

Chapter 5

Trees

Yi-Fen Liu

Department of IECS, FCU

References:

- E. Horowitz, S. Sahni and S. Anderson-Freed, *Fundamentals of Data Structures (2nd Edition)*
- Slides are credited from Prof. Chung, NTHU

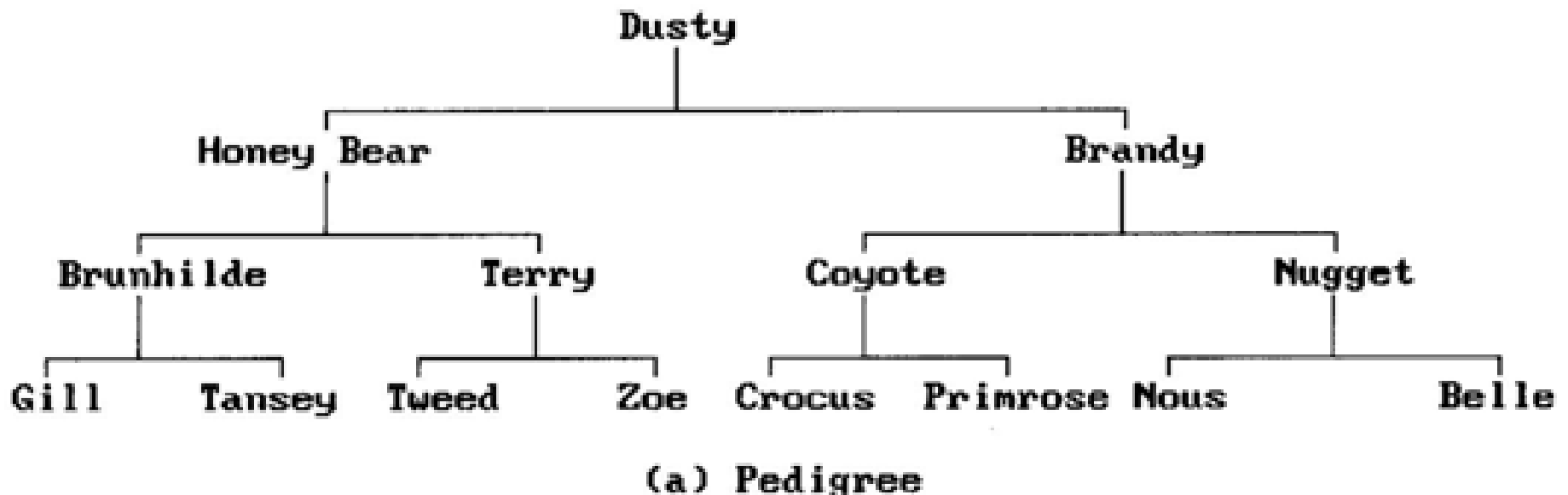
Outline

- Introduction
- Binary Trees
- Binary Tree Traversals
- Additional Binary Tree Operations
- Threaded Binary Tree
- Heaps
- Binary Search Trees
- Selection Trees
- Forest

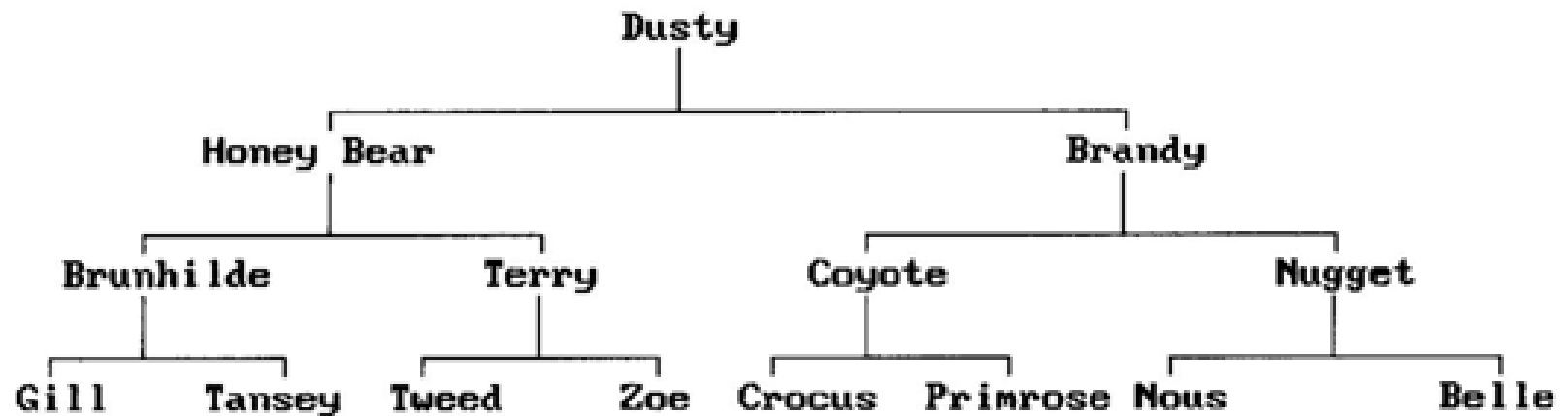
INTRODUCTION

Introduction (1)

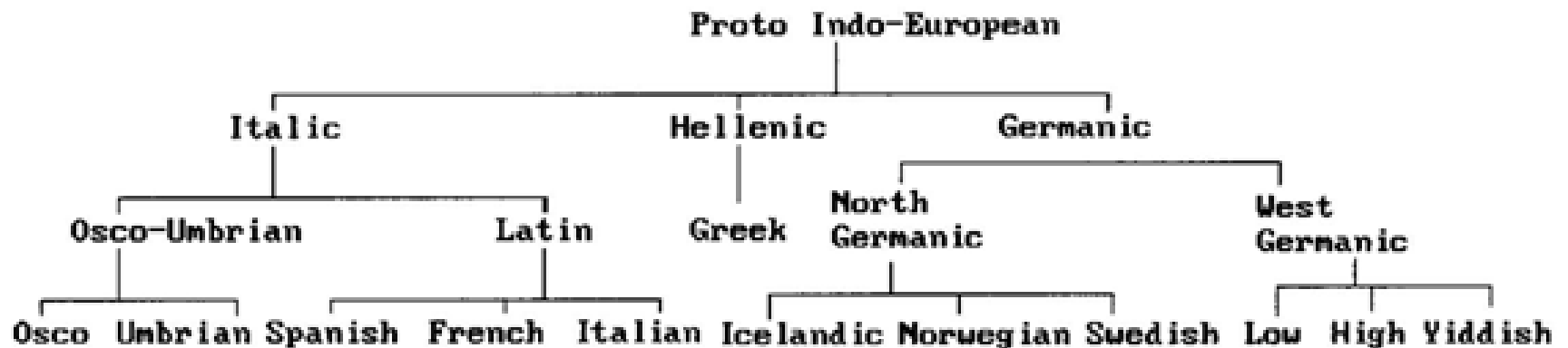
- A tree structure means that the data are organized so that items of information are related by branches



Introduction (2)



(a) Pedigree

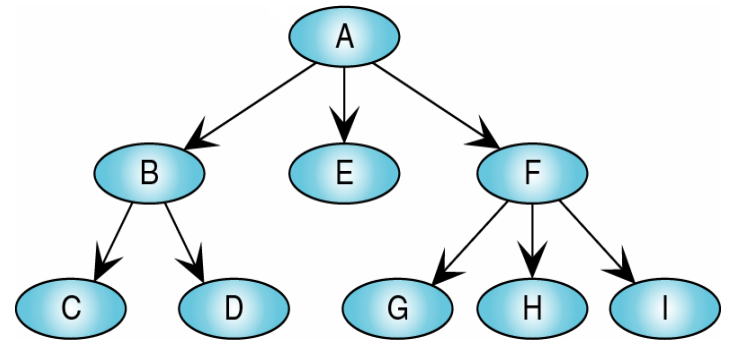


(b) Lineal

Introduction (3)

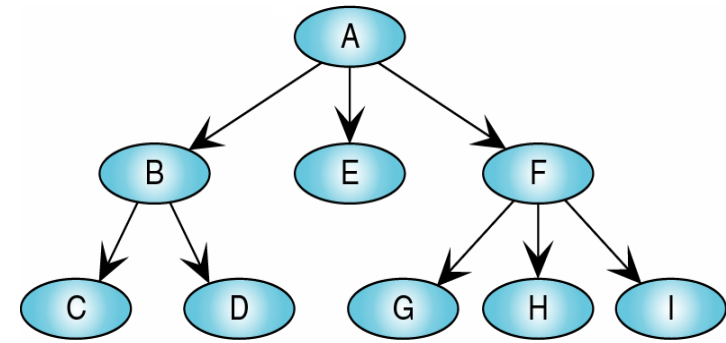
- **Definition** (recursively): A tree is a finite set of one or more nodes such that
 - There is a specially designated node called **root**
 - The remaining nodes are partitioned into $n \geq 0$ disjoint set T_1, \dots, T_n , where each of these sets is a tree
 - T_1, \dots, T_n are called the subtrees of the root
- Every node in the tree is the root of some subtree

Introduction (4)



- Some Terminology
 - *Node*: the item of information plus the branches to each node
 - *Degree*: the number of subtrees of a node
 - *Degree of a tree*: the maximum of the degree of the nodes in the tree
 - *Terminal nodes (or leaf)*: nodes that have degree zero
 - *Nonterminal nodes*: nodes that don't belong to terminal nodes
 - *Children*: the roots of the subtrees of a node X are the children of X
 - *Parent*: X is the parent of its children.

Introduction (5)



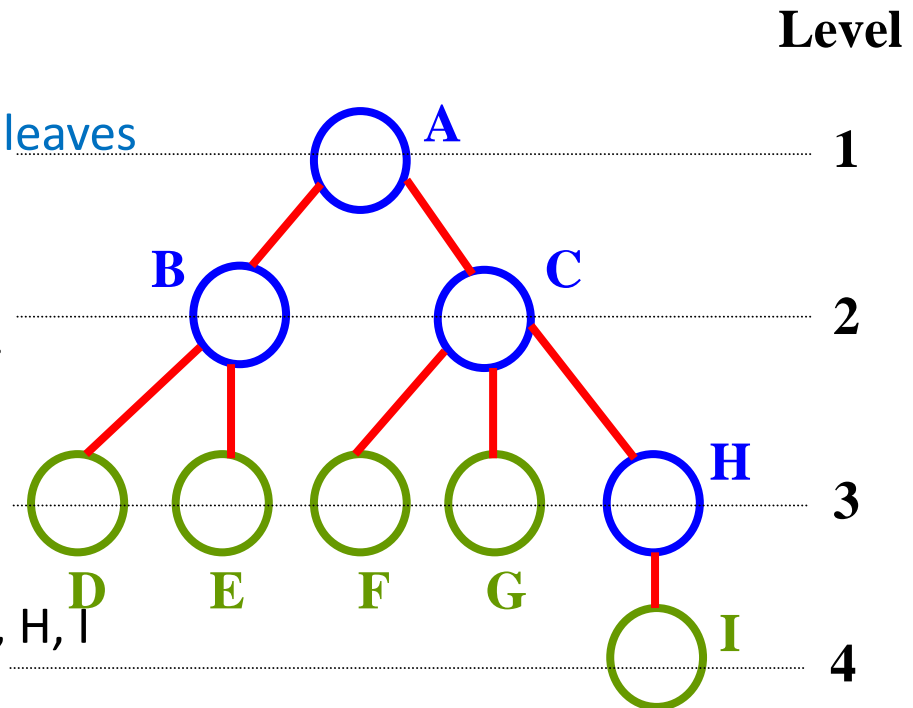
- Some Terminology (cont'd)
 - *Siblings*: children of the same parent are said to be siblings
 - *Ancestors* of a node: all the nodes along the path from the root to that node
 - The *level* of a node: defined by letting the root be at level one
 - If a node is at level l , then its children are at level $l+1$
 - *Height* (or *depth*): the maximum level of any node in the tree

Introduction (6)

- Example

- A is the **root** node
- B is the **parent** of D and E
- C is the **sibling** of B
- D and E are the **children** of B
- D, E, F, G, I are **terminal nodes**, or **leaves**
- A, B, C, H are **internal** nodes
- The **level** of E is 3
- The **height (depth)** of the tree is 4
- The **degree** of node B is 2
- The **degree** of the tree is 3
- The **ancestors** of node I is A, C, H
- The **descendants** of node C is F, G, H, I

Property: (# edges) = (#nodes) - 1

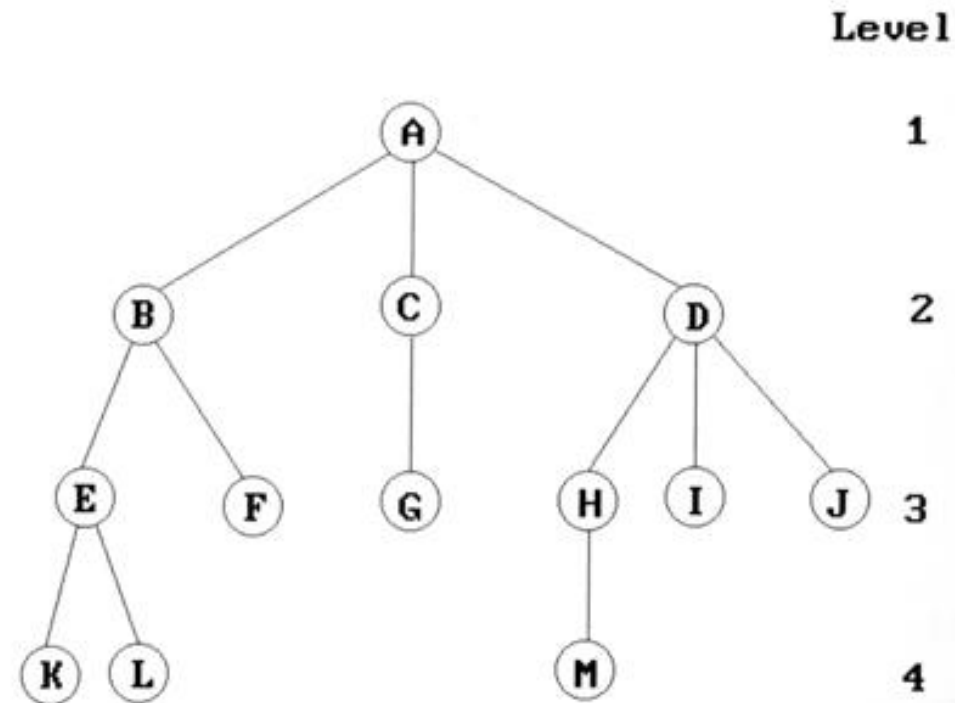


Introduction (7)

- Representation Of Trees
 - List Representation
 - We can write the figure as a list in which each of the subtrees is also a list
 - (A (B (E (K, L), F), C (G), D (H (M), I, J)))
 - The root comes first, followed by a list of sub-trees

<i>data</i>	<i>link 1</i>	<i>link 2</i>	<i>...</i>	<i>link n</i>
-------------	---------------	---------------	------------	---------------

Possible list representation for trees



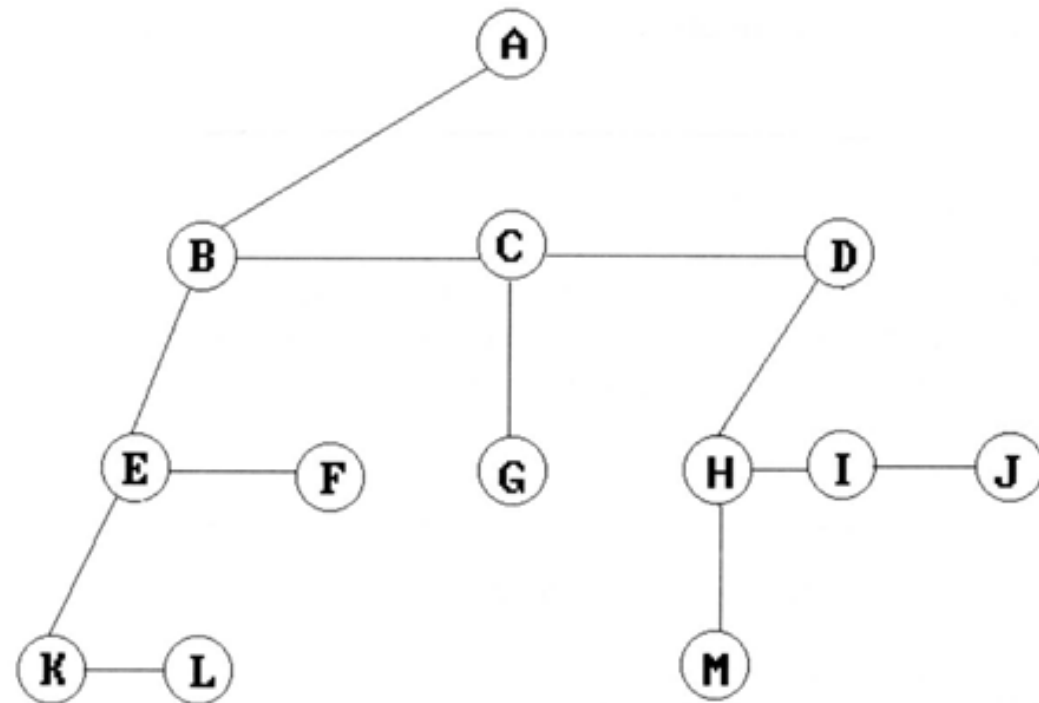
A sample tree

Introduction (8)

- Representation Of Trees (cont'd)
 - Left Child-Right Sibling Representation

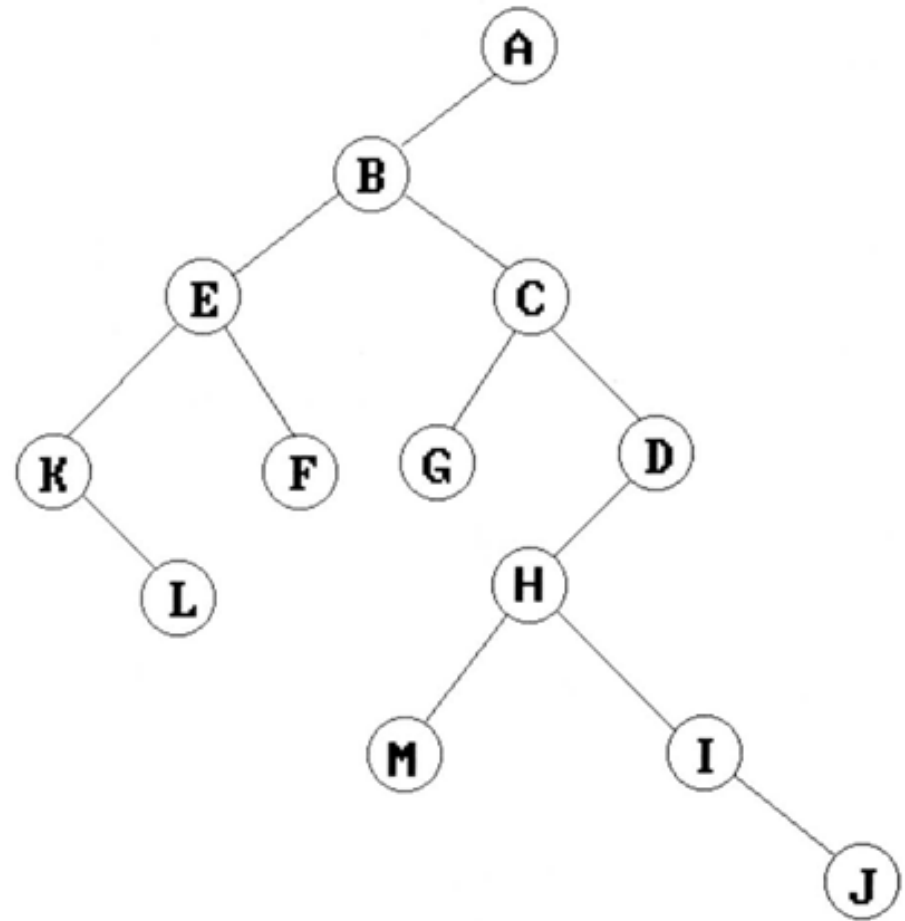
data	
left child	right sibling

Left child-right sibling node structure



Introduction (9)

- Representation Of Trees (cont'd)
 - Representation as a degree-two tree

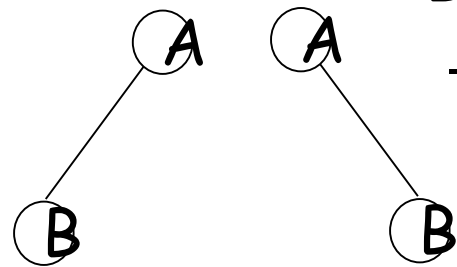


Left child-right child tree representation of a tree

BINARY TREES

Binary Trees (1)

- *Binary trees* are characterized by the fact that any node can have at most two branches
- Definition (recursive)
 - 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*
- Thus the left subtree and the right subtree are distinguished
- Any tree can be transformed into binary tree
 - by *left child-right sibling representation*



Binary Trees (2)

- The abstract data type of binary tree
-

structure *Binary_Tree* (abbreviated *BinTree*) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

