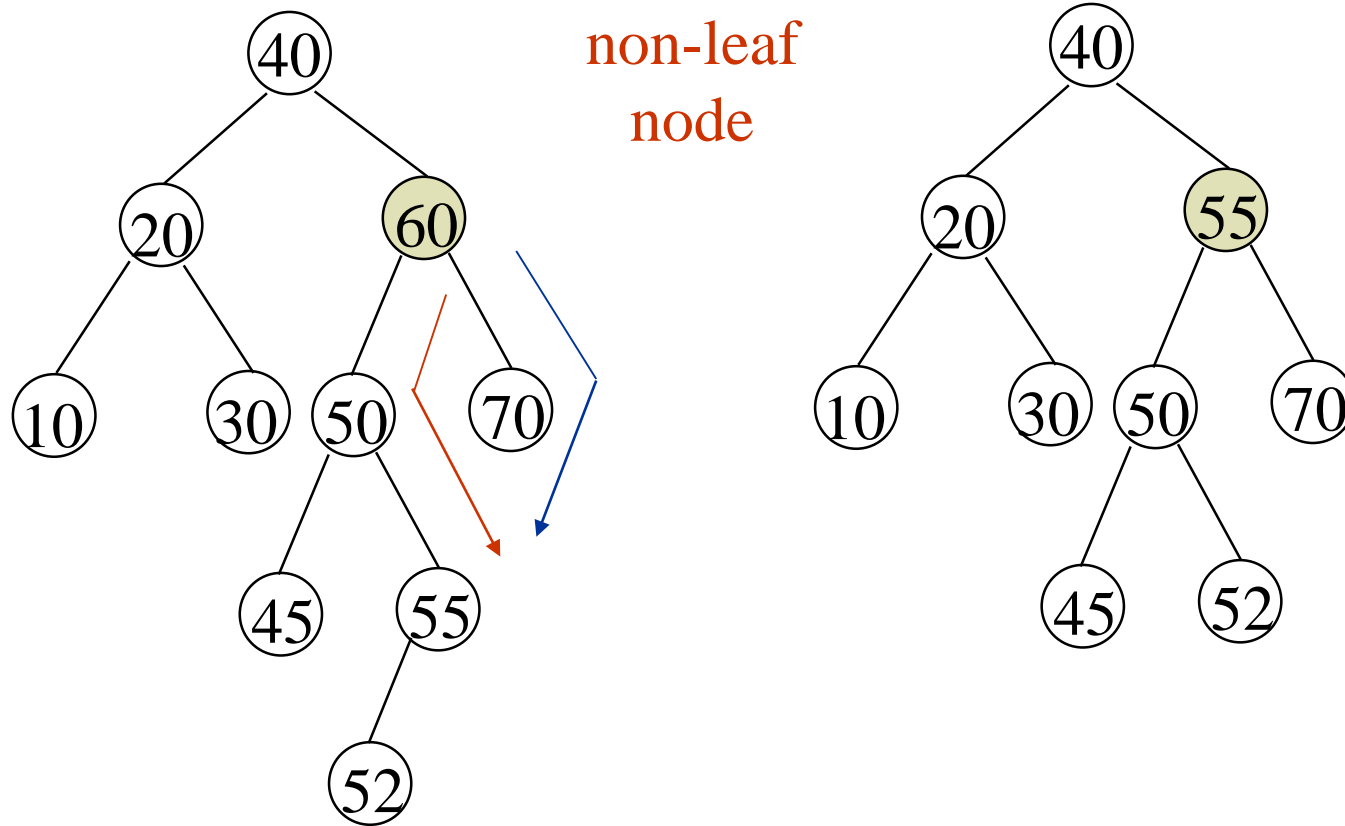
(a)



(b)



# Deletion for a Binary Search Tree



Before deleting 60

After deleting 60

In the left, to find the maximum

In the right, to find the minnum

# Split a Binary Search Tree

```
void split (nodePointer *theTree, int k, nodePointer *small,  
element *mid, nodePointer *big)
```

```
{ /* 根據鍵k來分割二元搜尋樹 */
```

```
    if (!theTree) { *small = *big = 0; (*mid).key = -1; return; }
```

```
    /* 空樹 */
```

```
    nodePointer sHead, bHead, s, b, currentNode;
```

```
    /* 替small和big建立標頭節點 */
```

```
    MALLOC(sHead, sizeof(*sHead));
```

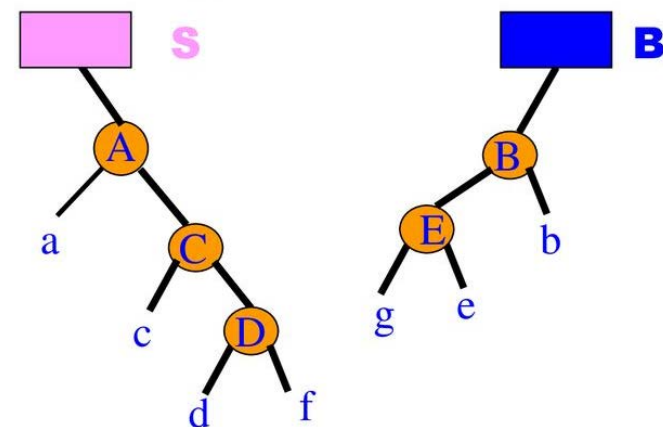
```
    MALLOC(bHead, sizeof(*bHead));
```

```
    s = sHead, b = bHead;
```

```
    /* 執行分割 */
```

```
    currentNode = *theTree;
```

```
    while (currentNode)
```



```

if (k < currentNode→data.key) { /* 加到big */
b→leftChild = currentNode; b = currentNode;
currentNode = currentNode→leftChild; }
else if (k > currentNode→data.key) { /* 加到 small */
s→rightChild = currentNode; s = currentNode;
currentNode = currentNode→rightChild; }

else { /* 在currentNode做分割 */
s→rightChild = currentNode→leftChild;
b→leftChild = currentNode→rightChild;
*small = sHead→rightChild; free(sHead);
*big = bHead→leftChild; free(bHead);
(*mid).item = currentNode→data.item;
(*mid).key = currentNode→data.key;
free(currentNode);
return; }

```

```
/* 沒有鍵為k的字典對 */  
s→rightChild = b→leftChild = 0;  
*small = sHead→rightChild; free(sHead);  
*big = bHead→leftChild; free(bHead);  
(*mid).key = -1;  
return;  
}
```

