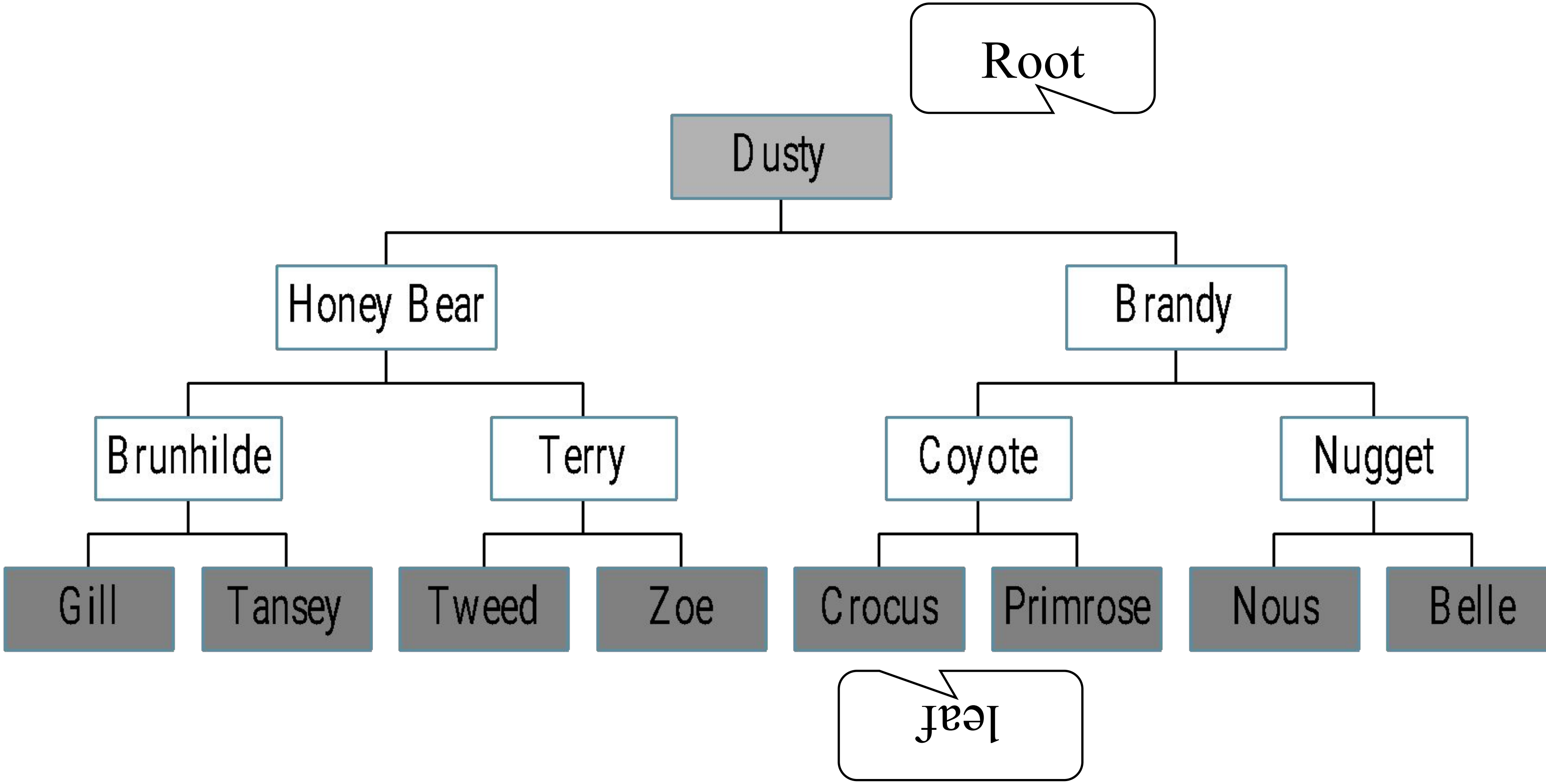
Trees



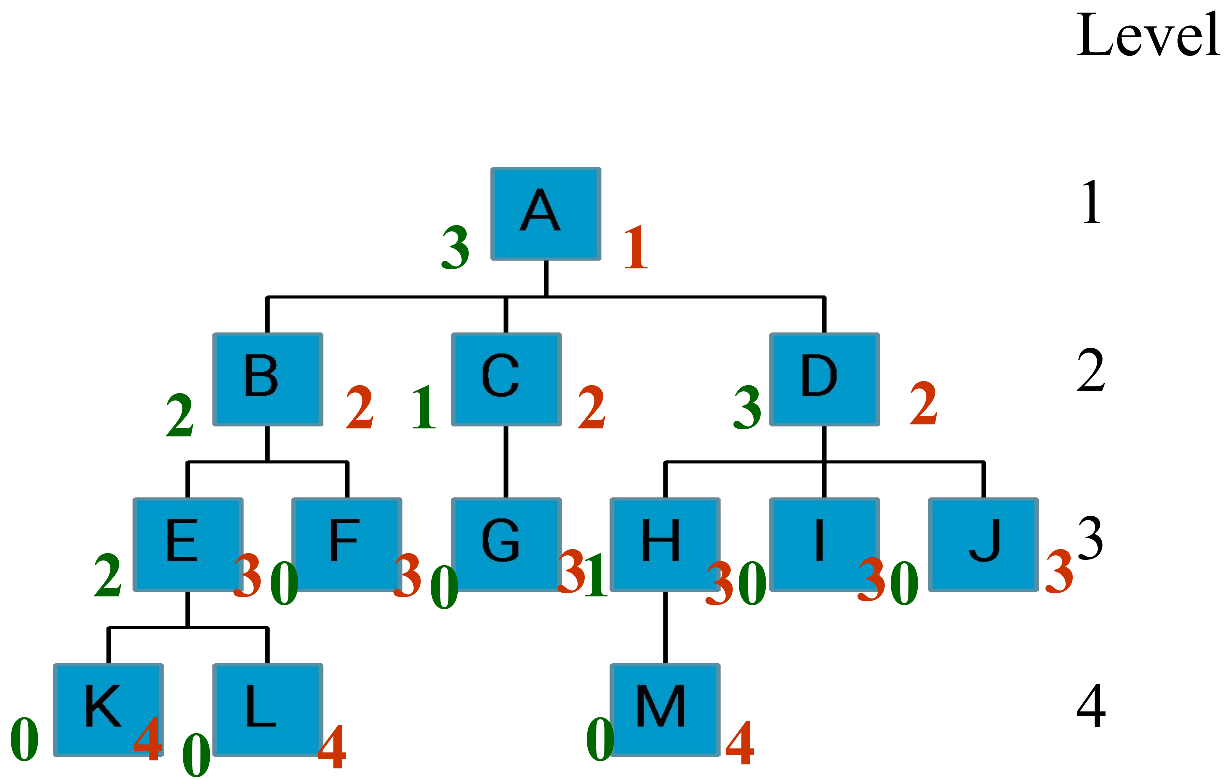
Root
Dusty
Honey Bear
Brandy
Brunhilde
Terry
Coyote
Nugget
Gill
Tansey
Tweed
Zoe
Crocus
Primrose
Nous
Belle
leaf

Definition of Tree

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

Level and Depth

node (13)
degree of a node
leaf (terminal)
nonterminal
parent
children
sibling
degree of a tree (3)
ancestor
level of a node
height of a tree (4)



Terminology

- The degree of a node is the number of subtrees of the node
  - The degree of A is 3; the degree of C is 1.
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the *parent* of the 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 along the path from the root to the node.