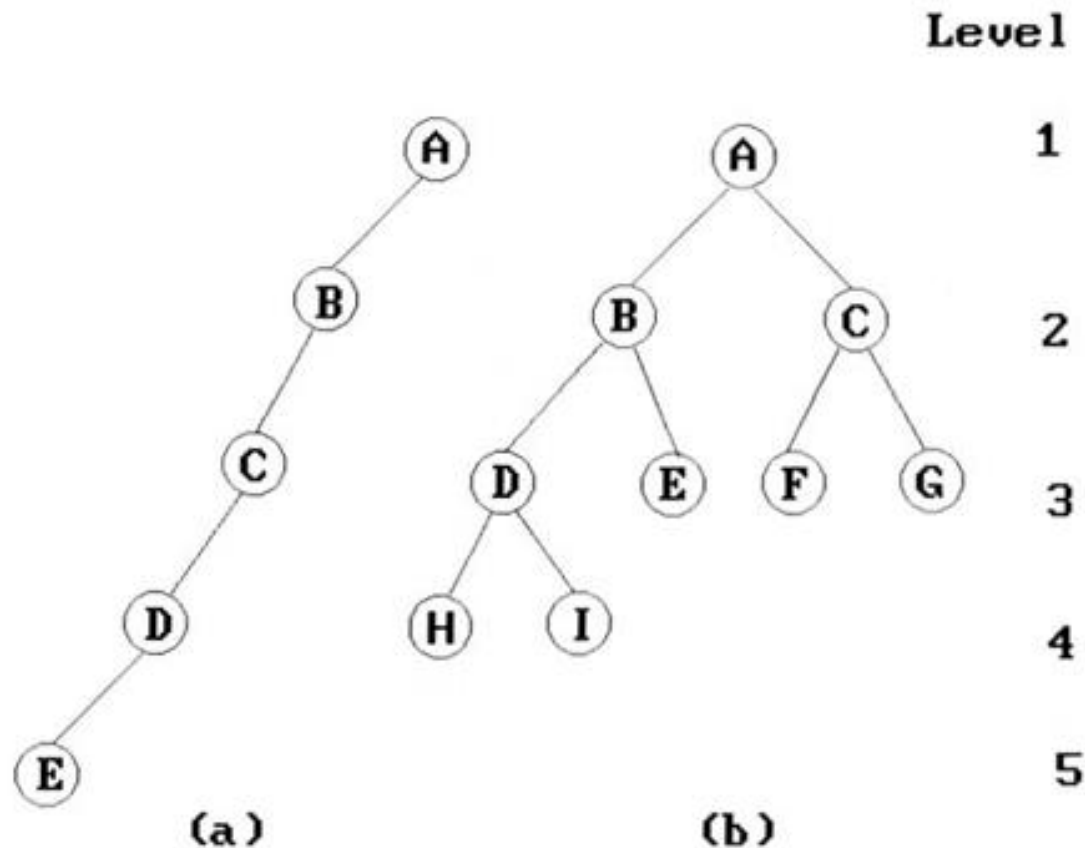
functions:

for all $bt, bt1, bt2 \in \text{BinTree}$, $item \in \text{element}$

<i>BinTree</i> Create()	::=	creates an empty binary tree
<i>Boolean</i> IsEmpty(<i>bt</i>)	::=	if (<i>bt</i> == empty binary tree) return <i>TRUE</i> else return <i>FALSE</i>
<i>BinTree</i> MakeBT(<i>bt1</i> , <i>item</i> , <i>bt2</i>)	::=	return a binary tree whose left subtree is <i>bt1</i> , whose right subtree is <i>bt2</i> , and whose root node contains the data <i>item</i> .
<i>BinTree</i> Lchild(<i>bt</i>)	::=	if (IsEmpty(<i>bt</i>)) return error else return the left subtree of <i>bt</i> .
<i>element</i> Data(<i>bt</i>)	::=	if (IsEmpty(<i>bt</i>)) return error else return the data in the root node of <i>bt</i> .
<i>BinTree</i> Rchild(<i>bt</i>)	::=	if (IsEmpty(<i>bt</i>)) return error else return the right subtree of <i>bt</i> .

Binary Trees (3)

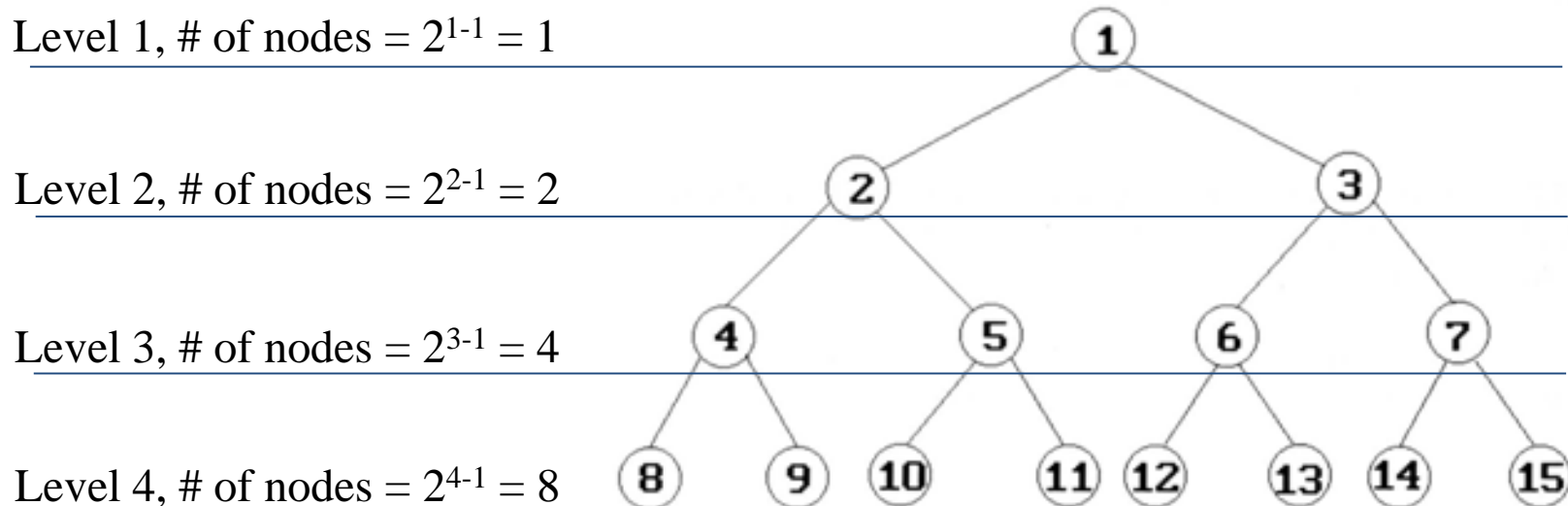
- Two special kinds of binary trees
 - skewed tree
 - complete binary tree
 - The all leaf nodes of these trees are on two adjacent levels



Skewed and complete binary trees

Binary Trees (4)

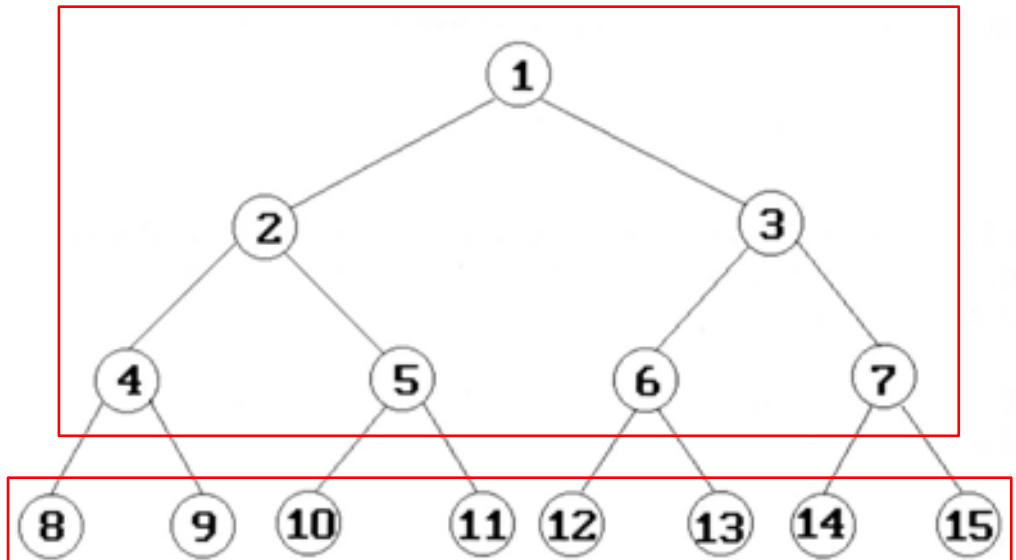
- Properties of binary trees
 - **Lemma** [Maximum number of nodes]
 - The maximum number of nodes on level i of a binary tree is 2^{i-1} , $i \geq 1$.
 - The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.



Binary Trees (5)

- **Lemma** [Relation between number of leaf nodes and degree-2 nodes]
 - For any nonempty binary tree, T , if n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2, then $n_0 = n_2 + 1$
- These lemmas allow us to define full and complete binary trees

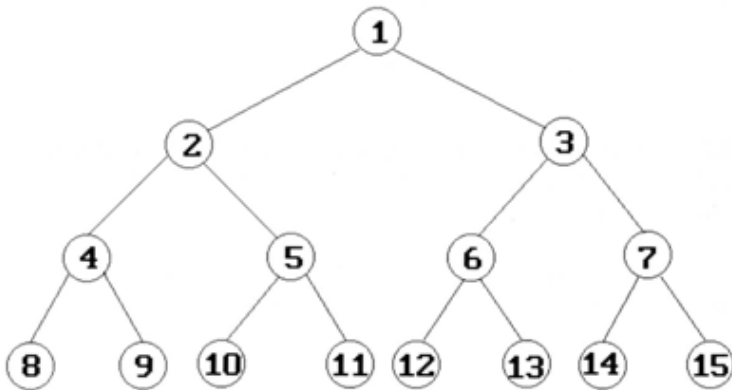
n_2



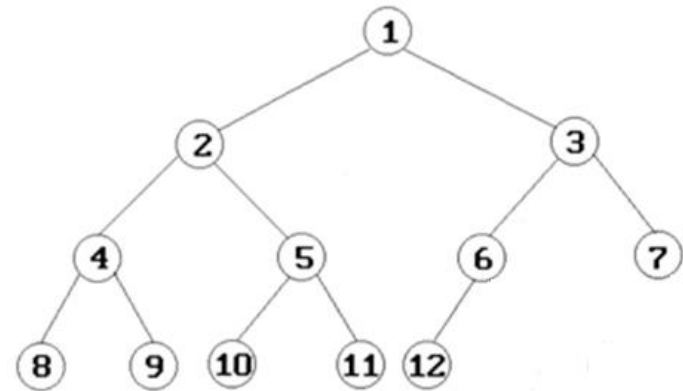
n_0

Binary Trees (6)

- A *full binary tree* of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$ (所有的 internal nodes 的 out degree 都是 2, 且所有的 external nodes 都在同一 level)
- 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 (最下一層也就是第 k 層的所有 nodes 都盡量擠在最左邊)
- **Lemma:** The height of a *complete binary tree* with n nodes is $\lceil \log_2(n+1) \rceil$



Full Binary Tree

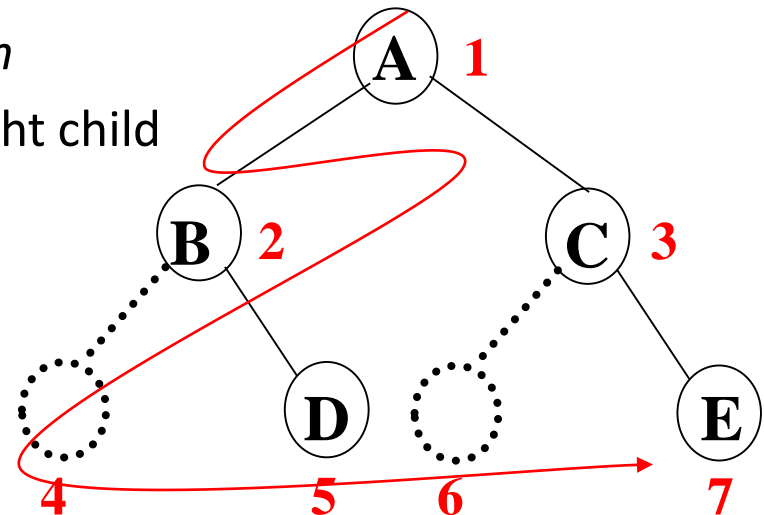
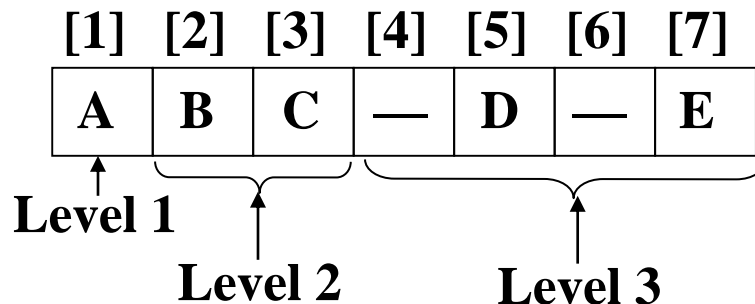


Complete Binary Tree

Binary Tree Representations

– Using Array (1)

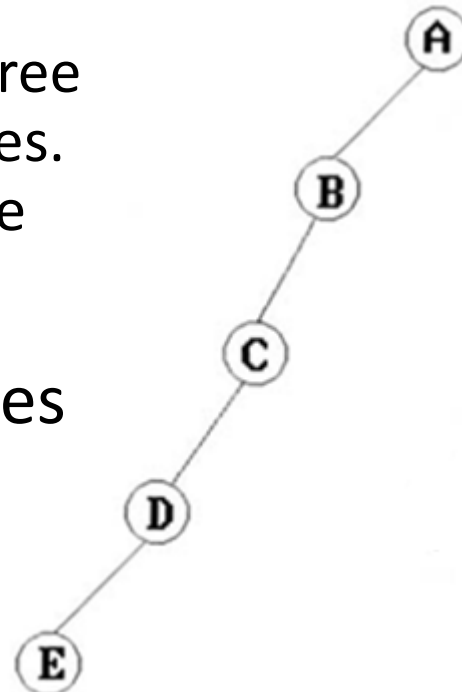
- **Lemma:** If a *complete binary tree* with n nodes is represented sequentially, then for any node with index i , $1 \leq i \leq n$, we have
 - *Parent(i)* is at $\lfloor i/2 \rfloor$ if $i \neq 1$
 - If $i = 1$, i is at the root and has no parent
 - *LeftChild(i)* is at $2i$ if $2i \leq n$
 - If $2i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



Binary Tree Representations

– Using Array (2)

- Waste spaces
 - In the worst case, a skewed tree of depth k requires $2^k - 1$ spaces. Of these, only k spaces will be occupied
- Insertion or deletion of nodes from the middle of a tree requires the movement of potentially many nodes to reflect the change in the level of these nodes

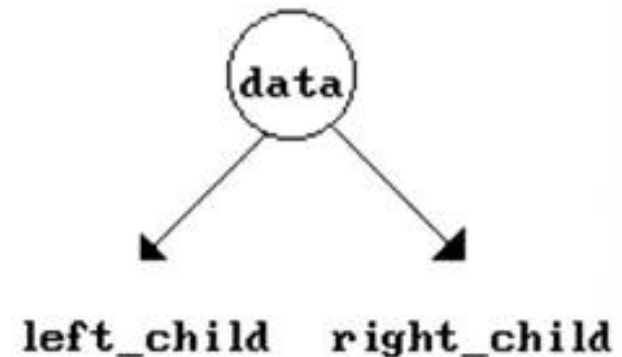
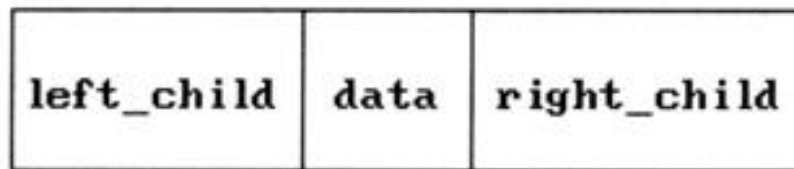


[1]	A
[2]	B
[3]	—
[4]	C
[5]	—
[6]	—
[7]	—
[8]	D
[9]	—
⋮	⋮
⋮	⋮
⋮	⋮
[16]	E

Binary Tree Representations

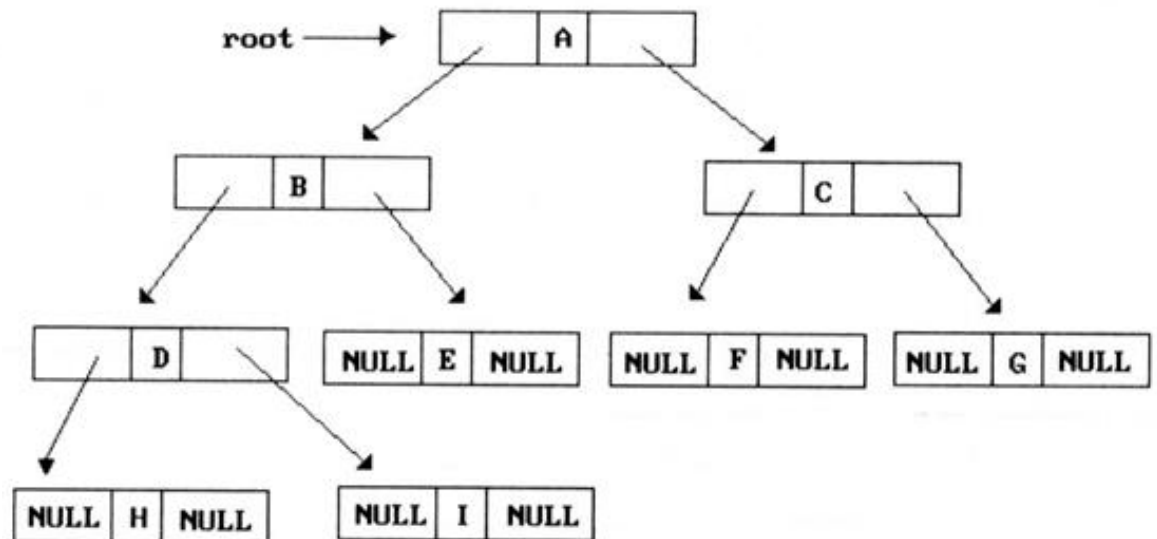
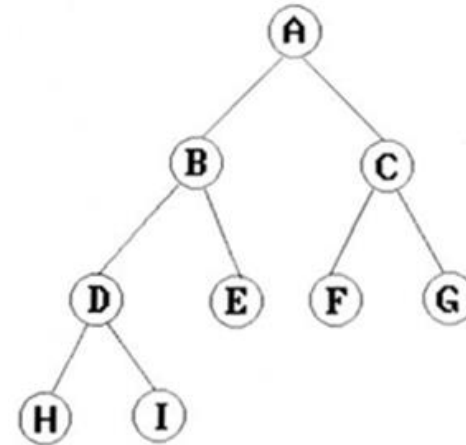
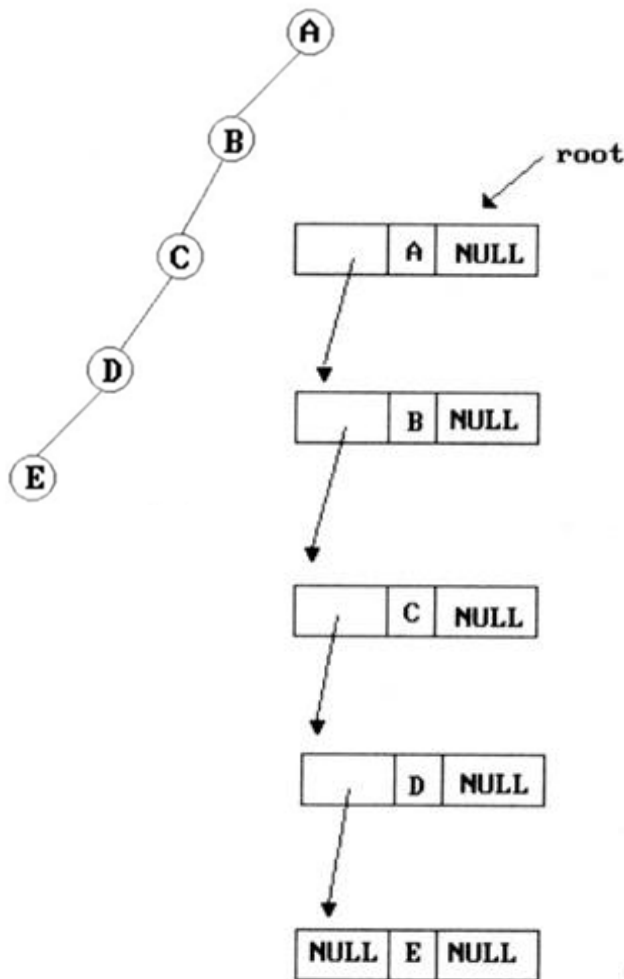
– Using Link (1)