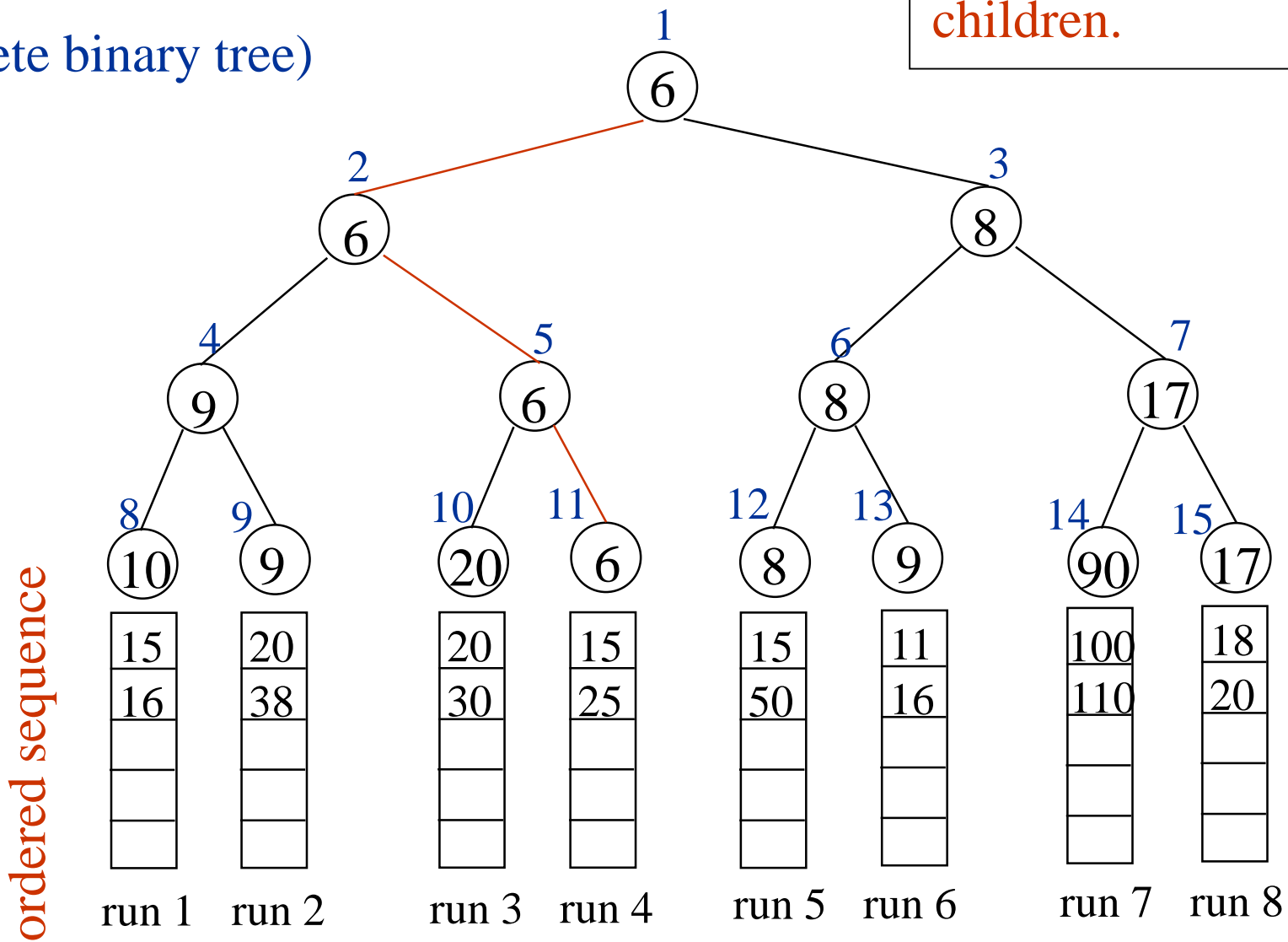
# Selection Trees

- (1) Winner tree
- (2) Loser tree

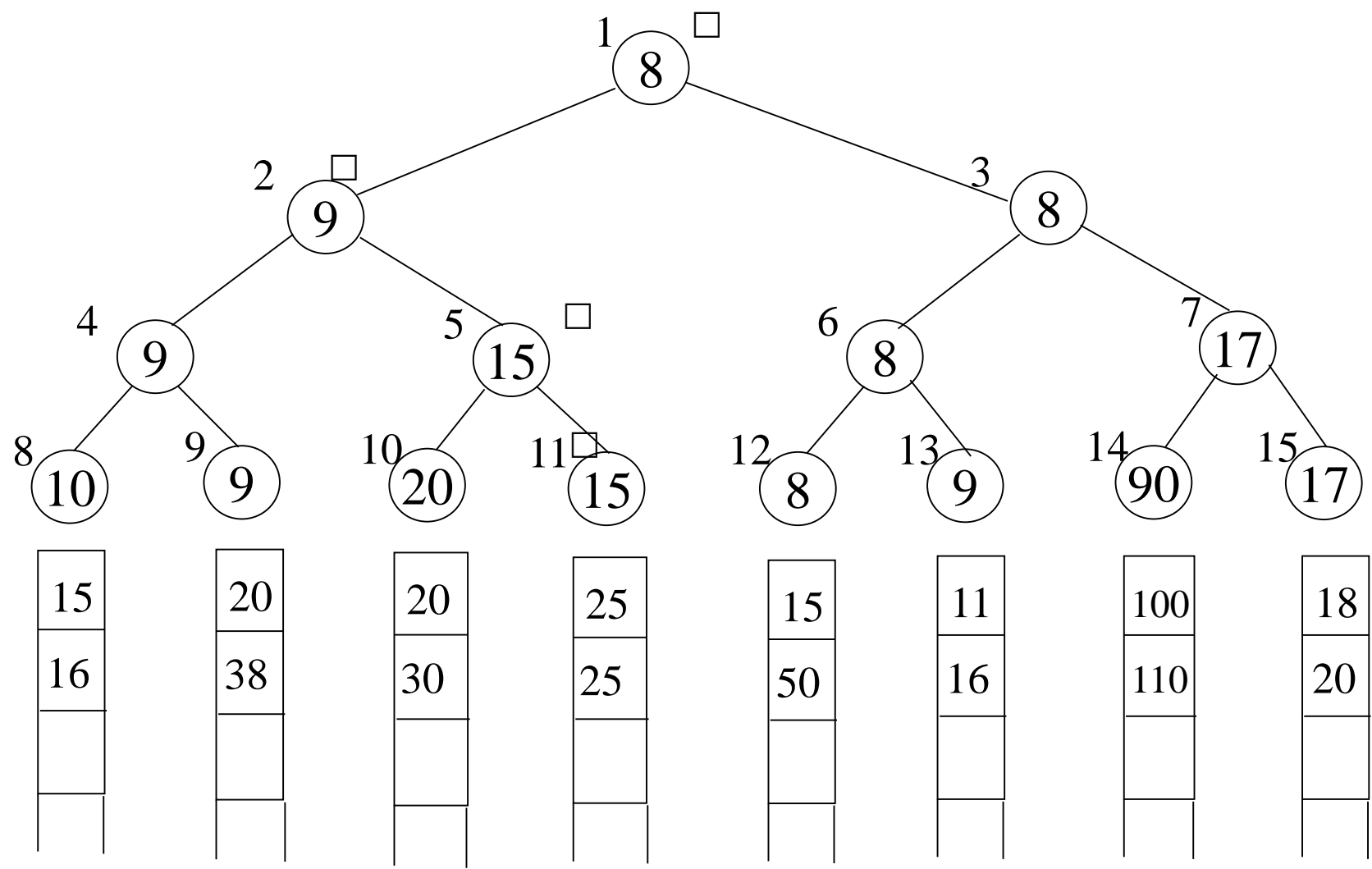
Sequential allocation scheme  
(complete binary tree)