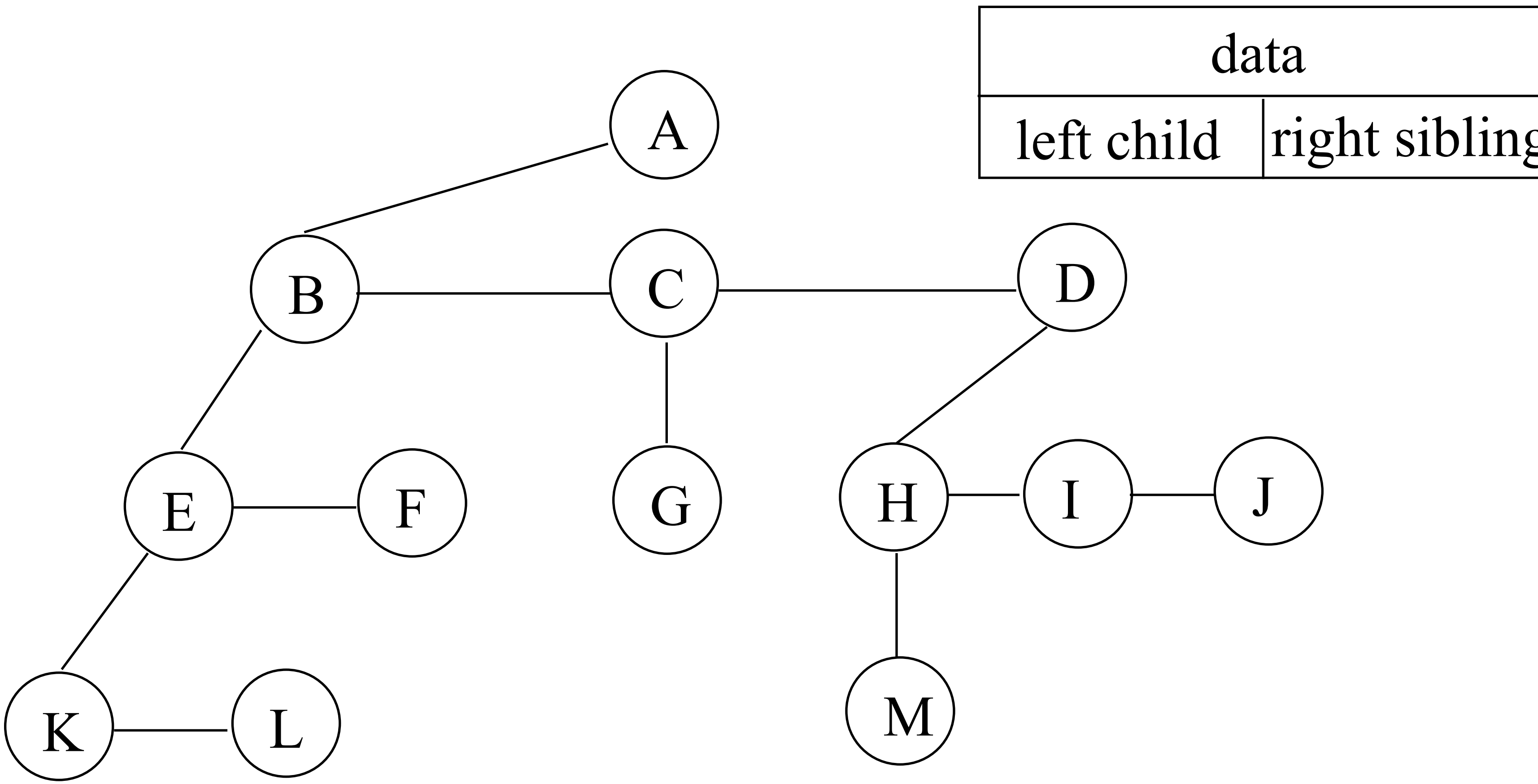
Representation of Trees

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

data	link 1	link 2	...	link n

How many link fields are needed in such a representation?

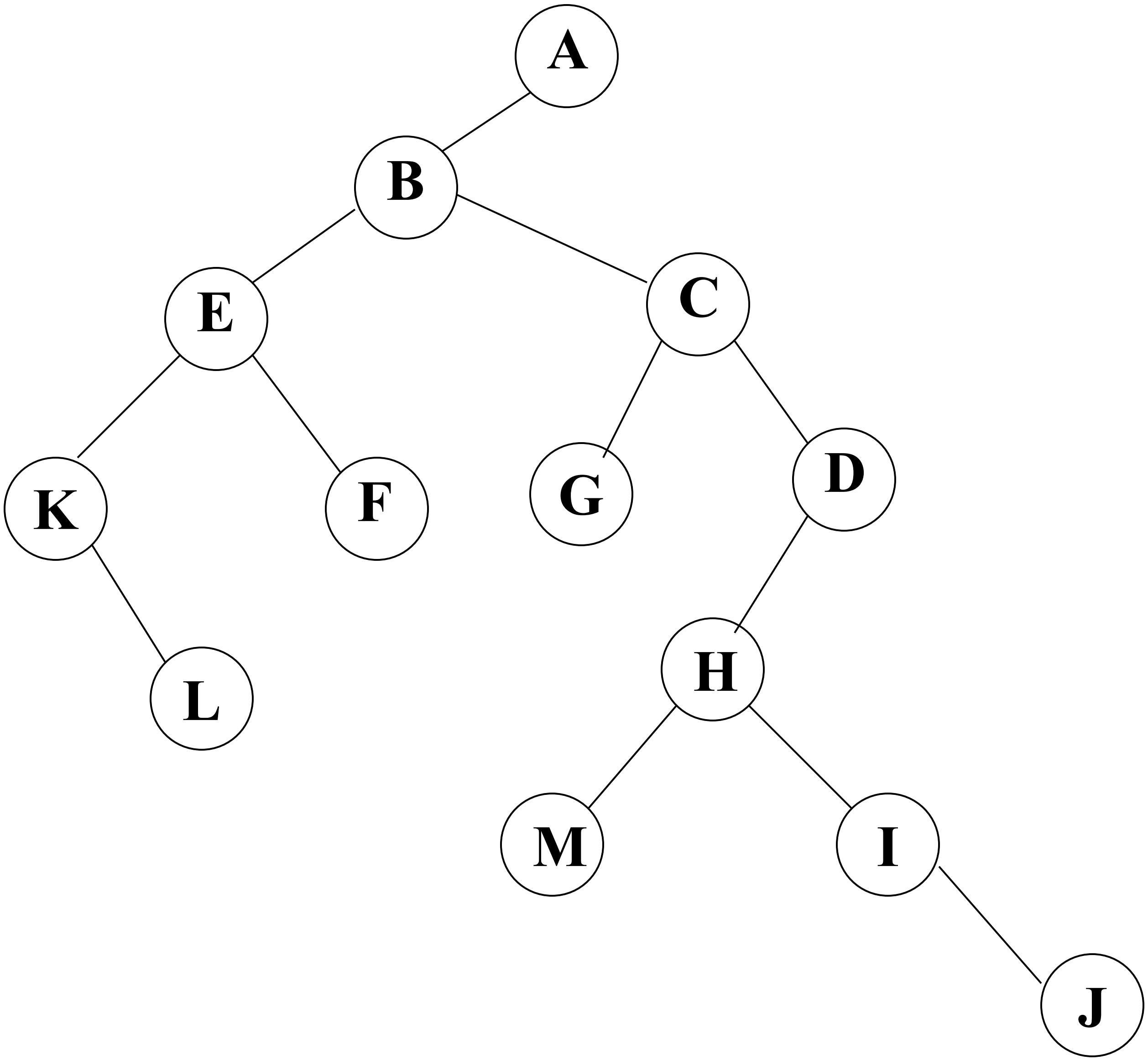
Left Child - Right Sibling



Binary Trees

- A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.
- Any tree can be transformed into binary tree.
  - by left child-right sibling representation
- The left subtree and the right subtree are distinguished.