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



Binary Tree Representations

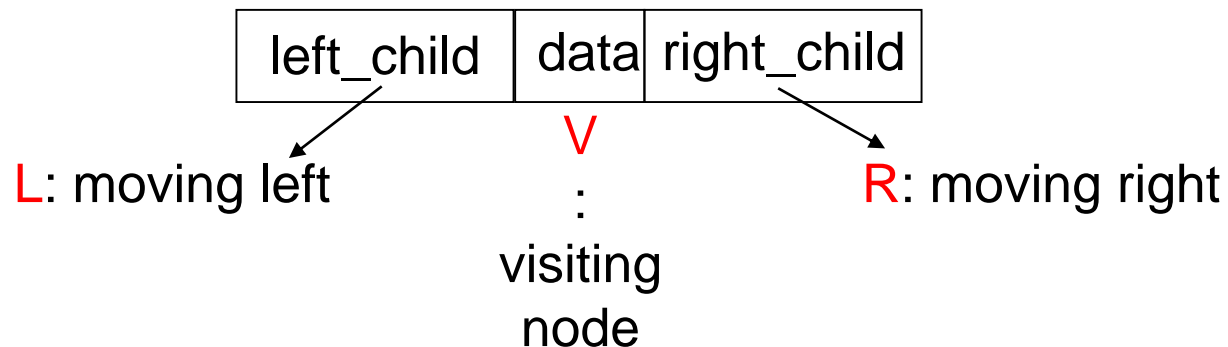
– Using Link (2)



BINARY TREE TRAVERSALS

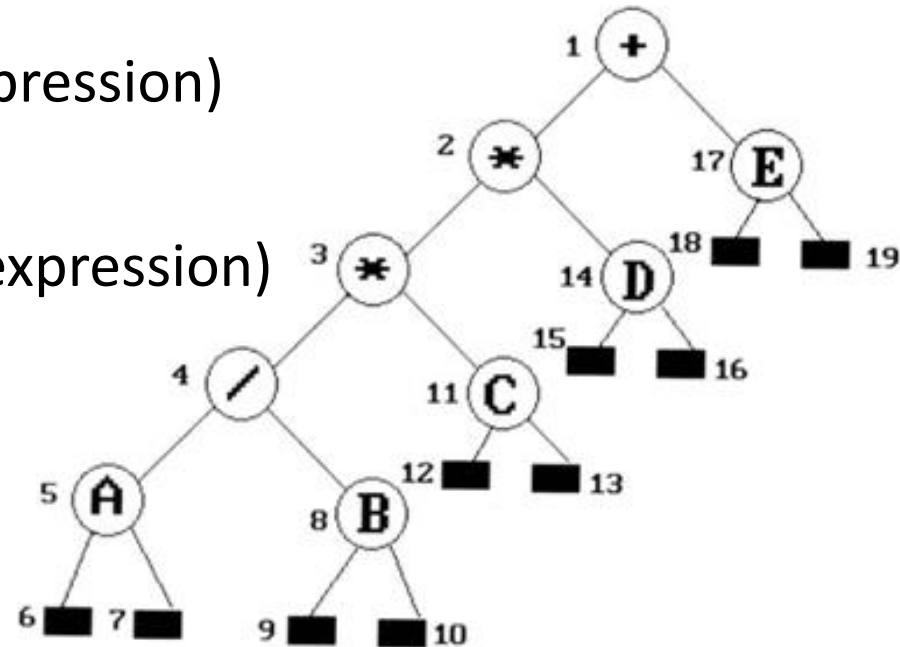
Binary Tree Traversals (1)

- How to traverse a tree or visit each node in the tree exactly once?
 - 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
 - **L**V**R** (**i**norder), **L**R**V** (**p**ostorder), **V**L**R** (**p**reorder)



Binary Tree Traversals (2)

- Arithmetic Expression using binary tree
 - Inorder traversal (infix expression)
 - $A / B * C * D + E$
 - Preorder traversal (prefix expression)
 - $+ * * / A B C D E$
 - Postorder traversal (postfix expression)
 - $A B / C * D * E +$
 - Level order traversal
 - $+ * E * D / C A B$

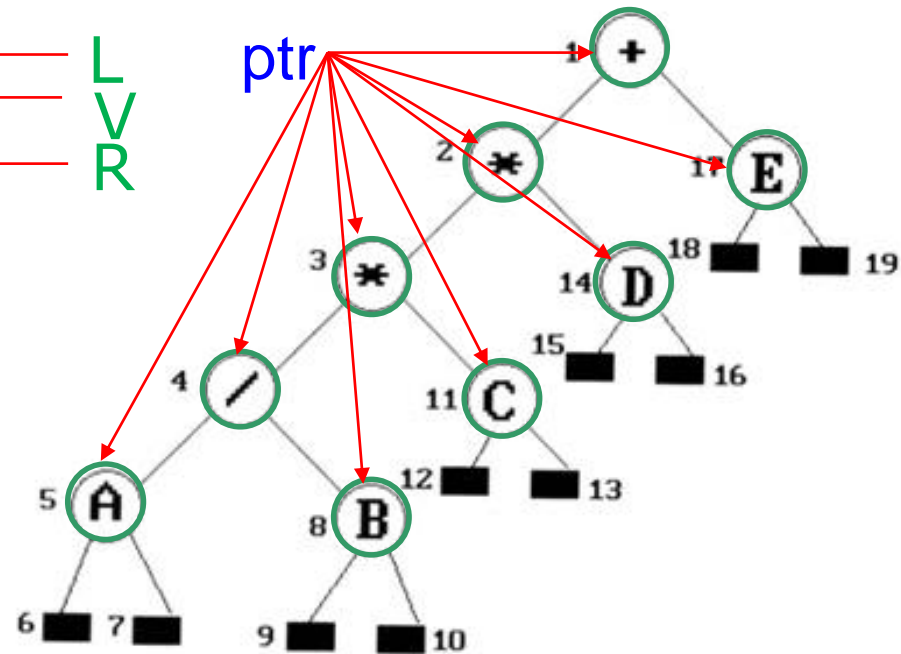


Binary Tree Traversals (3)

- Inorder traversal (*LVR*) (recursive version)

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

output: $A / B^* C^* D + E$



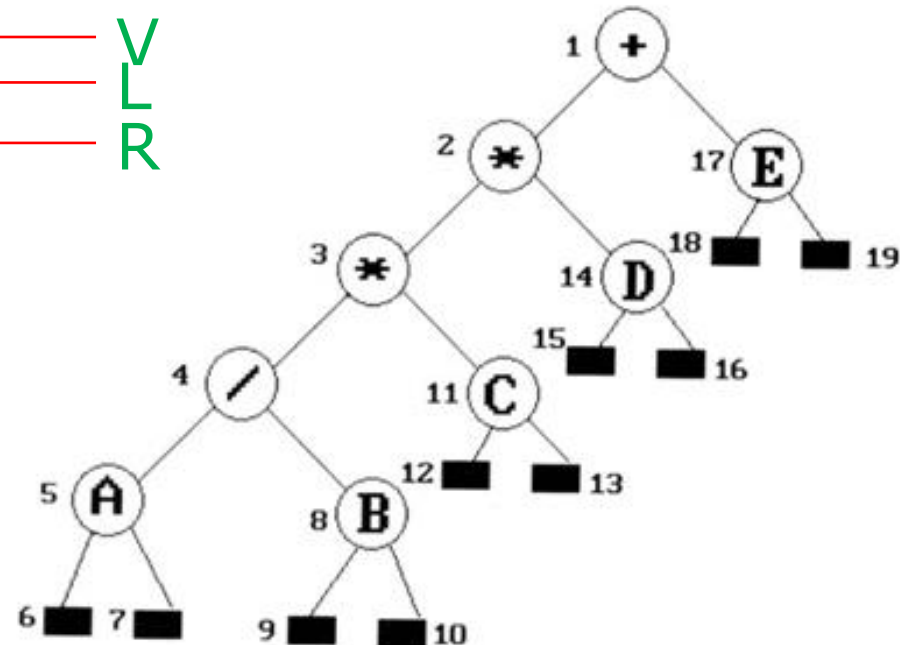
Inorder traversal of a binary tree

Binary Tree Traversals (4)

- Preorder traversal (VLR) (recursive version)

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

output: + * * / A B C D E



Preorder traversal of a binary tree

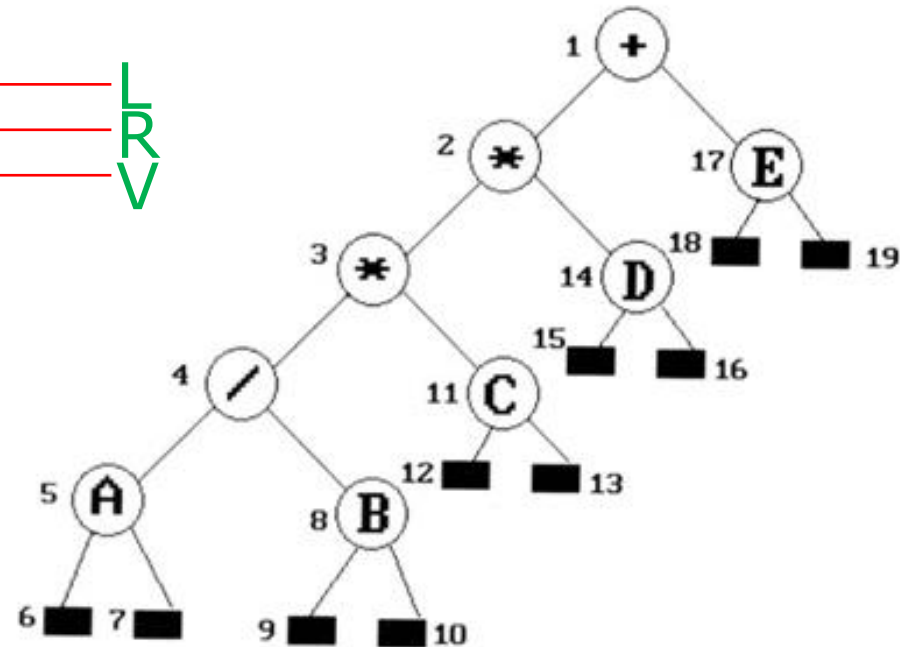
Binary Tree Traversals (5)

- Postorder traversal (*LRV*) (recursive version)

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

output: A B / C * D * E +

Postorder traversal of a binary tree

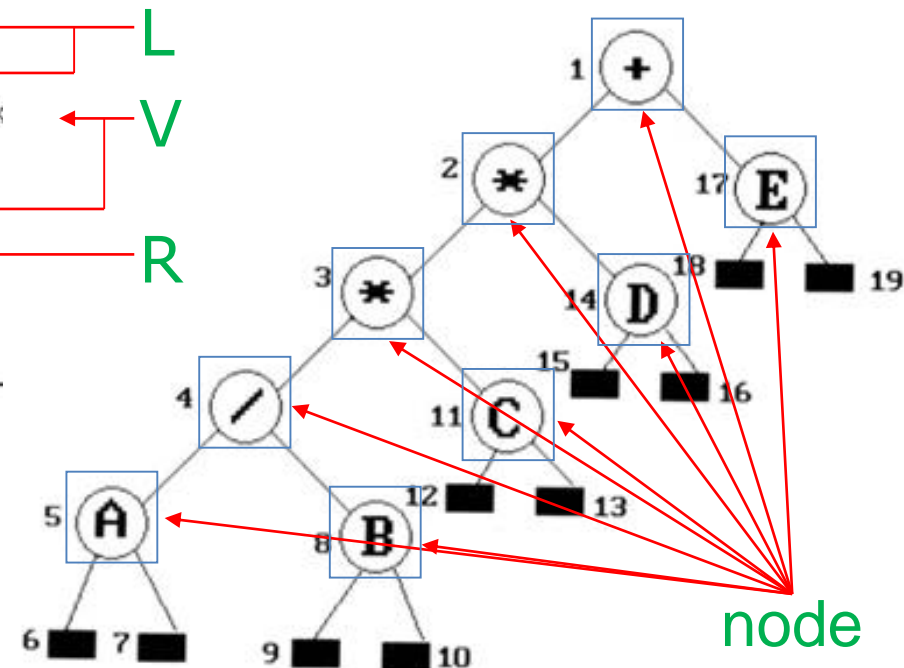


Binary Tree Traversals (6)

- Iterative inorder traversal
 - we use a **stack** to simulate recursion

```
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;
    }
}
```

5	8	11	14	17
A	B	C	D	E



Iterative inorder traversal

output: A / B * C * D + E

Binary Tree Traversals (7)

- Analysis of inorder (Non-recursive Inorder traversal)
 - Let n be the number of nodes in the tree
 - Time complexity: $O(n)$
 - Every node of the tree is placed on and removed from the stack exactly once
 - Space complexity: $O(n)$
 - Equal to the depth of the tree which (skewed tree is the worst case)

Level-Order Traversal

– Using Queue (1)

- Level-order traversal
 - Method
 - We visit the root first, then the root's left child, followed by the root's right child.
 - We continue in this manner, visiting the nodes at each new level from the leftmost node to the rightmost nodes
 - This traversal requires a **queue** to implement

Level-Order Traversal

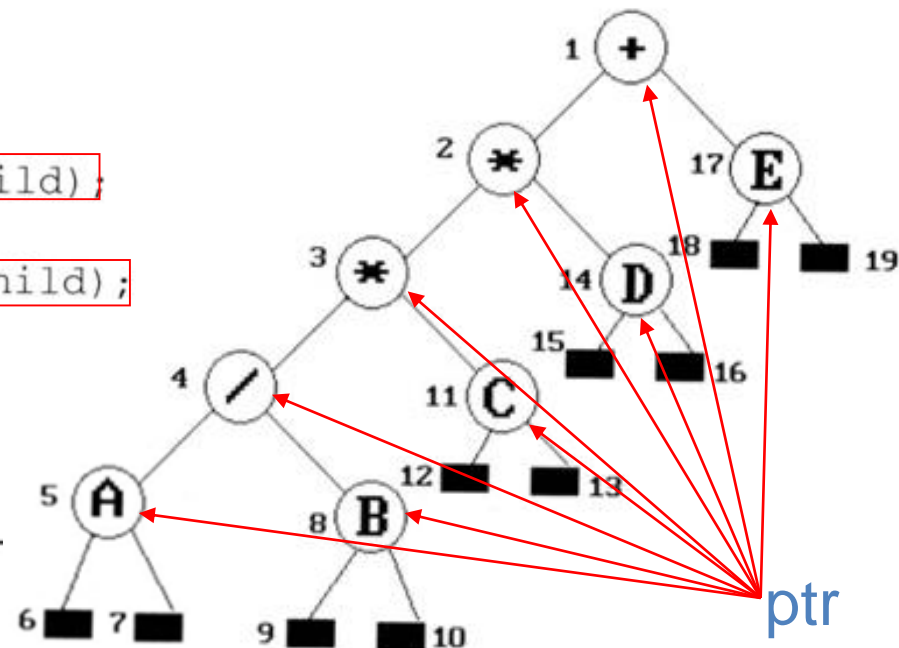
– Using Queue (2)

```
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;
    }
}
```

FIFO

output: + * E * D / C A B

2	17	3	14	4	11	5	8	
*	E	*	D	/	C	A	B	



Level order traversal of a binary tree

ADDITIONAL BINARY TREE OPERATIONS

Additional Binary Tree Operations

– Copying Binary Trees

- Modify the **postorder traversal** algorithm only slightly to copy the binary tree

```
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;
}
```

Additional Binary Tree Operations

– Testing Equality

- Binary trees are equivalent if they have the **same topology** and the **information** in corresponding nodes is identical

```
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 &&
V  —————> (first->data == second->data) &&
L  —————equal> first->left_child, second->left_child) &&
R  —————equal> first->right_child, second->right_child))
}
```

Testing for equality of binary trees

Satisfiability Problem (1)

- Variables x_1, x_2, \dots, x_n can hold only two possible values, *true* or *false*
 - Operators on variables: \wedge (and), \vee (or), \neg (not)
- 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: $x_1 \vee (x_2 \wedge \neg x_3)$

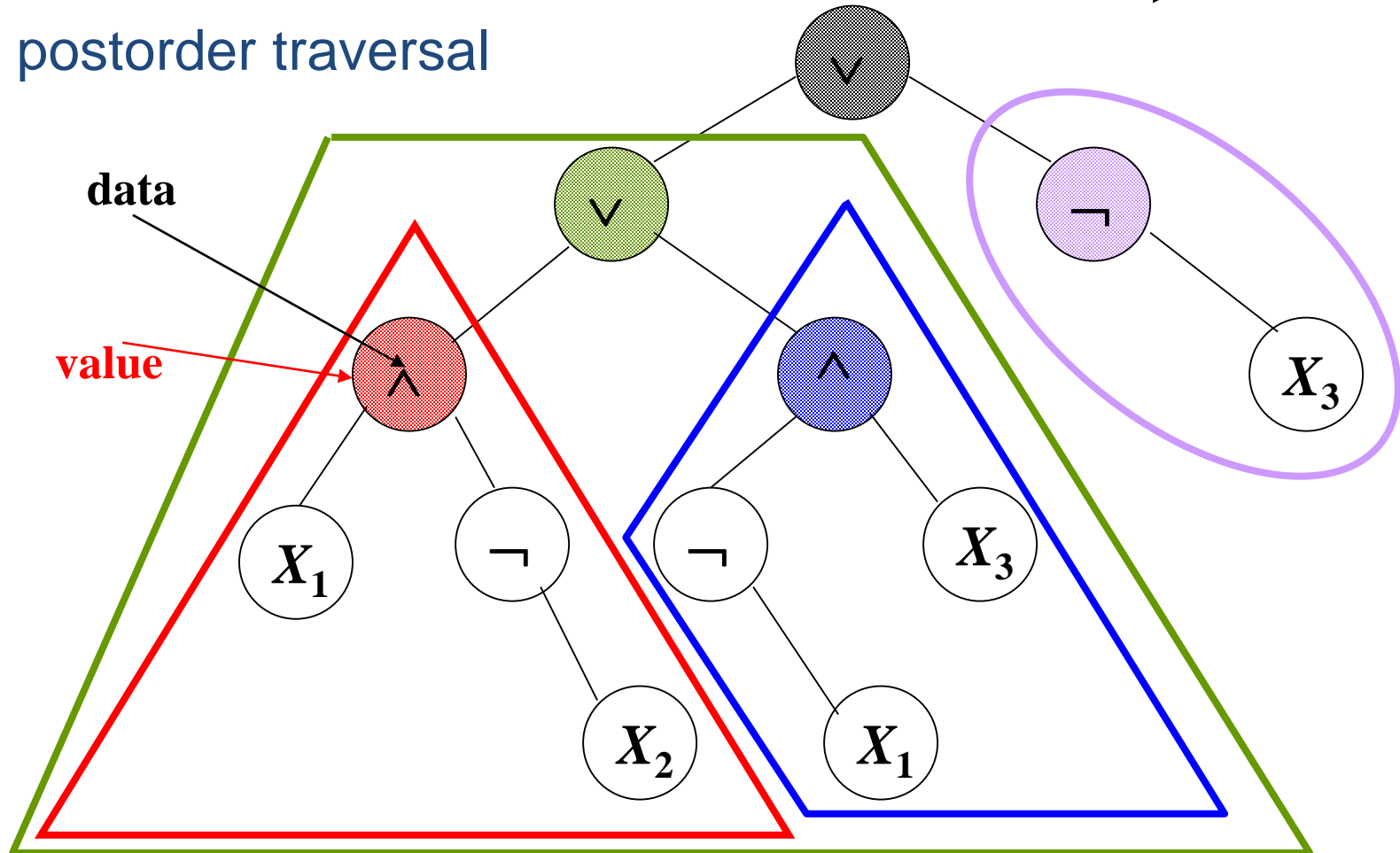
Satisfiability Problem (2)

- Satisfiability problem
 - Is there an assignment to make an expression true?
- Solution for the Example $x_1 \vee (x_2 \wedge \neg x_3)$
 - If x_1 and x_3 are false and x_2 is true
 - $\text{false} \vee (\text{true} \wedge \neg \text{false}) = \text{false} \vee \text{true} = \text{true}$
- For n value of an expression, there are 2^n possible combinations of true and false

Satisfiability Problem (3)

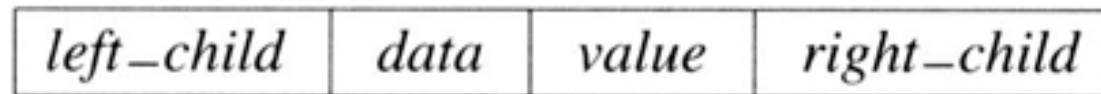
$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

postorder traversal



Satisfiability Problem (4)

- Node structure
 - For the purpose of our evaluation algorithm, we assume each node has four fields
 - We define this node structure in C as:

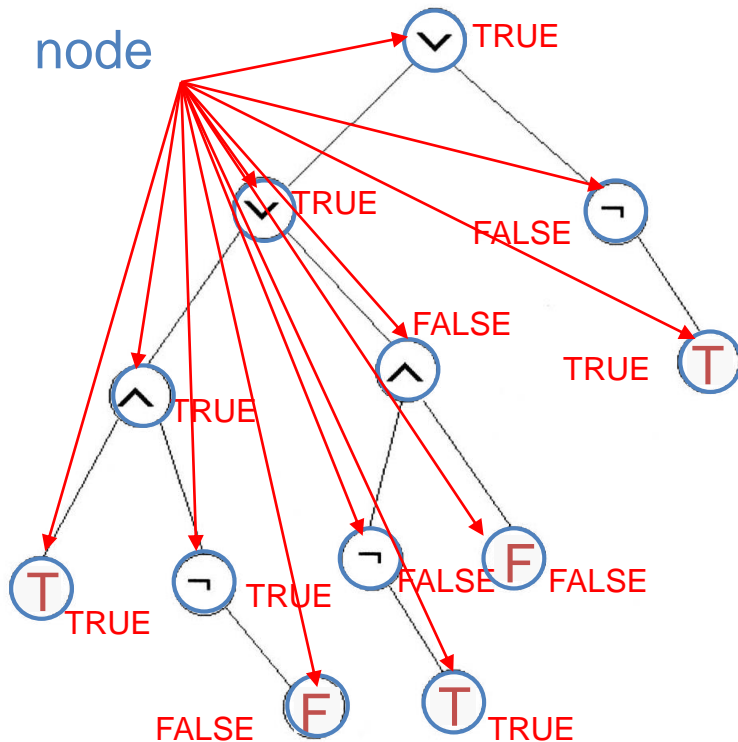


Node structure for the satisfiability problem

```
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;
} ;
```


Satisfiability Problem (5)

- Satisfiability function
 - To evaluate the tree is easily obtained by modifying the original recursive postorder traversal



```
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;
            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;
        }
    }
}
```

post-order-eval function

THREADED BINARY TREES

Threaded Binary Trees (1)

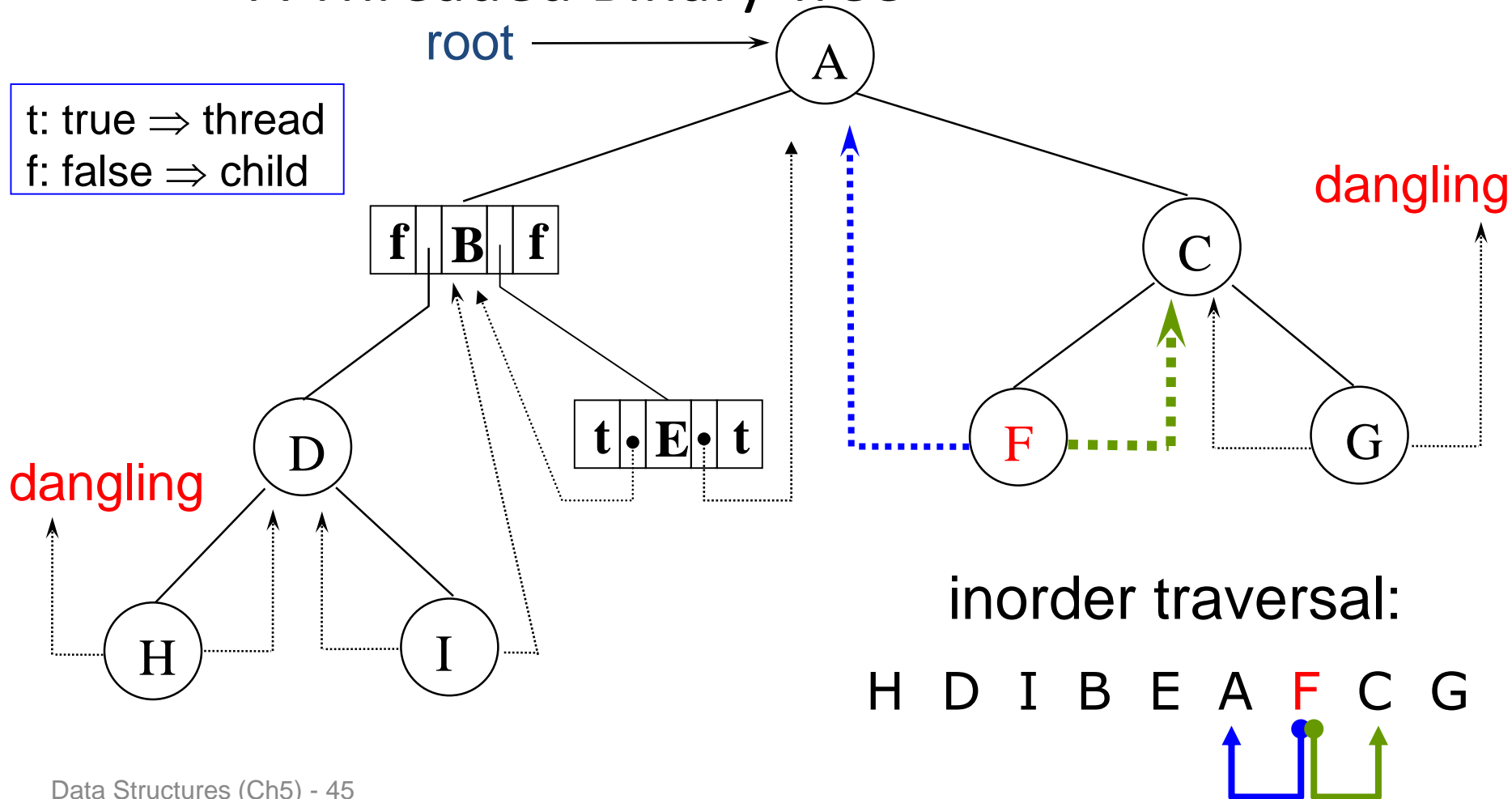
- Threads
 - The drawback of the binary tree
 - Too many null pointers in current representation of binary trees
 - n : number of nodes
 - number of non-null links: $n-1$
 - total links: $2n$
 - null links: $2n-(n-1) = n+1$
 - Solution: replace these null pointers with some useful “threads”

Threaded Binary Trees (2)

- Rules for constructing the threads
 - 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

Threaded Binary Trees (3)

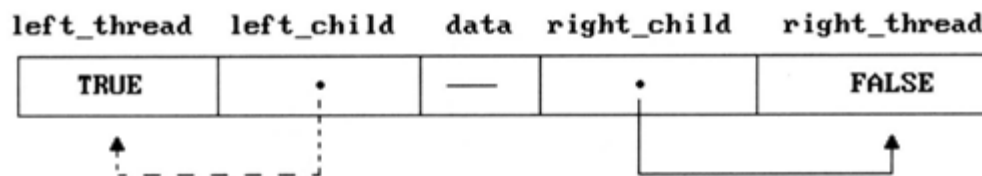
- A Threaded Binary Tree



Threaded Binary Trees (4)

- Two additional fields of the node structure, **left_thread** and **right_thread**
 - If **ptr->left_thread** = TRUE, then **ptr->left_child** contains a **thread**
 - Otherwise it contains a pointer to the left child
 - Similarly for the **right_thread**

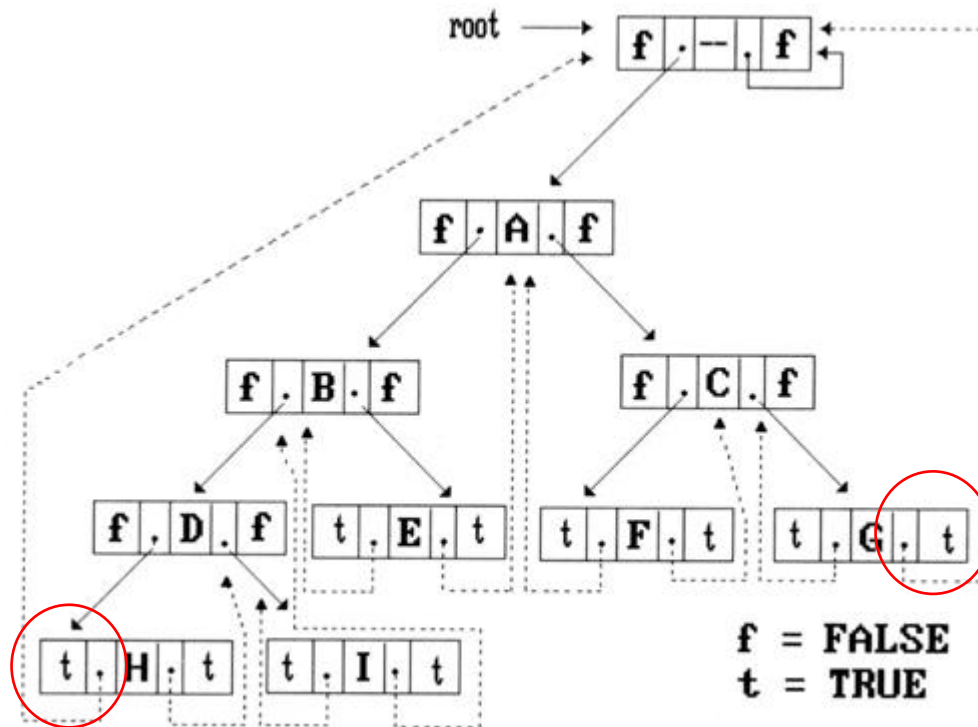
```
typedef struct threaded-tree *threaded_pointer;  
typedef struct threaded-tree {  
    short int left_thread;  
    threaded_pointer left_child;  
    char data;  
    threaded_pointer right_child;  
    short int right_thread;  
};
```



An empty threaded tree

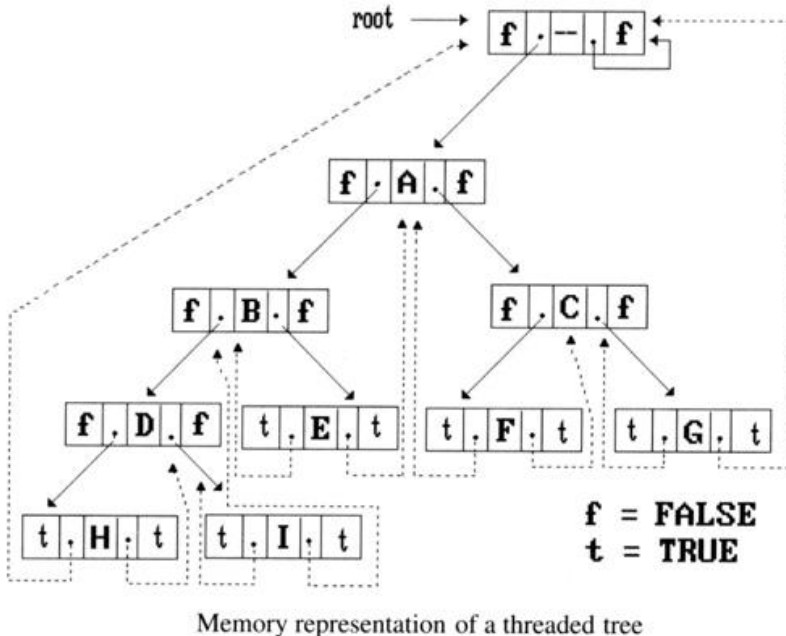
Threaded Binary Trees (5)

- If we don't want the left pointer of H and the right pointer of G to be dangling pointers, we may create root node and assign them pointing to the root node



Threaded Binary Trees (6)

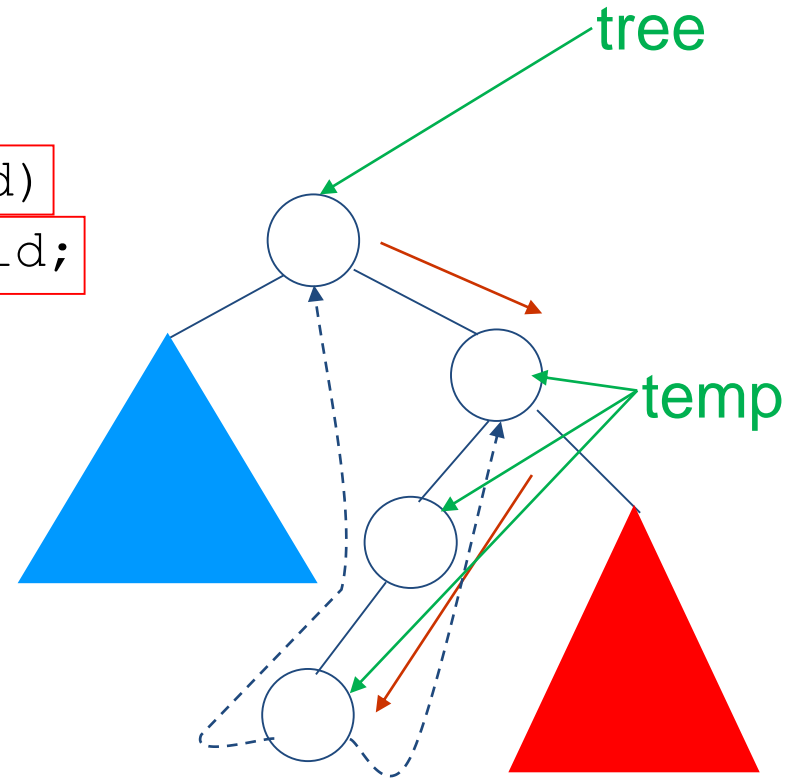
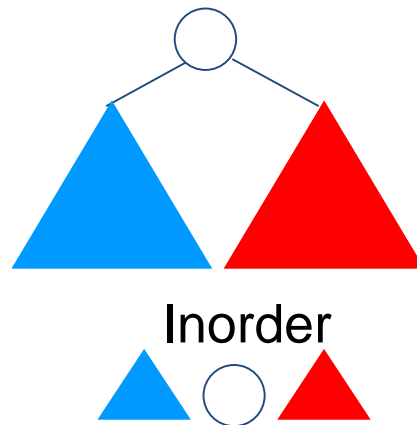
- Inorder traversal of a threaded binary tree
 - By using of threads, we can perform an inorder traversal **without using a stack** (simplifying the task)
 - Now, we can follow the thread of any node, *ptr*, to the “next” node of inorder traversal
 - If `ptr->right_thread = TRUE`, the inorder successor of `ptr` is `ptr->right_child` by definition of the threads
 - Otherwise we obtain the inorder successor of `ptr` by following a path of `left_child` links from the `right_child` of `ptr` until we reach a node with `left_thread = TRUE`



Threaded Binary Trees (7)

- Finding the inorder successor (next node) of a node

```
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;
}
```



Threaded Binary Trees (8)

- Inorder traversal of a threaded binary tree

```
void tinorder(threaded_pointer tree) {  
    /* traverse the threaded binary tree inorder */  
    threaded_pointer temp = tree; output: H D I B E A F C G
```

```
    for (;;) {
```

```
        temp = insucc(temp);
```

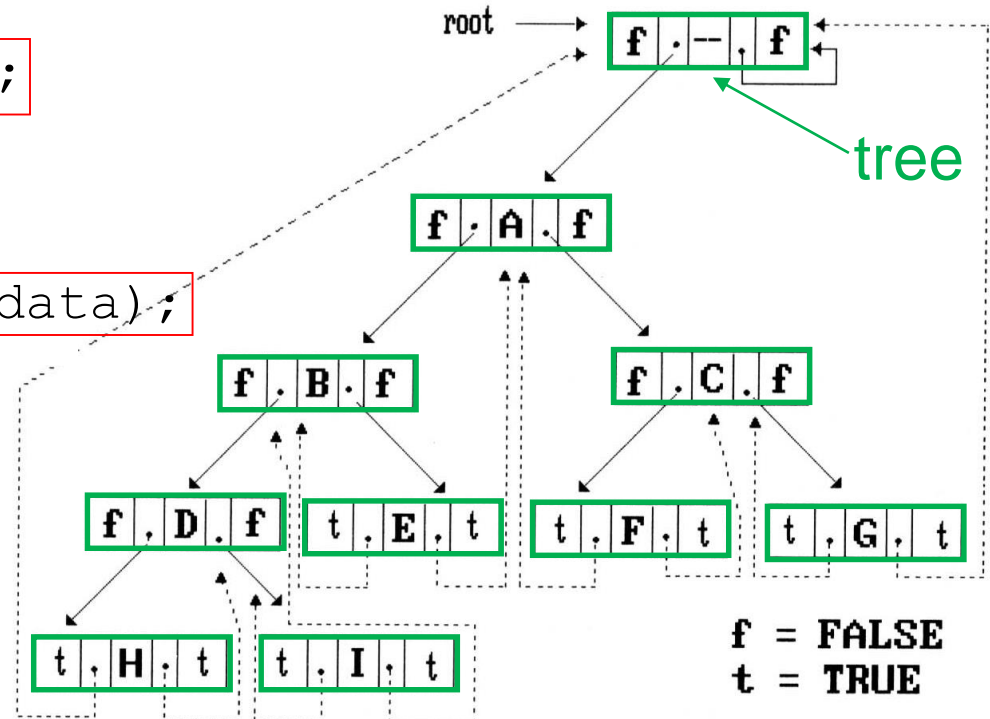
```
        if (temp==tree)
```

```
            break;
```

```
        printf("%3c", temp->data);
```

```
    }
```

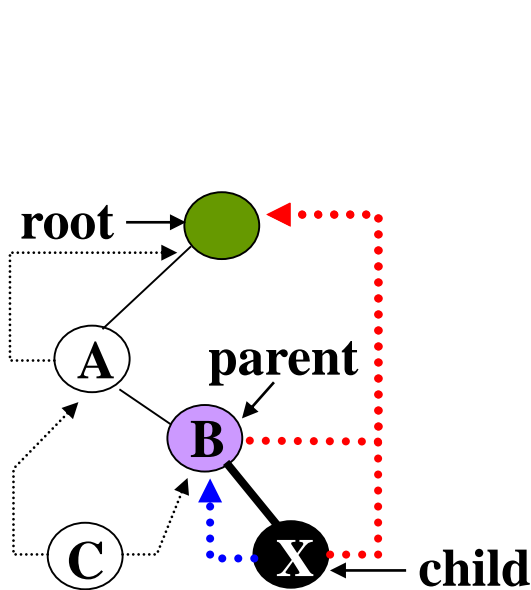
```
}
```



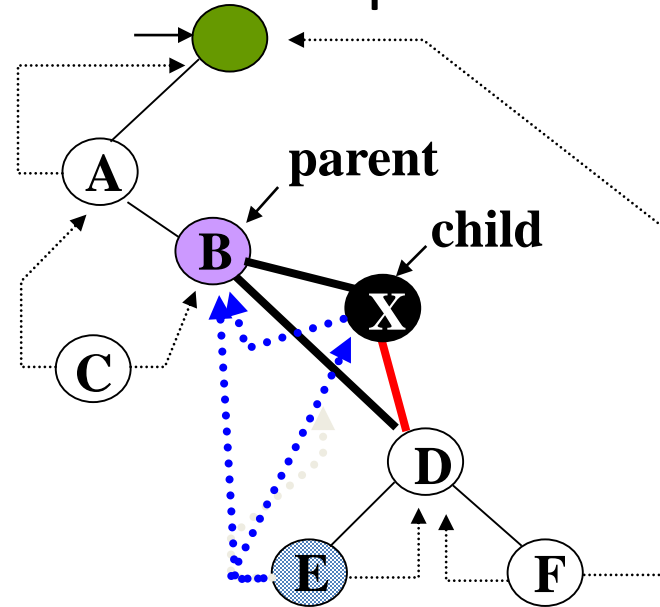
Time Complexity: $O(n)$

Threaded Binary Trees (9)

- Insertion of Threaded Binary Tree
 - Insert child as the right child of node parent



First Case



Second Case

Threaded Binary Trees (10)

- Right insertion in a threaded binary tree

```
void insert_right(thread_pointer parent, threaded_pointer
child){
```

```
/* insert child as the right child of
parent in a threaded binary tree */
```

```
threaded_pointer temp;
```

```
child->right_child = parent->right_child;
```

```
child->right_thread = parent->right_thread;
```

```
child->left_child = parent;
```

```
child->left_thread = TRUE;
```

```
parent->right_child = child;
```

```
parent->right_thread = FALSE;
```