# Winner tree

Each node represents the **smaller** of its two children.



**\*Figure 5.35:** Selection tree of Figure 5.34 after one record has been output and the tree restructured (nodes that were changed are ticked)

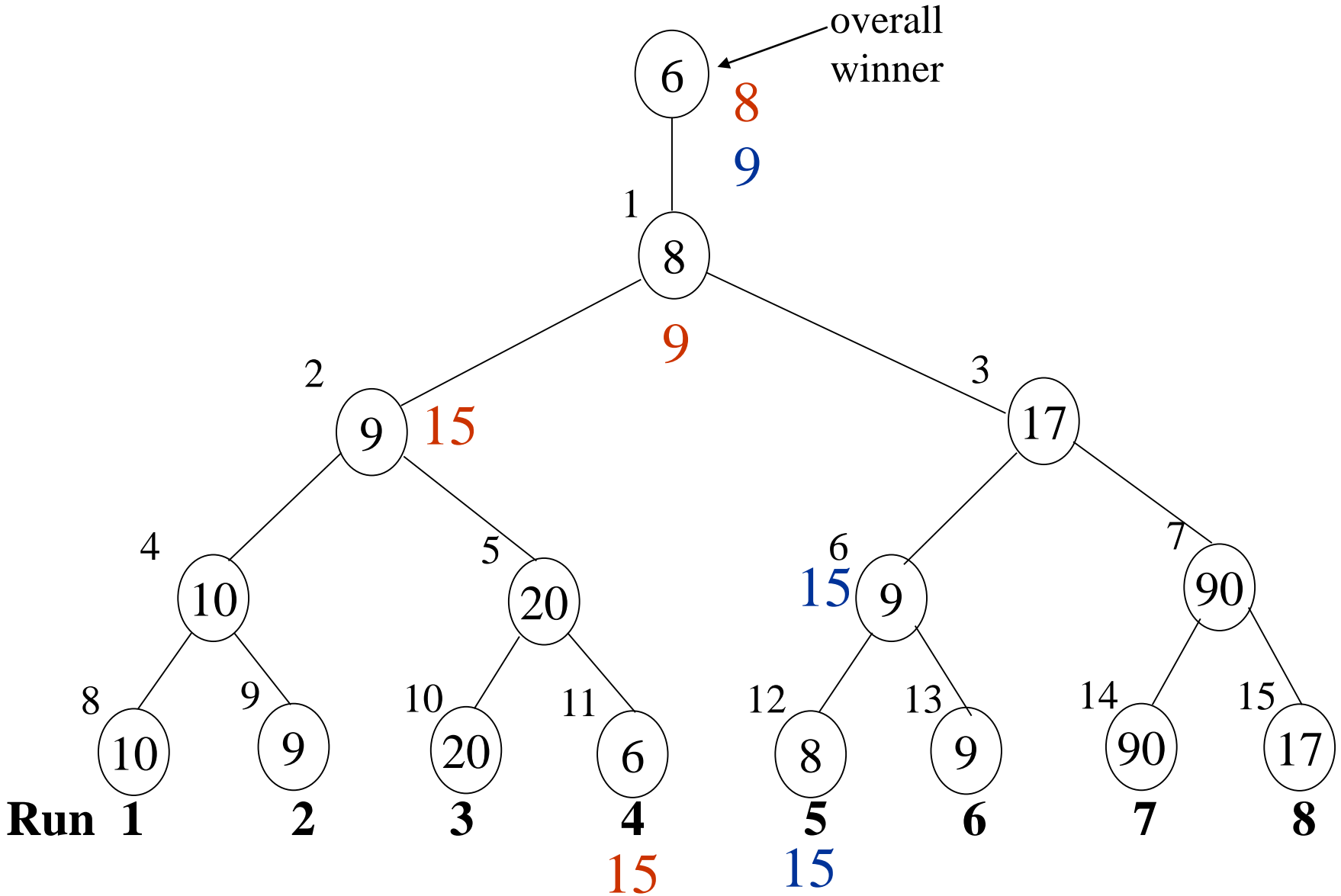


# Analysis

- $K$ : # of runs
- $n$ : # of records
- setup time:  $O(K)$   $(K-1)$
- restructure time:  $O(\log_2 K)$   $\lceil \log_2(K+1) \rceil$
- merge time:  $O(n \log_2 K)$
- slight modification: **loser tree**
  - consider the parent node only (vs. sibling nodes)

