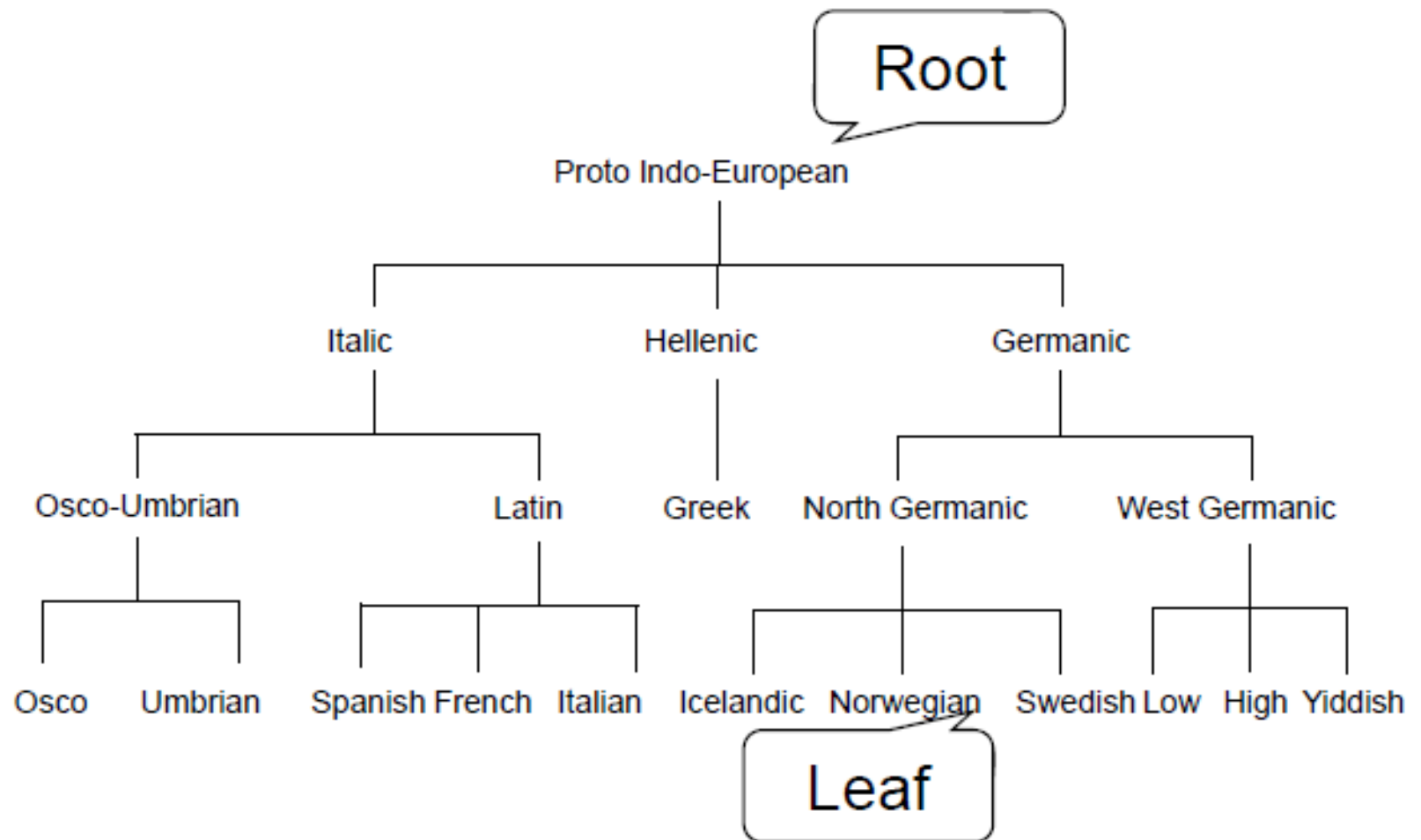




DATA STRUCTURE

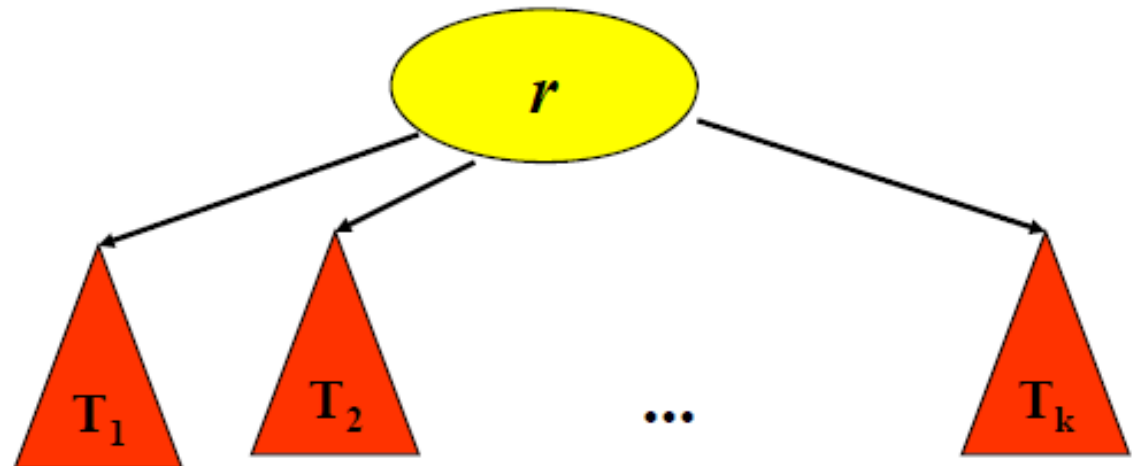
Chapter 05: Trees

TREES



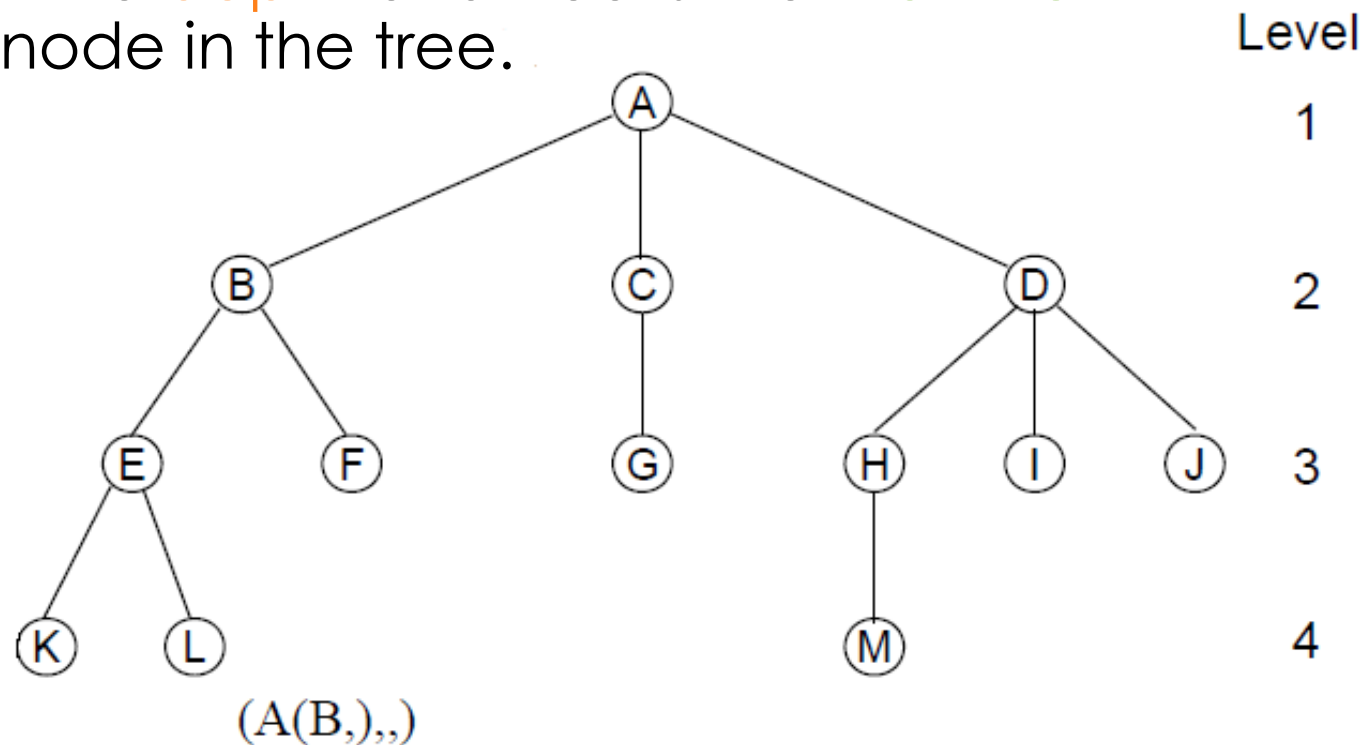
TREES (CONT'D)

- Tree: a finite set of one or more nodes such that
 - a distinguished node r (root)
 - zero or more nonempty (sub)trees T_1, T_2, \dots, T_k each of whose roots are connected by a directed edge from r



A SAMPLE TREE

- Assume the root is at level 1, then the level of a node is the level of the node's parent plus one.
- The **height** or the **depth** of a tree is the **maximum level** of any node in the tree.





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 roots of the subtrees.
- The roots of these subtrees are the *children* of the node.
- Children of the same parent are *siblings*.
- The ancestors of a node are all the nodes long the path from the root to the node.

REPRESENTATION OF TREES (1)

- Parenthesis notation

- The root comes first, followed by a list of sub-trees
- $T = (\text{root } (T_1, T_2, \dots, T_n))$

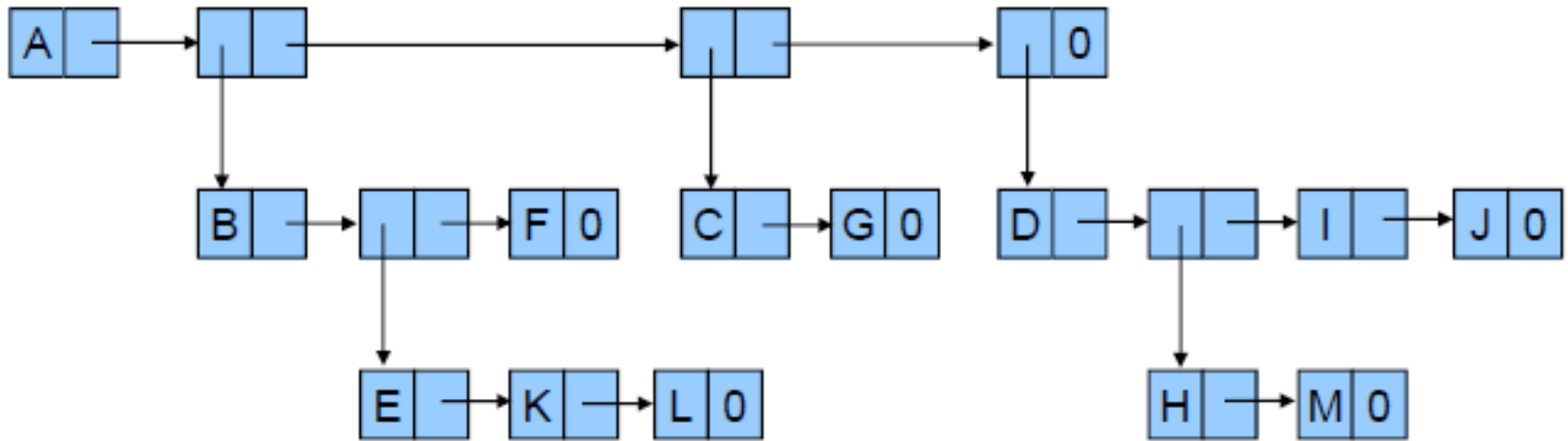
$(\underline{A} (\underline{B} (\underline{E} (K, L), F), \underline{C} (G), \underline{D} (H (M), I, J)))$

e.g. Draw the following tree:

$(A (B (E, F), C (G), D (H, I, J)))$

REPRESENTATION OF TREES (2)

- List representation



REPRESENTATION OF TREES (2)

(CONT'D)

- Possible node Structure for a tree of degree k

Lemma 5.1:

If T is a k -ary tree (i.e., a tree of degree k) with n nodes, each having a fixed size as in Figure 5.4, then $n(k-1) + 1$ of the nk child fields are 0, $n \geq 1$.

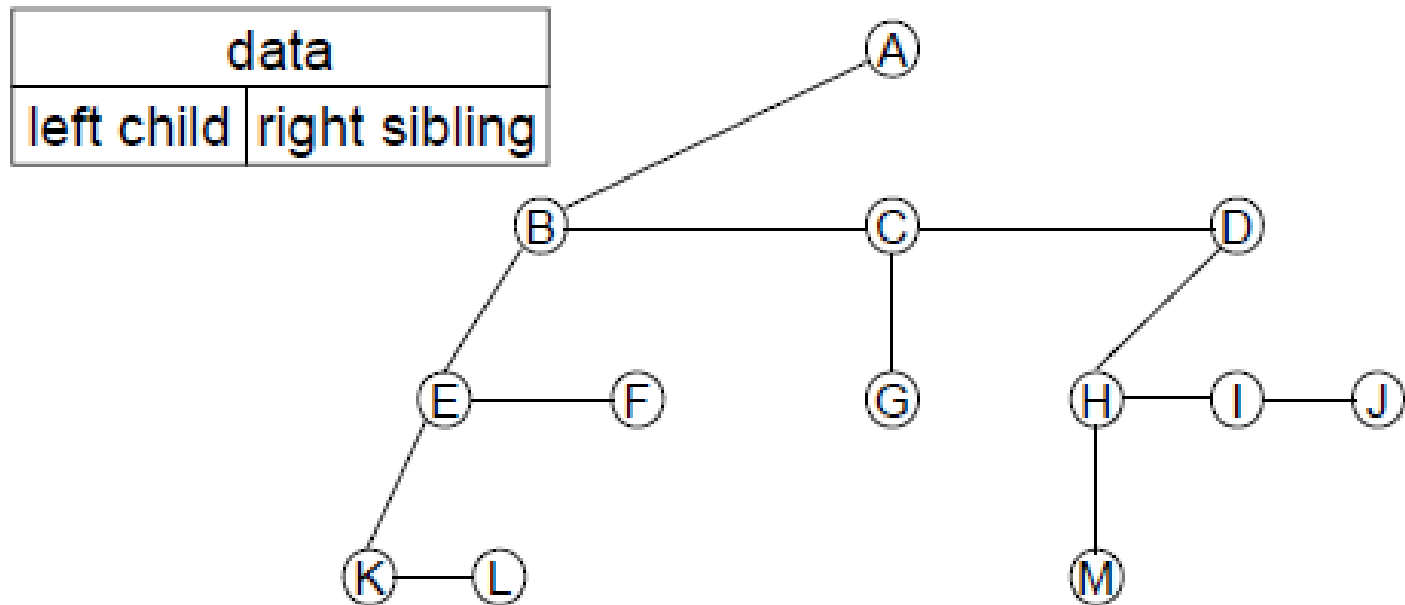
$$nk - (n-1) = n(k-1) + 1$$

Data	Child 1	Child 2	Child 3	Child 4	...	Child k
------	---------	---------	---------	---------	-----	-----------

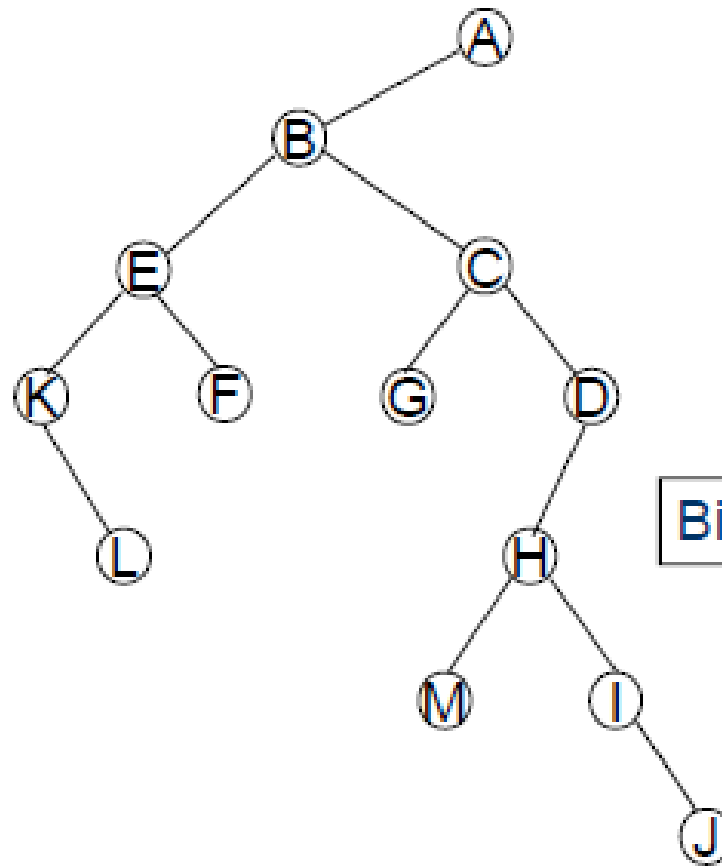
Wasting memory!

REPRESENTATION OF TREES (3)

- Left Child-Right Sibling Representation
 - Each node has two links (or pointers).
 - Each node only has one leftmost child and one closest sibling.



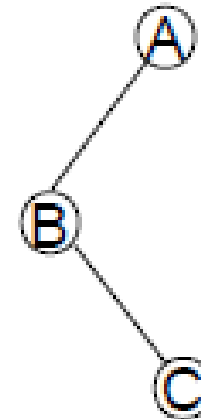
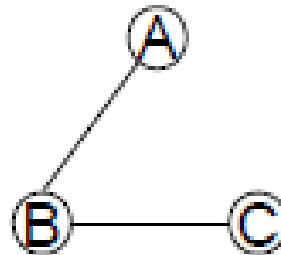
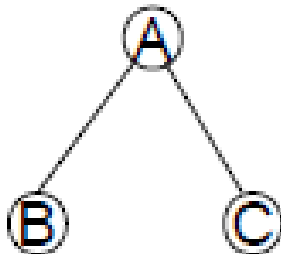
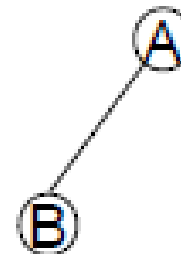
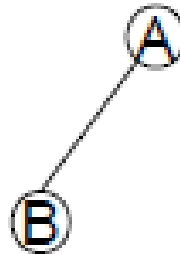
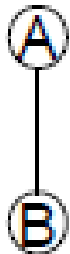
REPRESENTATION OF TREES (4)



Rotate the right-sibling pointers in a left child-right sibling tree clockwise by 45 degrees

Binary Tree!