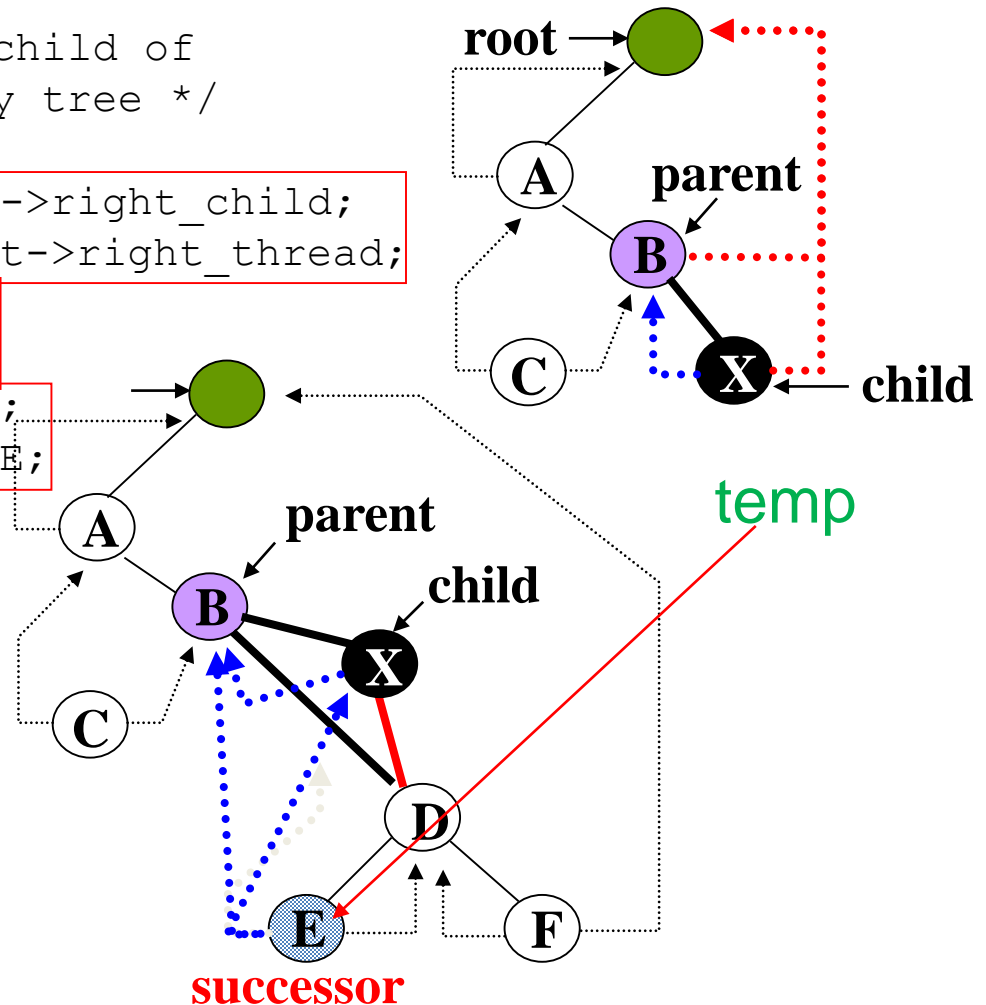
```
If(!child->right_thread){
```

```
temp = insucc(child);
```

```
temp->left_child = child;
```

```
}
```

```
}
```



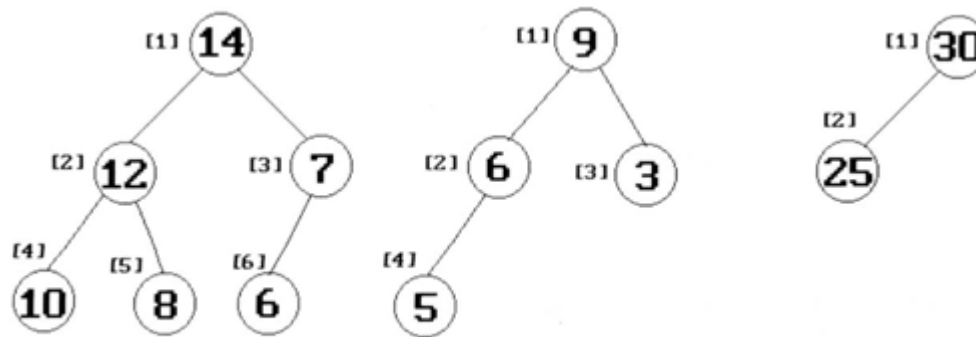
HEAP

Heaps (1)

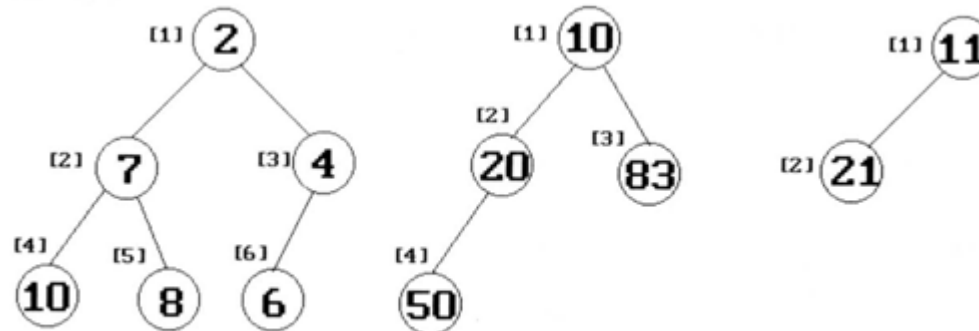
- The heap abstract data type
 - A *max(min) tree* is a tree in which the key value in each node is no smaller (larger) than the key values in its children
 - A *max (min) heap* is a complete binary tree that is also a max (min) tree
 - Basic Operations
 - Creation of an empty heap
 - Insertion of a new element into a heap
 - Deletion of the largest element from the heap

Heaps (2)

- The examples of max heaps and min heaps
 - Property: The root of max heap (min heap) contains the largest (smallest) element



Sample max heaps



Sample min heaps

Heaps (3)

- Abstract data type of Max Heap

structure *MaxHeap* is

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 $heap \in \text{MaxHeap}$, $item \in \text{Element}$, $n, \text{max_size} \in \text{integer}$

MaxHeap Create(max_size) ::= create an empty heap that can hold a maximum of max_size elements.

Boolean HeapFull($heap, n$) :: **if** ($n == \text{max_size}$) **return** *TRUE*
 else return *FALSE*

MaxHeap Insert($heap, item, n$) ::= **if** ($!\text{HeapFull}(heap, n)$)
 insert $item$ into $heap$ and return the resulting heap **else return** error.

Boolean HeapEmpty($heap, n$) :: **if** ($n > 0$) **return** *TRUE*
 else return *FALSE*

Element Delete($heap, n$) ::= **if** ($!\text{HeapEmpty}(heap, n)$) **return** one instance of the largest element in the heap and remove it from the heap **else return** error.

Heaps (4)

- Queue in Chapter 3: FIFO
- Priority queues
 - Heaps are frequently used to implement priority queues
 - delete the element with highest (lowest) priority
 - insert the element with arbitrary priority
 - Heaps is the only way to implement **priority queue**

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	$O(\log_2 n)$	$O(\log_2 n)$

Priority queue representations

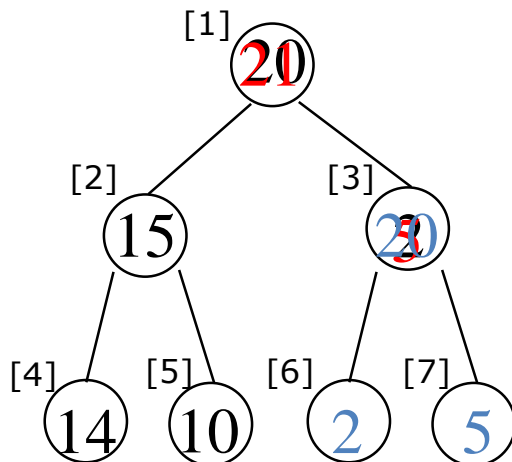
Heaps (5)

- Insertion into a Max Heap
 - Analysis of insert_max_heap
 - The complexity of the insertion function is $O(\log_2 n)$

insert 21

*n=6

i=7



```
void insert_max_heap(element item, int *n)
{
    /*insert item into a max heap of current size *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;
}
```

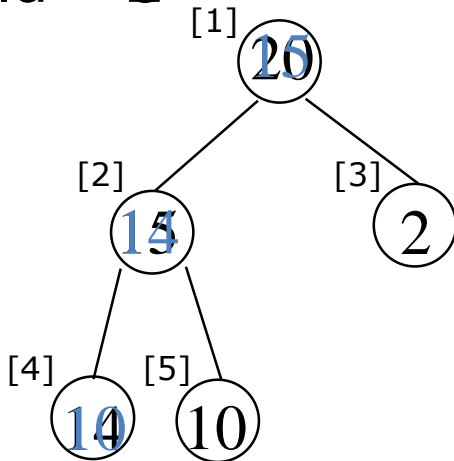
parent sink
item upheap

Insertion into a max heap

Heaps (6)

- Deletion from a max heap
 - After deletion, the heap is still a complete binary tree
 - Analysis of *delete_max_heap*
 - The complexity of the insertion function is $O(\log_2 n)$

parent = 2
child = 2
*n=5



```
element delete_max_heap(int *n)
{
    /* delete element with the highest key from the heap */
    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];
        parent = child;
        child *= 2;
    }
    heap[parent] = temp;
    return item;
}
```

item.key = 20
temp.key = 10

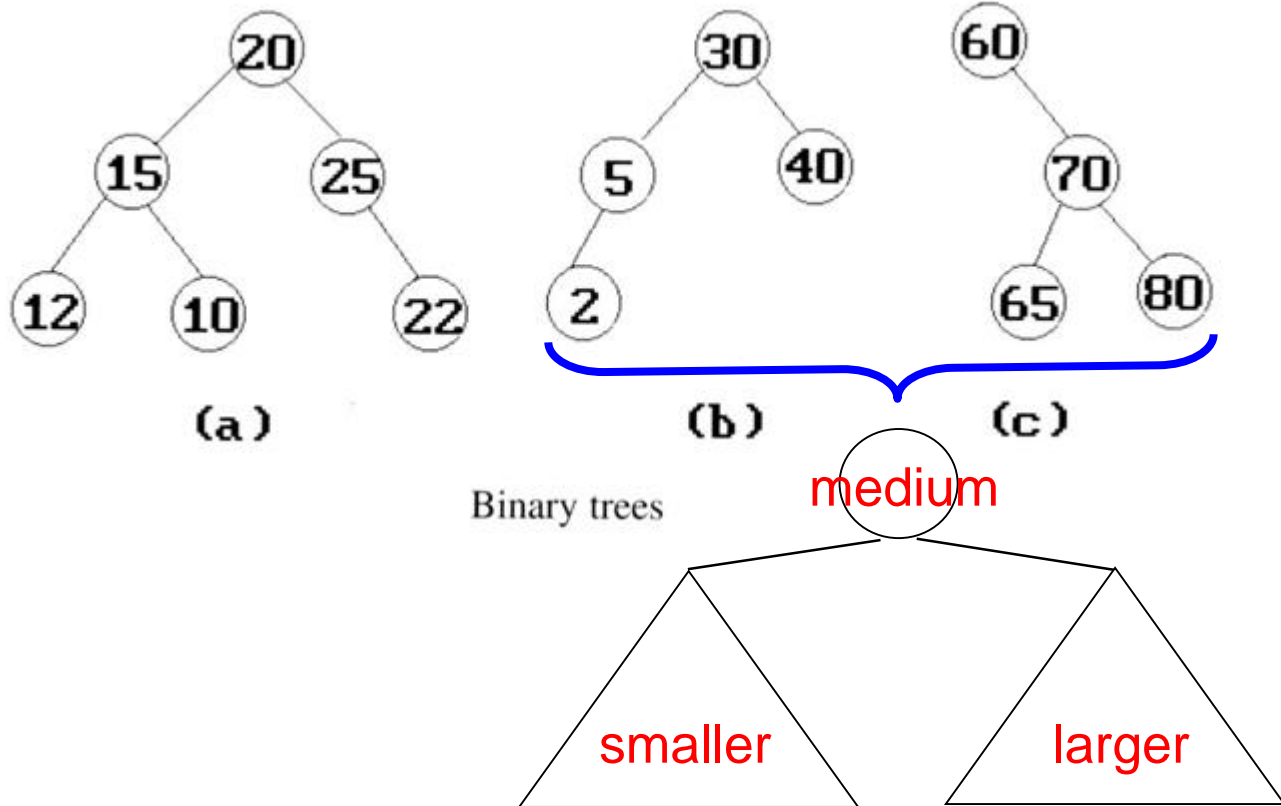
BINARY SEARCH TREES

Binary Search Trees (1)

- Why do binary search trees need?
 - Heap is not suited for applications in which arbitrary elements are to be deleted from the element list
 - A min (max) element is deleted $O(\log_2 n)$
 - Deletion of an arbitrary element $O(n)$
 - Search for an arbitrary element $O(n)$
- Definition of 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

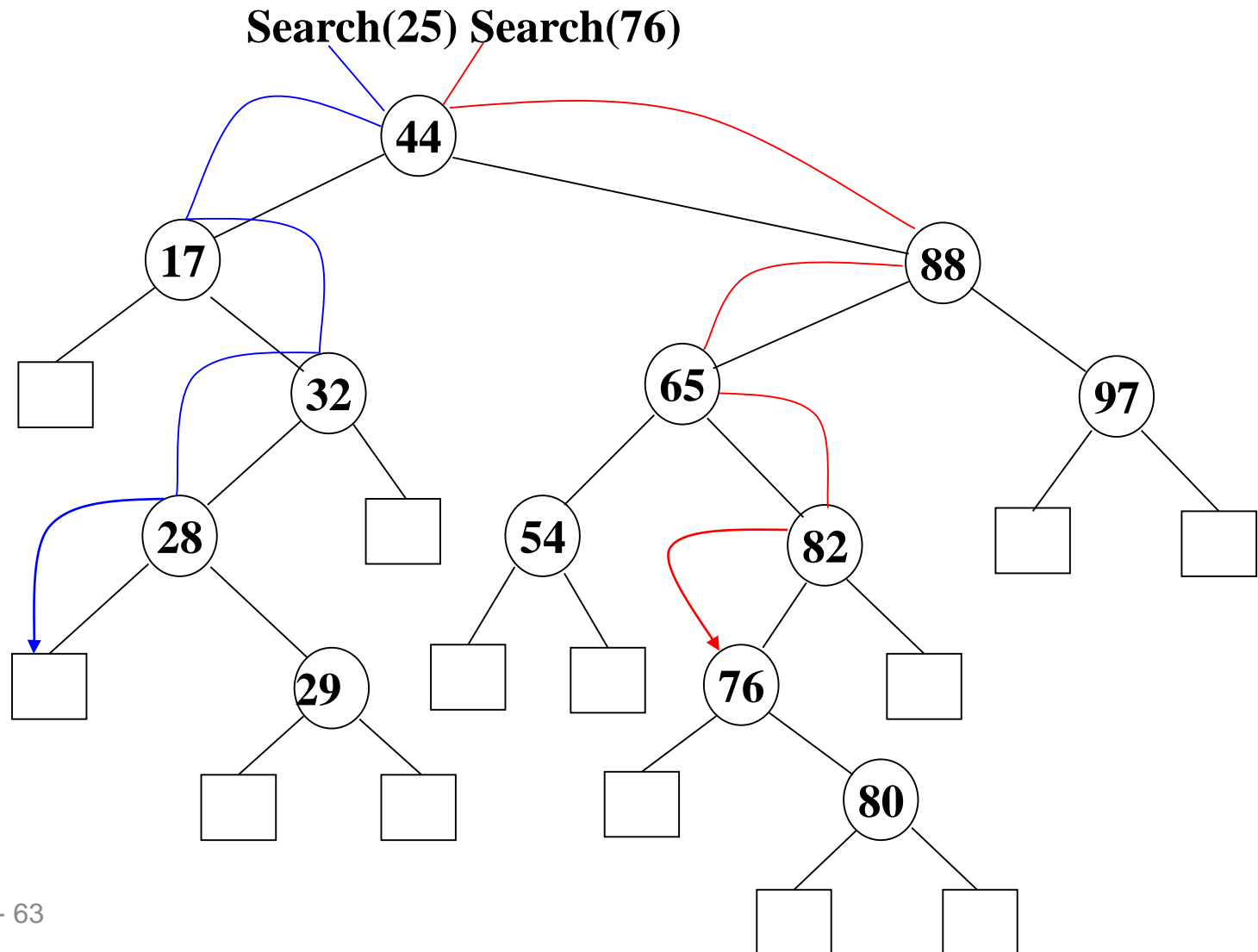
Binary Search Trees (2)

- Example: (b) and (c) are binary search trees



Binary Search Trees (3)

- Search



Binary Search Trees (4)

- 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);
}
```

Recursive search of a binary search tree

Binary Search Trees (5)

```
tree_pointer search2(tree_pointer tree, int key)
{
    /* return a pointer to the node that contains key.  If
    there is no such node, return NULL. */
    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)



Iterative search of a binary search tree

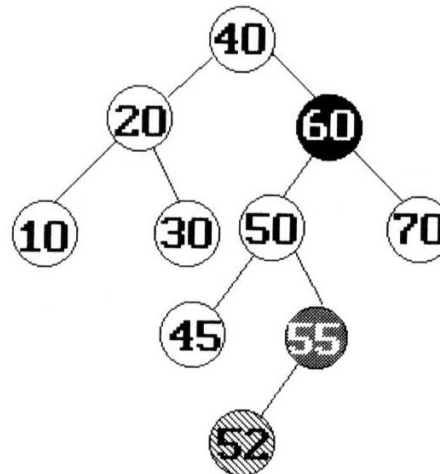
Binary Search Trees (6)

- Inserting into a binary search tree

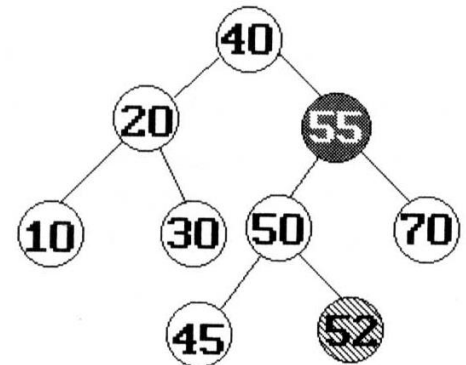
```
void insert_node(tree_pointer *node, int num)
/* If num is in the tree pointed at by node do nothing;
otherwise add a new node with data = num */
{
    tree_pointer ptr, temp = modified_search(*node, num);
    if (temp || !(*node)) → An empty tree
        /* num is not in the tree */
        ptr = (tree_pointer)malloc(sizeof(node));
        if (IS_FULL(ptr)) {
            fprintf(stderr, "The memory is full\n");
            exit(1);
        }
        ptr->data = num;
        ptr->left_child = ptr->right_child = NULL;
        if (*node) /* insert as child of temp */
            if (num < temp->data) temp->left_child = ptr;
            else temp->right_child = ptr;
        else *node = ptr;
    }
}
```

Binary Search Trees (7)

- Deletion from a binary search tree
 - Three cases should be considered
 - case 1. leaf → delete
 - case 2.
one child → delete and change the pointer to this child
 - case 3. two child →
either the smallest element in the right subtree or the largest element in the left subtree



(a) tree before deletion of 60



(b) tree after deletion of 60

Binary Search Trees (8)

- Height of a binary search tree
 - The height of a binary search tree with n elements can become as large as n
 - It can be shown that when insertions and deletions are made at random, the height of the binary search tree is $O(\log_2 n)$ on the average
 - Search trees with a worst-case height of $O(\log_2 n)$ are called *balance search trees*

Binary Search Trees (9)

- Time Complexity
 - Searching, insertion, removal
 - $O(h)$, where h is the height of the tree
 - Worst case - skewed binary tree
 - $O(n)$, where n is the # of internal nodes
- Prevent worst case
 - Rebalancing scheme
 - AVL, 2-3, and Red-black tree

SELECTION TREES

Selection Trees (1)

- Problem
 - Suppose we have k order sequences, called runs, that are to be merged into a single ordered sequence
- Solution
 - Straightforward : $k-1$ comparison
 - Selection tree : $\lceil \log_2 k \rceil + 1$
- There are two kinds of selection trees
 - Winner trees
 - Loser trees

Selection Trees (2)

- Definition (Winner tree)
 - A selection tree is the **binary** tree where each node represents the **smaller** of its two children
 - Root node is **the smallest node** in the tree
 - A winner is the record with smaller key
- Rules
 - Tournament : between sibling nodes
 - put **X** in the parent node \Rightarrow **X** tree
where **X** = **winner** or **loser**

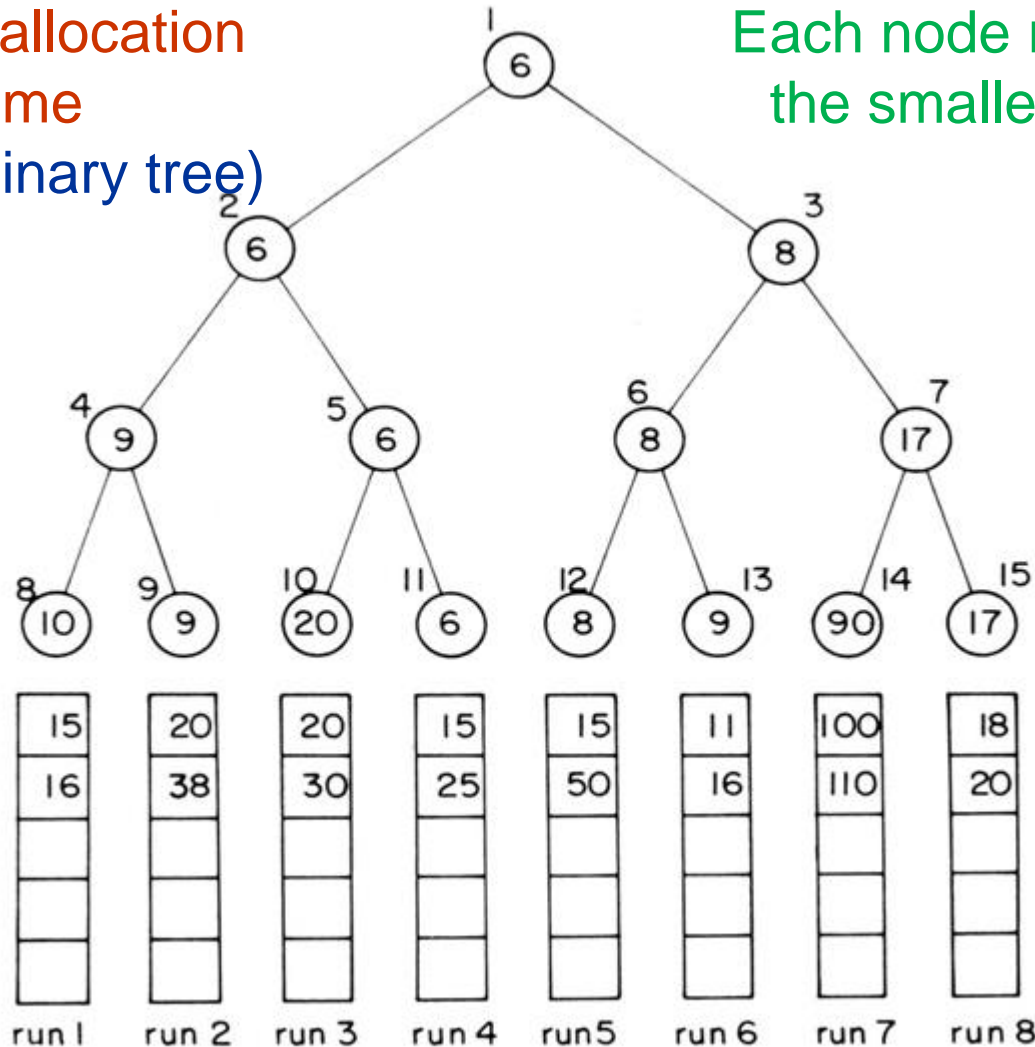
Selection Trees (3)

sequential allocation
scheme
(complete binary tree)