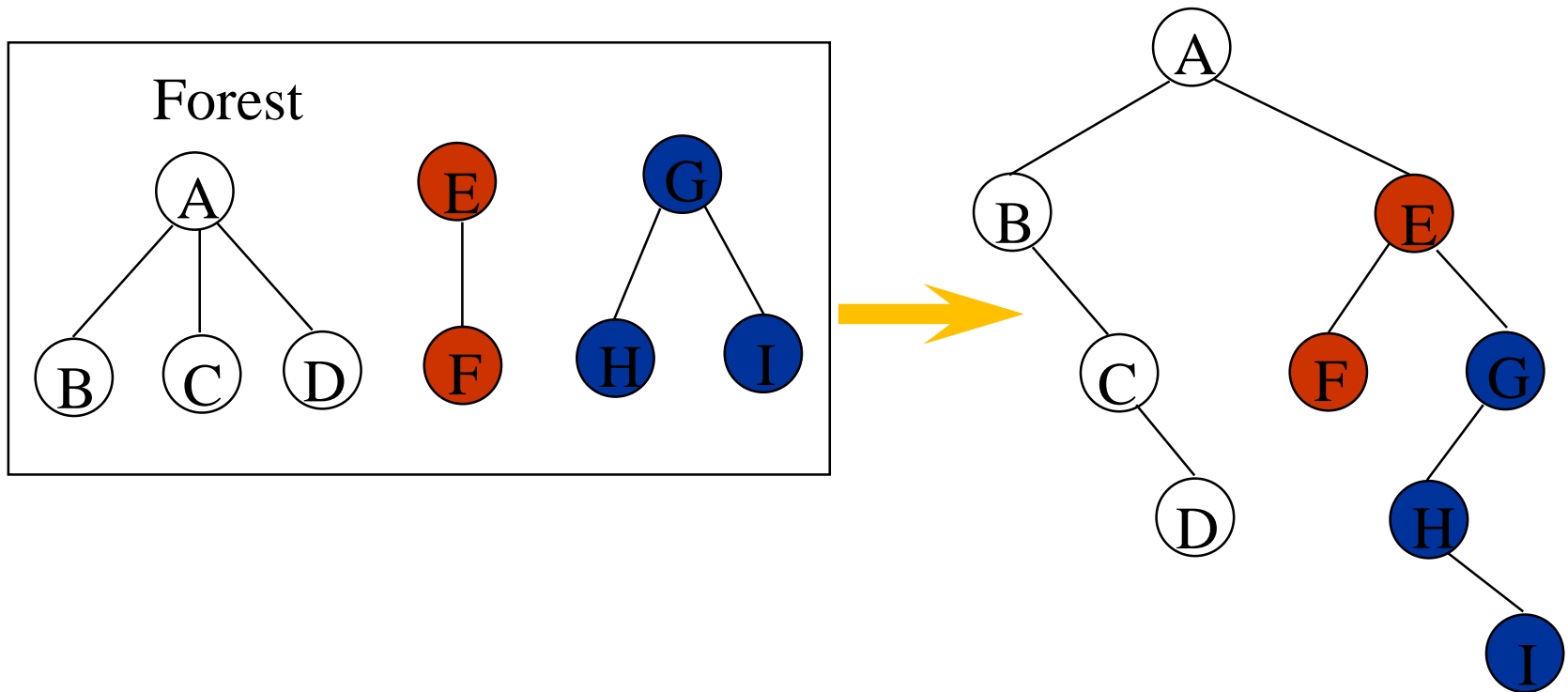


\*Figure 5.34: Tree of losers corresponding to Figure 5.32



# Forest

- Definition: **A forest is a set of  $n \geq 0$  disjoint trees**



# Transform a forest into a binary tree

- $T_1, T_2, \dots, T_n$ : a forest of trees
- $B(T_1, T_2, \dots, T_n)$ : a binary tree corresponding to this forest
- Algorithm
  - (1) empty, if  $n = 0$
  - (2) has root equal to  $\text{root}(T_1)$ 
    - has left subtree equal to  $B(T_{11}, T_{12}, \dots, T_{1m})$
    - has right subtree equal to  $B(T_2, T_3, \dots, T_n)$