***Figure 5.6:** Left child-right child tree representation of a tree (p.191)



Abstract Data Type 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* $\in$ *BinTree*, *item* $\in$ *element*

Bintree Create()::= creates an empty binary tree

Boolean IsEmpty(*bt*)::= if (*bt*==empty binary tree) return *TRUE* else return *FALSE*

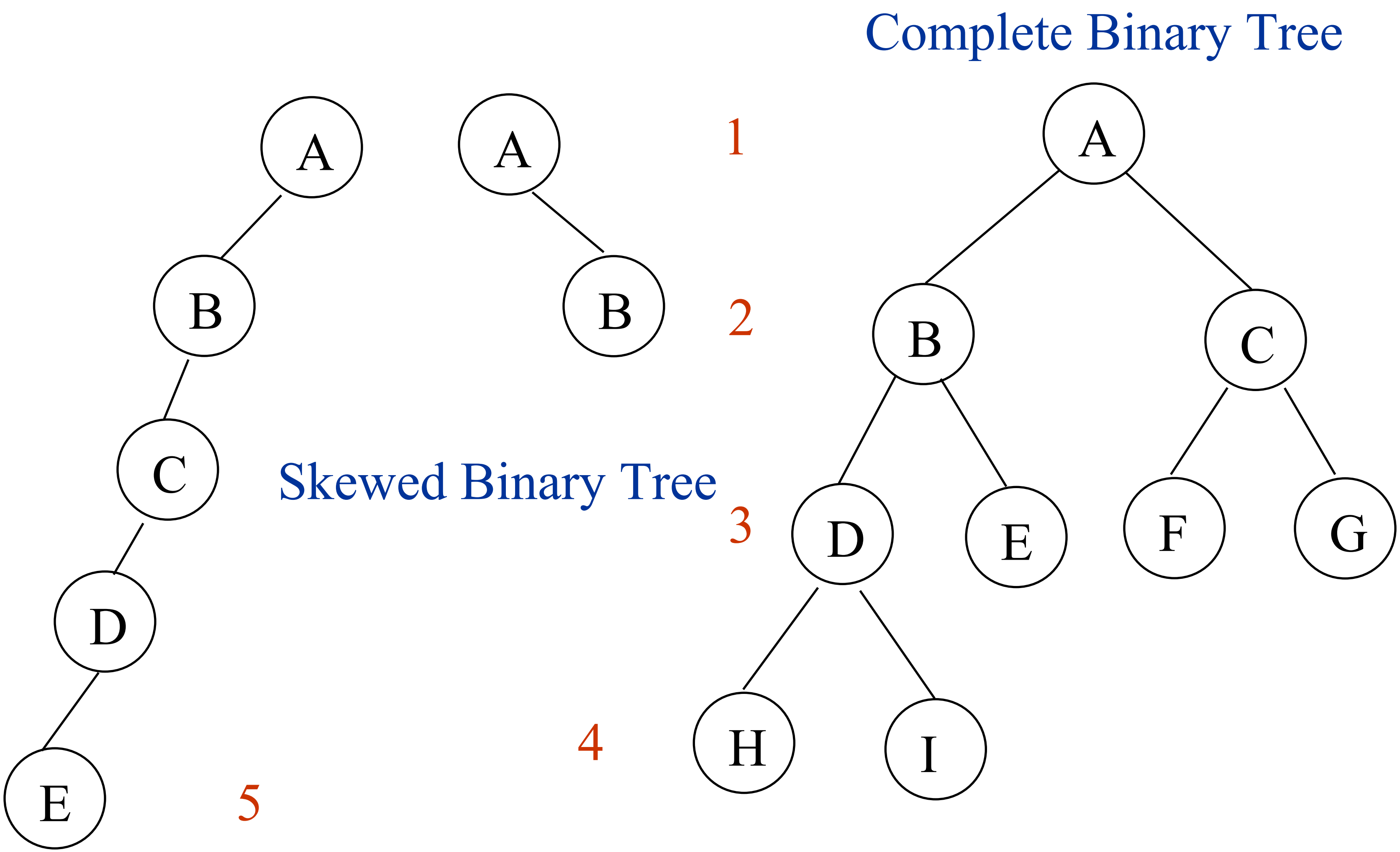
BinTree MakeBT(*bt1*, *item*, *bt2*)::= return a binary tree
whose left subtree is *bt1*, whose right subtree is *bt2*,
and whose root node contains the data *item*

Bintree Lchild(*bt*)::= if (IsEmpty(*bt*)) return error
else return the left subtree of *bt*

element Data(*bt*)::= if (IsEmpty(*bt*)) return error
else return the data in the root node of *bt*

Bintree Rchild(*bt*)::= if (IsEmpty(*bt*)) return error
else return the right subtree of *bt*

Samples of Trees



Maximum Number of Nodes in BT

- The maximum number of nodes on level i of a
i-1 binary tree is 2^{i-1}, i>=1.
- The maximum nubmer of nodes in a binary tree of depth k is 2^k-1, k>=1.

Prove by induction.

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

Relations between Number of Leaf Nodes and Nodes of Degree 2

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

proof:

Let n and B denote the total number of nodes & branches in T.

Let n_0, n_1, n_2 represent the nodes with no children, single child, and two children respectively.

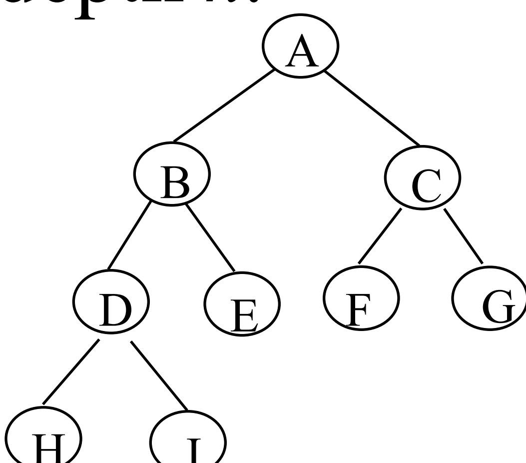
$n= n_0+n_1+n_2$, $B+1=n$, $B=n_1+2n_2$ ==> $n_1+2n_2+1= n$

$n_1+2n_2+1= n_0+n_1+n_2$ ==> $n_0=n_2+1$

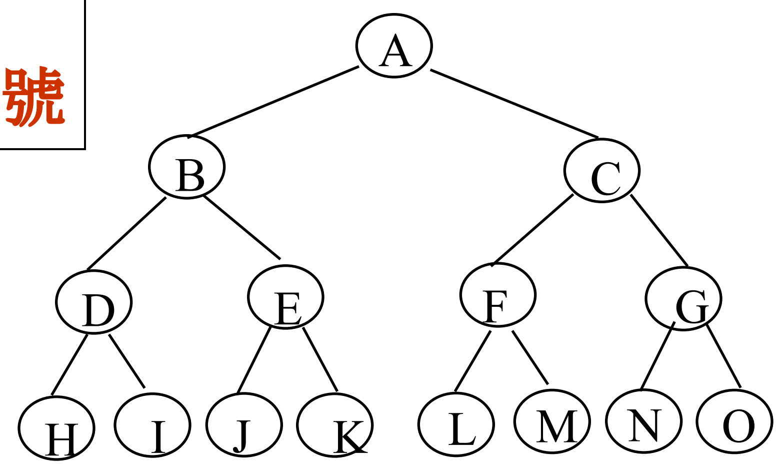
Full BT VS Complete BT

- A full binary tree of depth k is a binary tree of depth k having 2^k-1 nodes, k>=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.