Each node represents
the smaller of its two
children

Winner Tree

ordered sequence

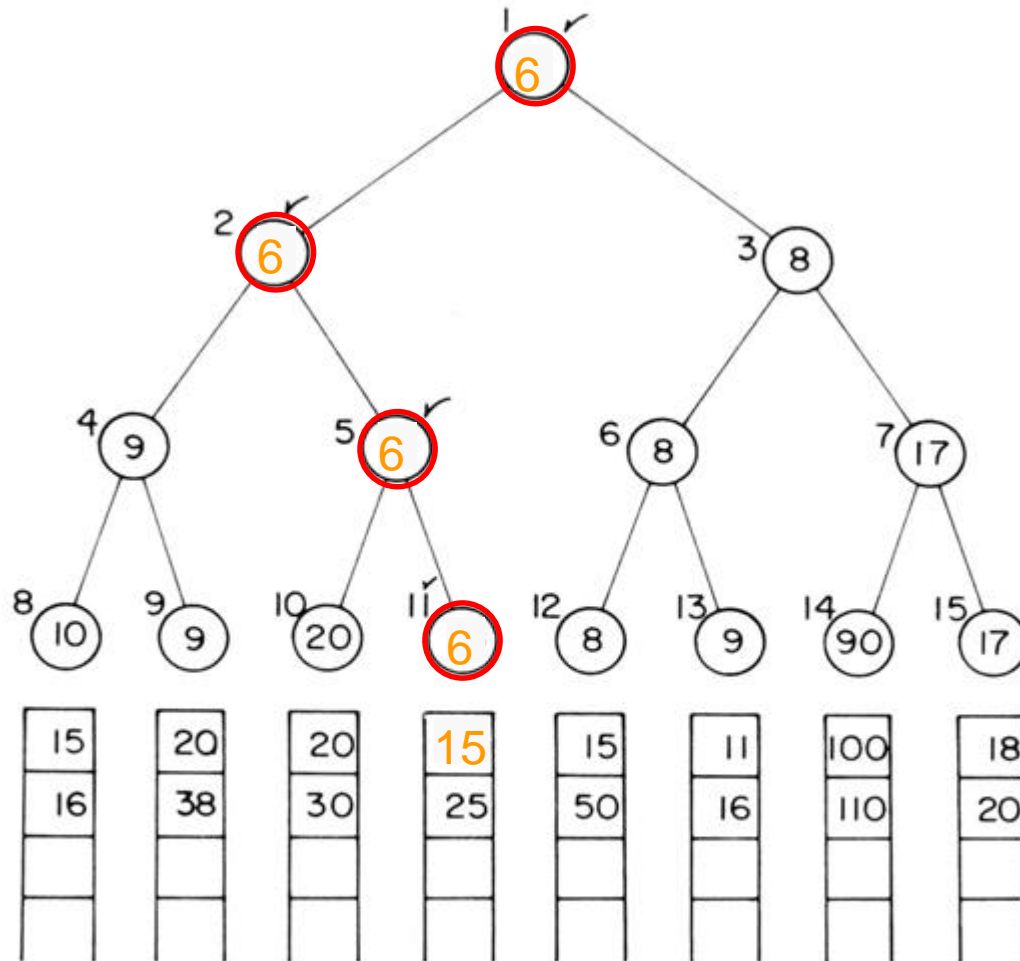


Selection tree for $k=8$ showing the first three keys in each of the eight runs

Selection Trees (4)

- Analysis of merging runs using winner trees (k is the number of sequence)
 - # of levels: $\lceil \log_2 K \rceil + 1 \Rightarrow$ restructure time: $O(\log_2 K)$
 - Merge time: $O(n \log_2 K)$
 - Setup time: $O(K)$
- Slight modification: tree of loser
 - Consider the parent node only (v.s. sibling nodes)

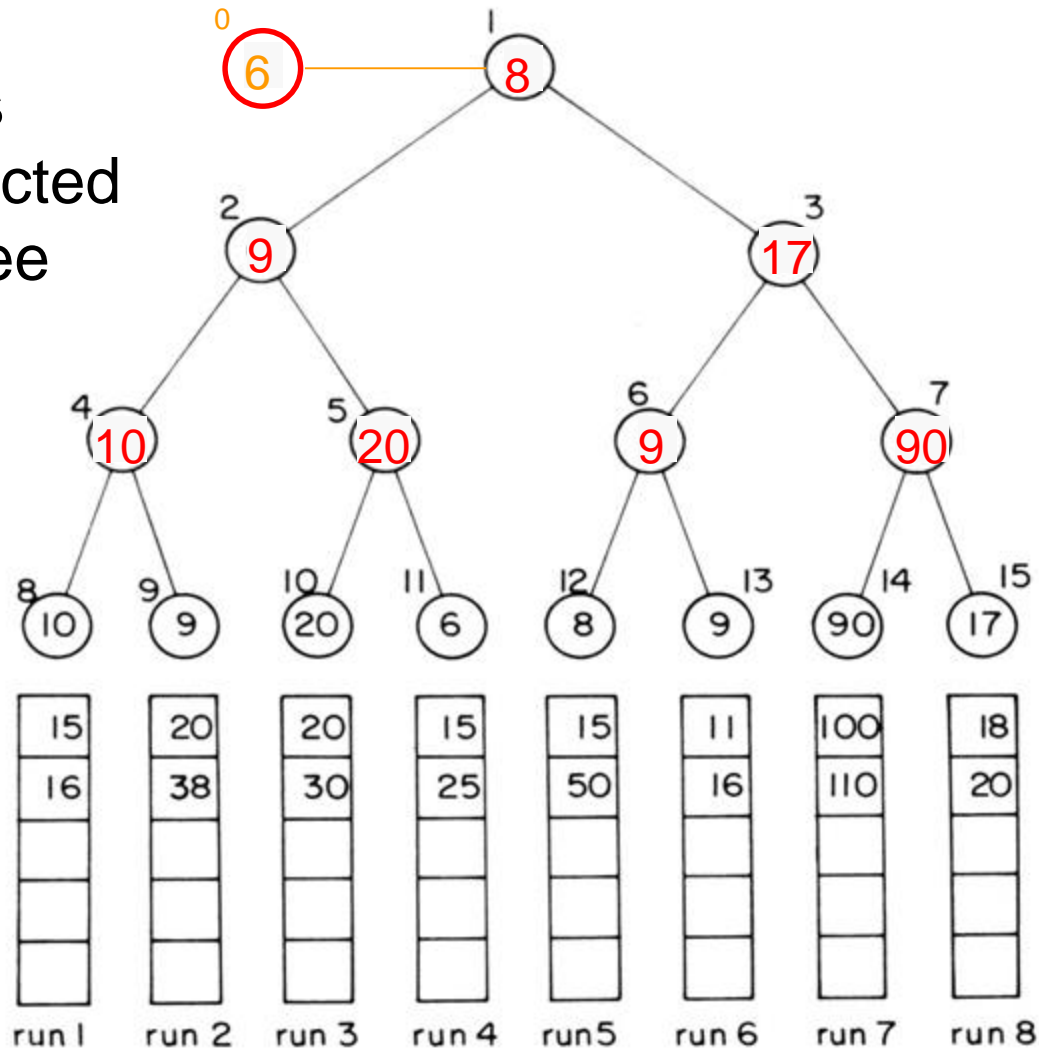
Selection Trees (5)



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

Selection Trees (6)

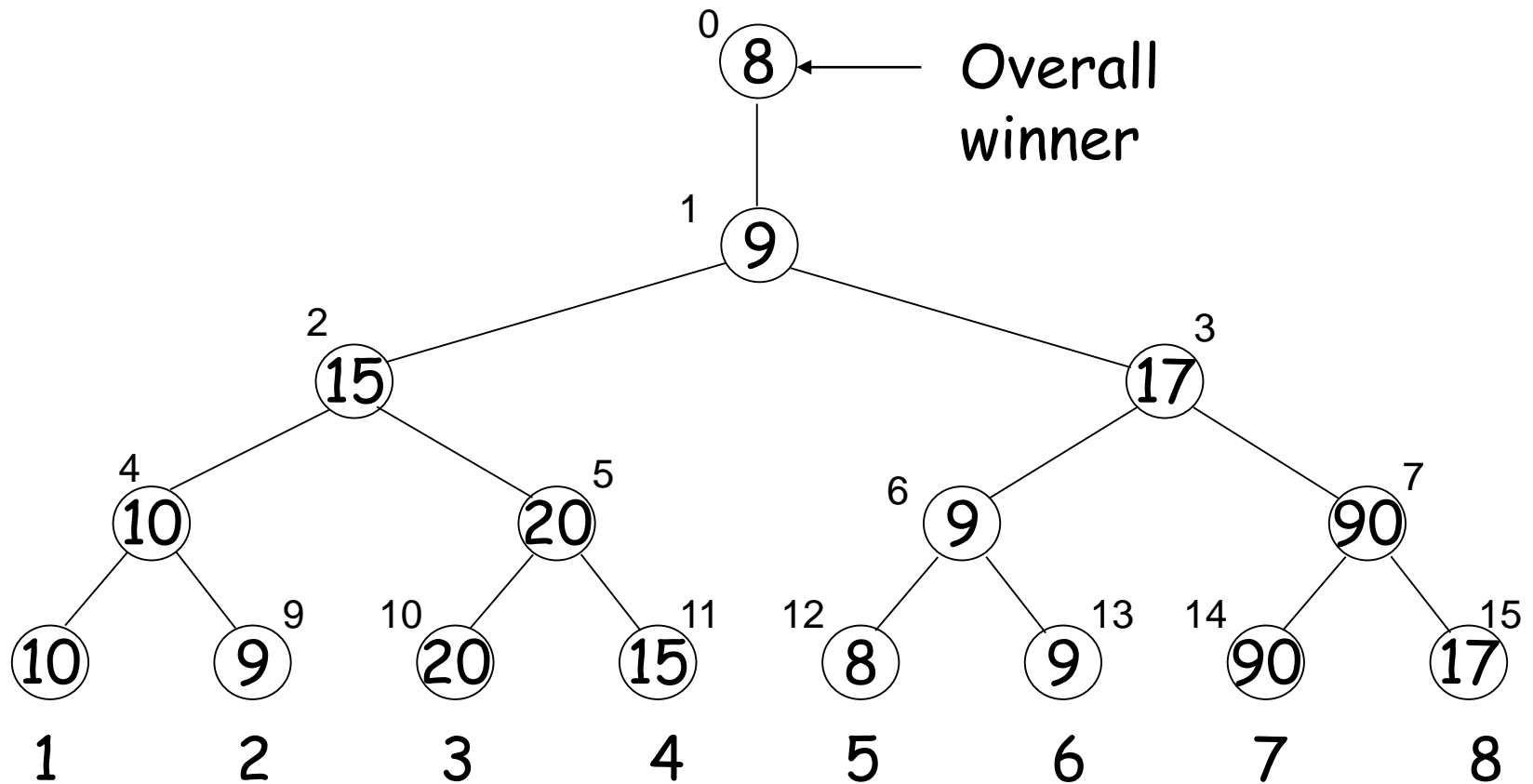
Tree of losers
can be conducted
by Winner tree



Selection tree for $k=8$ showing the first three keys in each of the eight runs

Selection Trees (7)

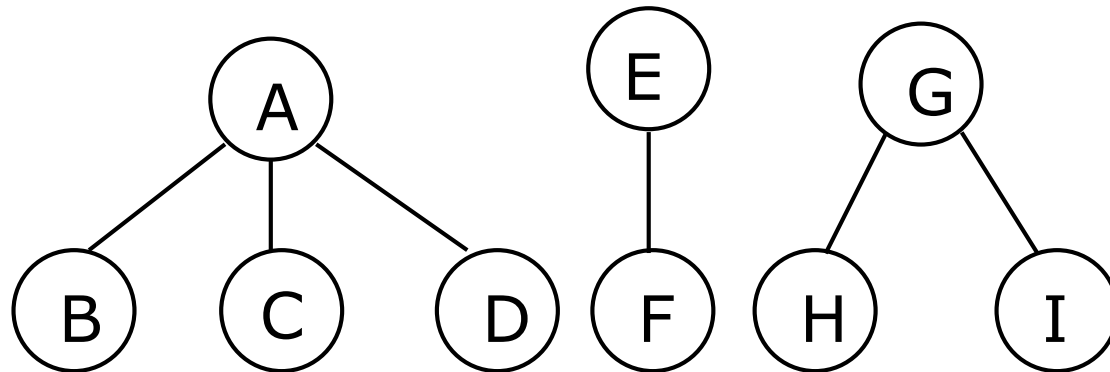
- The loser tree after output 6



FORESTS

Forests

- Definition
 - A forest is a set of $n \geq 0$ disjoint trees
- When we remove a root from a tree, we'll get a forest
 - Removing the root of a binary tree will get a forest of two trees

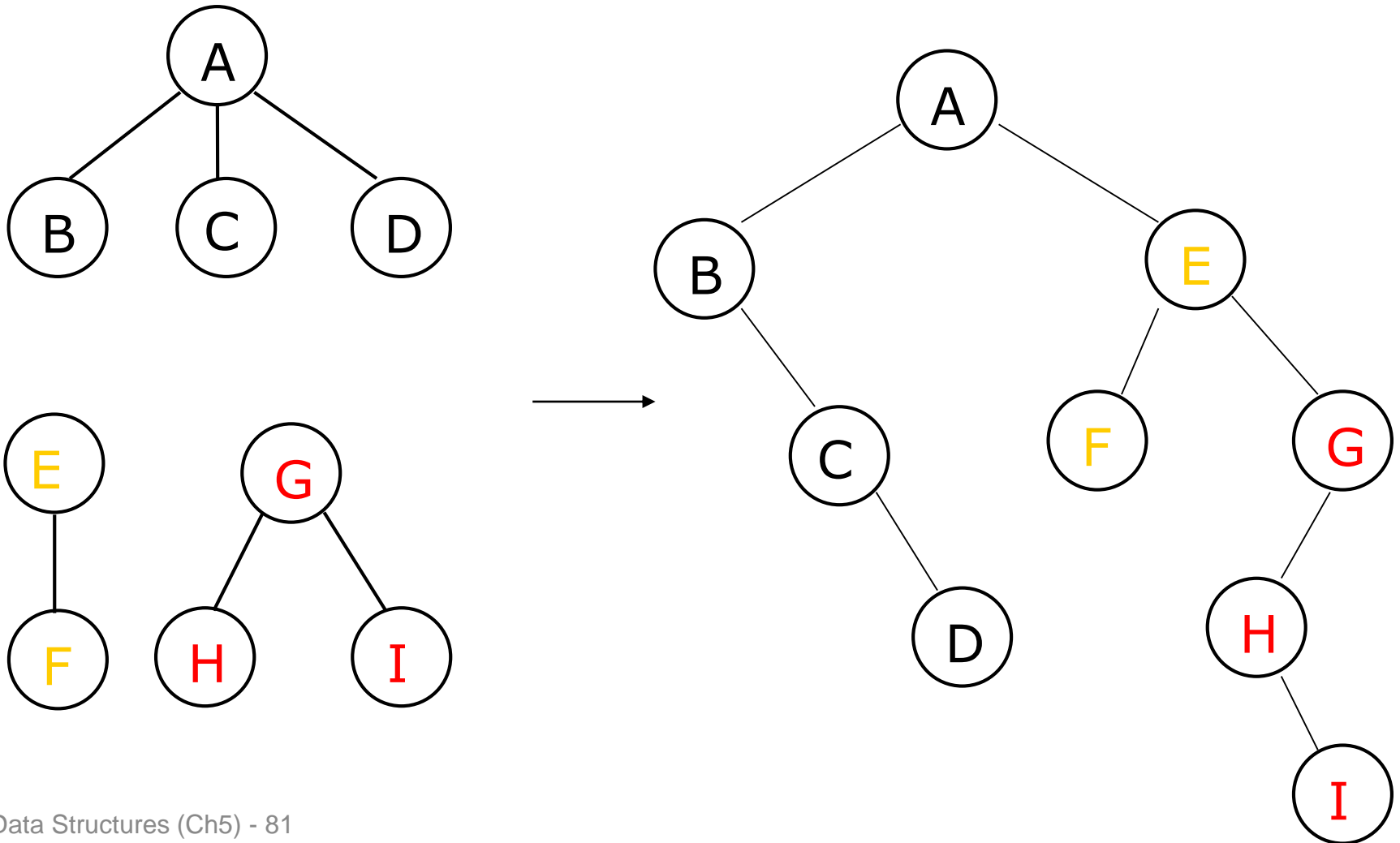


Three-tree forest

Transforming A Forest Into A Binary Tree (1)

- T_1, \dots, T_n
 - Forest of trees
- $B(T_1, \dots, T_n)$
 - Binary tree
- Algorithm
 - Is empty if $n = 0$
 - Has root equal to $\text{root}(T_1)$
 - Has left subtree equal to $B(T_{11}, \dots, T_{1m})$
 - T_{11}, \dots, T_{1m} are the subtree of $\text{root}(T_1)$
 - Has right subtree equal to $B(T_2, \dots, T_n)$

Transforming A Forest Into A Binary Tree (2)



SET REPRESENTATION

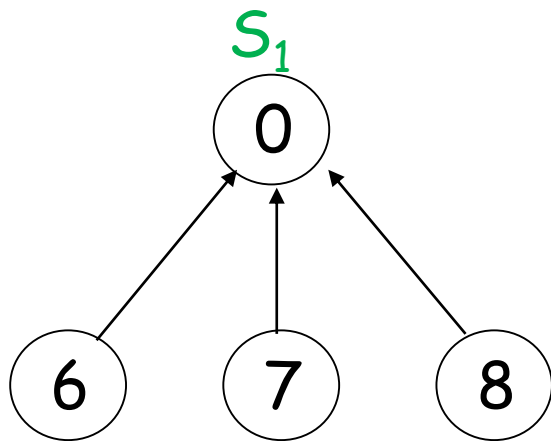
UNION & FIND

Set Representation

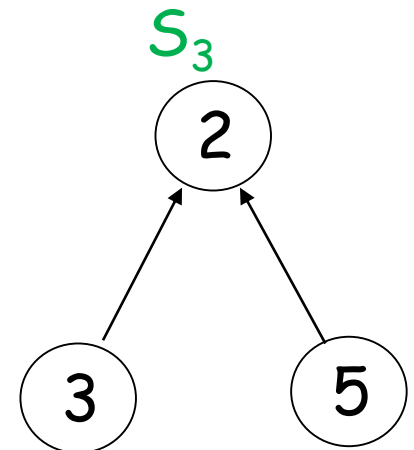
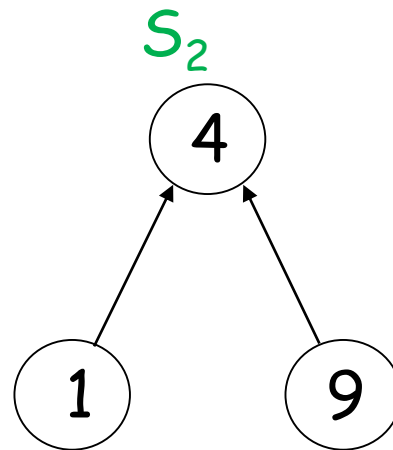
- Trees can be used to represent **sets**
- Disjoint set Union
 - If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j = \{\text{all elements } x \text{ such that } x \text{ is in } S_i \text{ or } S_j\}$.
- Find (i)
 - Find the set containing element i