TREE REPRESENTATIONS



Left child-right sibling

Binary tree



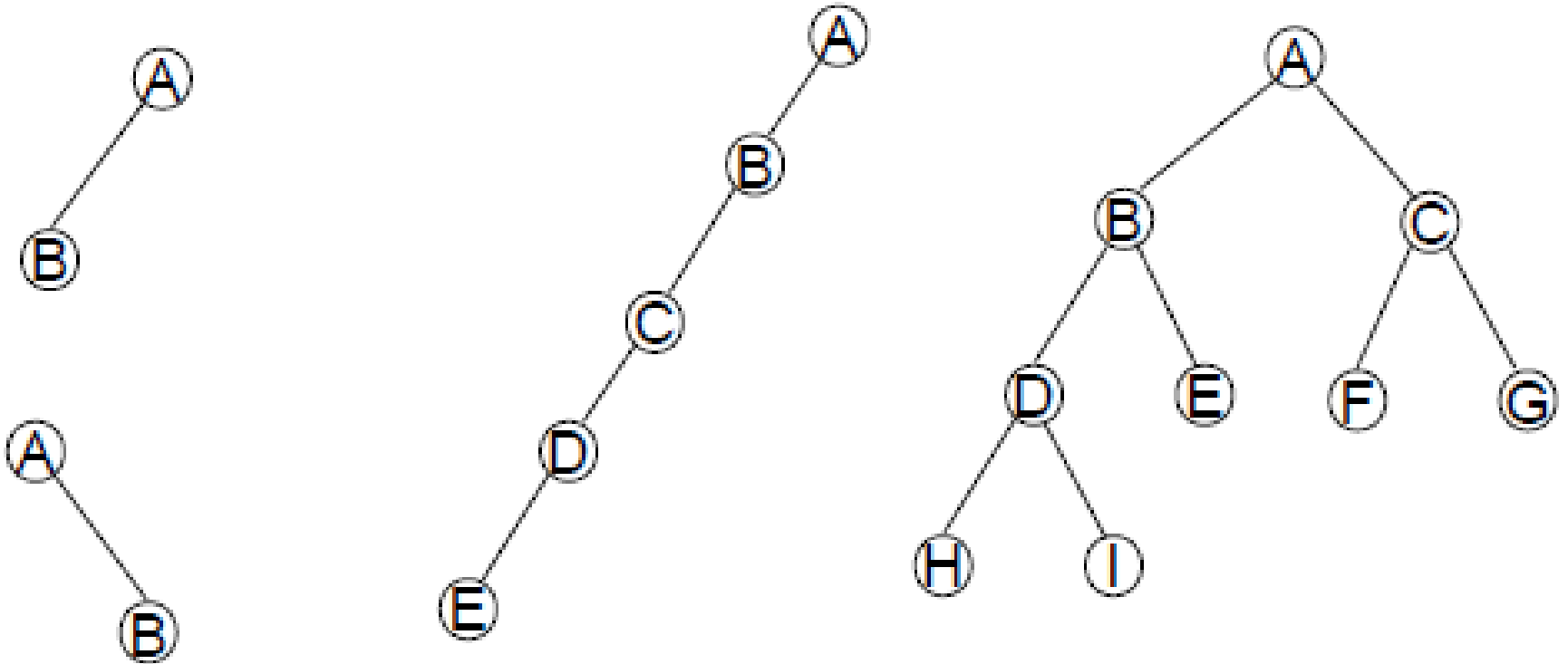
BINARY TREE

- Definition:
 - 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.
- There is no tree with zero nodes. But there is an empty binary tree.
- Binary tree distinguishes between the **order** of the children while in a tree we do not.

DISTINCTIONS BETWEEN A BINARY TREE AND A TREE

	Binary Tree	Tree
Node Degree	≤ 2	Not limited
Order of the subtrees	✓	✗
Allow zero nodes	✓	✗

BINARY TREE EXAMPLES



Skewed binary tree

Complete binary tree

THE PROPERTIES OF BINARY TREES

Lemma 5.2 [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 $2^k - 1$, $k \geq 1$.

Lemma 5.3 [Relation between number of leaf nodes and nodes of degree 2]:

For any non-empty binary tree, T , if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Definition:

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

MAXIMUM NUMBER OF NODES IN BINARY TREES

- The maximum number of nodes on level i of a binary tree is 2^{i-1} , $i \geq 1$.
Prove by induction.

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

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

RELATION BETWEEN NUMBER OF LEAF NODES AND NODES OF DEGREE 2

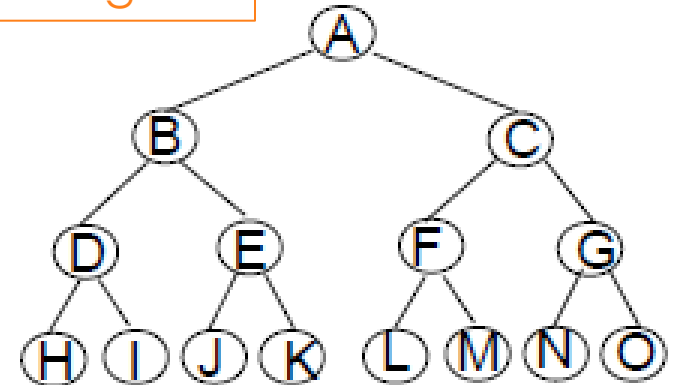
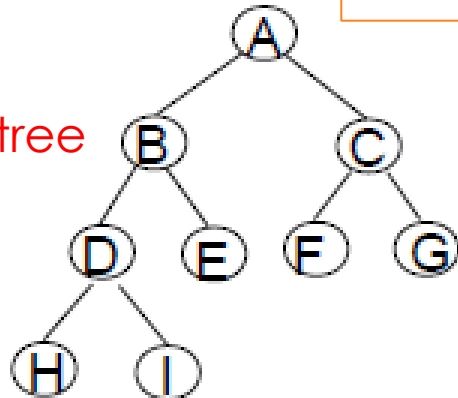
- For any nonempty binary tree, T , if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$
- proof:
 - Let n and B denote the total number of nodes & branches in T .
 - Let n_0, n_1, n_2 represent the nodes with no children, single child, and two children respectively.
 - $n = n_0 + n_1 + n_2$, $B + 1 = n$, $B = n_1 + 2n_2 \Rightarrow n_1 + 2n_2 + 1 = n$,
 - $n_1 + 2n_2 + 1 = n_0 + n_1 + n_2 \Rightarrow n_0 = n_2 + 1$

FULL BT VERSUS 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 .

Numbering from top to bottom, to left to right

Complete binary tree



Full binary tree of depth 4

ARRAY REPRESENTATION OF A BINARY TREE

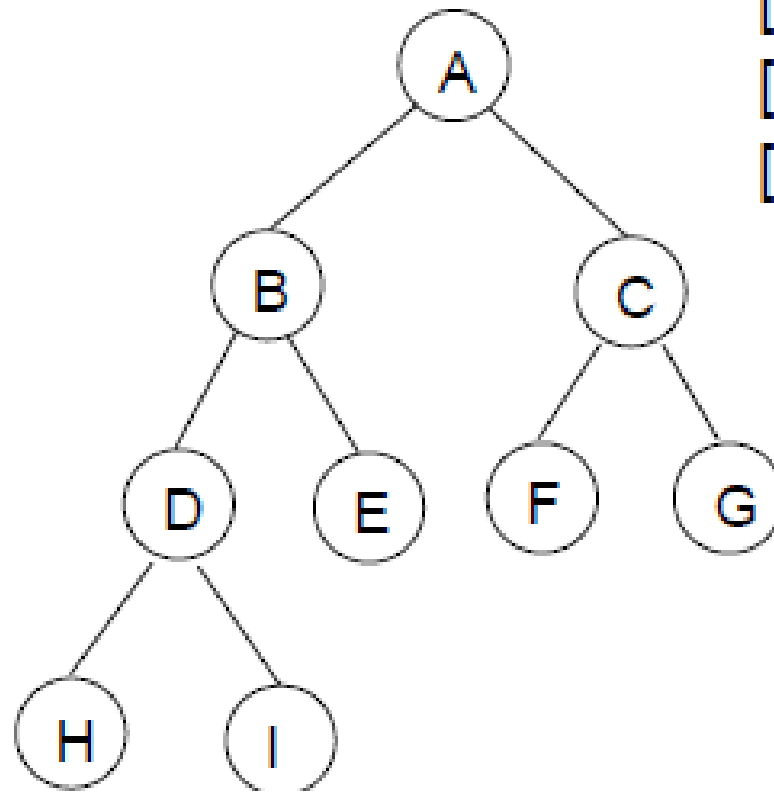
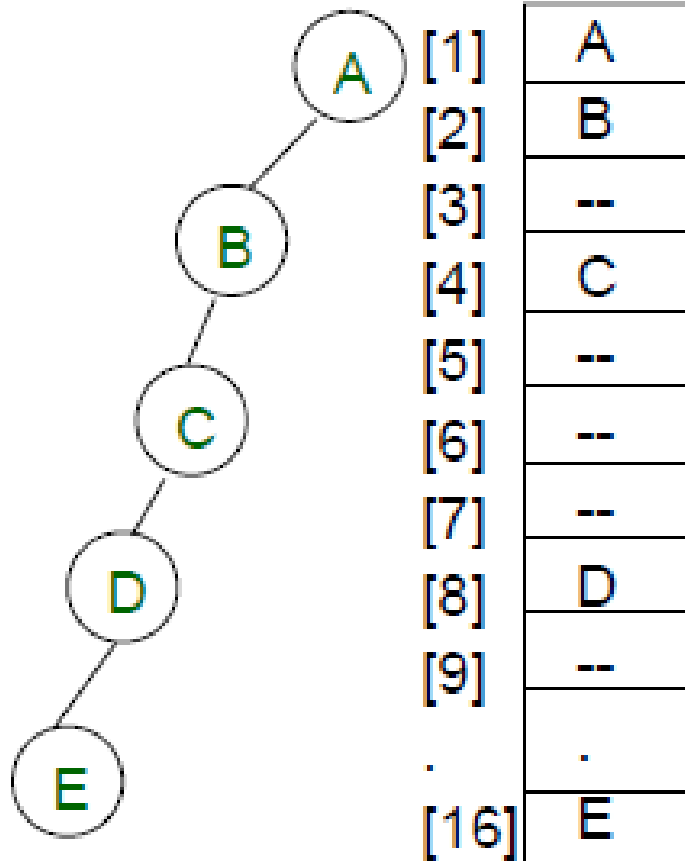
Lemma 5.4:

If a complete binary tree with n nodes is represented sequentially, then for any node with index i , $1 \leq i \leq n$, we have:

- $\text{parent}(i)$ is at $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i = 1$, i is at the root and has no parent.
 - $\text{left_child}(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, then i has no left child.
 - $\text{right_child}(i)$ is at $2i + 1$ if $2i + 1 \leq n$. If $2i + 1 > n$, then i has no right child.
- Position zero of the array is not used.

SEQUENTIAL REPRESENT,

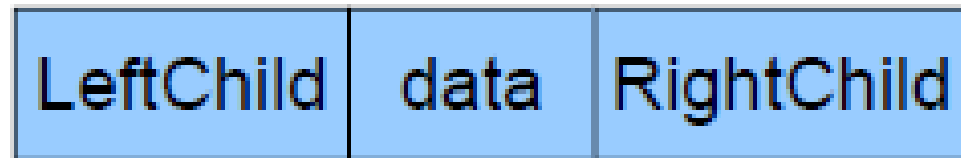
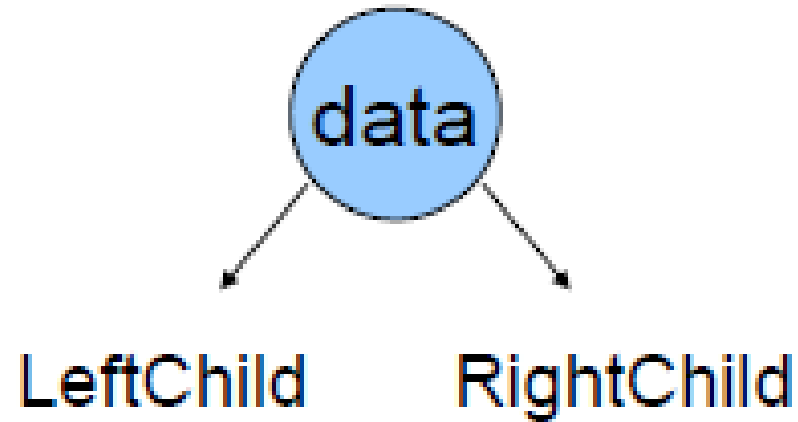
- (1) Waste space
- (2) Insertion / deletion problem



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

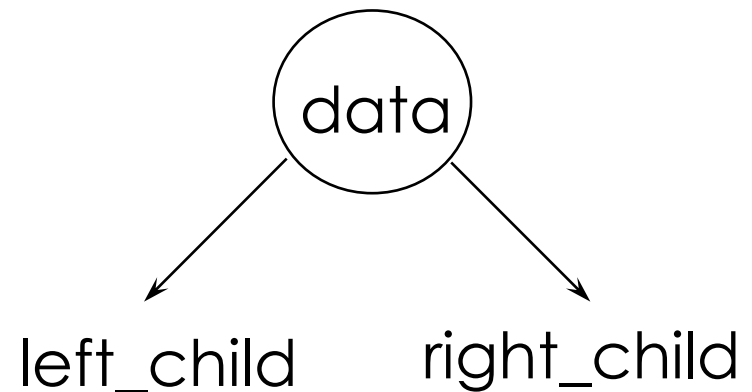
A
B
C
D
E
F
G
H
I

NODE REPRESENTATION



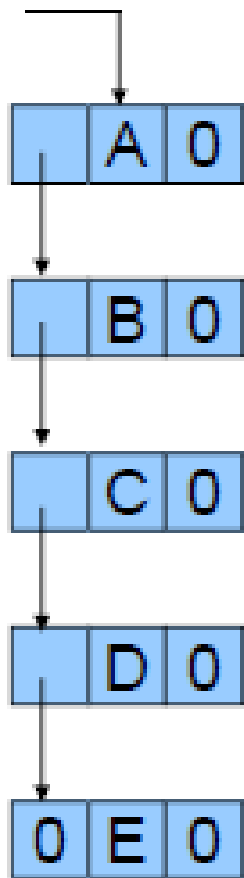
LINKED LIST REPRESENTATION FOR THE BINARY TREES

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

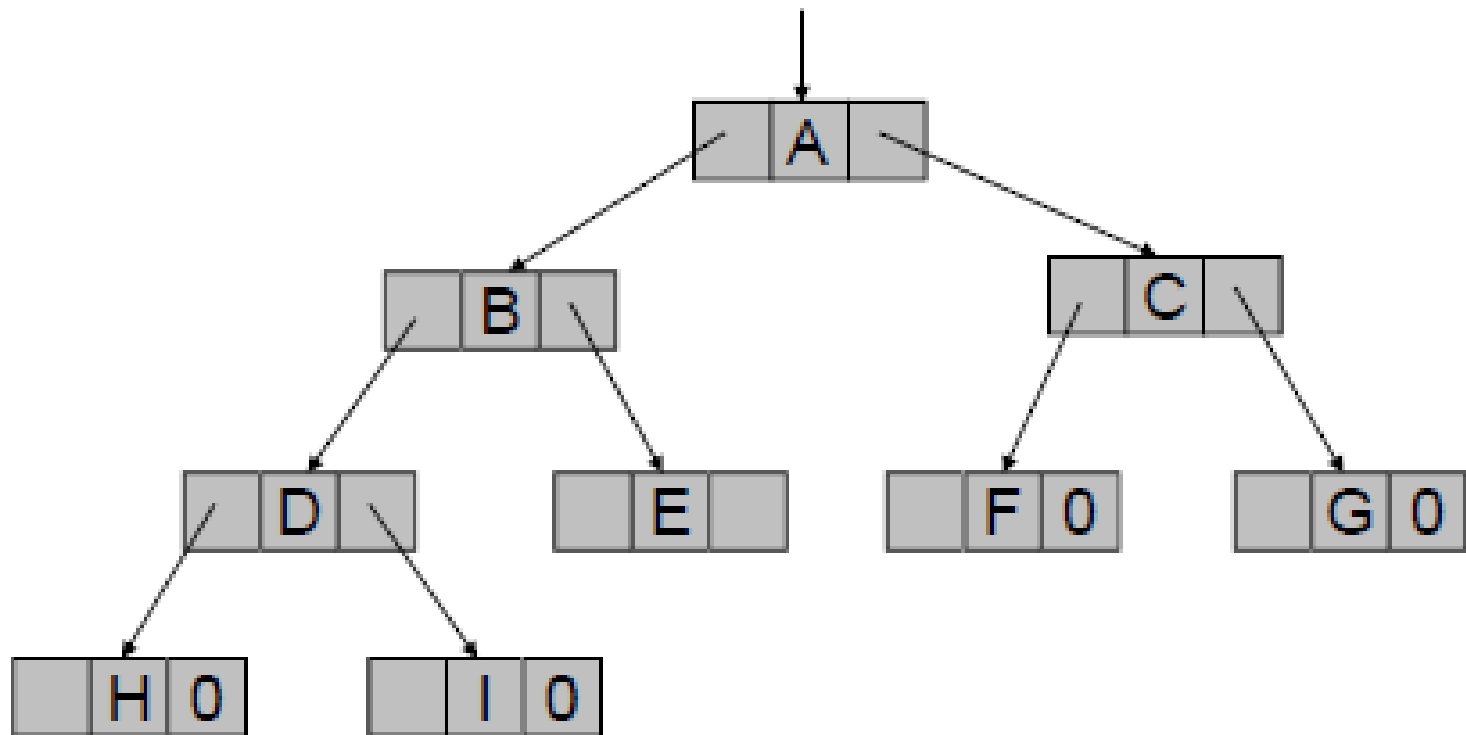


LINKED LIST REPRESENTATION FOR THE BINARY TREES (CONT'D)

root



root



COMPARE TWO BINARY TREE REPRESENTATIONS

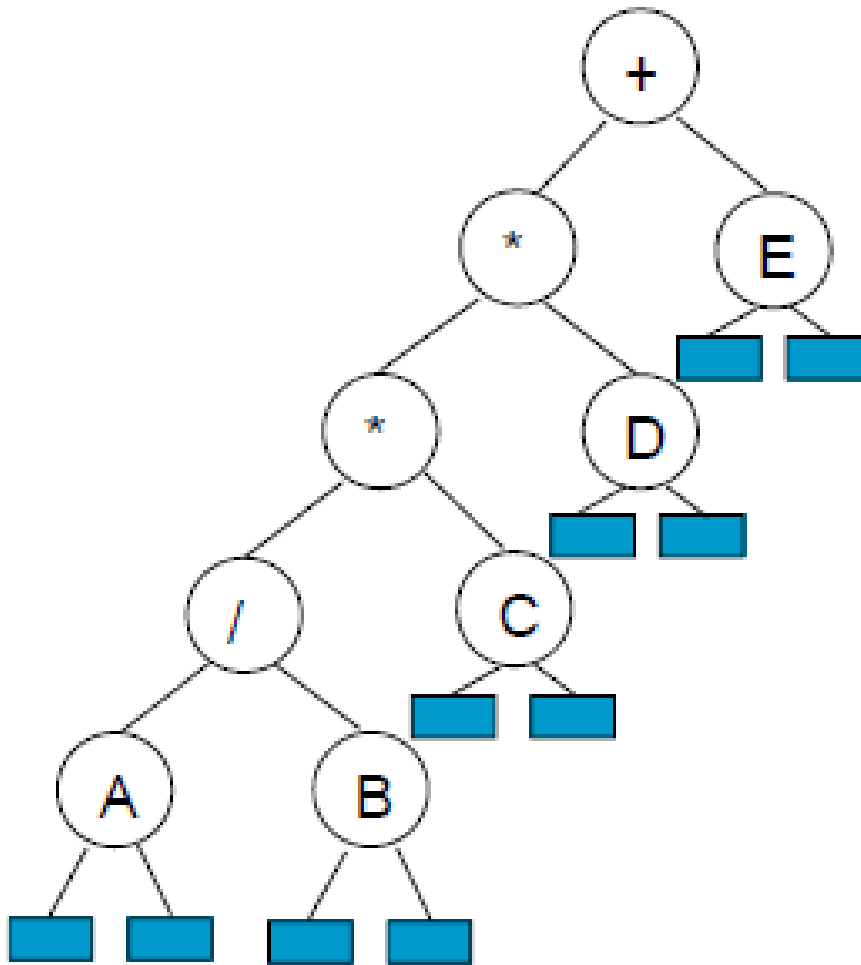
	Array representation	Linked list representation
Determination the locations of the parent, left child and right child	Easy	Difficult
Space overhead	Much	Little
Insertion and deletion	Difficult	Easy