由上至下，
由左至右編號



Complete binary tree

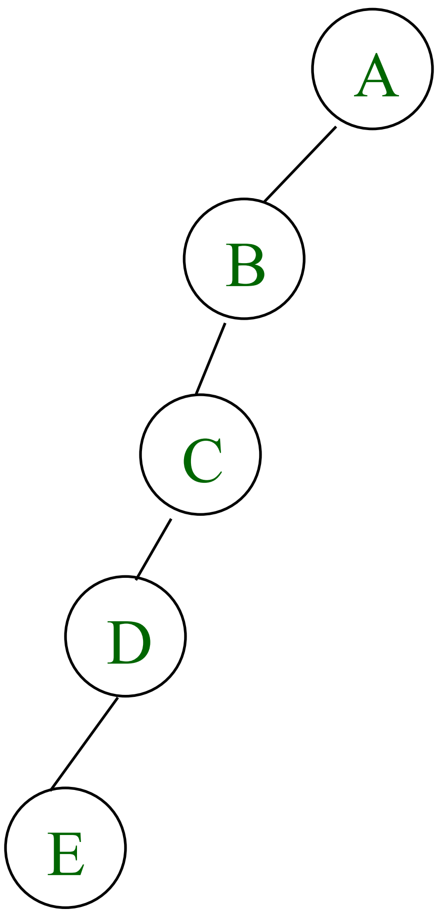


Full binary tree of depth 4

Binary Tree Representations

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

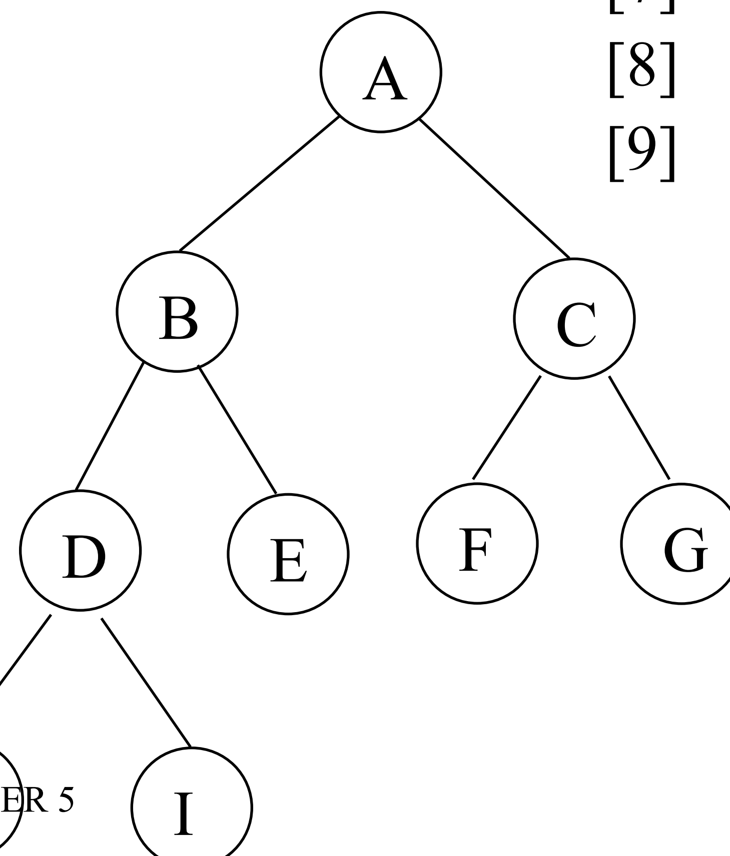
Sequential Representation



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

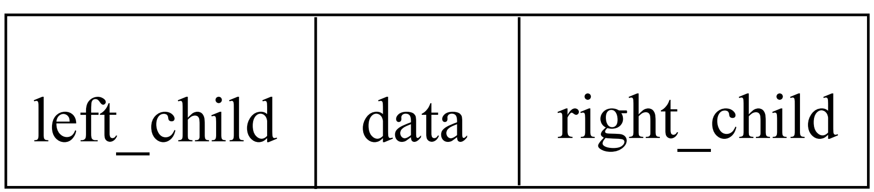
(1) waste space
(2) insertion/deletion problem

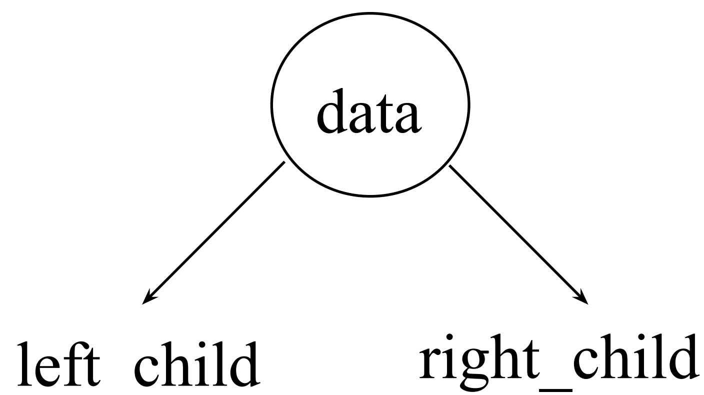
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I



Linked Representation

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

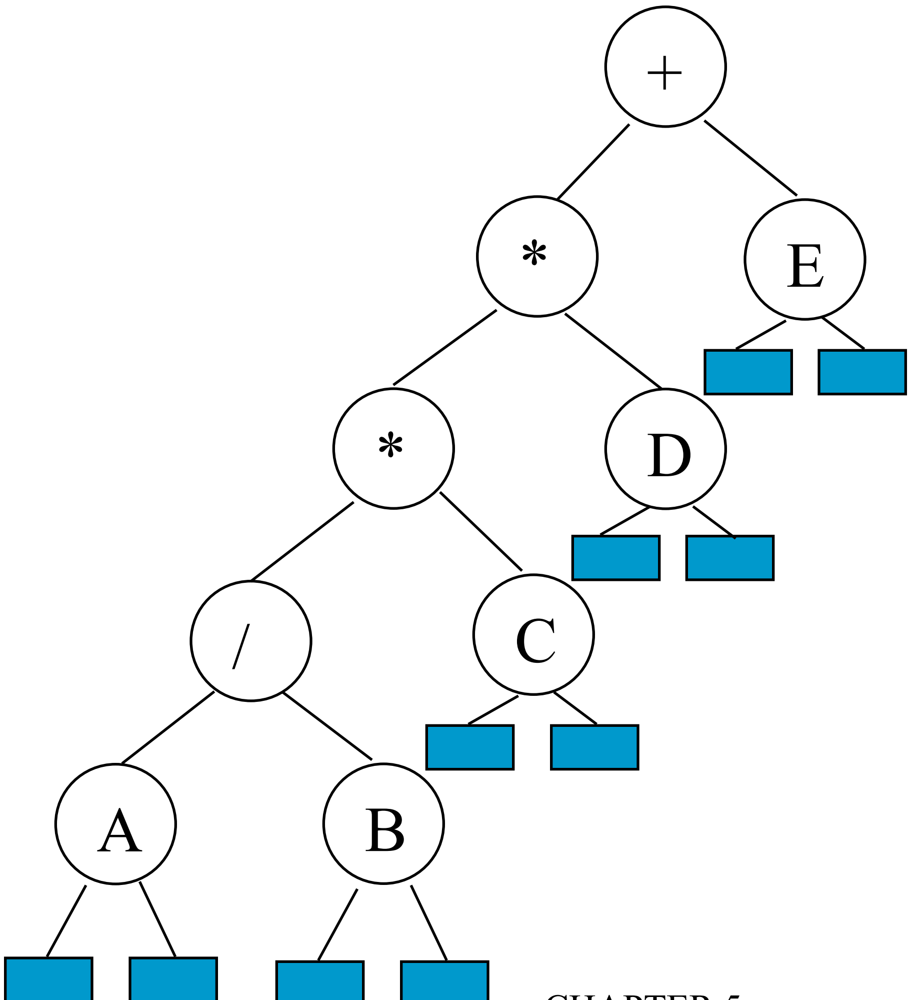




Binary Tree Traversals

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

Arithmetic Expression Using BT



inorder traversal

A / B * C * D + E

infix expression

preorder traversal

+ * * / A B C D E

prefix expression

postorder traversal

A B / C * D * E +

postfix expression

level order traversal

+ * E * D / C A B

Inorder Traversal (recursive version)

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

A / B * C * D + E

Preorder Traversal (recursive version)

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

+ * * / A B C D E

Postorder Traversal (recursive version)

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

A B / C * D * E +

Iterative Inorder Traversal

(using stack)

```
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)

Trace Operations of Inorder Traversal

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

Level Order Traversal

(using queue)

```
void 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;
  }
}
```

+ * E * D / C A B

Copying Binary Trees