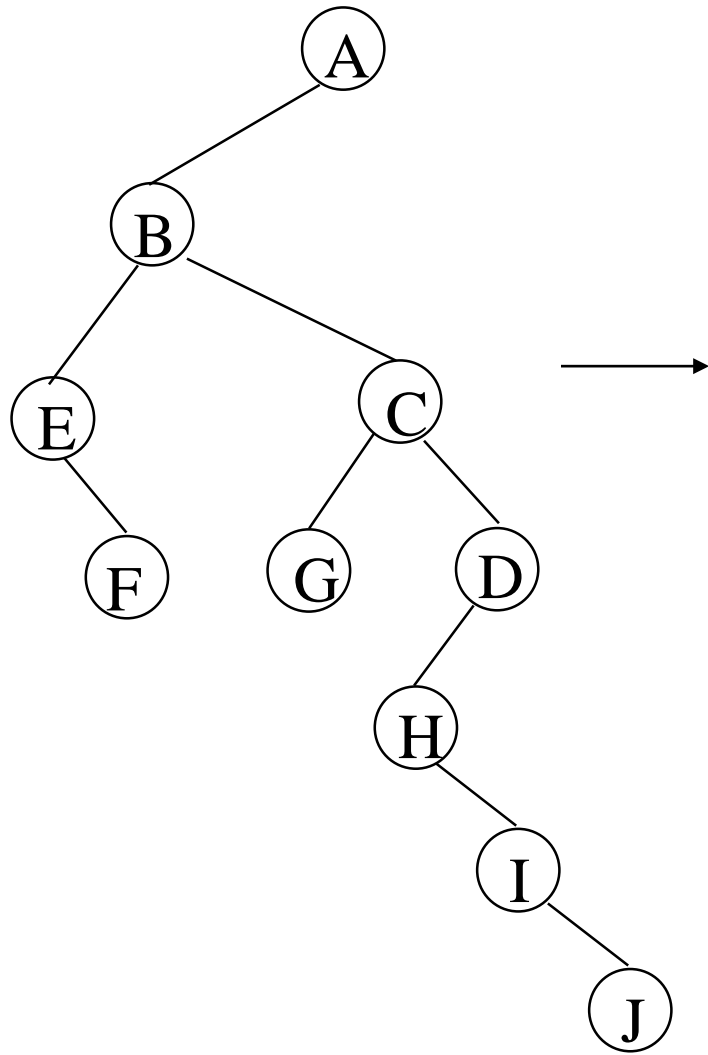
# Forest Traversals

## □ Preorder (VLR)

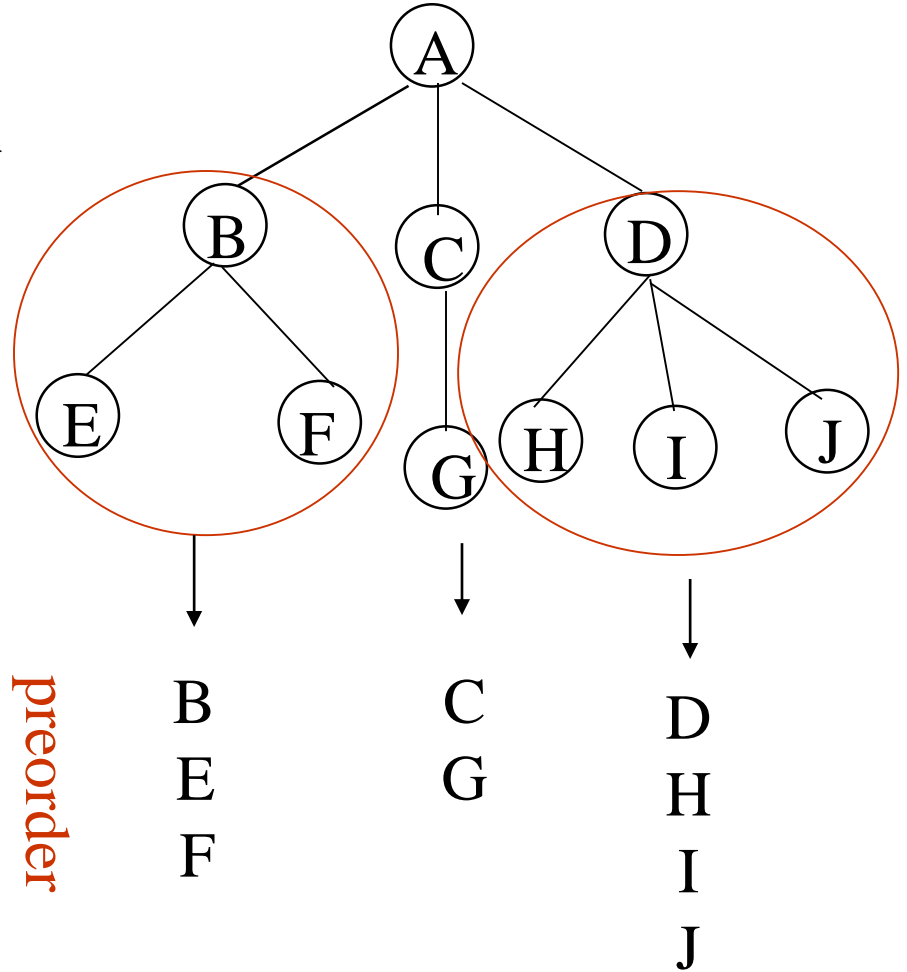
- If F is empty, then return
- Visit the root of the first tree of F
- Traverse the subtrees of the first tree in tree preorder
- Traverse the remaining trees of F in preorder

## □ Inorder (LVR)

- If F is empty, then return
- Traverse the subtrees of the first tree in tree inorder
- Visit the root of the first tree
- Traverse the remaining trees of F in inorder

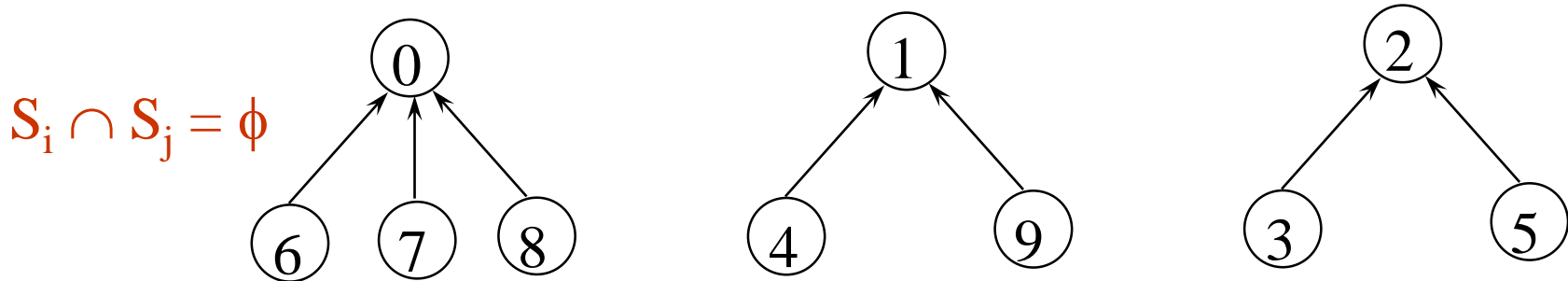


inorder: EFBGCHIJDA  
 preorder: ABEFCGDHIJ



# Set Representation

□  $S_1 = \{0, 6, 7, 8\}$ ,  $S_2 = \{1, 4, 9\}$ ,  $S_3 = \{2, 3, 5\}$



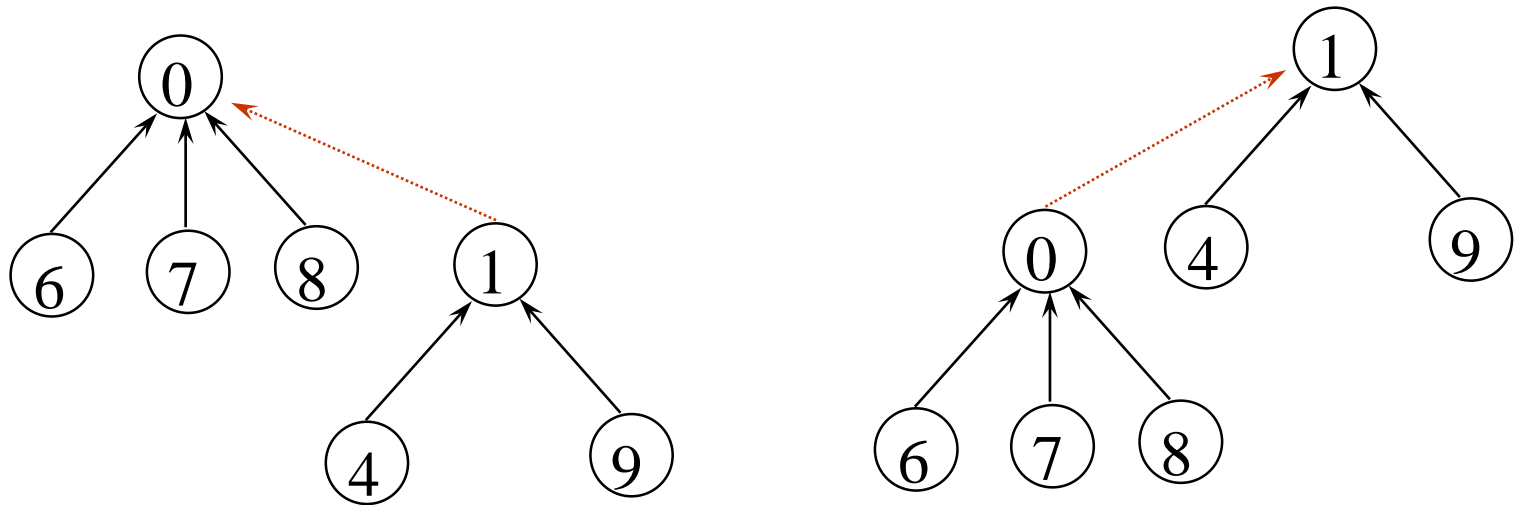
□ Two operations considered here

- *Disjoint set union*  $S_1 \cup S_2 = \{0, 6, 7, 8, 1, 4, 9\}$
- *Find(i)*: Find the set containing the element  $i$ .