Possible Tree Representation of Sets

Set 2 = {4 , 1 ,9}



Set 1 = {0 , 6 ,7 ,8 }



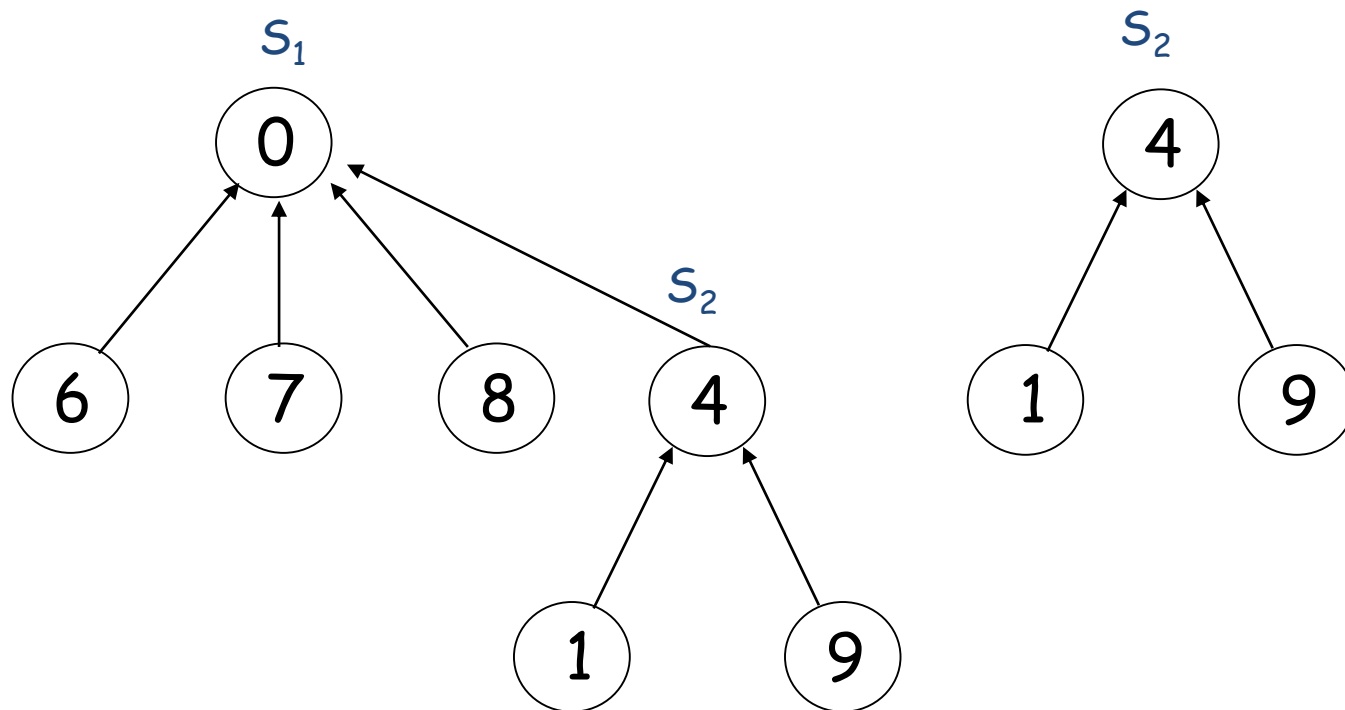
Set 3 = {2 , 3 , 5}

Link the nodes from the children to the parent

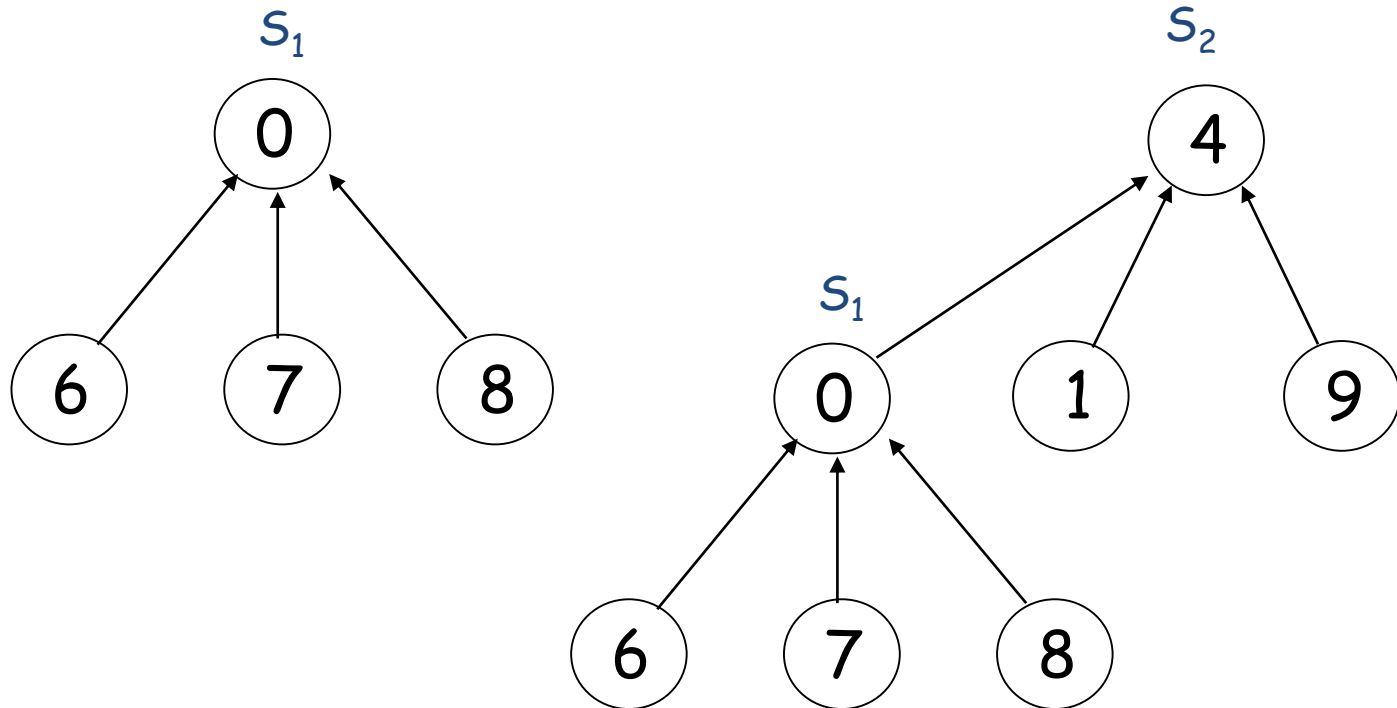
Unions of Sets

- To obtain the union of two sets, just set the parent field of one of the roots to the other root
- To figure out which set an element is belonged to, just follow its parent link to the root and then follow the pointer in the root to the set name

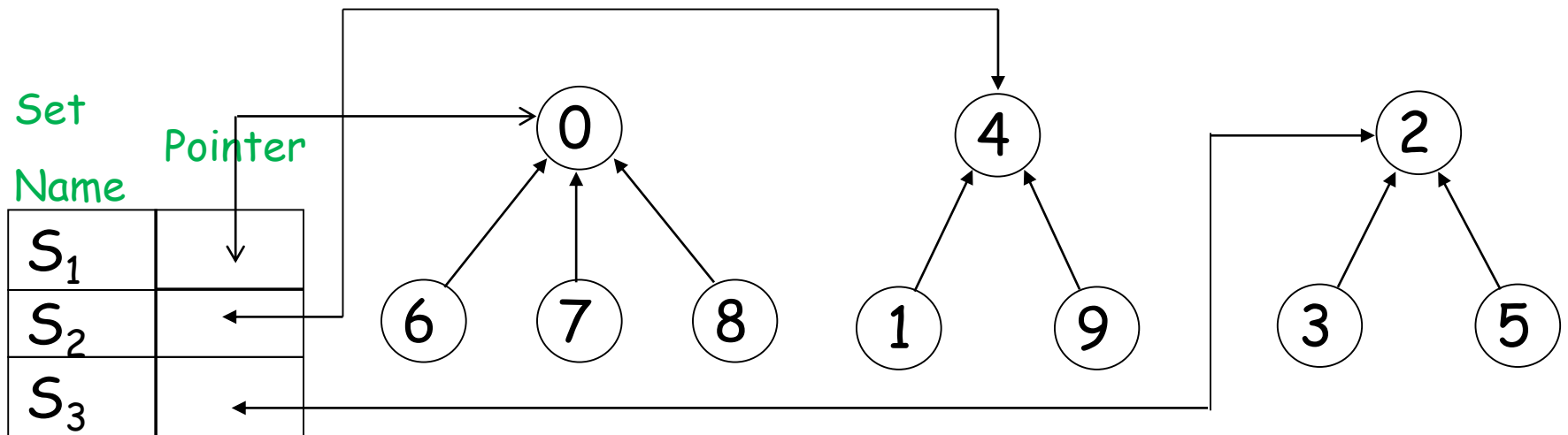
Possible Representations of $S_1 \cup S_2$



Possible Representations of $S_1 \cup S_2$



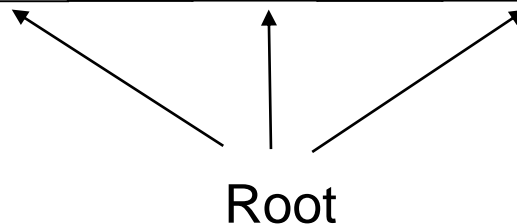
Data Representation for S_1, S_2, S_3



Array Representation (1)

- 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




Array Representation (2)

```
void Union1( int i , int j )  
{  
    parent[i] = j ;  
}
```

EX: S1 \cup S2 Union1(0 , 2) ;

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

i = 2



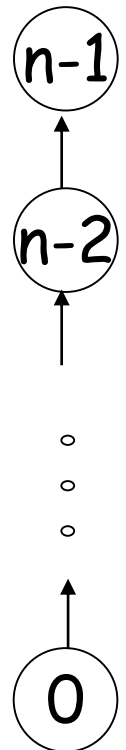
EX: Find1(5) ;

```
int Find1( int i )  
{  
    for(;parent[i] >= 0 ; i = parent[i]) ;  
    return i ;  
}
```

Analysis Union/Find Operations

- For a set of n elements each in a set of its own, then the result of the union function is a degenerate tree
- The time complexity of the following union-find operation is $O(n^2)$
- The complexity can be improved by using weighting rule for union

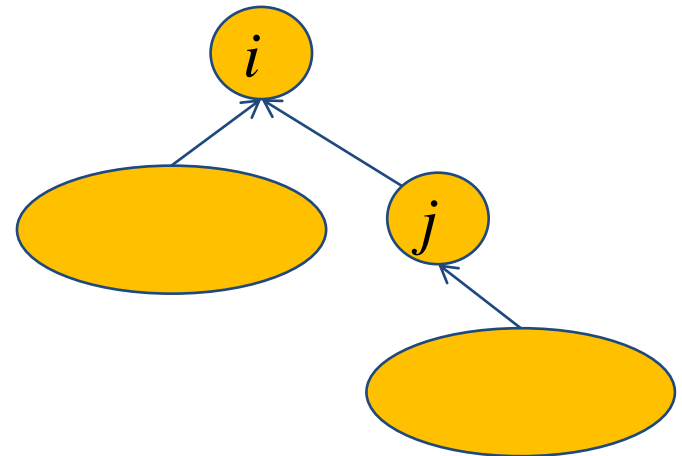
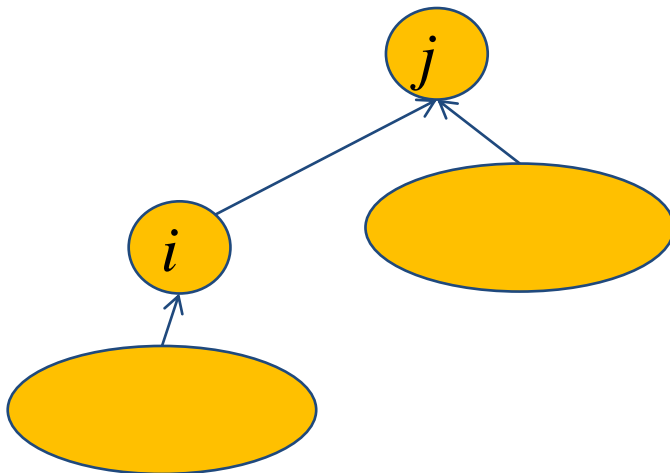
$\text{union}(0, 1), \text{find}(0)$	Union operation
$\text{union}(1, 2), \text{find}(0)$	$O(n)$
\vdots	
$\text{union}(n-2, n-1), \text{find}(0)$	Find operation
	$O(n^2)$



Weighting Rule

- Definition : 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

of nodes in $i < \#$ of nodes in j # of nodes in $i \geq \#$ of nodes in j



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

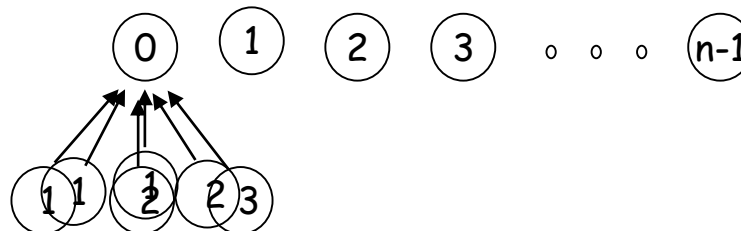
```

void union2 (int i, int j)
{
    int temp = parent[i] + parent[j];
    if ( parent[i]>parent[j]) {
        parent[i]=j;
        parent[j]=temp;
    }
    else {
        parent[j]=i;
        parent[i]=temp;
    }
}

```

unoin2 (0 , 1)
 unoin2 (0 , 2)
 unoin2 (0 , 3)

temp = -2
 temp = -3

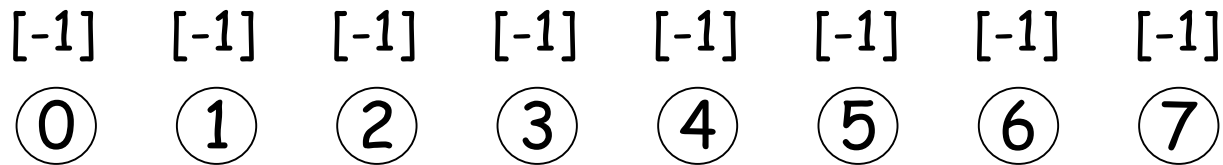


EX: unoin2 (0 , 1) , unoin2 (0 , 2) , unoin2 (0 , 2)

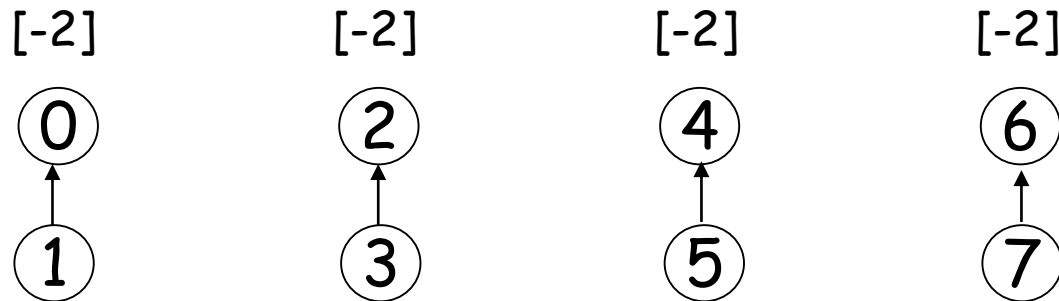
Weighted Union

- Assume we start with a forest of trees, each having one node
 - **Lemma:** Let T be a tree with n nodes created as a result of WeightedUnion. No node in T has level greater than $\lfloor \log_2 n \rfloor + 1$
- For the processing of an intermixed sequence of $u - 1$ unions and f find operations, the time complexity is $O(u + f \log u)$

Trees Achieving Worst-Case Bound

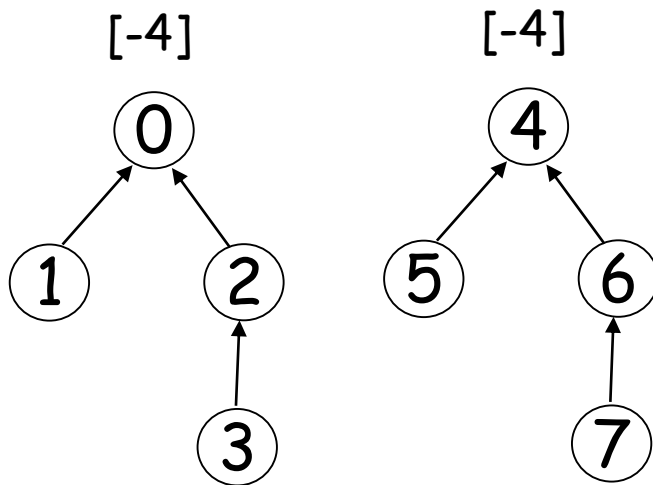


(a) Initial height trees

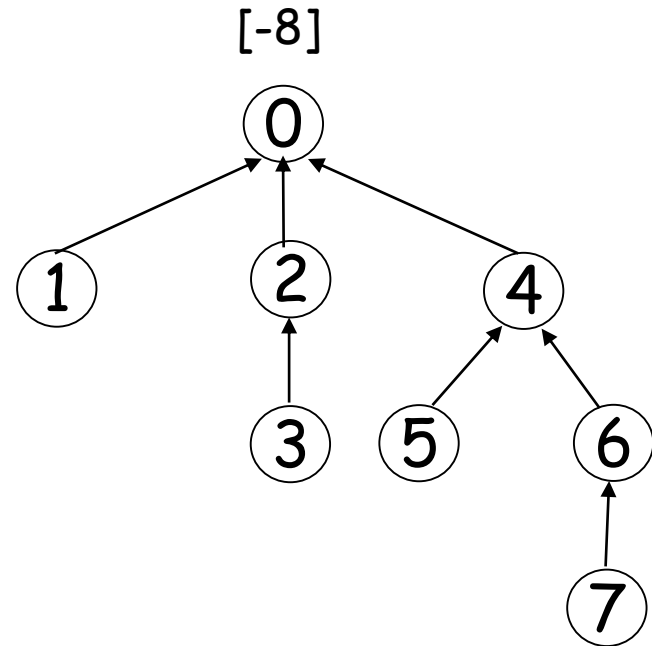


(b) Height-2 trees following union (0, 1), (2, 3), (4, 5), and (6, 7)

Trees Achieving Worst-Case Bound (Cont.)



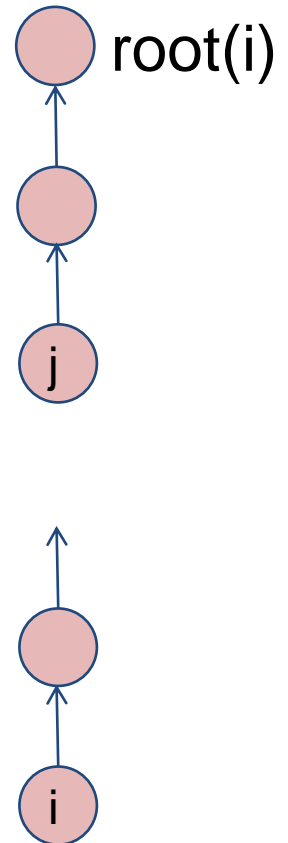
(c) Height-3 trees following
union (0, 2), (4, 6)



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

Collapsing Rule (1)

- **Definition** : 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, all following find operation of the same element only goes up one link to find the root



Collapsing Rule (2)

```
int find2(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];
```

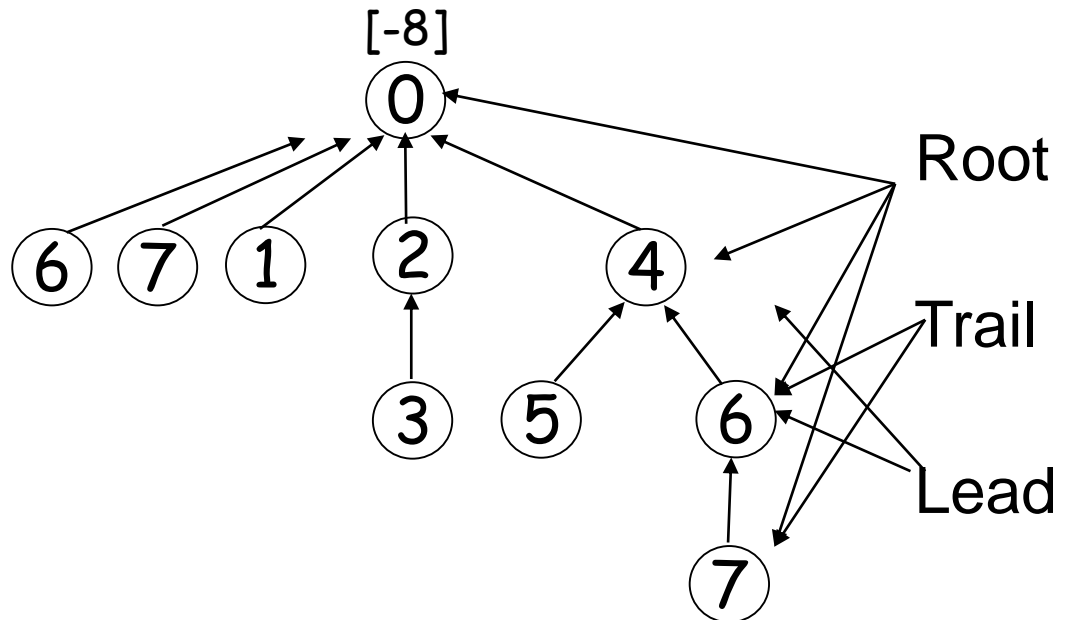
```
        parent[trail]= root;
```

```
    }
```

```
    return root;
```

```
}
```

Ex: find2 (7)



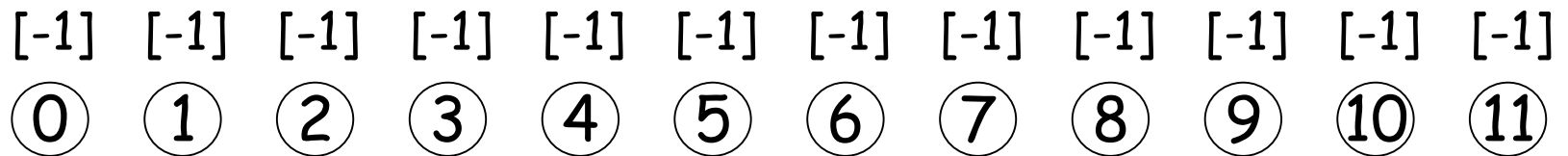
Collapsing Rule (3)

- Analysis of WeightedUnion and CollapsingFind
 - The use of collapsing rule roughly double the time for an individual find. However, it reduces the worst-case time over a sequence of finds

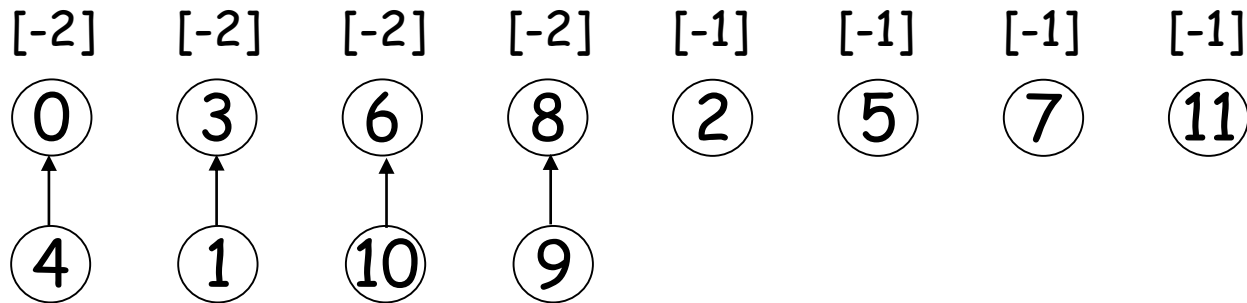
Revisit Equivalence Class

- The aforementioned techniques can be applied to the equivalence class problem
- Assume initially all n polygons are in an equivalence class of their own: $\text{parent}[i] = -1, 0 \leq i < n$.
 - Firstly, we must determine the sets that contains i and j
 - If the two are in different set, then the two sets are to be replaced by their union
 - If the two are in the same set, then nothing need to be done since they are already in the equivalence class
 - So we need to perform two finds and at most one union.