```
tree_poointer copy(tree_pointer original)
{
tree_pointer temp;
if (original) {
 temp=(tree_pointer) malloc(sizeof(node));
 if (IS_FULL(temp)) {
   fprintf(stderr, "the memory is full\n");
   exit(1);
 }
 temp->left_child=copy(original->left_child);
 temp->right_child=copy(original->right_child)
 temp->data=original->data;
 return temp;
}
return NULL;
}
```

postorder

Equality of Binary Trees

the same topology and data

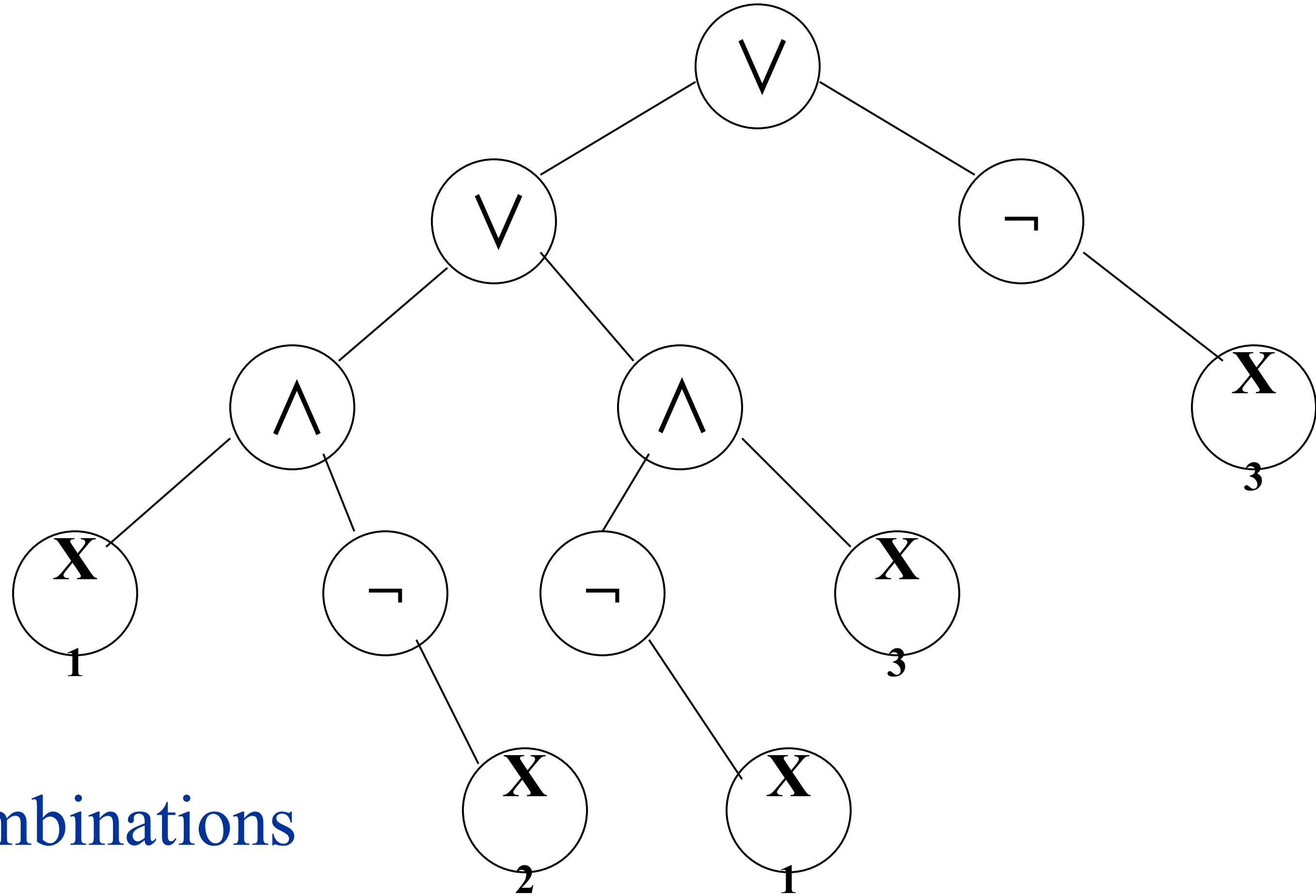
```
int equal(tree_pointer first, tree_pointer second)
{
/* function returns FALSE if the binary trees first and
   second are not equal, otherwise it returns TRUE */

  return ((!first && !second) || (first && second &&
        (first->data == second->data) &&
        equal(first->left_child, second->left_child) &&
        equal(first->right_child, second->right_child))
}
```

Propositional Calculus Expression

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

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

(t,t,t)
(t,t,f)
(t,f,t)
(t,f,f)
(f,t,t)
(f,t,f)
(f,f,t)
(f,f,f)



2ⁿ possible combinations for n variables

postorder traversal (postfix evaluation)

node structure

left_child	***data***	***value***	***right_child***

```
typedef emun {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");
```

Post-order-eval function