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

$A / B * C * D + E$

preorder traversal

$+ * * / A B C D E$

postorder traversal

$A B / C * D * E +$

level order traversal

$+ * E * D / C A B$

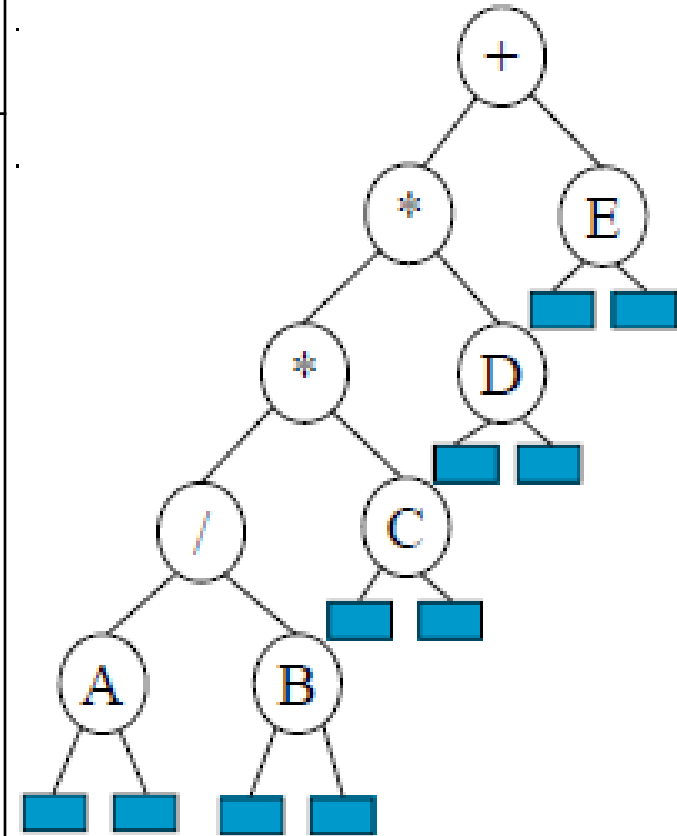
INORDER TRAVERSAL (RECURSIVE VERSION)

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

$A / B * C * D + E$

TREE 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	Cout
4	/		13	NULL	
5	A		2	*	Cout
6	NULL		14	D	
5	A	Cout	15	NULL	
7	NULL		14	D	Cout
4	/	Cout	16	NULL	
8	B		1	+	Cout
9	NULL		17	E	
8	B	Cout	18	NULL	
10	NULL		17	E	Cout
3	*	Cout	19	NULL	



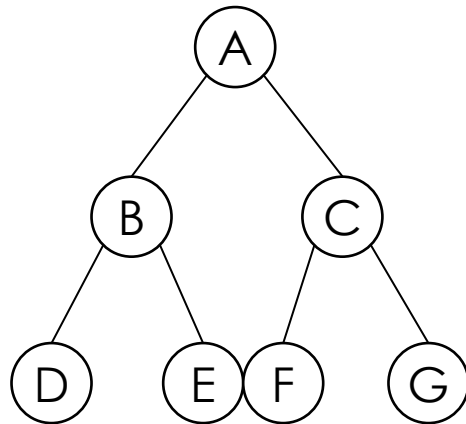
PREORDER TRAVERSAL (RECURSIVE VERSION)

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

POSTORDER TRAVERSAL (RECURSIVE VERSION)

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

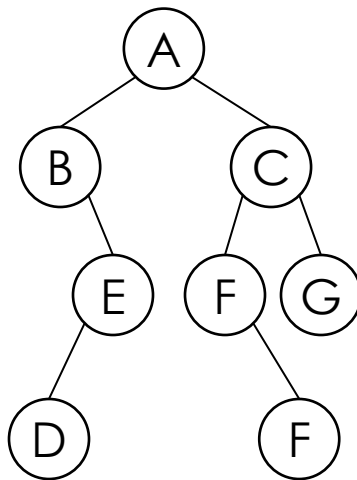
EXERCISE



Preorder: _____

Inorder: _____

Postorder: _____



Preorder: _____

Inorder: _____

Postorder: _____

ITERATIVE INORDER TRAVERSAL

```
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;
    }
}
```

$O(n)$



LEVEL-ORDER TRAVERSAL

- All previous mentioned schemes use stacks
- Level-order traversal uses a queue
- Level-order scheme visit the root first, then the root's left child, followed by the root's right child
- All the nodes at a level are visited before moving down to another level

LEVEL-ORDER TRAVERSAL OF A BINARY TREE

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

+*ED/CAB



SOME OTHER BINARY TREE FUNCTIONS

- With the inorder, postorder, or preorder mechanisms, we can implement all needed binary tree functions. e.g.,
 - Copying Binary Trees
 - Testing Equality
 - Two binary trees are equal if their topologies are the same and the information in corresponding nodes is identical.

PROGRAM 5.9: COPYING A B.T.

```
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;
}
```

PROGRAM 5.10: B.T. EQUIVALENCE

```
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)))
}
```

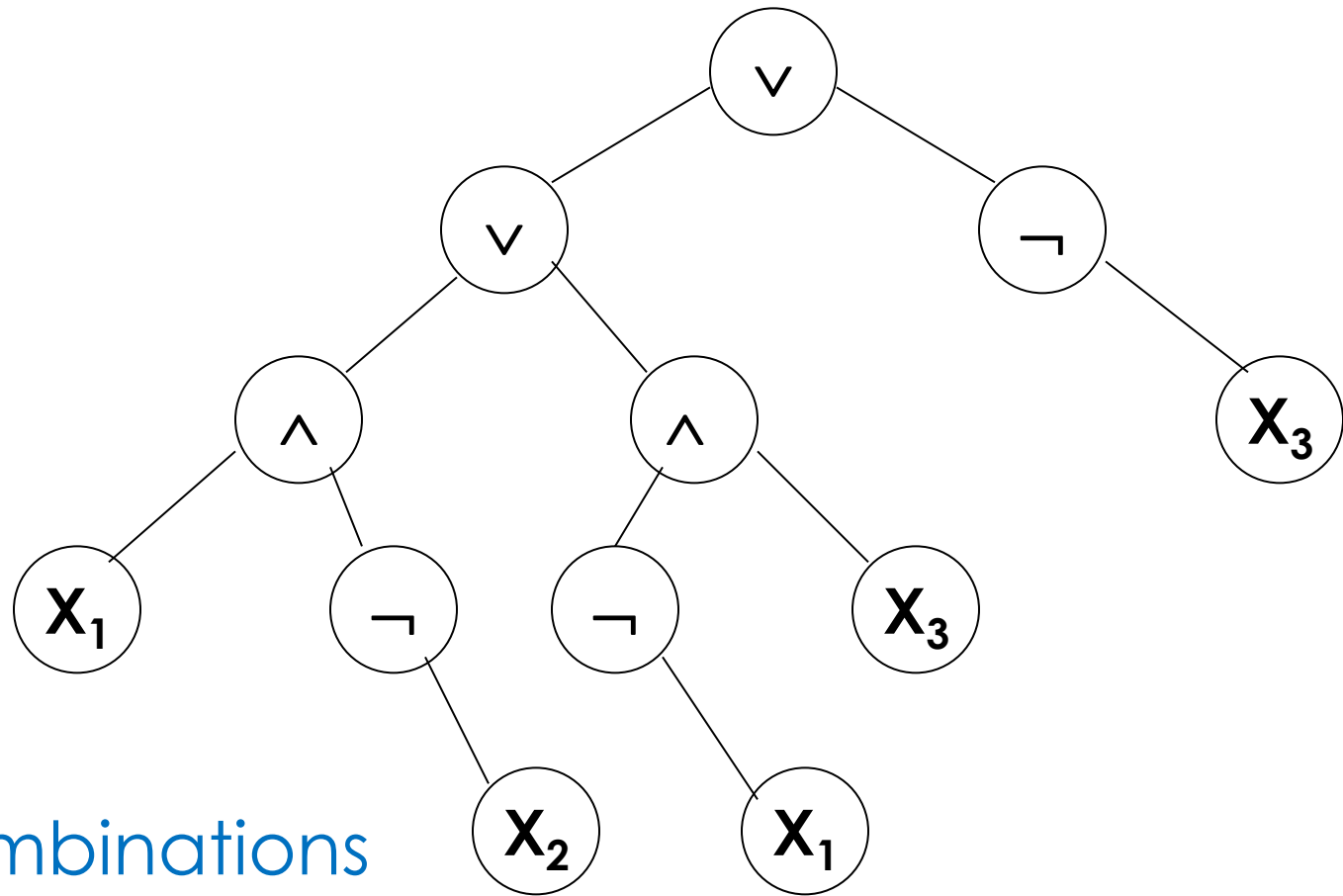
PROPOSITIONAL CALCULUS EXPRESSION

$$x_1 \vee (x_2 \wedge \neg x_3)$$

- 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 ($\neg > \wedge > \vee$).
- **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)

PERFORM FORMULA EVALUATION

- To evaluate an expression, we can traverse its tree in **postorder**.
- To perform evaluation, assume that each node has four fields
 - LeftChild
 - Data
 - Value
 - RightChild

<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 list_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");
```

EVALUATING A FORMULA (POSTORDER)

```
void post_order_eval(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;
```

EVALUATING A FORMULA (POSTORDER)

```
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;
```

```
}
```

```
}
```

```
}
```

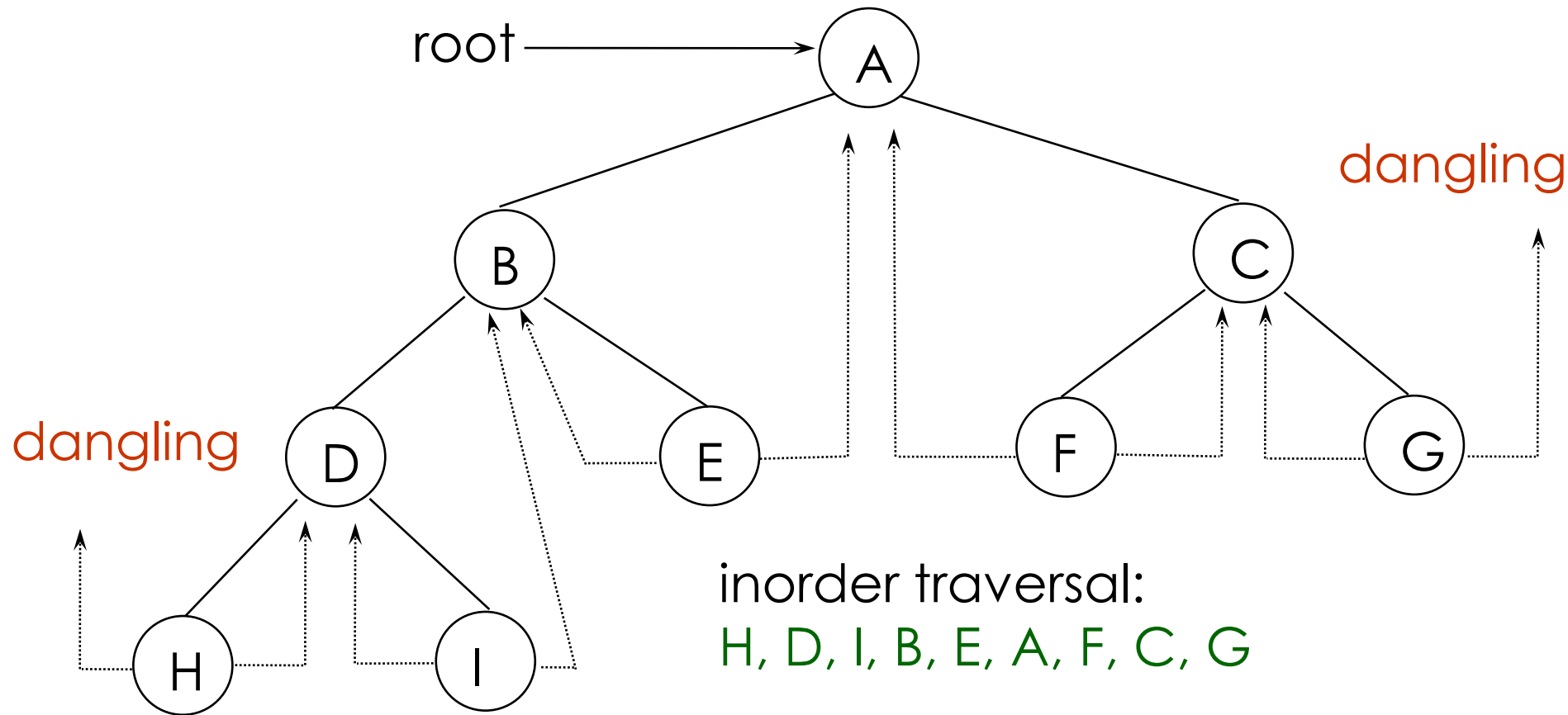
THREADED BINARY TREE

- Two many null pointers in current representation of binary trees
 - n : number of nodes
 - number of non-null links: $n-1$
 - total links: $2n$
 - null links: $2n-(n-1)=n+1$
- Replace these null pointers with some useful “threads”.

THREADED BINARY TREE (CONT'D)

- If $\text{ptr} \rightarrow \text{left_child}$ is null,
 - replace it with a pointer to the node that would be visited *before ptr* in an *inorder traversal*
- If $\text{ptr} \rightarrow \text{right_child}$ is null,
 - replace it with a pointer to the node that would be visited *after ptr* in an *inorder traversal*

THREADED BINARY TREE (CONT'D)

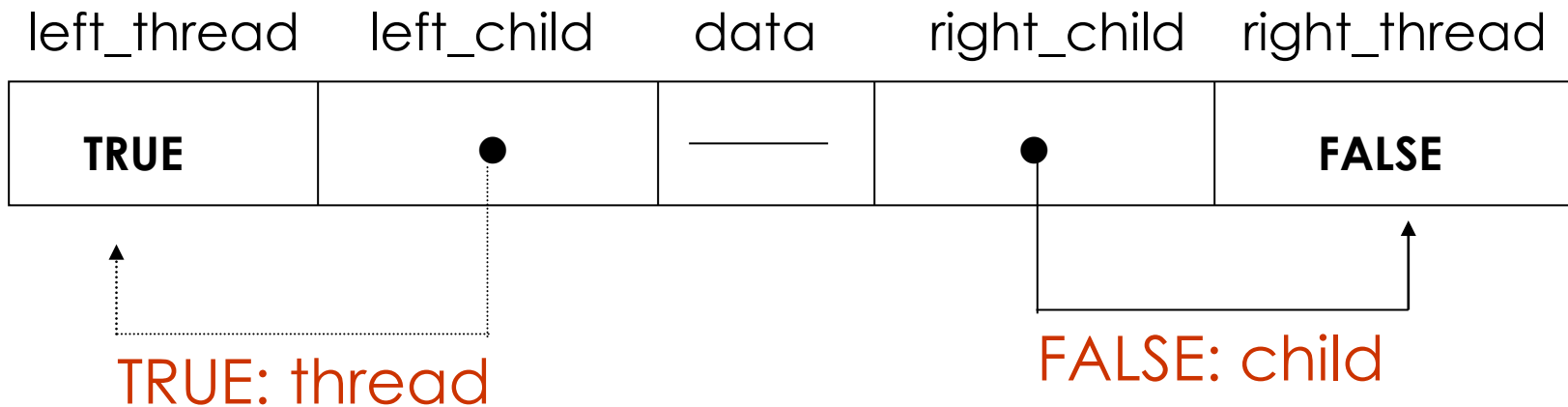




THREADS

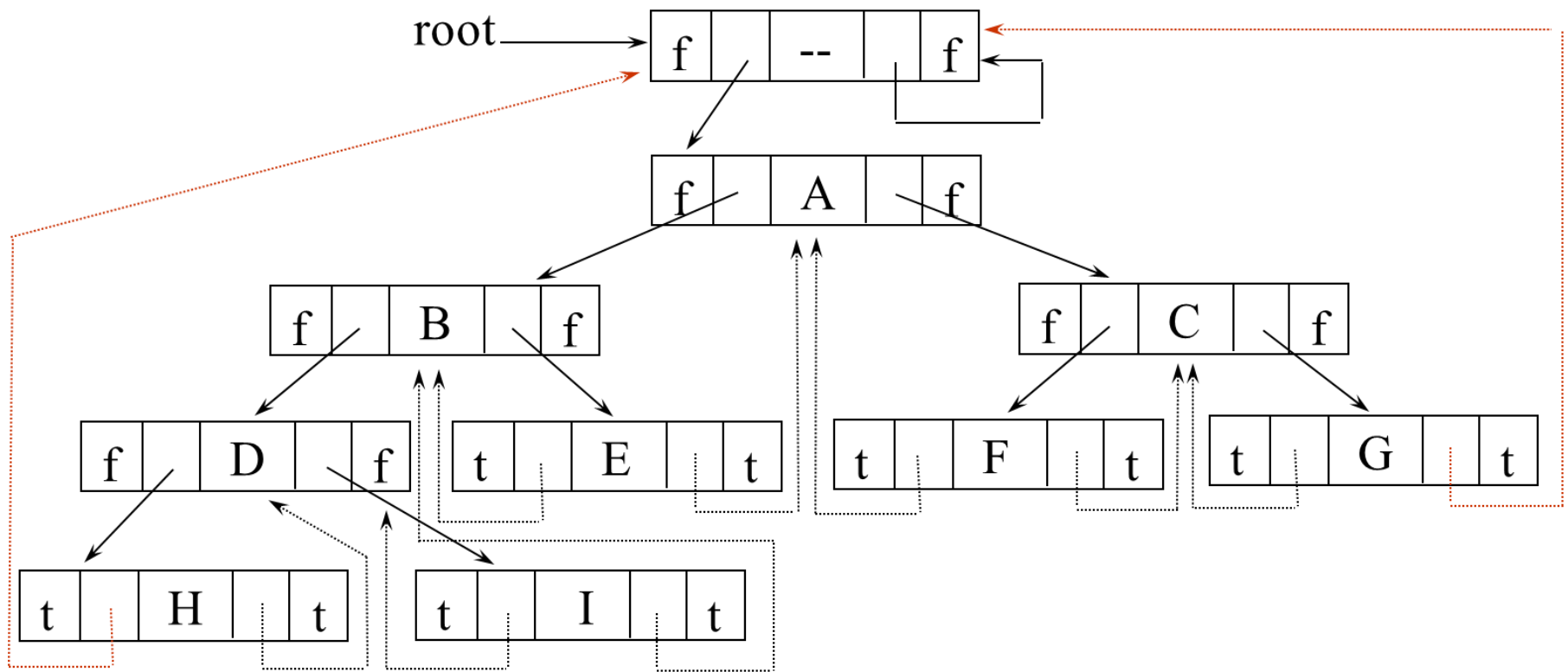
- To distinguish between normal pointers and threads, two boolean fields, LeftThread and RightThread, are added to the record in memory representation.
 - $t \rightarrow \text{LeftThread} = \text{TRUE}$
 - ➔ $t \rightarrow \text{LeftChild}$ is a **thread**
 - $t \rightarrow \text{LeftThread} = \text{FALSE}$
 - ➔ $t \rightarrow \text{LeftChild}$ is a **pointer** to the left child.

DATA STRUCTURES FOR THREADED BT



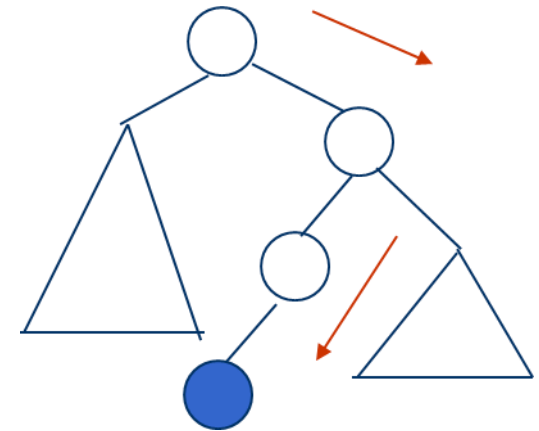
```
typedef struct threaded_tree *threaded_pointer;
typedef struct threaded_tree {
    short int left_thread;
    threaded_pointer left_child;
    char data;
    threaded_pointer right_child;
    short int right_thread; };
```

MEMORY REPRESENTATION OF THREADED B.T.



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

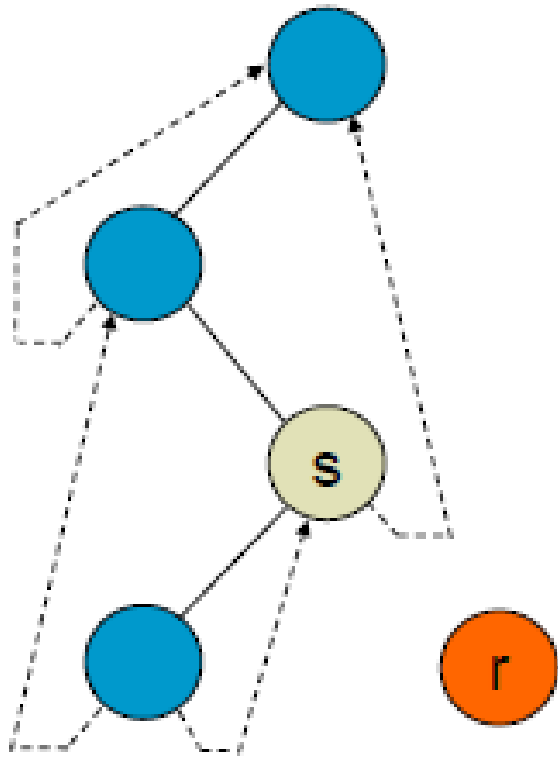
```
void tinorder(threaded_pointer tree)
{
    /* traverse the threaded binary tree
    inorder */
    threaded_pointer temp = tree;
    for (;;) {
         $O(n)$     temp = insucc(temp);
        if (temp==tree) break;
        printf("%3c", temp->data);
    }
}
```

INSERTING A NODE TO A THREADED BINARY TREE

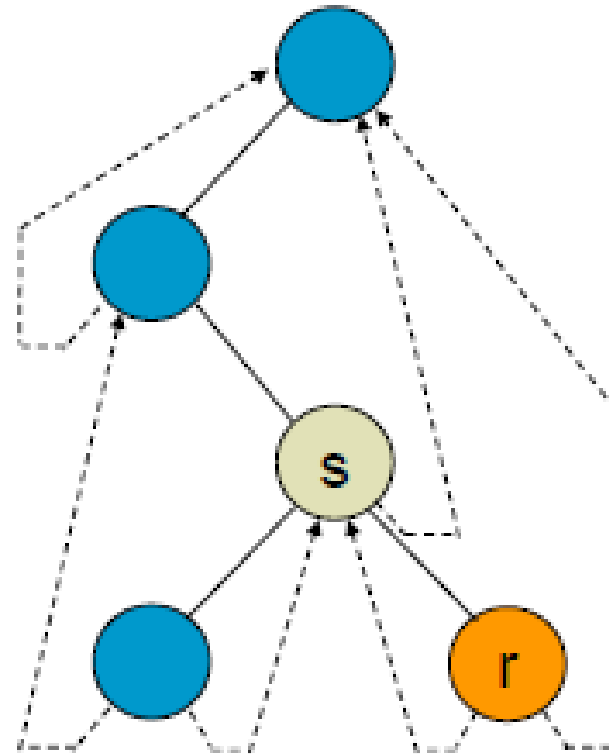
- Insert `child` as the right child of node `parent`
 - change `parent->right_thread` to `FALSE`
 - set `child->left_thread` and `child->right_thread` to `TRUE`
 - set `child->left_child` to point to `parent`
 - set `child->right_child` to `parent->right_child`
 - change `parent->right_child` to point to `child`

EXAMPLE

Insert a node r as a right child of s



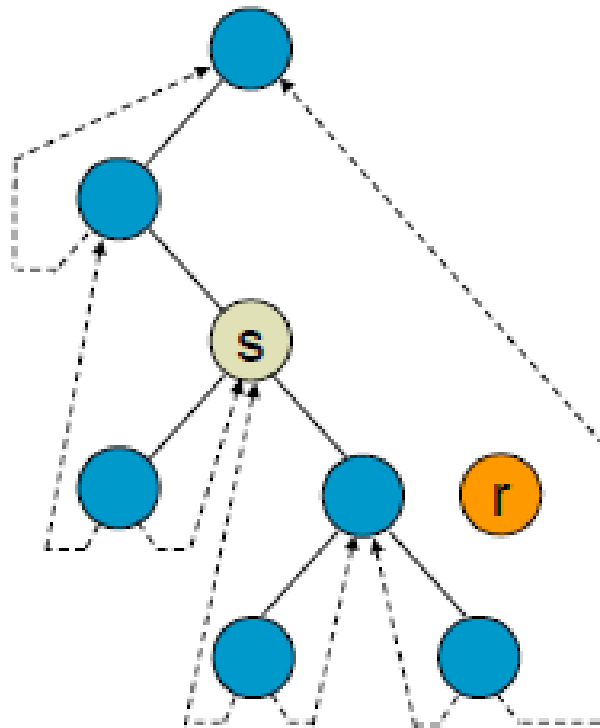
before



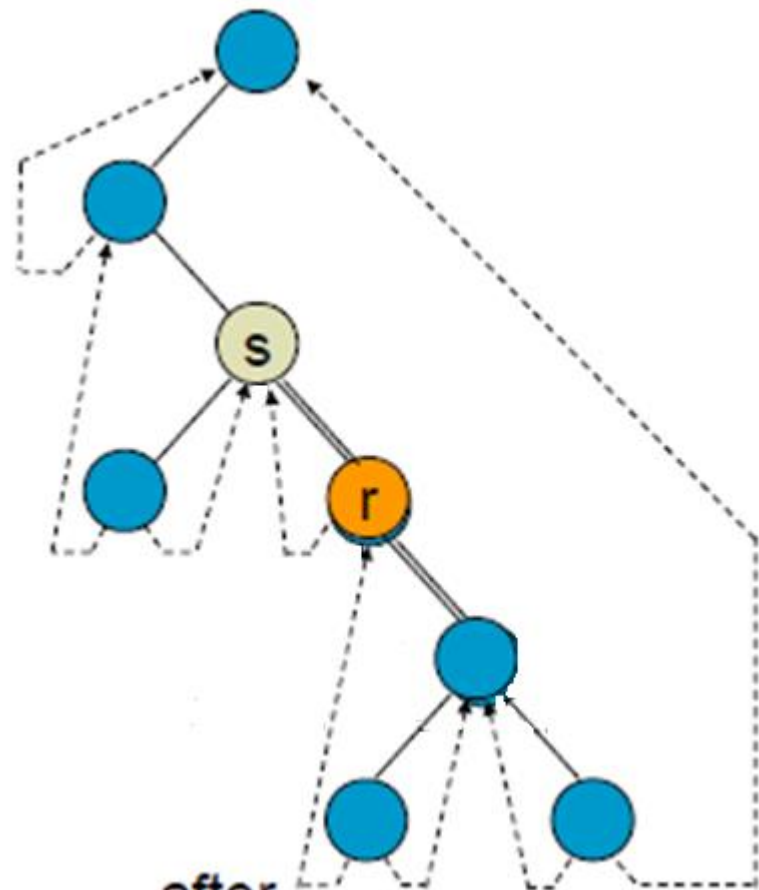
after

EXAMPLE

Insert a node r as a right child of s



before



after

RIGHT INSERTION IN THREADED BTS