Example (1)

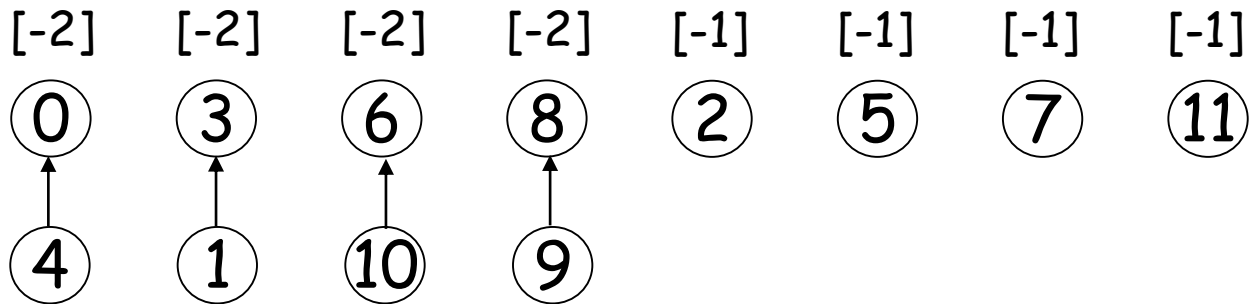


(a) Initial trees

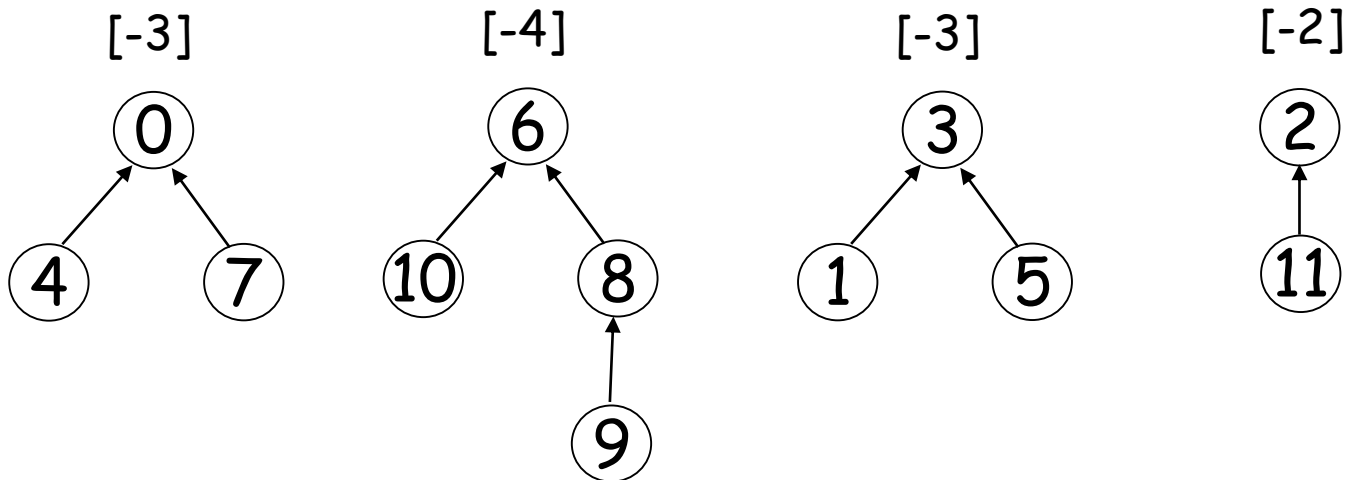


(b) Height-2 trees following $0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, and $8 \equiv 9$

Example (2)

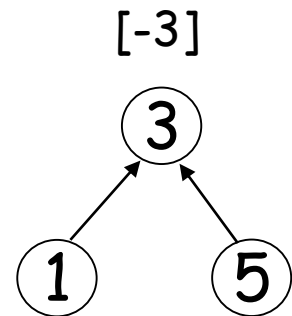
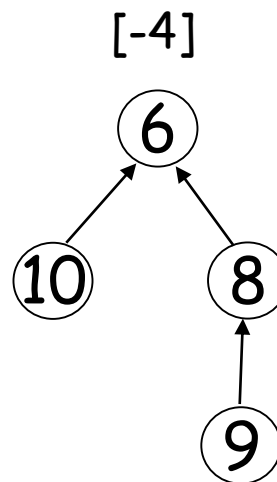
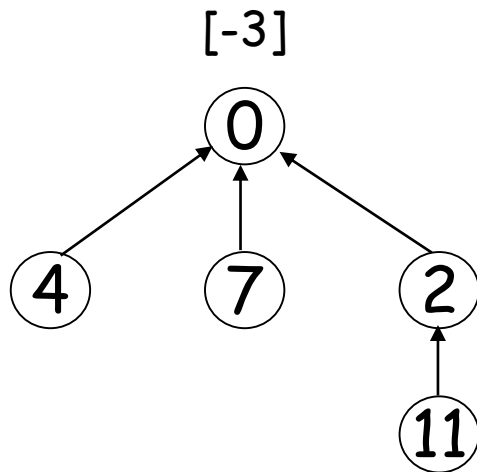


(b) Height-2 trees following $0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, and $8 \equiv 9$



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

Example (3)

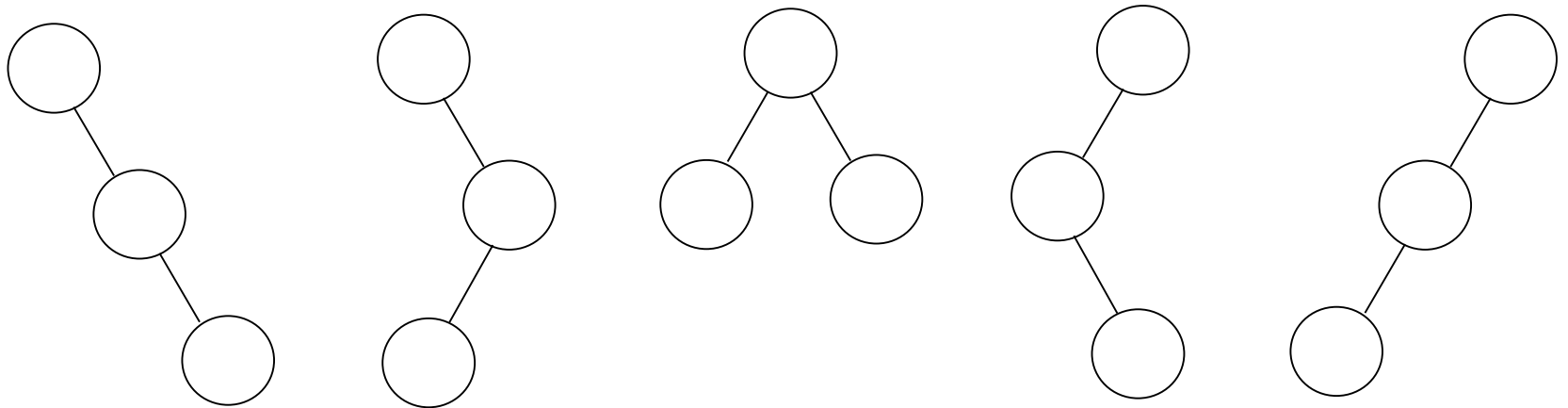


(d) Tree following $11 \equiv 0$

Counting Binary Tree

- We consider the following three disparate problems
 - Determine the number of distinct binary trees having n nodes
 - Determine the number of distinct permutations of the numbers from 1 through n obtainable by a stack
 - Determine the number of distinct ways of multiplying $n+1$ matrices .
- They amazingly have the same solution

Distinct Binary Trees

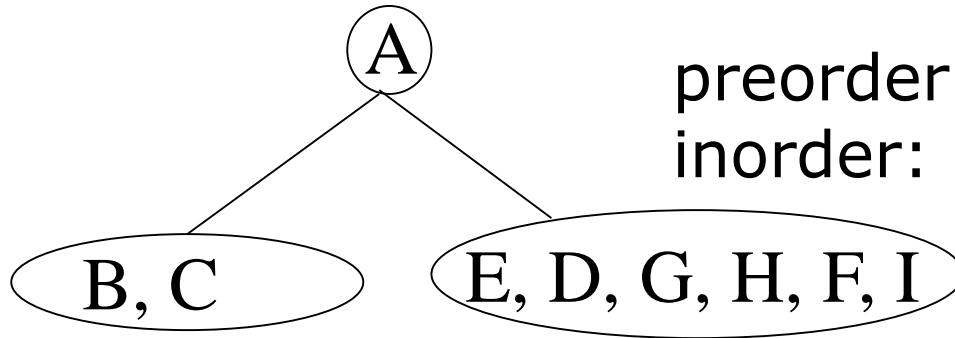


There are 5 distinct binary tree

Stack permutation (1)

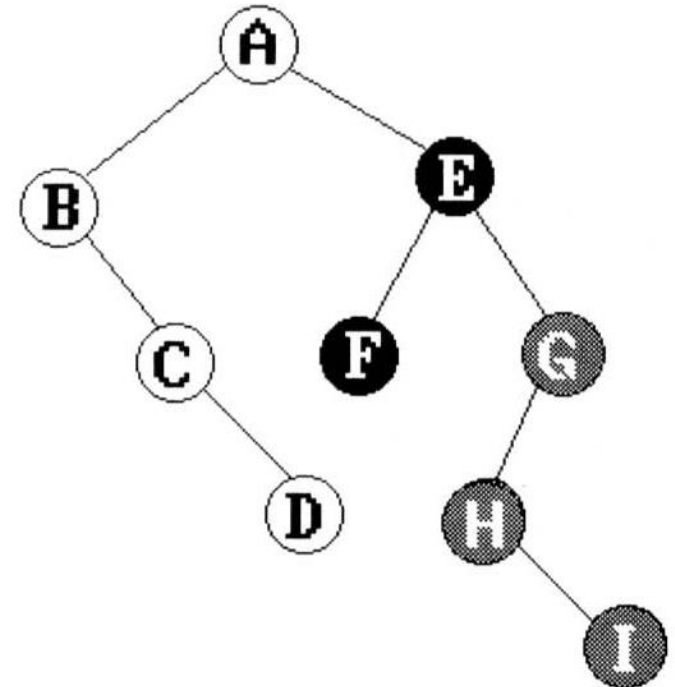
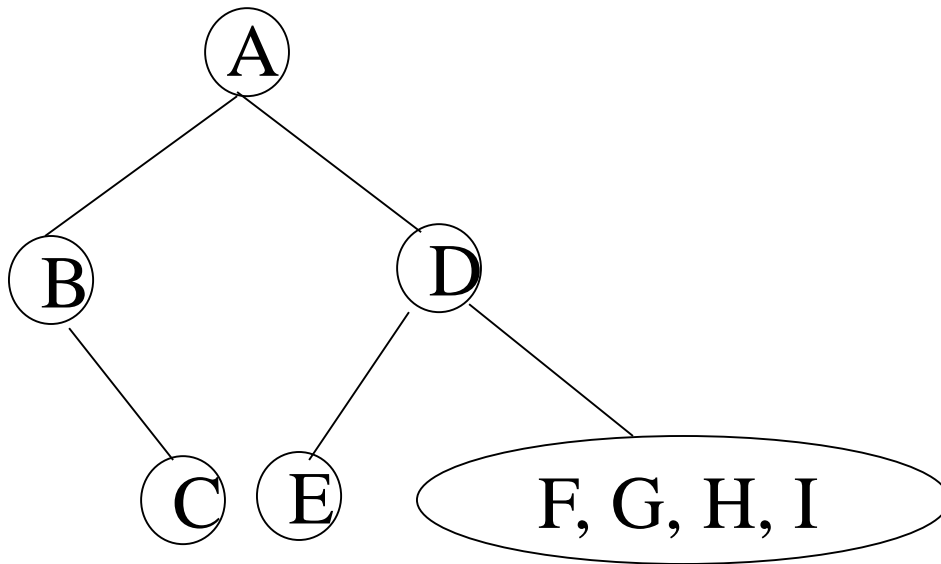
- In section 5.3 we introduced preorder, inorder, and postorder traversal of a binary tree and indicated that each traversal requires a stack
- Every binary tree has a unique pair of preorder/inorder sequences
- The number of distinct binary trees is equal to the number of inorder permutations obtainable from binary trees having the preorder permutation, $1, 2, \dots, n$.

Stack permutation (2)



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

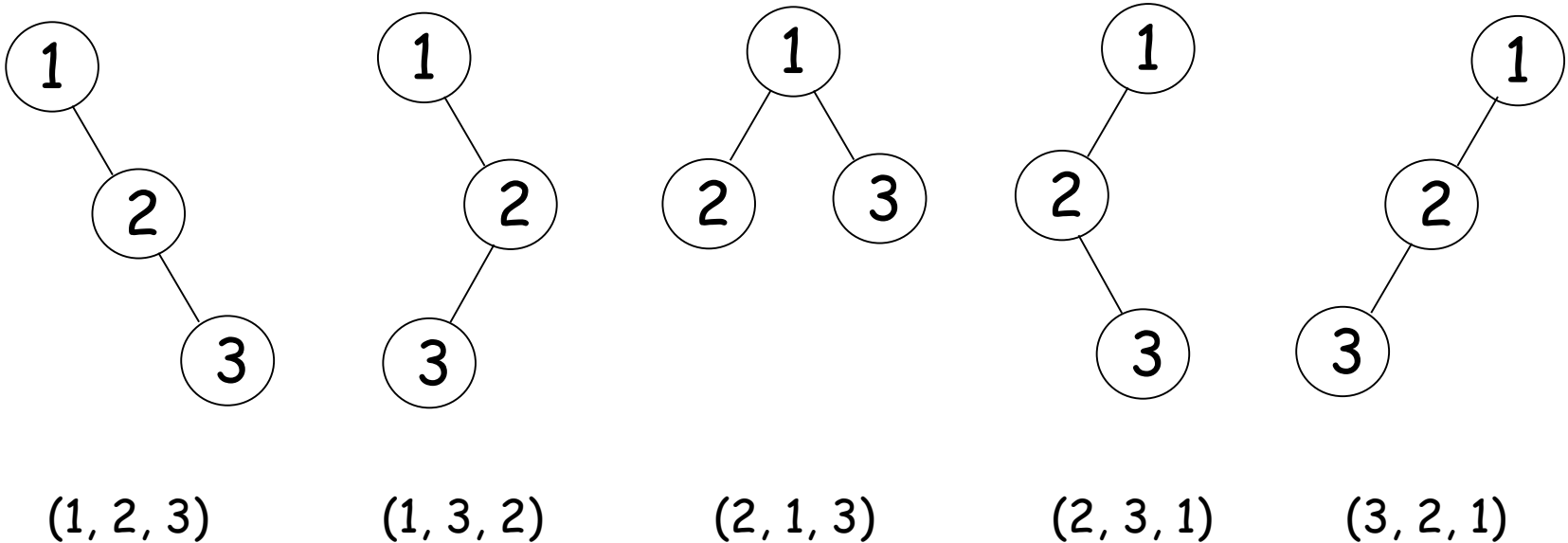
preorder: A B C (D E F G H I)
inorder: B C A (E D G H F I)



Stack permutation (3)

- The number of distinct permutations obtainable by passing the numbers 1 through n through a stack is equal to the number of distinct binary trees with n nodes
- Example
 - If we start with the numbers 1 , 2 , and 3 , then the possible permutations obtainable by a stack are
(1,2,3) (1,3,2) (2,1,3) (2,3,1) (3,2,1)

Stack permutation (4)



Binary trees corresponding to five permutation

Matrix Multiplication (1)

- The number of distinct binary trees is equal to the number of distinct inorder permutations obtainable from binary trees having the preorder permutation, $1, 2, \dots, n$.
- Computing the product of n matrices are related to the distinct binary tree problem

$$M_1 * M_2 * \dots * M_n$$

$$n = 3 \quad (M_1 * M_2) * M_3 \qquad M_1 * (M_2 * M_3)$$

$$\begin{aligned} n = 4 \quad & ((M_1 * M_2) * M_3) * M_4 \\ & (M_1 * (M_2 * M_3)) * M_4 \\ & M_1 * ((M_2 * M_3) * M_4) \\ & (M_1 * (M_2 * (M_3 * M_4))) \\ & ((M_1 * M_2) * (M_3 * M_4)) \end{aligned}$$

Matrix Multiplication (2)

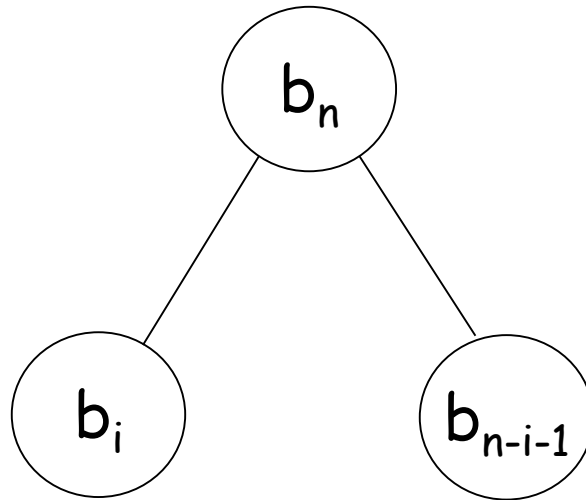
- Let b_n be the number of different ways to compute the product of n matrices
 - $b_2 = 1$, $b_3 = 2$, and $b_4 = 5$

$$b_n = \sum_{i=1}^{n-1} b_i b_{n-i}, \quad n > 1$$

Number of Distinct Binary Trees (1)

- The number of distinct binary trees of n nodes is

$$b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1}, \quad n \geq 1, \text{ and } b_0 = 1$$



Number of Distinct Binary Trees (2)

- Assume we let $B(x) = \sum_{i \geq 0} b_i x^i$ which is the generating function for the number of binary trees.
- By the recurrence relation we get $xB^2(x) = B(x) - 1$

$$B(x) = \frac{1 - \sqrt{1 - 4x}}{2x}$$

$$B(x) = \frac{1}{2x} \left(1 - \sum_{n \geq 0} \binom{1/2}{n} (-4x)^n \right) = \sum_{m \geq 0} \binom{1/2}{m+1} (-1)^m 2^{2m+1} x^m$$

$$b_n = \frac{1}{n+1} \binom{2n}{n} \approx b_n = O(4^n / n^{3/2})$$