```
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;
    }
  }
}
```

Threaded Binary Trees

- 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

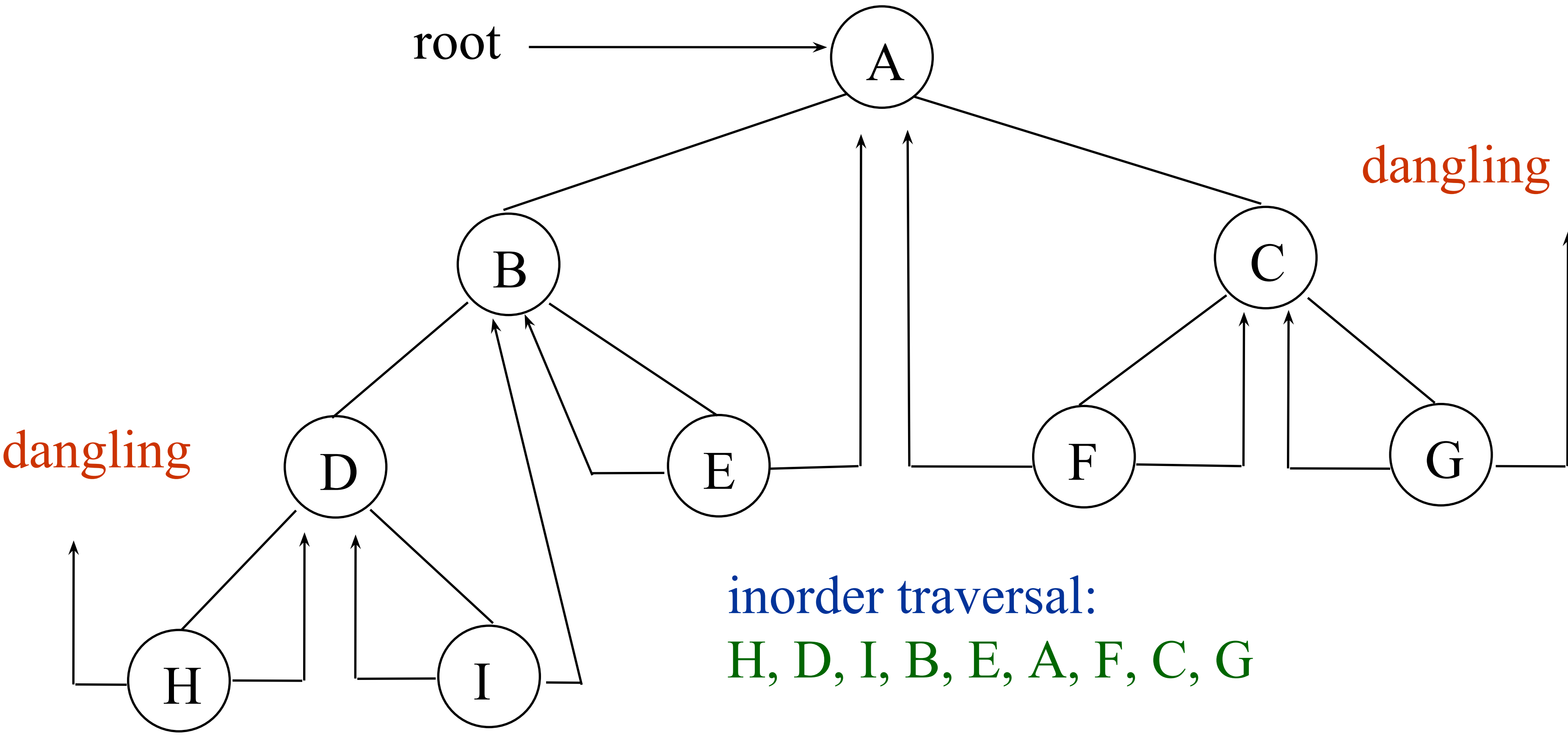
- Replace these null pointers with some useful "threads".

Threaded Binary Trees (*Continued*)

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

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

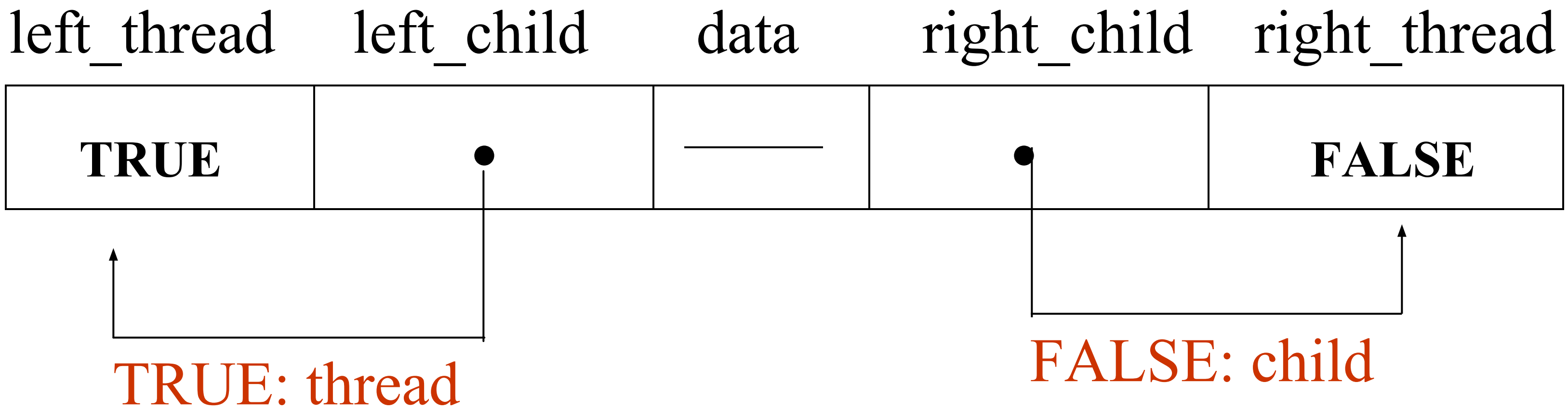
A Threaded Binary Tree



inorder traversal:

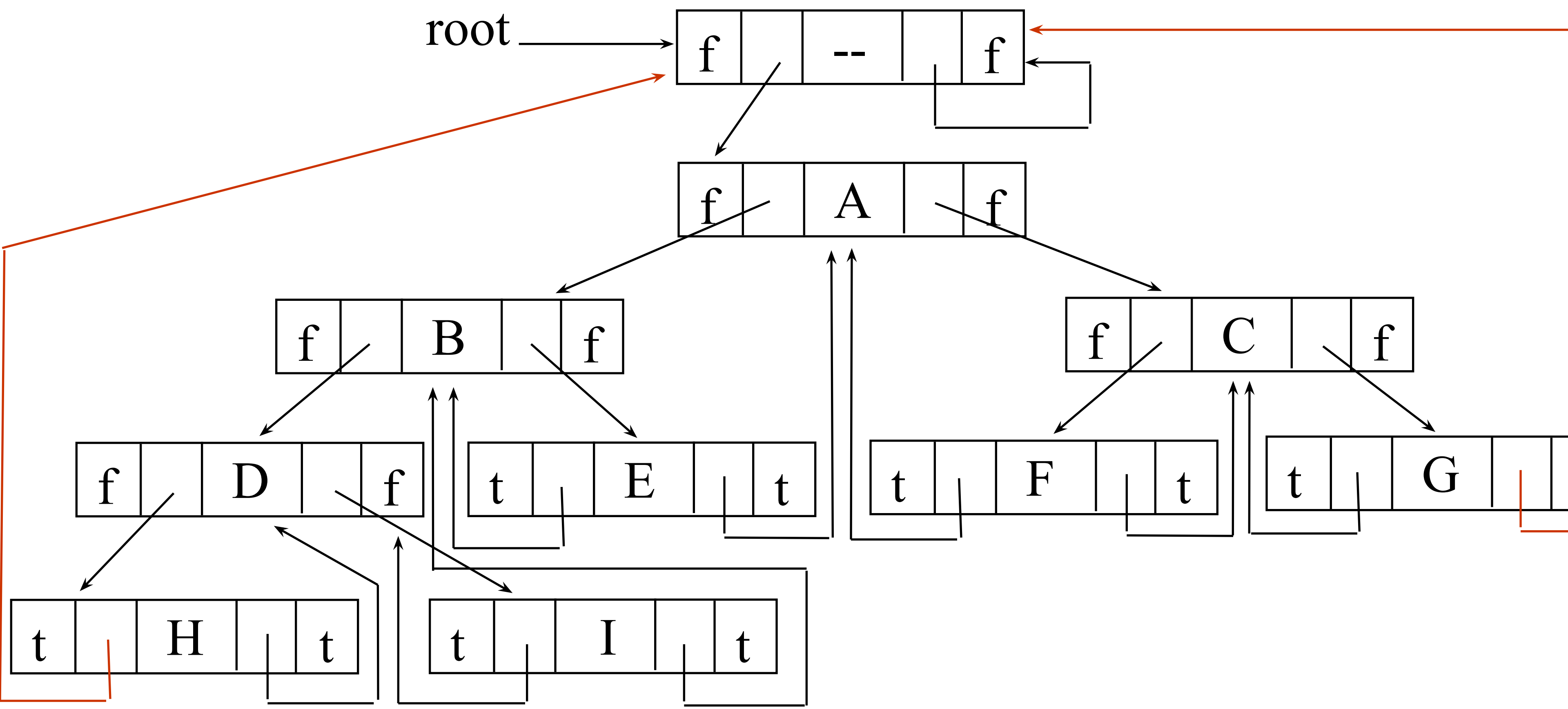
H, D, I, B, E, A, F, C, G

Data Structures for Threaded BT



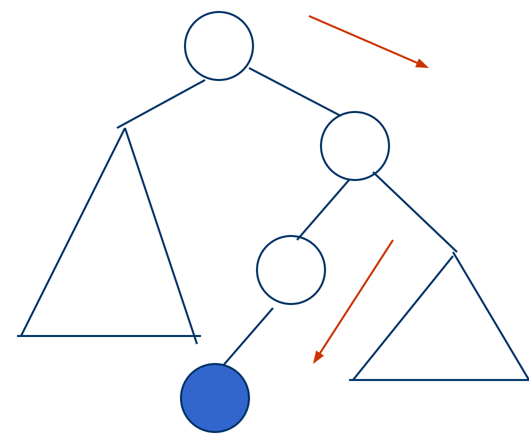