```
void insert_right(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



PRIORITY QUEUES

- In a priority queue, the element to be deleted is the one with highest (or lowest) priority.
- An element with arbitrary priority can be inserted into the queue according to its priority.
- A data structure supports the above two operations is called max (min) priority queue.



APPLICATION: PRIORITY QUEUE

- machine service
 - amount of time (min heap)
 - amount of payment (max heap)



DATA STRUCTURES

- Unordered linked list
- Unordered array
- Sorted linked list
- Sorted array
- Heap

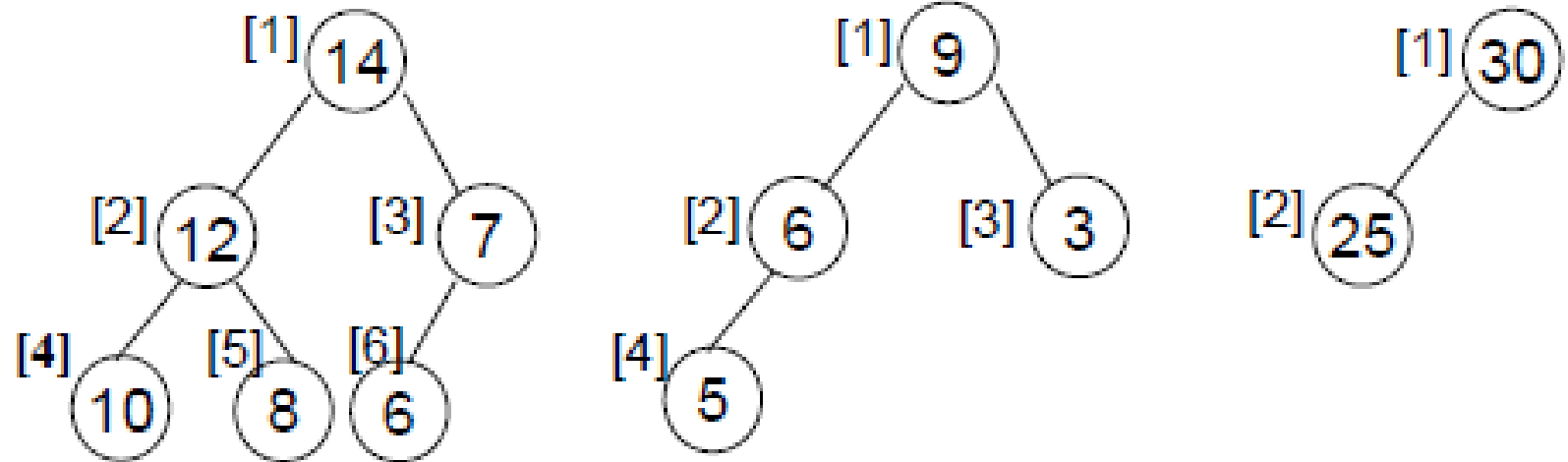
PRIORITY QUEUE REPRESENTATIONS

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

MAX (MIN) HEAP

- Heaps are frequently used to implement priority queues. The complexity is **$O(\log n)$** .
- Definition:
 - A max (min) tree is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any).
 - A max heap is a **complete binary tree** that is also a max tree.
 - A min heap is a **complete binary tree** that is also a min tree.

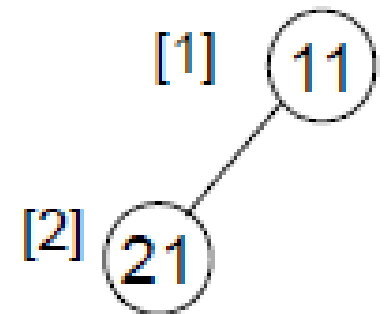
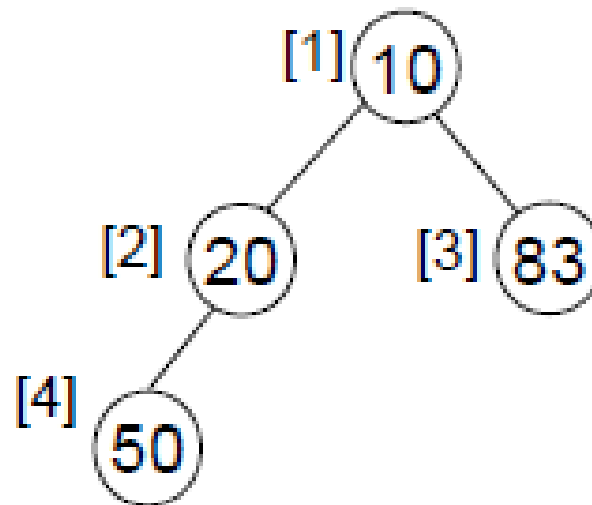
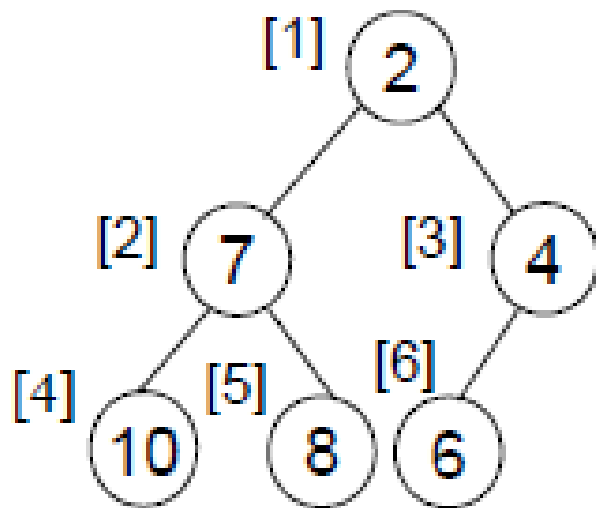
MAX HEAP EXAMPLES



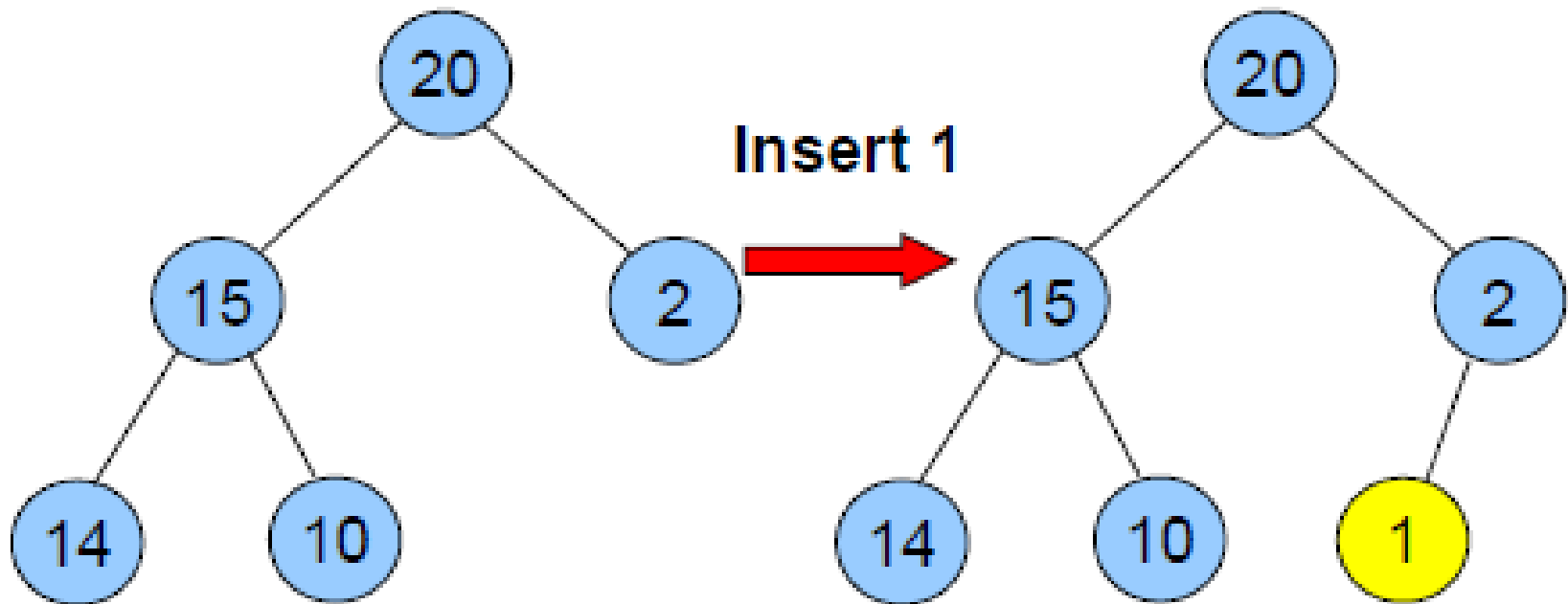
Property:

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

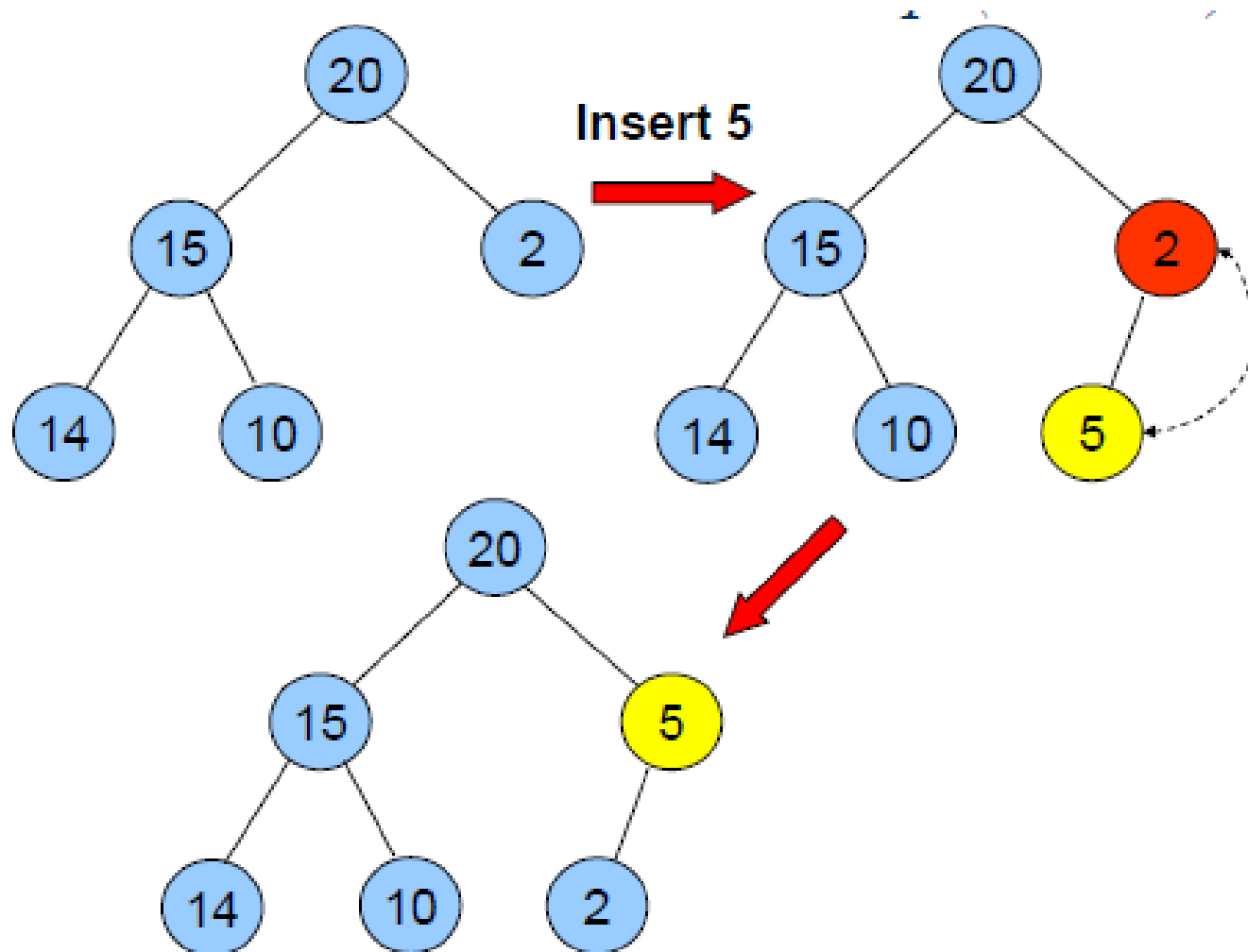
MIN HEAP EXAMPLES



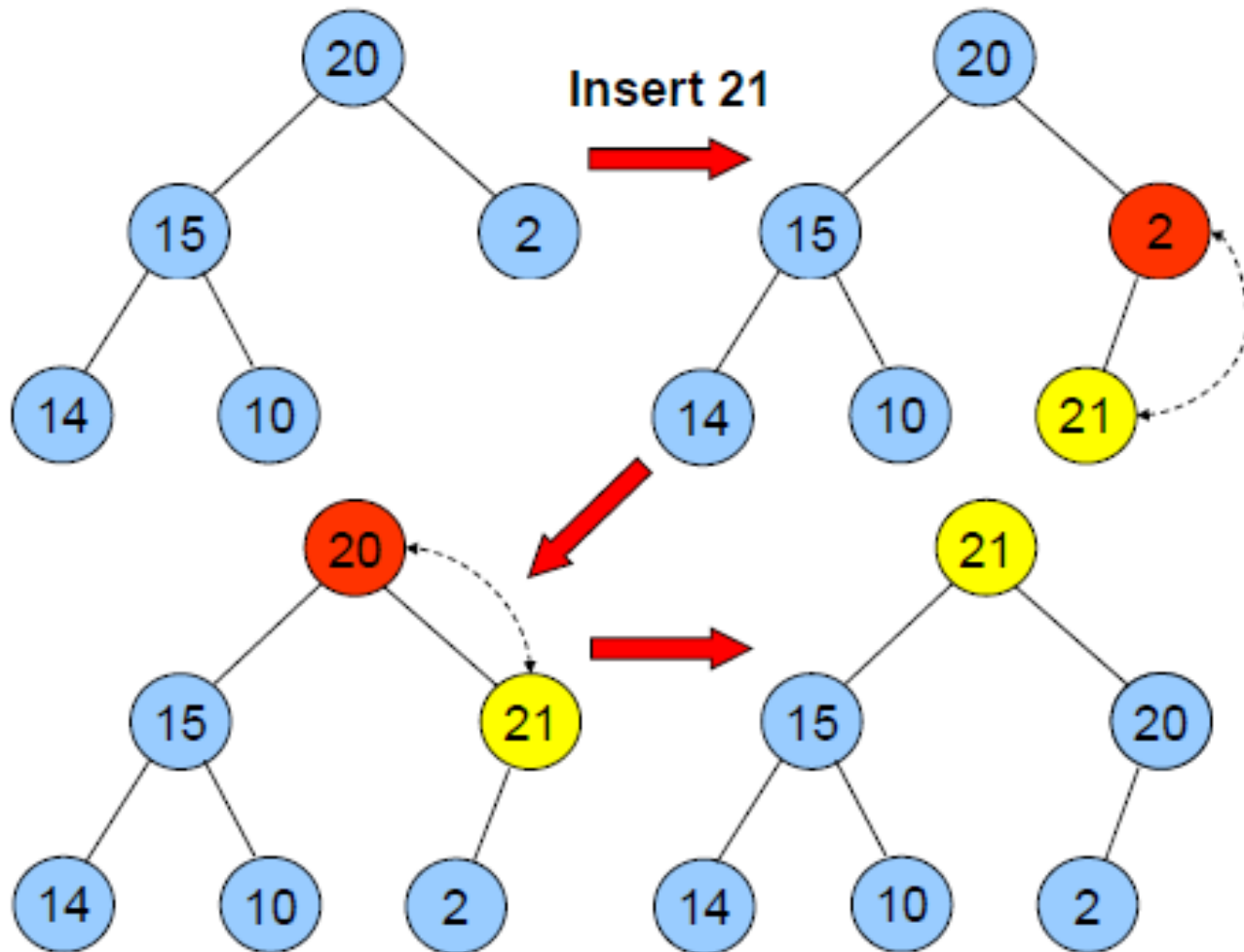
INSERTION INTO A MAX HEAP



INSERTION INTO A MAX HEAP (CONT'D)



INSERTION INTO A MAX HEAP (CONT'D)



PROGRAM 5.18:

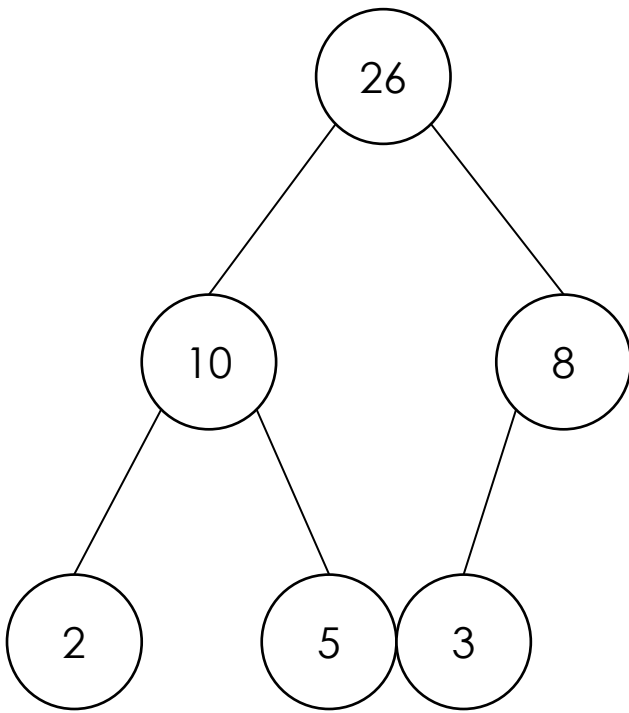
```
void insert_max_heap(element item, int *n)
{
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "the heap is full.\n");
        exit(1);
    }
    i = ++(*n);
    while ((i!=1) && (item.key>heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}
```

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

$O(\log_2 n)$

EXERCISE

- Insert: 15, 30, 20, 50



DELETION FROM A MAX HEAP

```
element delete_max_heap(int *n)
{
    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;
```

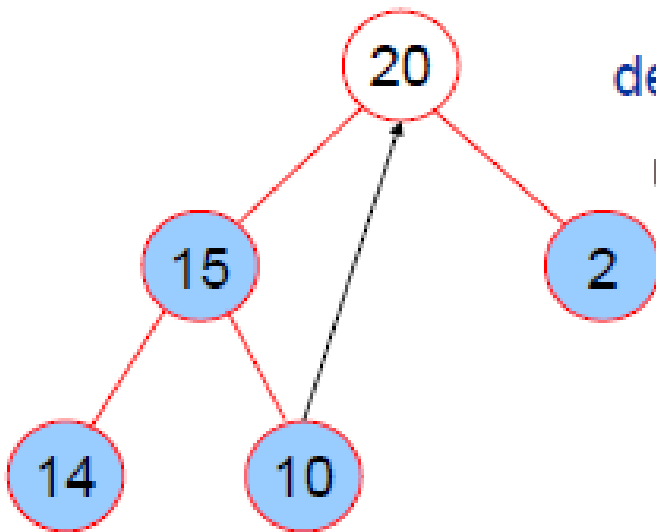
DELETION FROM A MAX HEAP

```
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];
    child *= 2;
}
heap[parent] = temp;
return item;
}
```

DELETION FROM A MAX HEAP

1	2	3	4	5
20	15	2	14	10

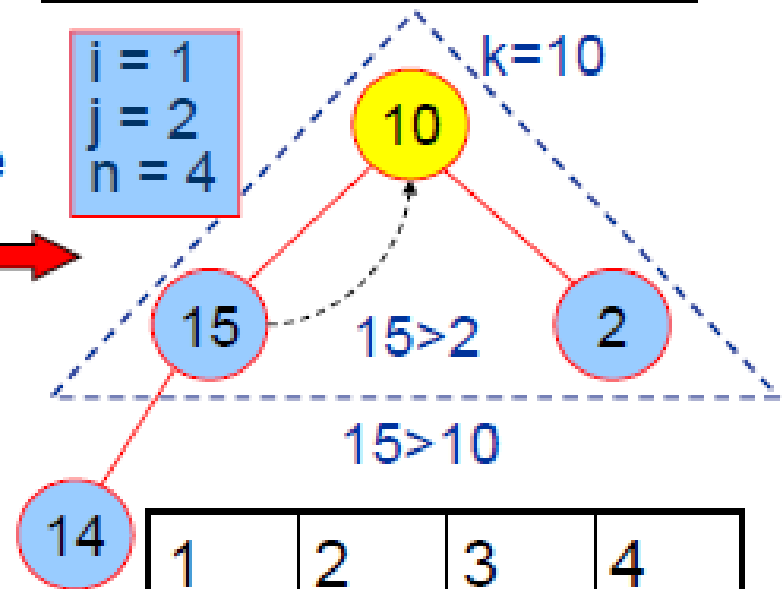
1	2	3	4
20	15	2	14



delete



$i = 1$
 $j = 2$
 $n = 4$

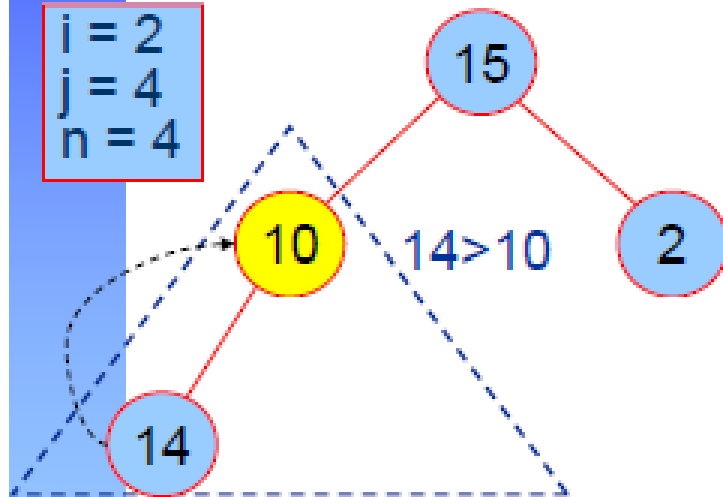


1	2	3	4
15	15	2	14

DELETION FROM A MAX HEAP (CONT'D)

1	2	3	4
15	15	2	14

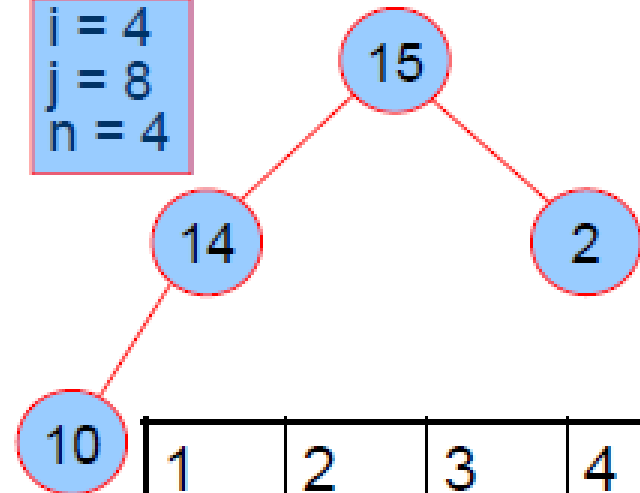
i = 2
j = 4
n = 4



1	2	3	4
15	14	2	14

1	2	3	4
15	14	2	14

i = 4
j = 8
n = 4



1	2	3	4
15	14	2	10



EXERCISE

Using the new heap we just construct. Now, draw the results after performing following functions:

- (1) Delete Max.
- (2) Delete Max again.

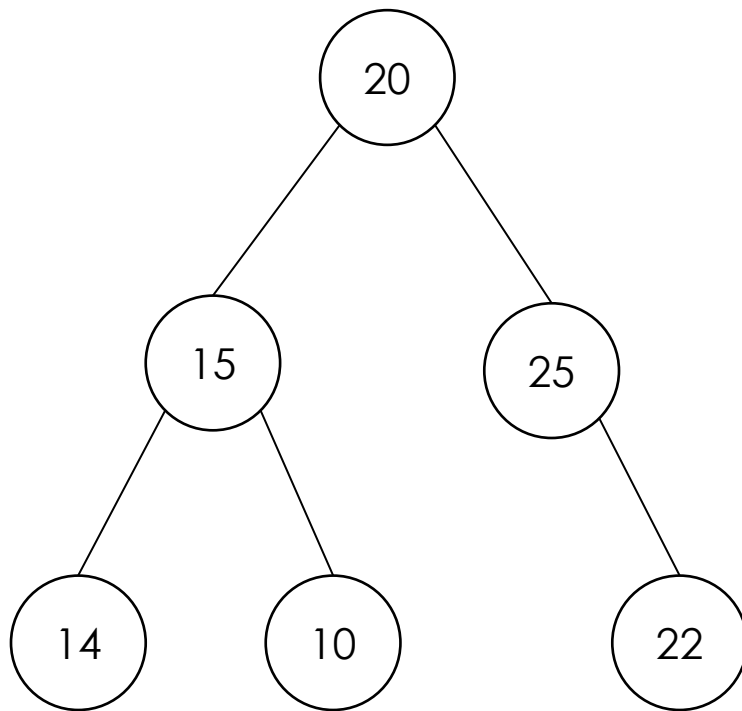
BUILD A HEAP

- Top-down method:
 - using insert function to insert all the nodes to build a heap
 - $O(n \log n)$
- Bottom-up method:
 - First, input all the data as a complete B.T.
 - Second, adjust each subtree to heap from the last parent
 - $O(n)$
- Example: Given 5,26,77,13,49,8,90,60, build a max heap

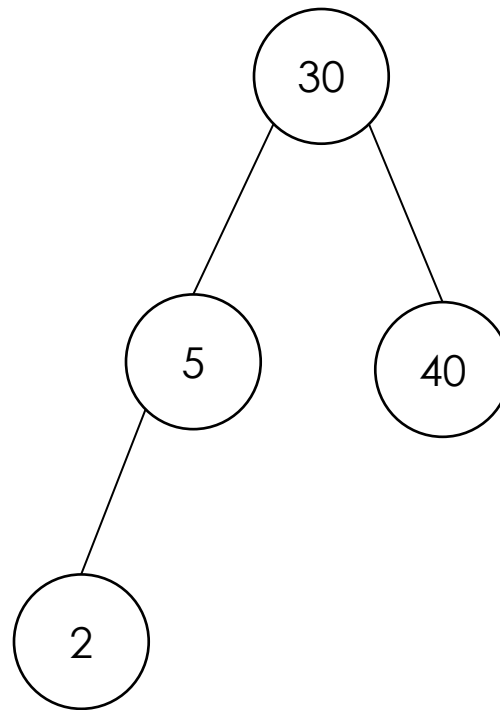
BINARY SEARCH TREE

- 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 subtree are also binary search tree.

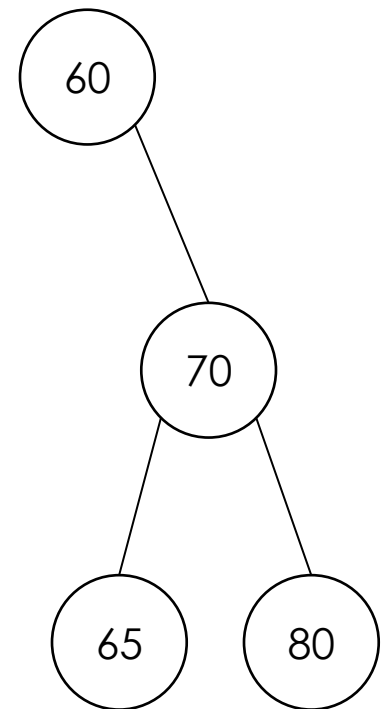
BINARY TREES



Not a B.S.T.



B.S.T.





SEARCHING A BINARY SEARCH TREE

- If the root is null, then this is an empty tree. No search is needed.
- If the root is not null, compare the x with the key of root.
 - If x is equal to the key of the root, then it's done.
 - If x is less than the key of the root, then no elements in the right subtree have key value x . We only need to search the left tree.
 - If x is larger than the key of the root, only the right subtree is to be searched.

SEARCHING A B.S.T.

```
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);
}
```

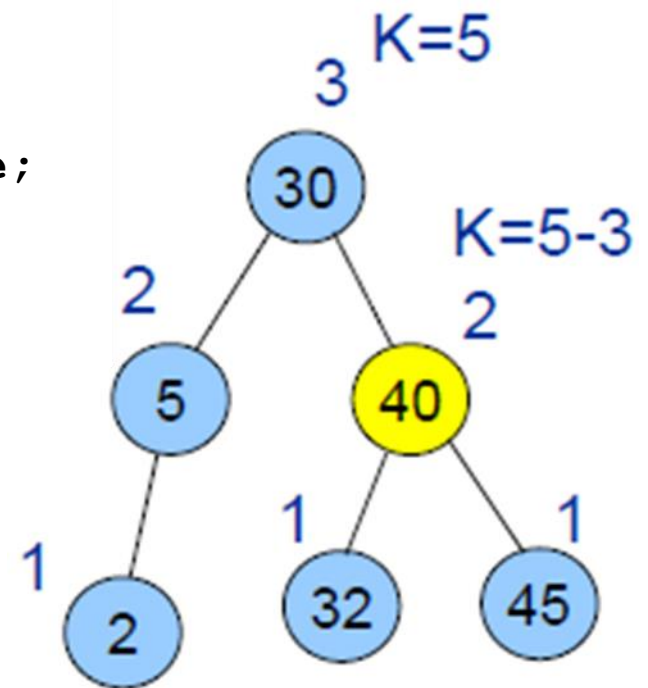


SEARCH BINARY SEARCH TREE BY RANK

- To search a binary search tree by the ranks of the elements in the tree, we need additional field *LeftSize*.
- LeftSize is **the number of the elements** in the left subtree of a node plus one.
- It is obvious that a binary search tree of height h can be searched by key as well as by rank in $O(h)$ time.
 - What is the range of h ?

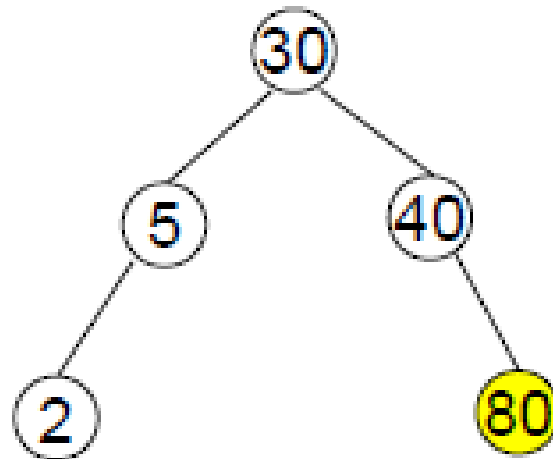
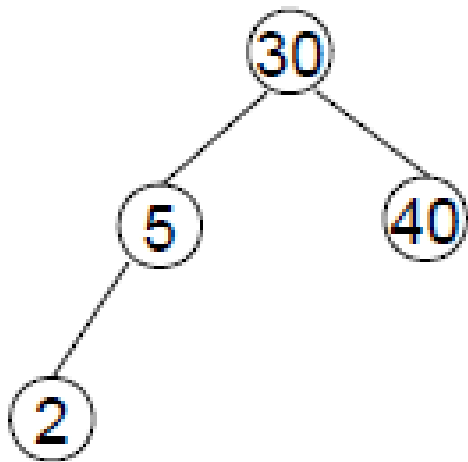
ANOTHER SEARCHING ALGORITHM

```
tree_pointer search2(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;
}
```

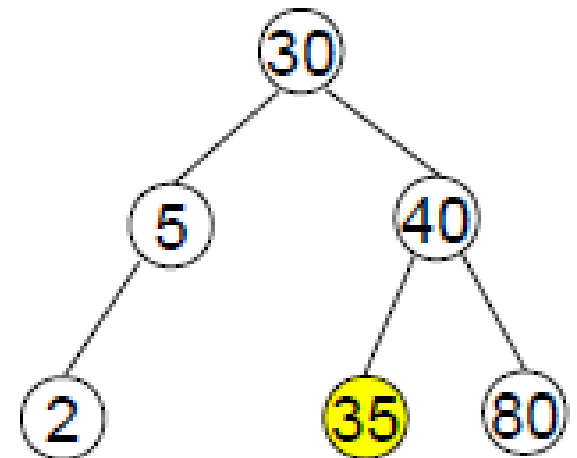


$O(h)$

INSERTING A NODE INTO A B.S.T.