$$3 \in S_3, 8 \in S_1$$

# Disjoint Set Union

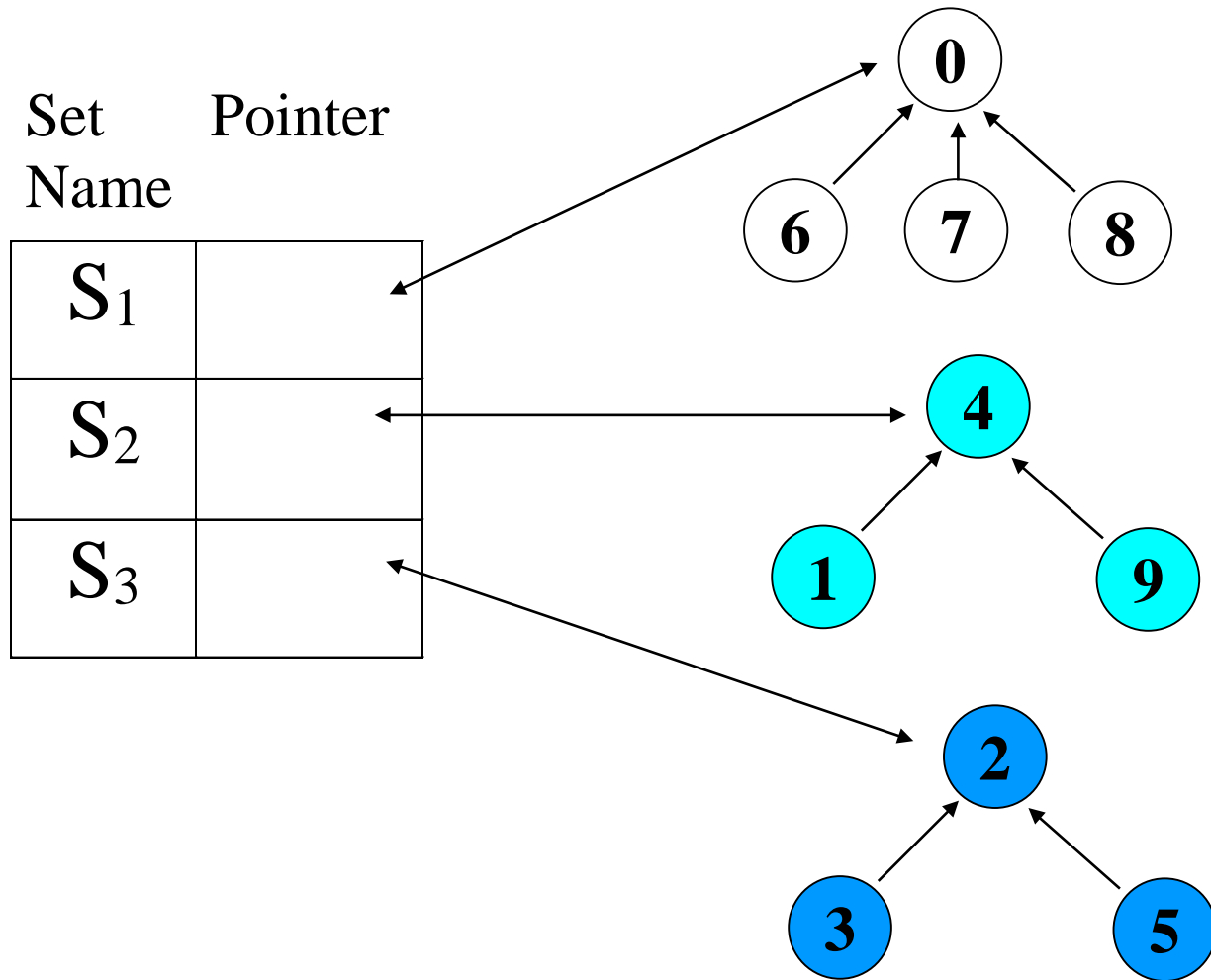
Make one of the trees a subtree of the other



Possible representation for  $S_1 \cup S_2$

$$S_1 \cup S_2$$

**\*Figure 5.39: Data Representation of  $S_1$ ,  $S_2$  and  $S_3$**





# Array Representation for Set

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

```
int simpleFind(int i)
{
    for (; parent[i] >= 0; i = parent[i]);
    return i;
}

void simpleUnion(int i, int j)
{
    parent[i] = j;
}
```

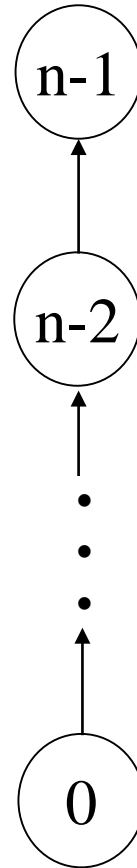
\*Figure 5.41: Degenerate tree (退化樹)

union operation

$O(n)$   $n-1$

find operation

$O(n^2)$   $\sum_{i=2}^n i$



union(0,1), find(0)

union(1,2), find(0)

.

.

.

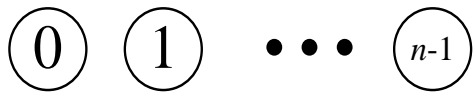
union(n-2,n-1), find(0)

degenerate tree

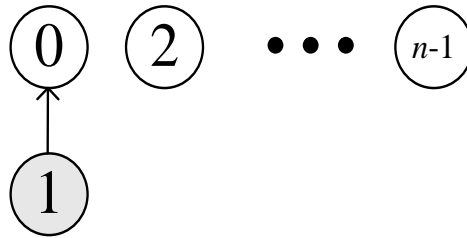
**\*Figure 5.42:** Trees obtained using the weighting rule

weighting rule for union(i,j):

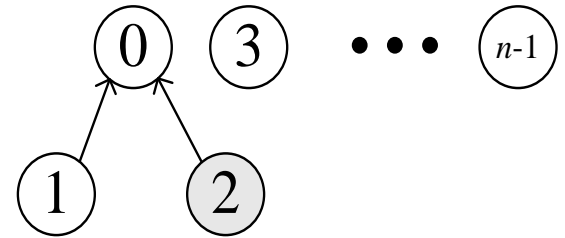
if # of nodes in i < # in j then make j the parent of i



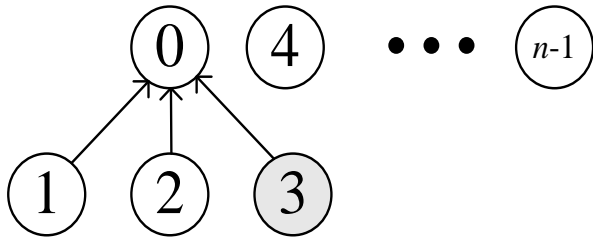
起始狀況



Union (0,1)

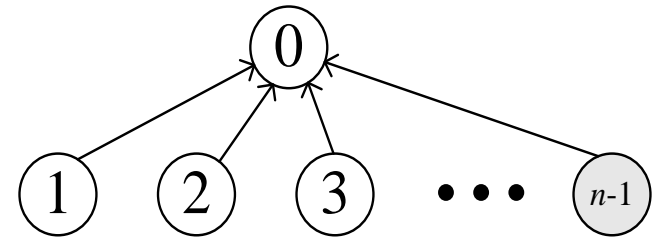


Union (0,2)



Union (0,3)

...