```
typedef struct threaded_tree
  *threaded_pointer;
typedef struct threaded_tree {
     short int left_thread;
     threaded_pointer left_child;
     char data;
     threaded_pointer right_child;
     short int right_thread;   };
```

Memory Representation of A Threaded BT



Next Node in Threaded BT

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



Inorder Traversal of Threaded BT

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

O(n)

Inserting Nodes into Threaded BTs

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

Examples

Insert a node D as a right child of B.

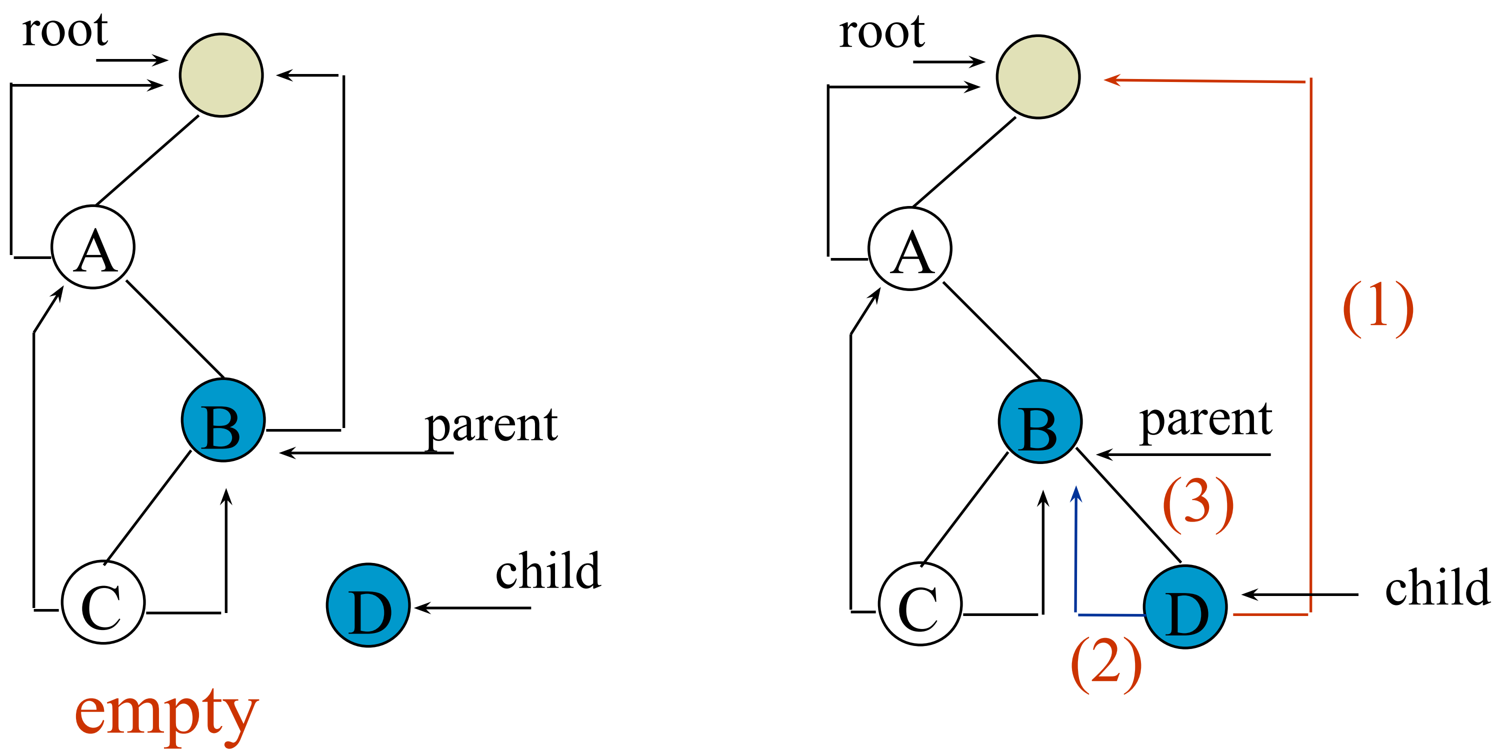
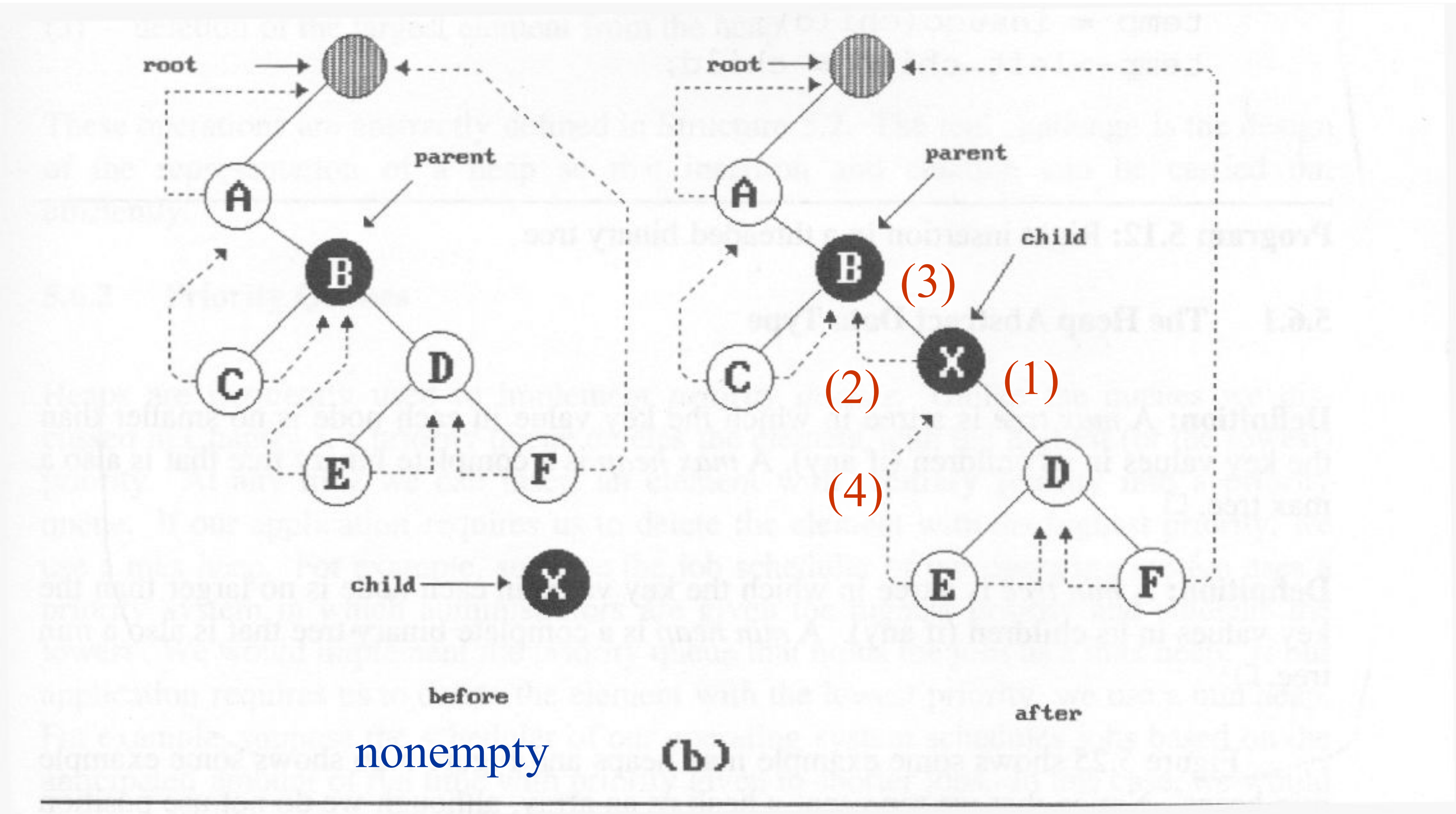


***Figure 5.24:** Insertion of child as a right child of parent in a threaded binary tree (p.217)



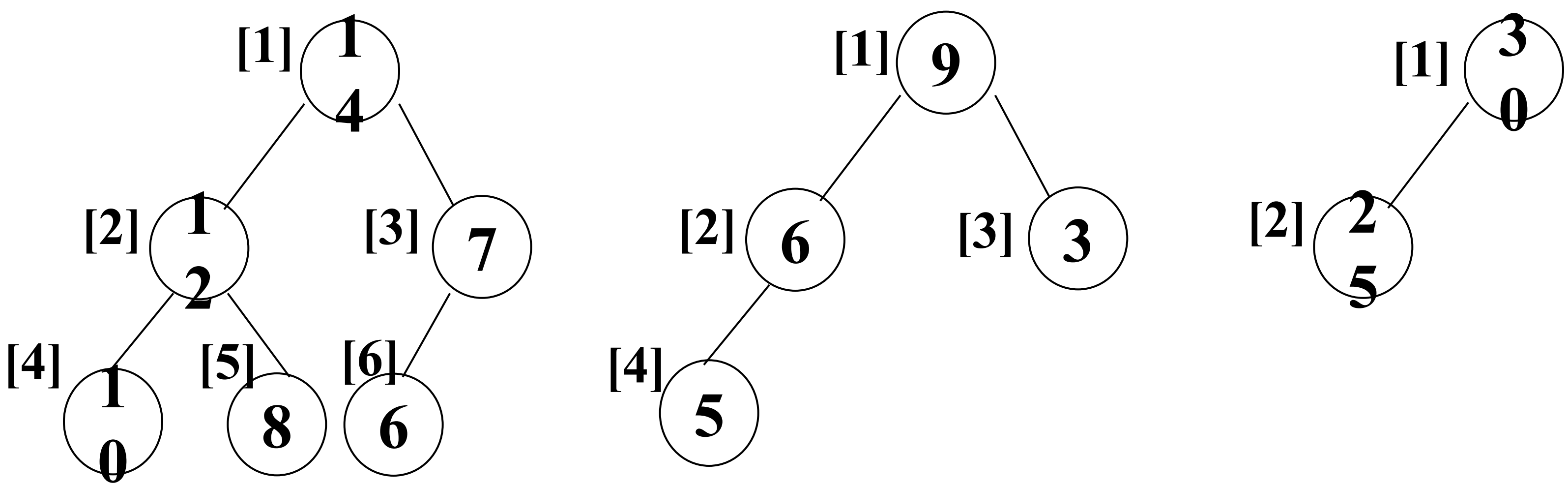
Right Insertion in Threaded BTs