Insert 80



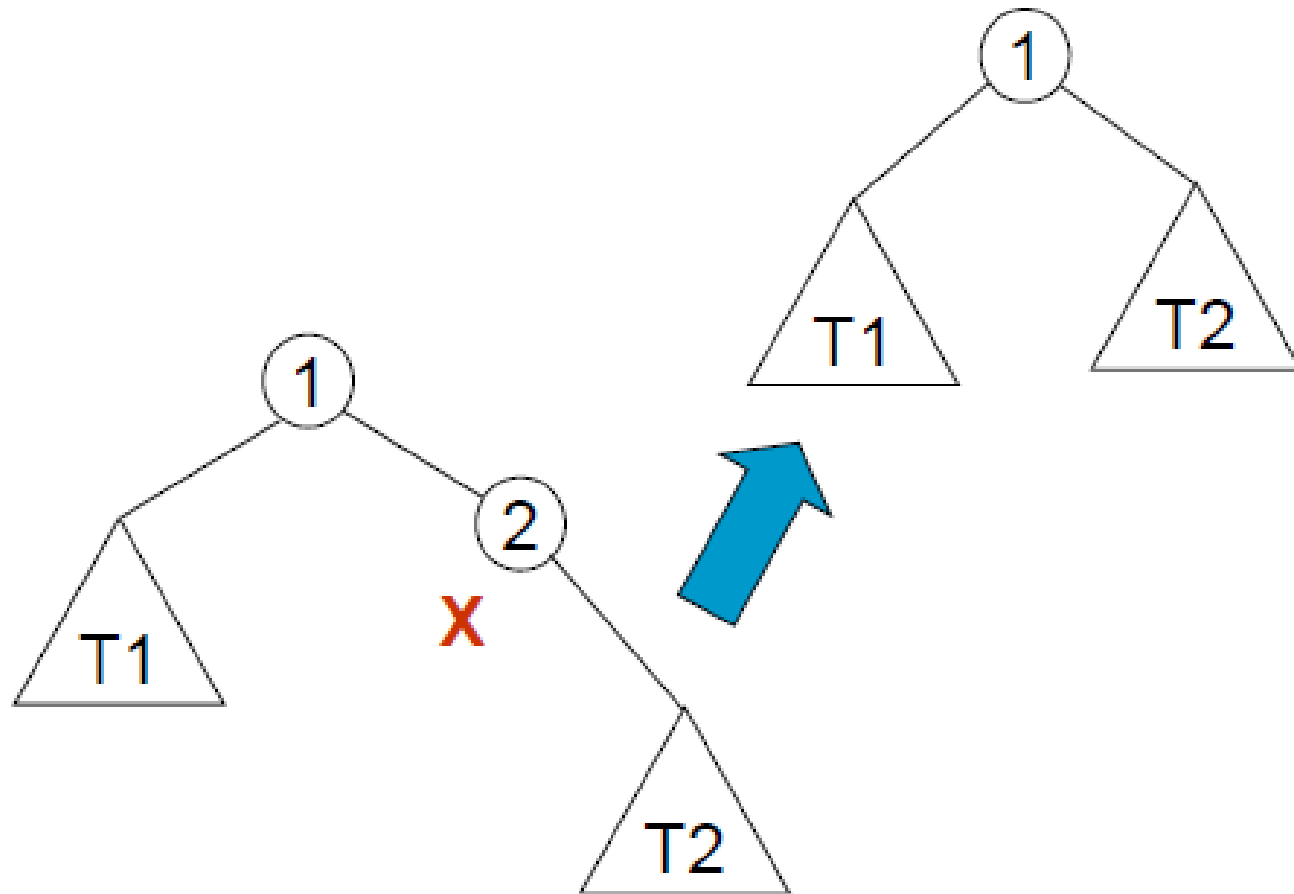
Insert 35



EXERCISE:

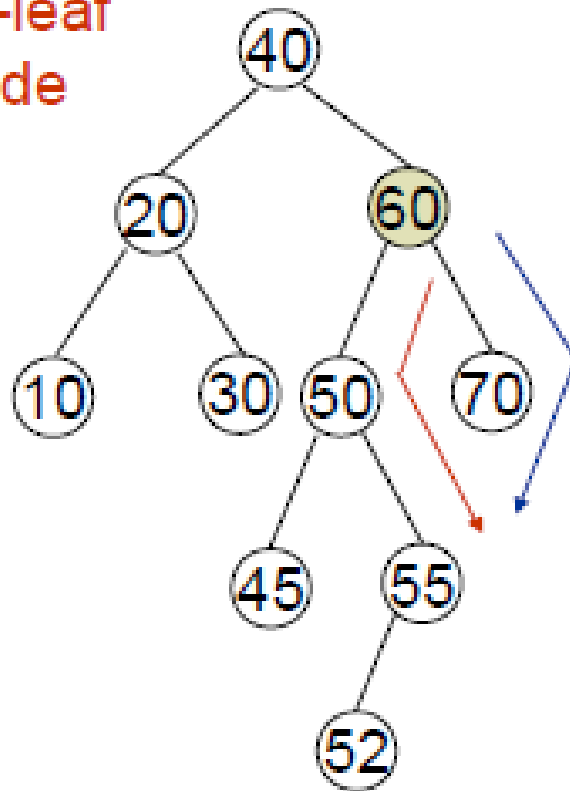
- Given 26, 5, 77, 13, 8, 2, 3, 49, 33, build a binary search tree.

DELETION FROM A BINARY SEARCH TREE

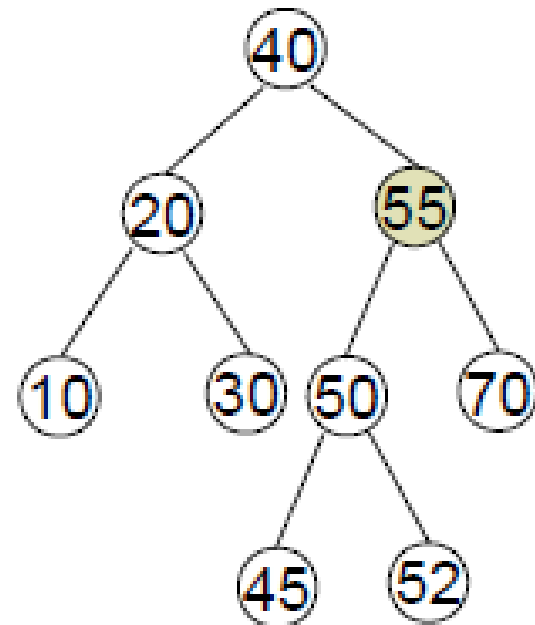


DELETION FROM A BINARY SEARCH TREE (CONT'D)

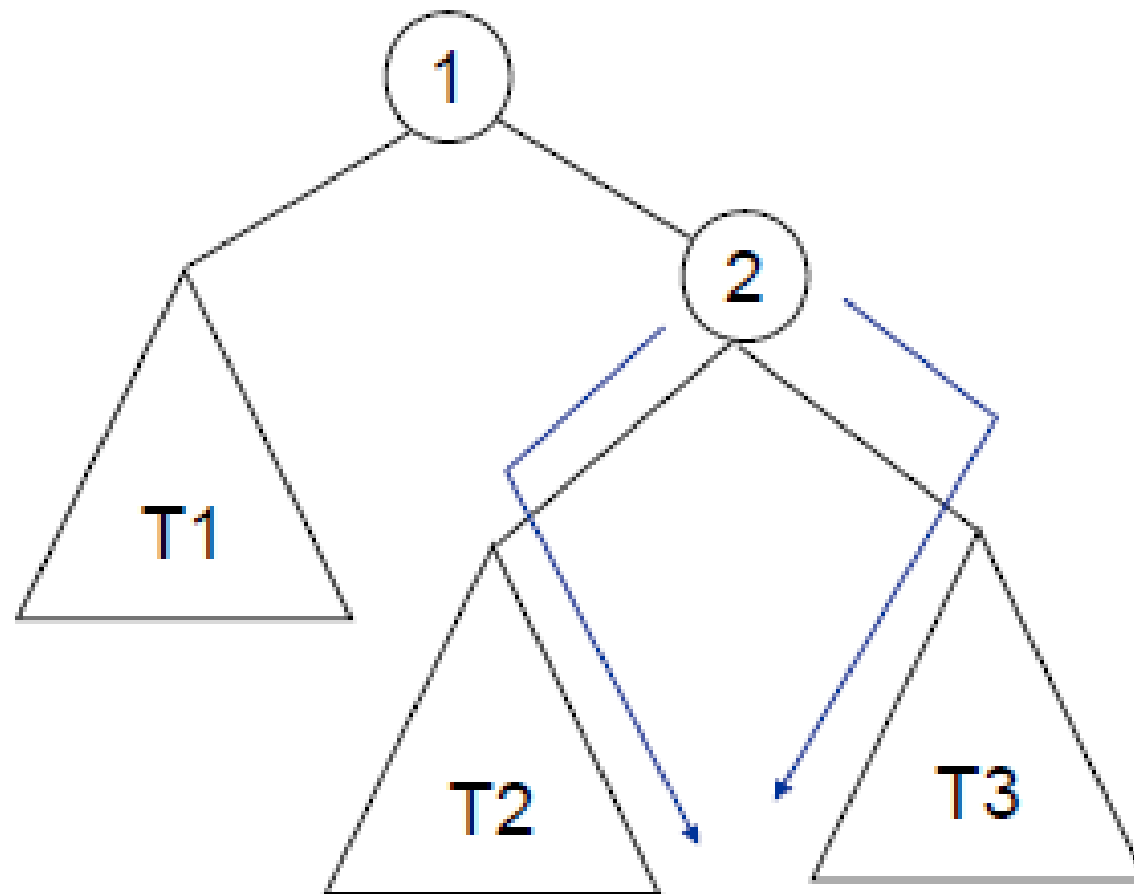
non-leaf
node



Before deleting 60



After deleting 60





EXERCISE:

- According to the binary search tree you just built, delete 8, 15, 26.



SELECTION TREES

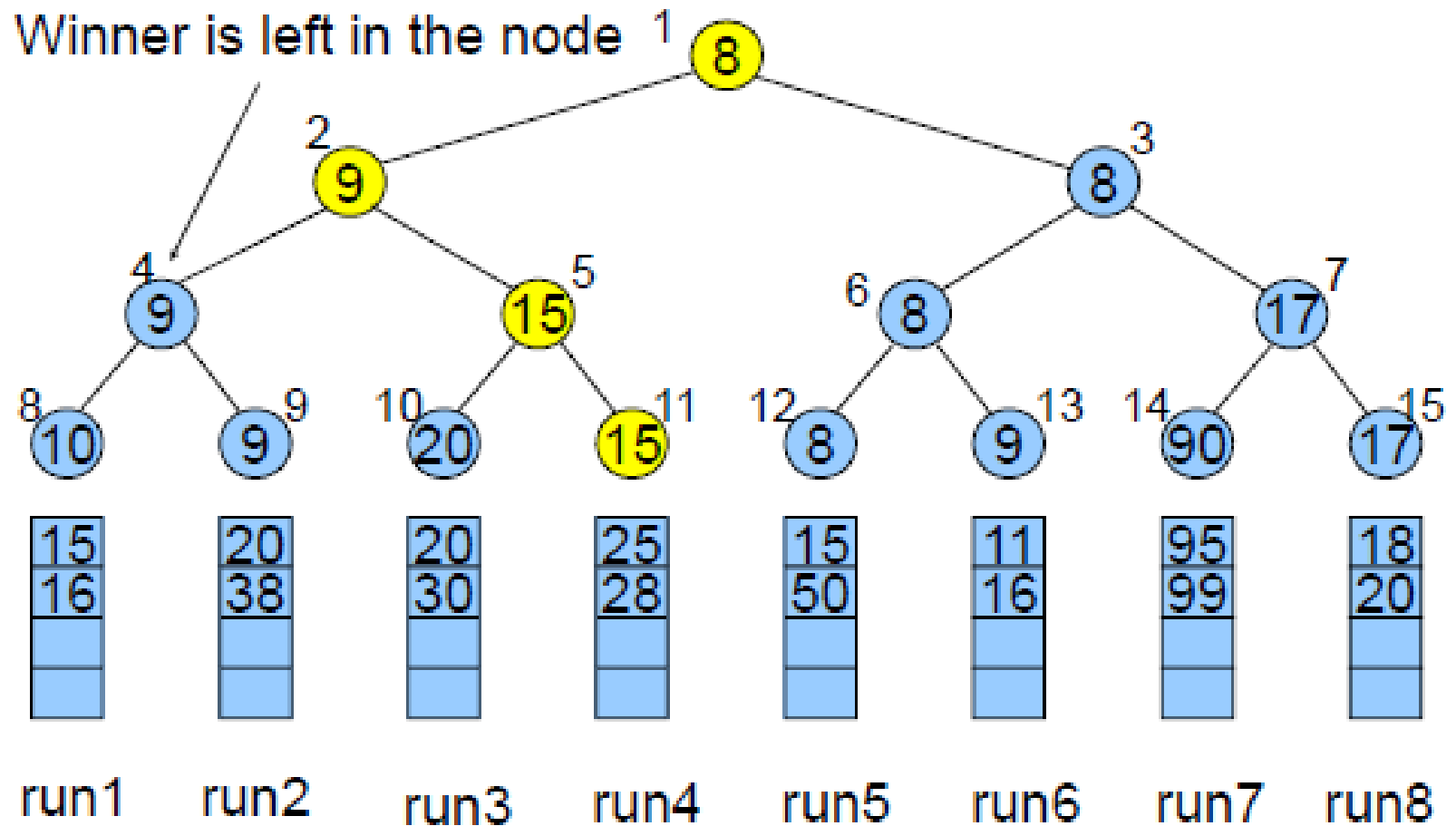
- Winner tree
- Loser tree



WINNER TREE



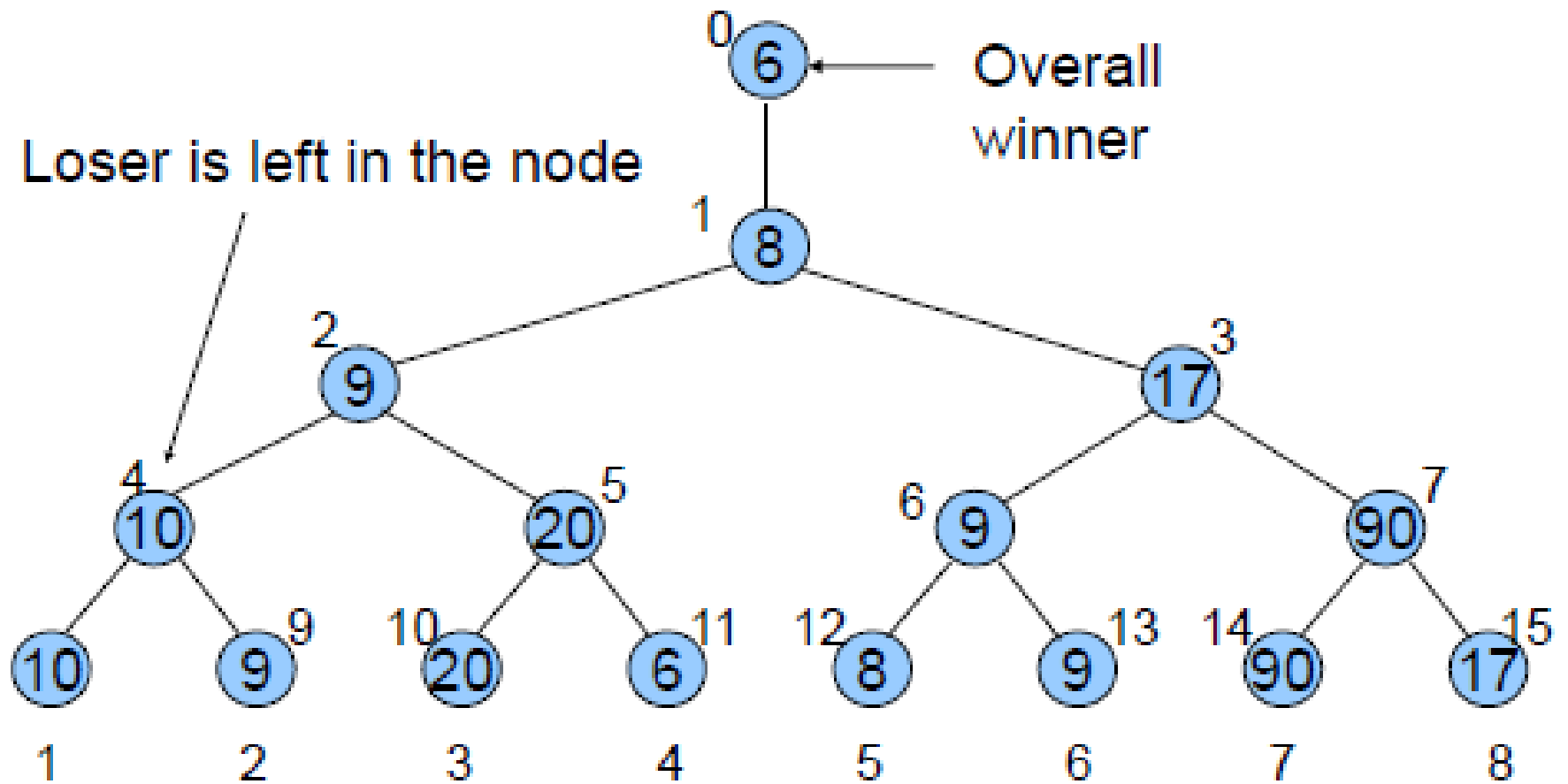
WINNER TREE (CONT'D)