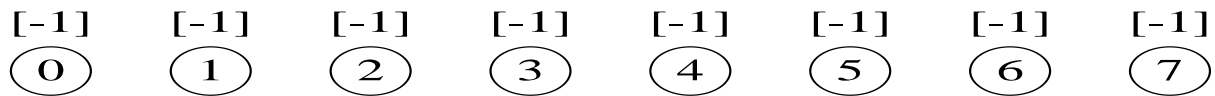


Union (0,n-1)

# Modified Union Operation

```
void weightedUnion(int i, int j)
{
    Keep a count in the root of tree
    //parent[i]=-count[i] and parent[j]=-count[j]
    int temp = parent[i]+ parent[j];
    if (parent[i]>parent[j]) {
        parent[i]=j;
        /* make j the new root*/
        parent[j]=temp;
    }
    else {
        parent[j]=i;
        /* make i the new root*/
        parent[i]=temp;
    }
}
```

If the number of nodes in tree i is less than the number in tree j, then make j the parent of i; otherwise make i the parent of j.



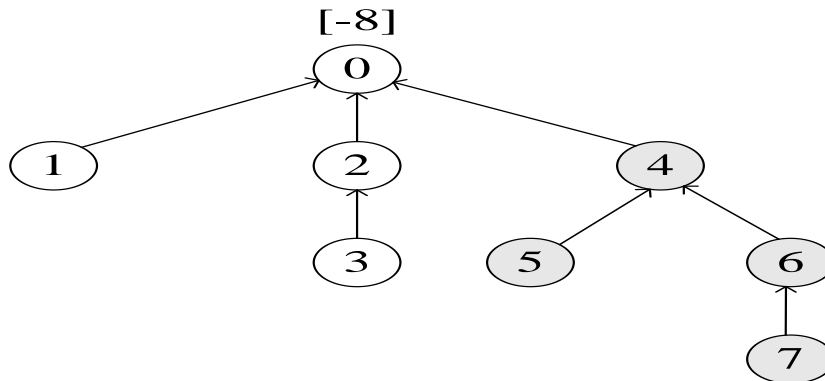
(a) 一開始樹的高度都是1



(b) 執行 *Union* (0,1) , (2,3) , (4,5) , 與 (6,7) 後樹之高度為 2



(c) 執行 *Union* (0,2) 與 (4,6) 後樹之高度為 3

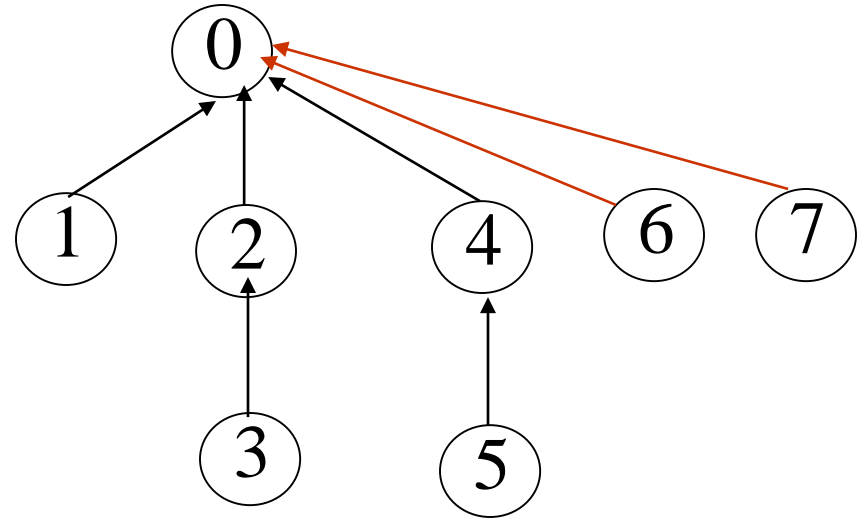
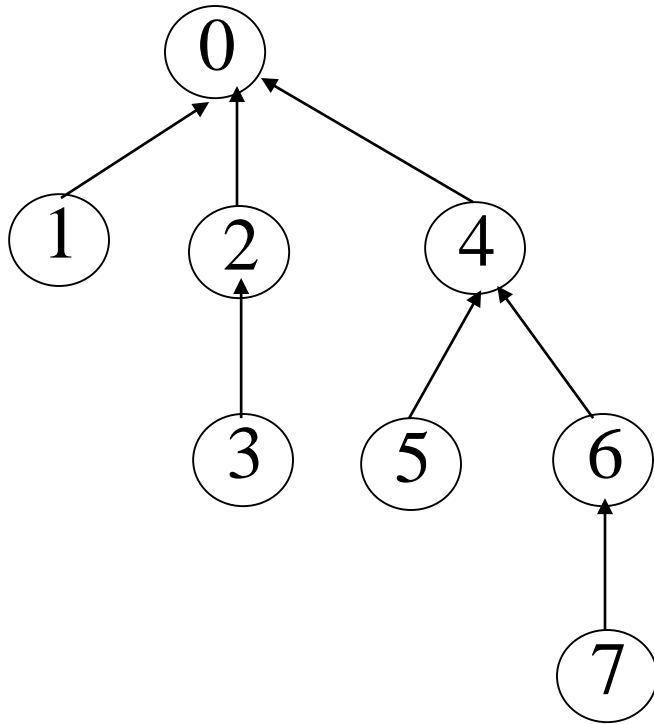


(d) 執行 *Union* (0,4) 後樹之高度為 4 **Figure 5.43: Trees ach**

# collapsingFind(i) Operation

```
int collapsingFind(int i)
{
    int root, trail, lead;
    for (root=i; parent[root]>=0;
         root=parent[root]);
    for (trail=i; trail!=root;
         trail=lead) {
        lead = parent[trail];
        parent[trail]= root;
    }
    return root;
}
```