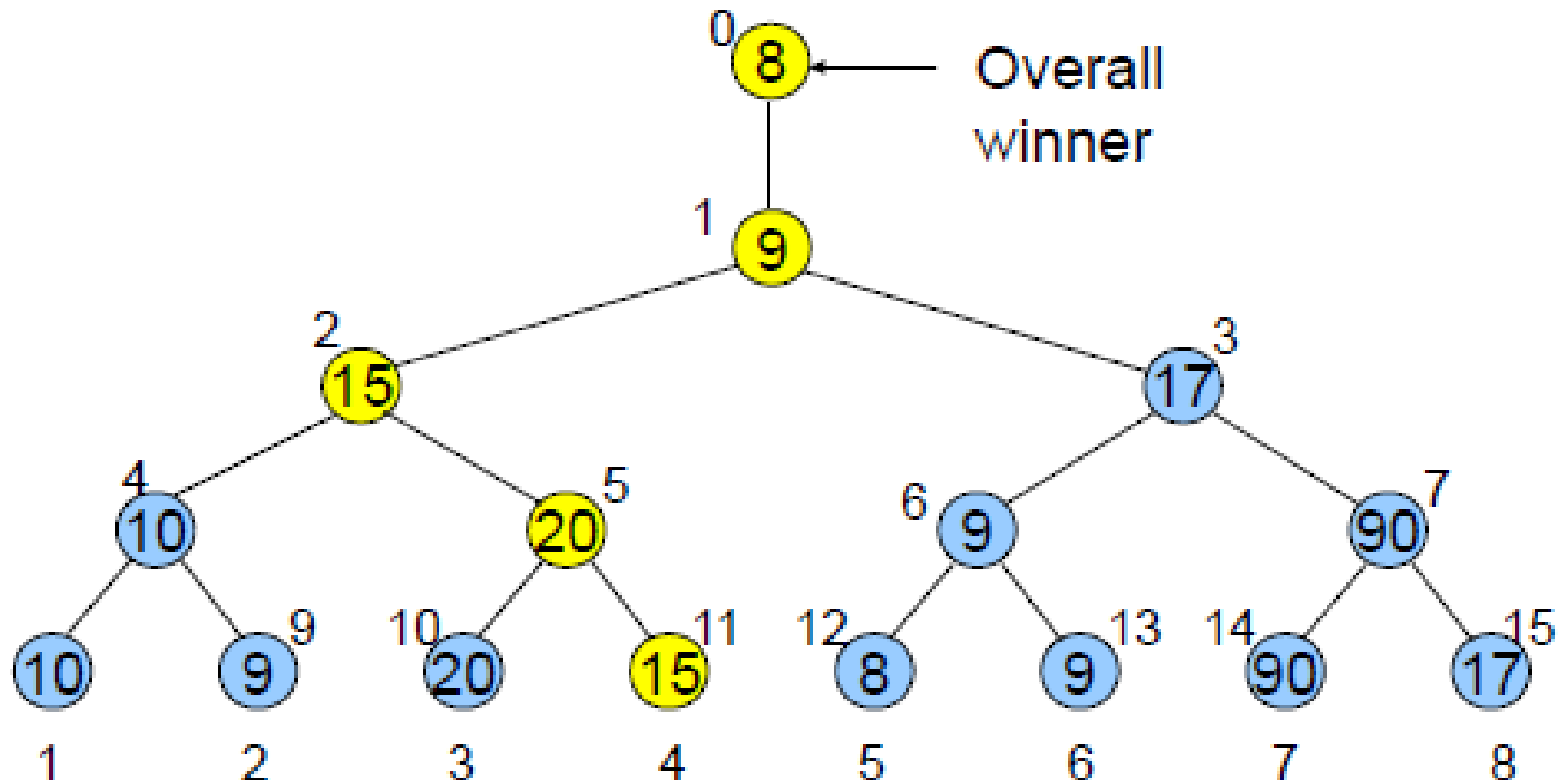
ANALYSIS

- K: # of runs
- n: # of records
- Setup time: $O(K)$
- Reconstruct time: $O(\log_2 K)$
- Merge time: $O(n \log_2 K)$
- Slight modification: tree of loser
 - Consider the parent node only (vs. sibling nodes)

LOSER TREE

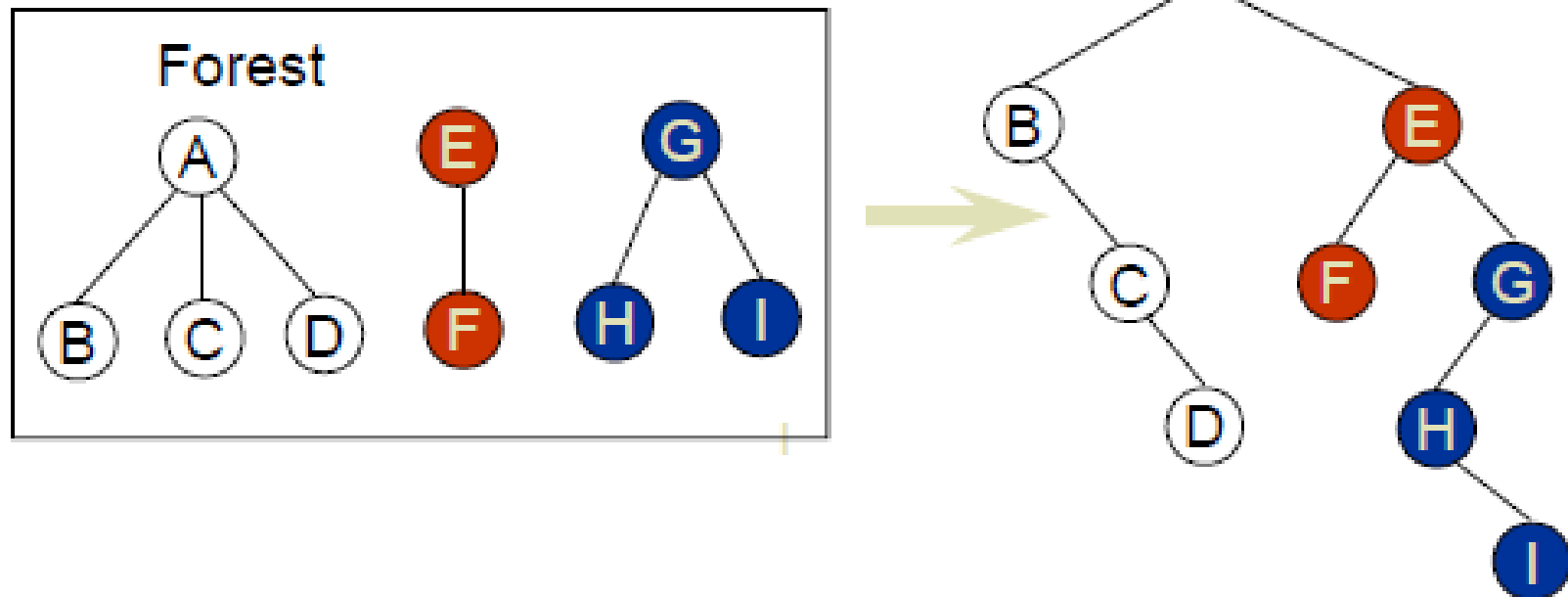


LOSER TREE (CONT'D)



FOREST

- A forest is a set of $n \geq 0$ disjoint trees
- Forest to BST



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
 - empty, if $n=0$
 - has root equal to $\text{root}(T_1)$; has left subtree equal to $B(T_{11}, T_{12}, \dots, T_{1m})$; where $B(T_{11}, T_{12}, \dots, T_{1m})$ are subtrees of $\text{root}(T_1)$; and has right subtree equal to $B(T_2, T_3, \dots, T_n)$



FOREST TRAVERSALS

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

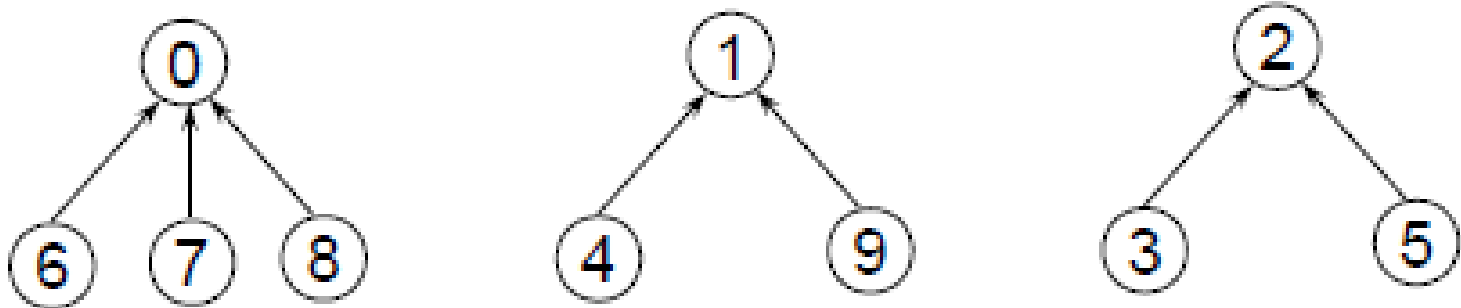


FOREST TRAVERSALS (CONT'D)

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

SET REPRESENTATION

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

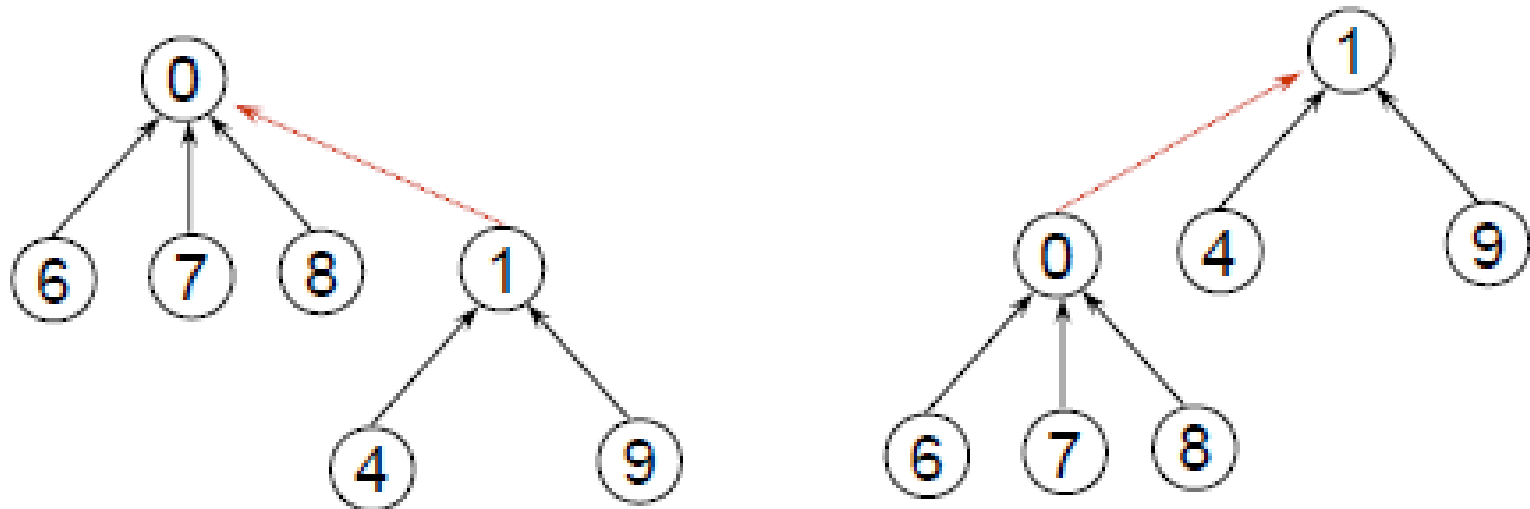


$$S_i \cap S_j = \phi$$

- 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

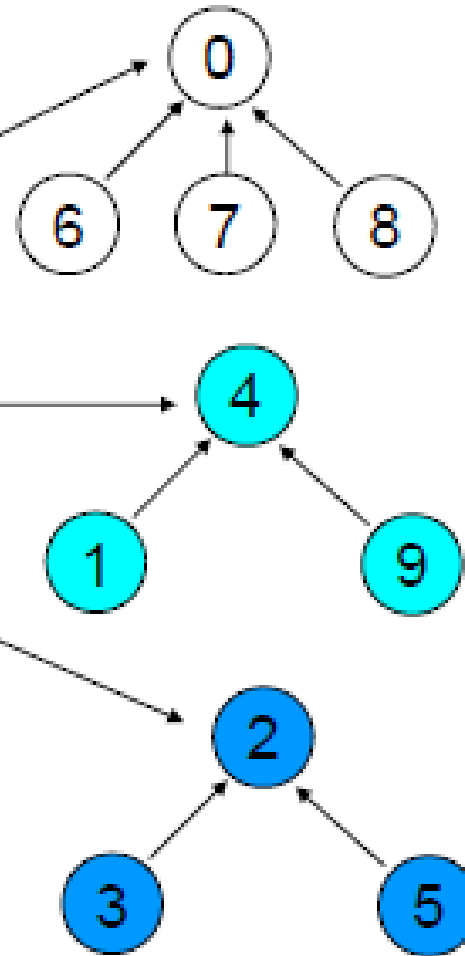
Make one of trees a subtree of the other



Possible representation for $S_1 \cup S_2$

DATA REPRESENTATION OF S_1 , S_2 AND S_3

Set name	pointer
S_1	
S_2	
S_3	



ARRAY REPRESENTATION OF S_1 , S_2 AND S_3

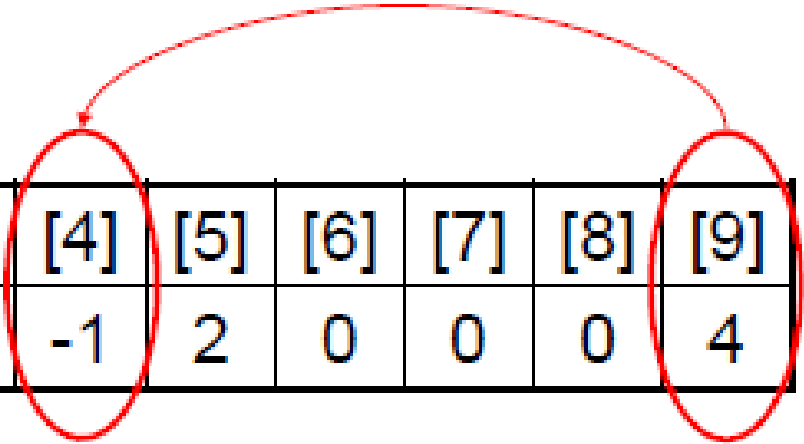
- We could use an array for the set name. Or the set name can be an element at the root.
- Assume set elements are numbered 0 through $n-1$.

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

root

SIMPLE FIND

```
SimpleFind(int i)
// Find the root of the tree
// containing element i
{
    while (parent[i] >= 0)
        i = parent[i];
    return i;
}
```



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

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



degenerate tree

DEGENERATE TREE

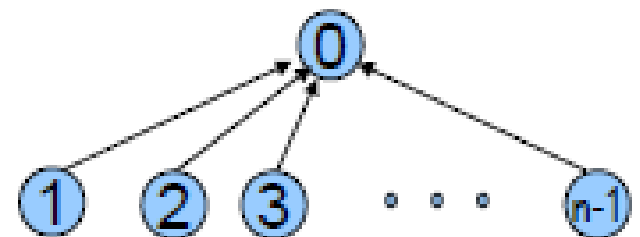
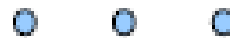
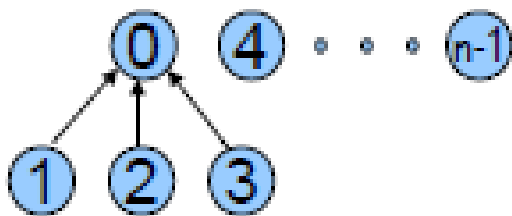
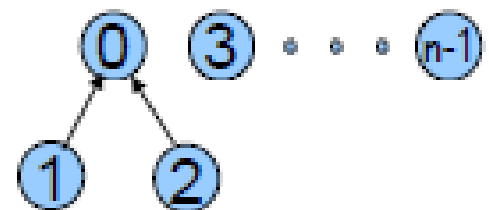
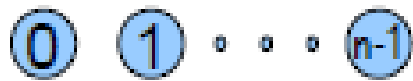
- n-1 union operations
 - $O(n)$
- One union operation
 - $O(1)$
- n find operations
 - $O(n^2)$ $\sum_{i=2}^n i$
- One find operations
 - $O(n)$



WEIGHTING RULE

- Weighting rule for *union*(i, j)
 - If the number of nodes in the tree with root i is less than the number in the tree with root j , then make j the parent of i ; otherwise make i the parent of j .
- Use the weighting rule on the union operation to avoid the creation of degenerate trees.

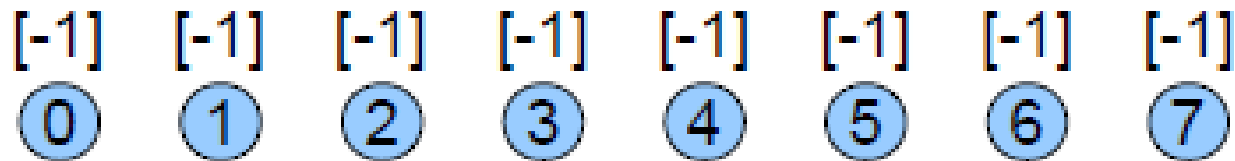
TREES OBTAINED USING THE WEIGHTING RULE



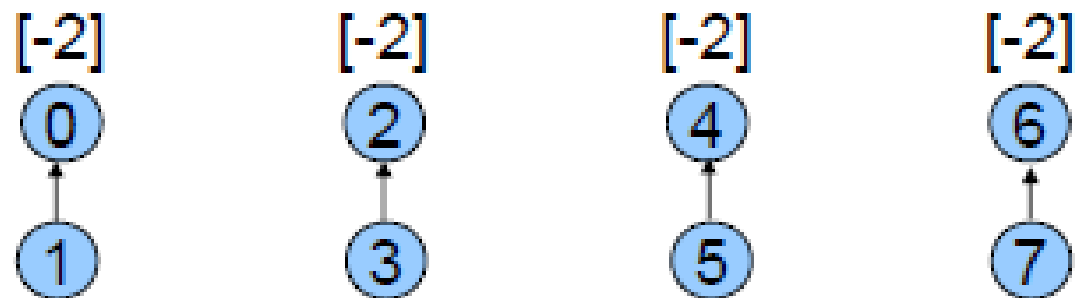
WEIGHTED UNION

- Lemma 5.5: Assume that we start with a forest of trees, each having one node. Let T be a tree with m nodes created as a result of a sequence of unions each performed using function `WeightedUnion`. The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$
- For the processing of an intermixed sequence of $u-1$ unions and f find operations, the time complexity is $O(u + f \log u)$.
 - No tree has more than u nodes in it.
 - We need $O(n)$ additional time to initialize the n -tree forest

TREES ACHIEVING WORST-CASE BOUND

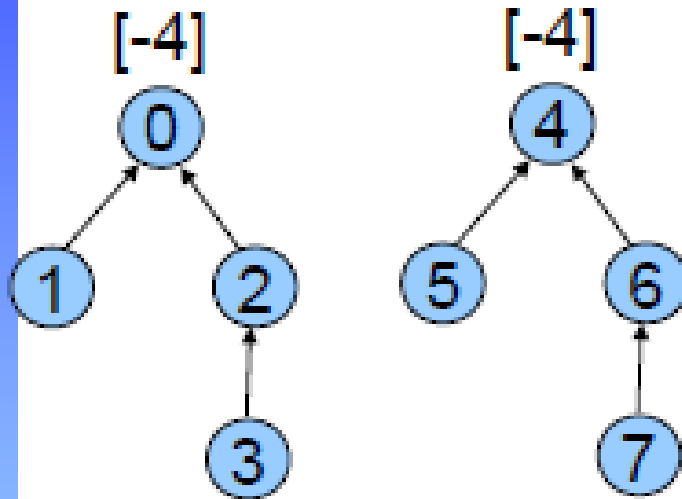


(a) Initial height trees

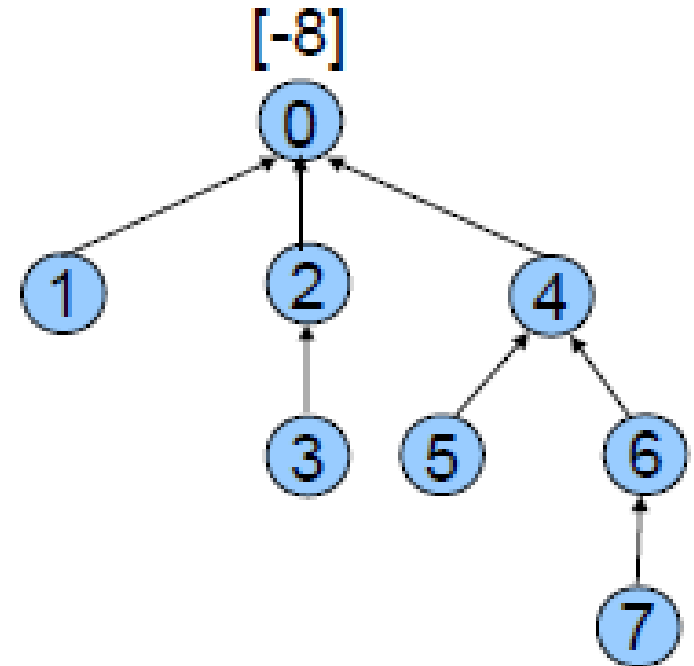


(b) Height-2 trees following union (0, 1), (2, 3), (4, 5), and (6, 7)