If  $j$  is a node on the path from  $i$  to its root then make  $j$  a child of the root



find(7) find(7) find(7) find(7) find(7) find(7) find(7) find(7)

go up	3	1	1	1	1	1	1	1	1
reset	2								

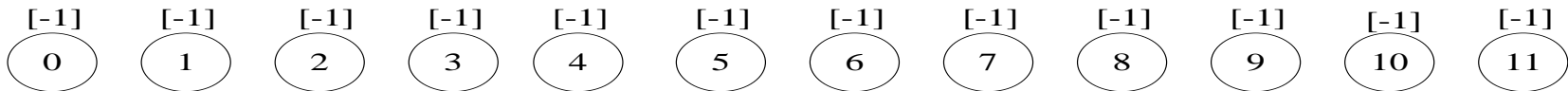
---

13 moves (vs. 24 moves)

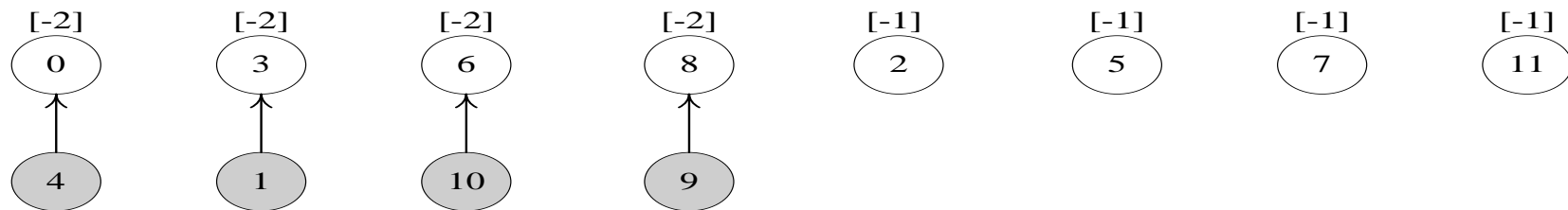
# Application to Equivalence Classes

- Find equivalence class  $i \equiv j$
- Find  $S_i$  and  $S_j$  such that  $i \in S_i$  and  $j \in S_j$   
(two finds)
  - $S_i = S_j$  **do nothing**
  - $S_i \neq S_j$  **union( $S_i$ ,  $S_j$ )**
- example
  - $0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4, 6 \equiv 8,$
  - $3 \equiv 5, 2 \equiv 11, 11 \equiv 0$
  - $\{0, 2, 4, 7, 11\}, \{1, 3, 5\}, \{6, 8, 9, 10\}$

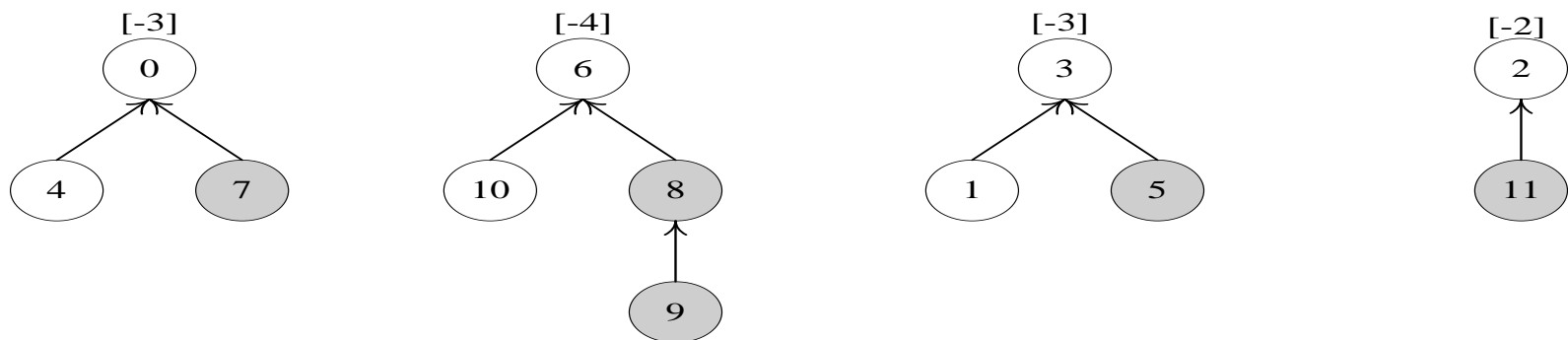




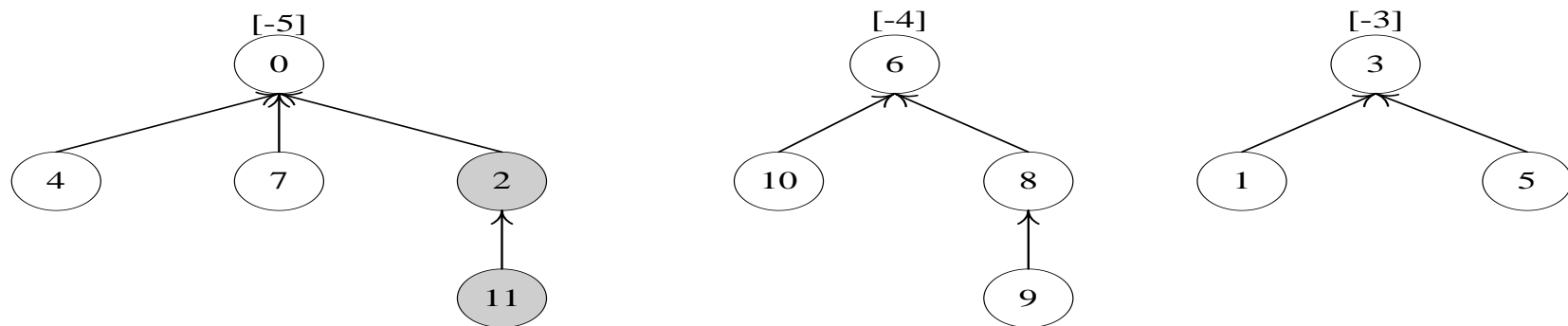
(a) 起始樹



(b) 處理完  $0 \equiv 4$ ,  $3 \equiv 1$ ,  $6 \equiv 10$ ,  $8 \equiv 9$  後高度為 2 的樹



(c) 處理完  $7 \equiv 4$ ,  $6 \equiv 8$ ,  $3 \equiv 5$ ,  $2 \equiv 11$  後的樹



(d) 處理完  $11 \equiv 0$  後的樹

preorder: A B C D E F G H I  
inorder: B C A E D G H F I