TREES ACHIEVING WORST-CASE BOUND (CONT'D)



(c) Height-3 trees following
union (0, 2), (4, 6)

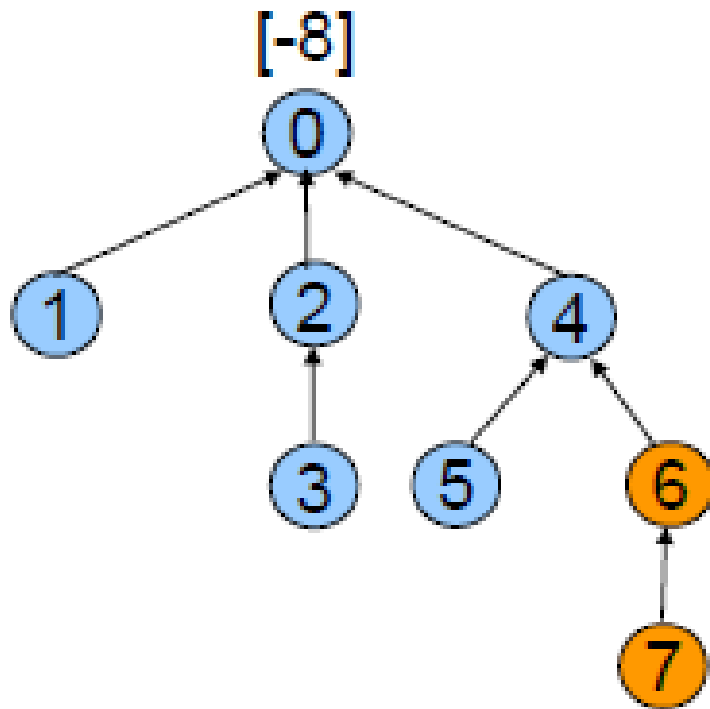


(d) Height-4 trees following
union (0, 4)

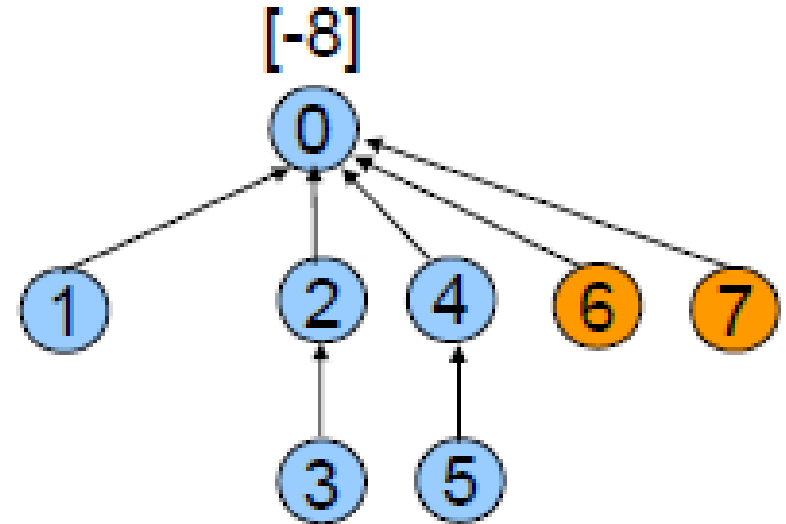
COLLAPSING RULE

- Collapsing rule:
 - If j is a node on the path from i to its root and $parent[i] \neq root(i)$, then set $parent[j]$ to $root(i)$.
- The first run of find operation will collapse the tree. Therefore, each following find operation of the same element only goes up one link to find the root.

COLLAPSING FIND



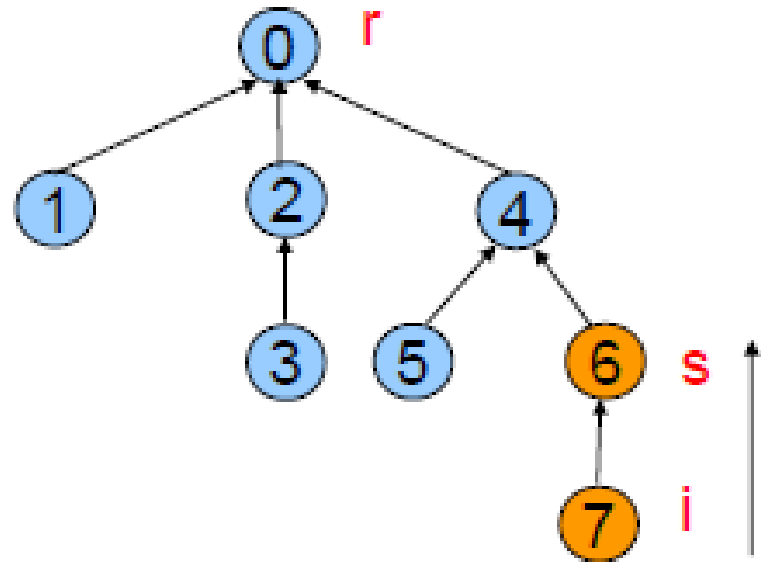
Before collapsing



After collapsing

COLLAPSING FIND (CONT'D)

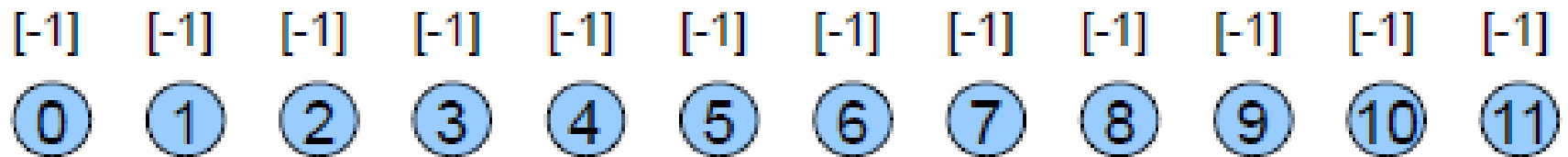
```
CollapsingFind(int i)
{
    // find root
    for(int r=i;parent[r]>=0;r=parent[r]);
    //collapse
    while(i!=r)
    {
        int s=parent[i];
        parent[i]=r;
        i=s;
    }
}
```



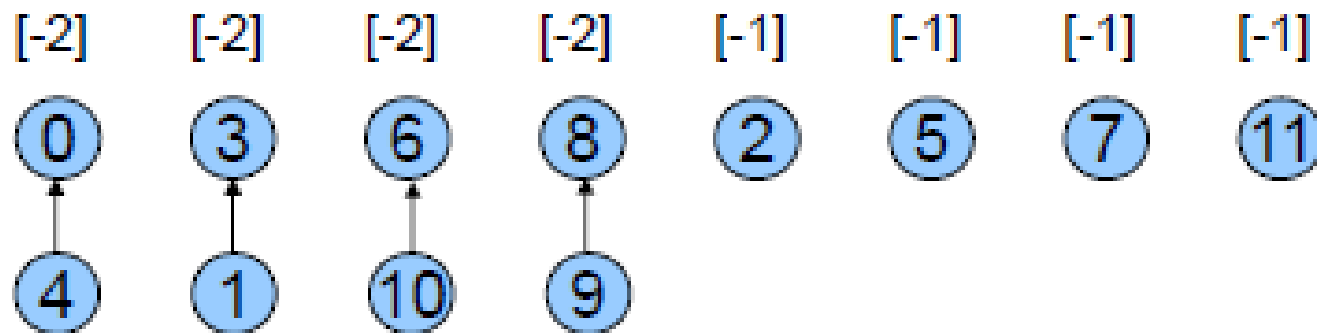
APPLICATIONS

- 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
 $0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4, 6 \equiv 8, 3 \equiv 5, 2 \equiv 11, 11 \equiv 0$
 $\{0, 2, 4, 7, 11\}, \{1, 3, 5\}, \{6, 8, 9, 10\}$

EXAMPLE

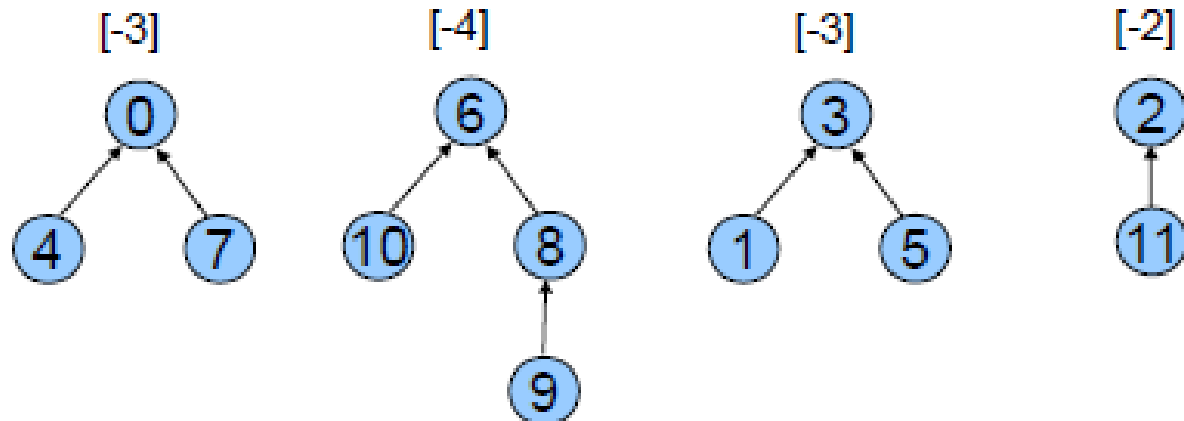


(a) Initial trees

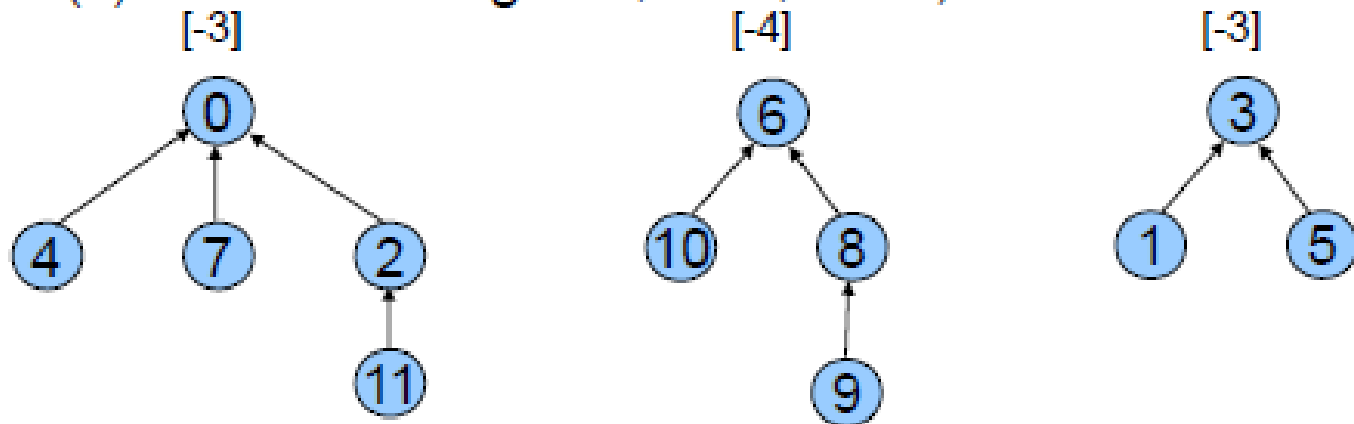


(b) Height-2 trees following $0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, and $8 \equiv 9$

EXAMPLE (CONT'D)



(c) Trees following $7 \equiv 4$, $6 \equiv 8$, $3 \equiv 5$, and $2 \equiv 11$



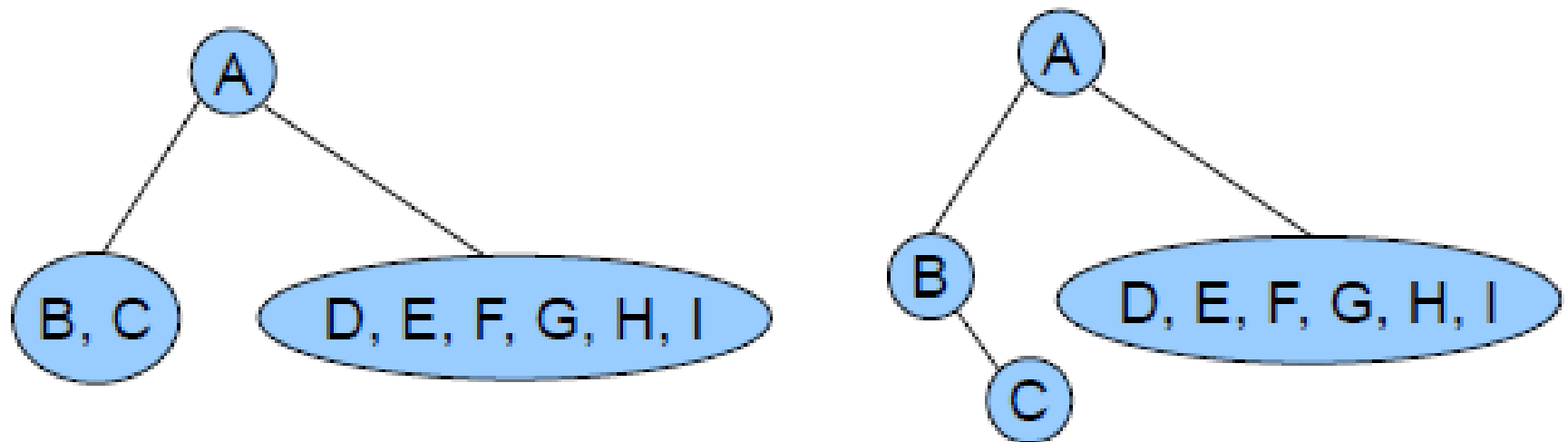
(d) Trees following $11 \equiv 0$



UNIQUENESS OF A BINARY TREE

- Suppose that we have the preorder sequence ABCDEFGHI and the inorder sequence BCAEDGHI of the same binary tree.
- Does such a pair of sequence uniquely define a binary tree?
 - Yes.
 - How to prove it?

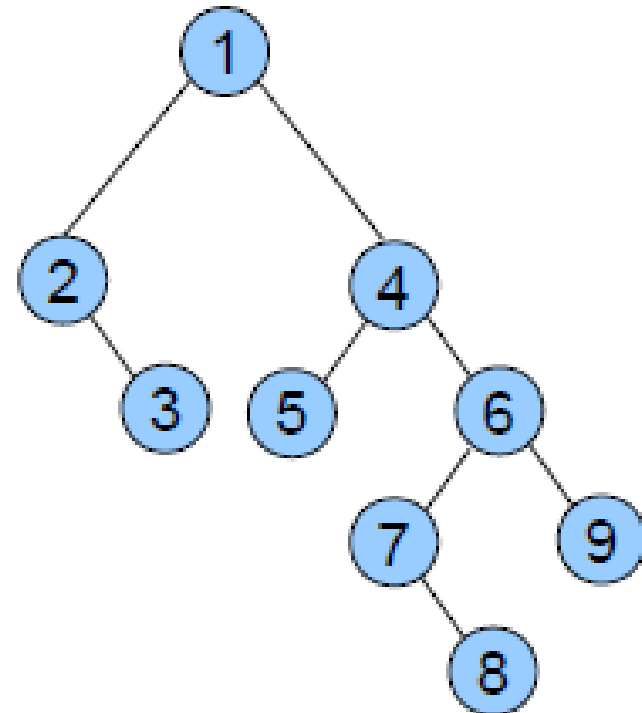
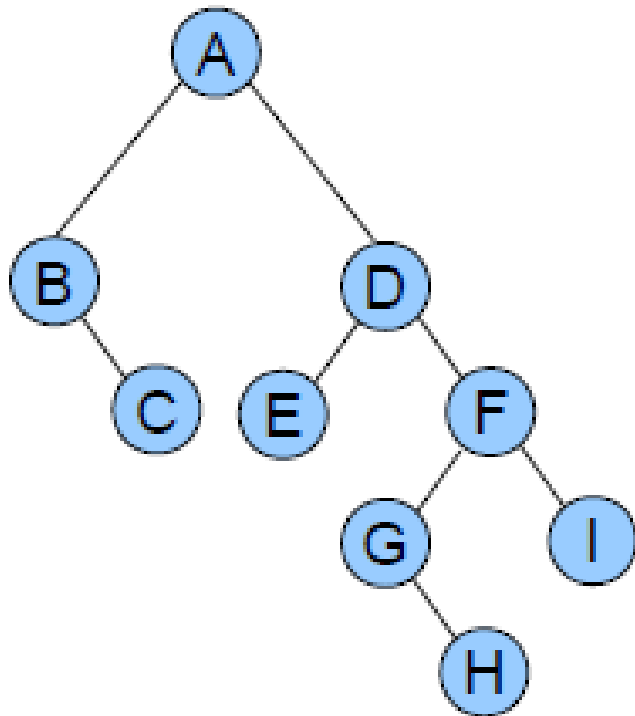
CONSTRUCTING A BINARY TREE FROM ITS PREORDER AND INORDER SEQUENCES



Preorder: **A**BCDEFGHI

Inorder: B**C**AEDGFHI

CONSTRUCTING A BINARY TREE FROM ITS PREORDER AND INORDER SEQUENCES (CONT'D)

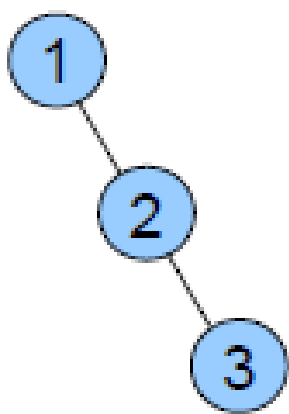


Preorder: 1, 2, 3, 4, 5, 6, 7, 8, 9

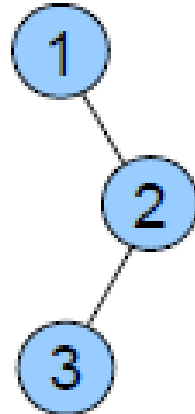
Inorder: 2, 3, 1, 5, 4, 7, 8, 6, 9

DISTINCT BINARY TREES

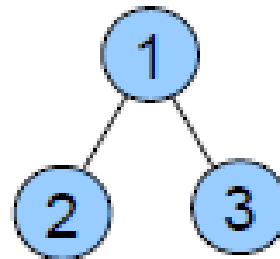
preorder: (1, 2, 3)



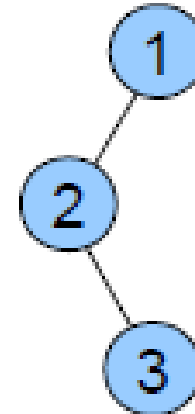
(1, 2, 3)



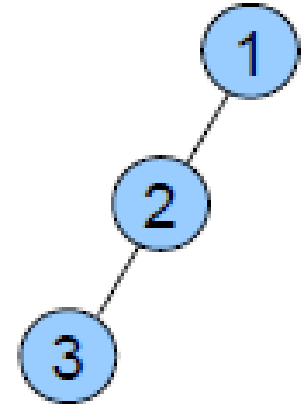
(1, 3, 2)



(2, 1, 3)



(2, 3, 1)



(3, 2, 1)

inorder