```
void insert_right(threaded_pointer parent,
                  threaded_pointer child)
{
    threaded_pointer temp;
    child->right_child = parent->right_child;
(1) child->right_thread = parent->right_thread;
    child->left_child = parent;  case (a)
(2) child->left_thread = TRUE;
    parent->right_child = child;
(3) parent->right_thread = FALSE;
    if (!child->right_thread) { case (b)
      temp = insucc(child);
(4)   temp->left_child = child;
    }
}
```

Heap

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

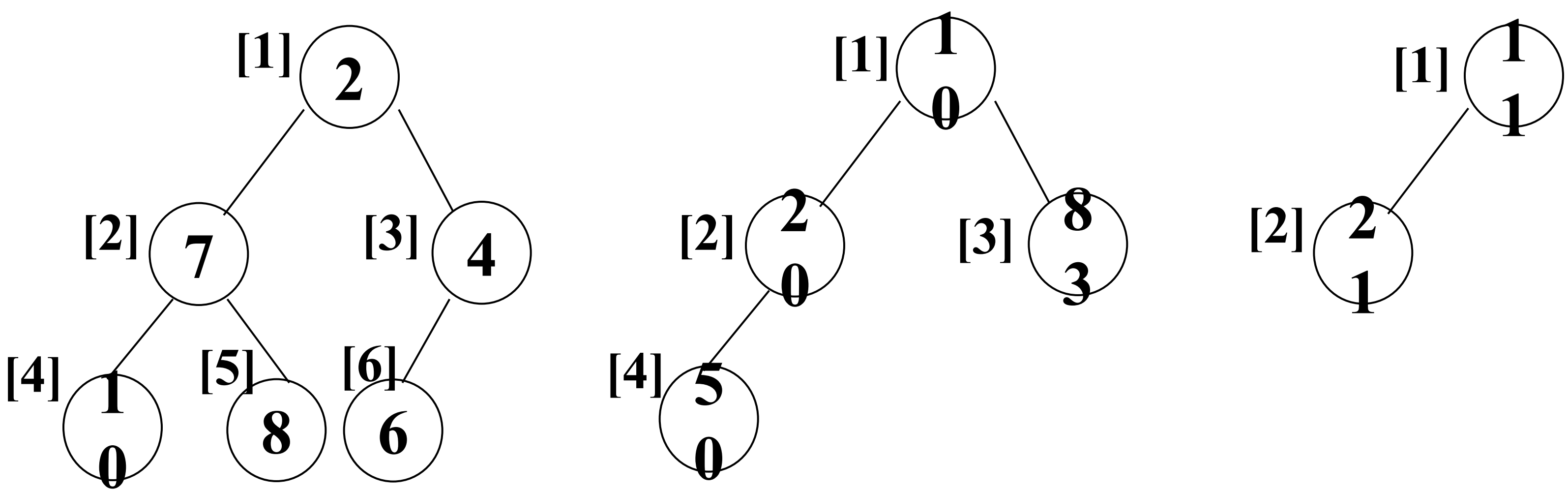
***Figure 5.25:** Sample max heaps (p.219)



Property:

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

*Figure 5.26:Sample min heaps (p.220)



ADT for Max Heap

structure MaxHeap

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

functions:

for all *heap* belong to *MaxHeap*, *item* belong to *Element*, *n*, *max_size* belong to integer

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

Boolean HeapFull(heap, n)::= if (n==max_size) return TRUE
else return FALSE

MaxHeap Insert(heap, item, n)::= if (!HeapFull(heap,n)) insert item into heap and return the resulting heap
else return error

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

Element Delete(heap,n)::= if (!HeapEmpty(heap,n)) return one instance of the largest element in the heap and remove it from the heap
else return error

Application: priority queue

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

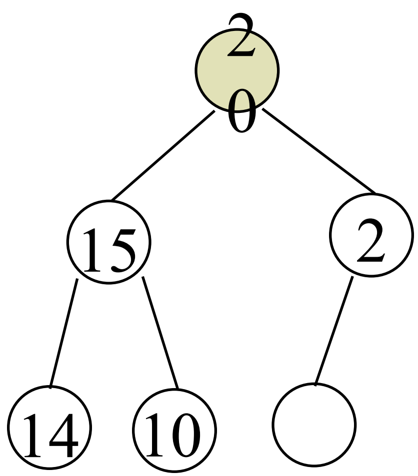
Data Structures

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

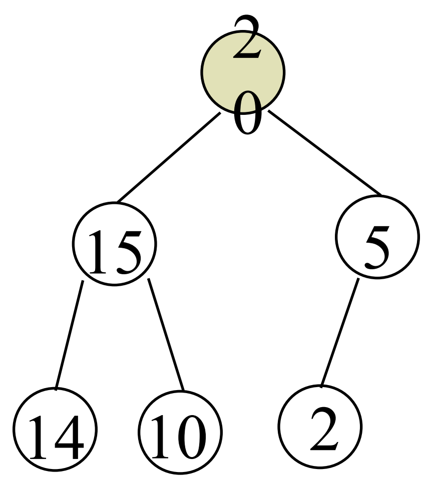
***Figure 5.27:** Priority queue representations (p.221)

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

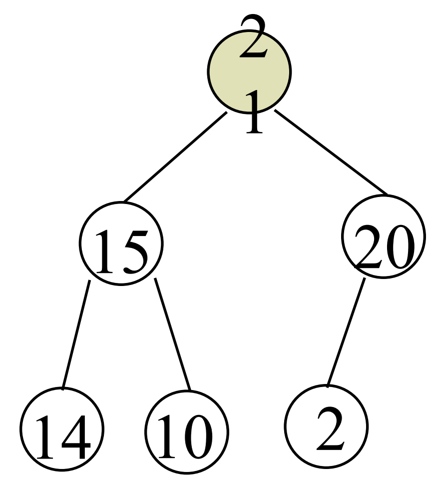
Example of Insertion to Max Heap



initial location of new node



insert 5 into heap



insert 21 into heap

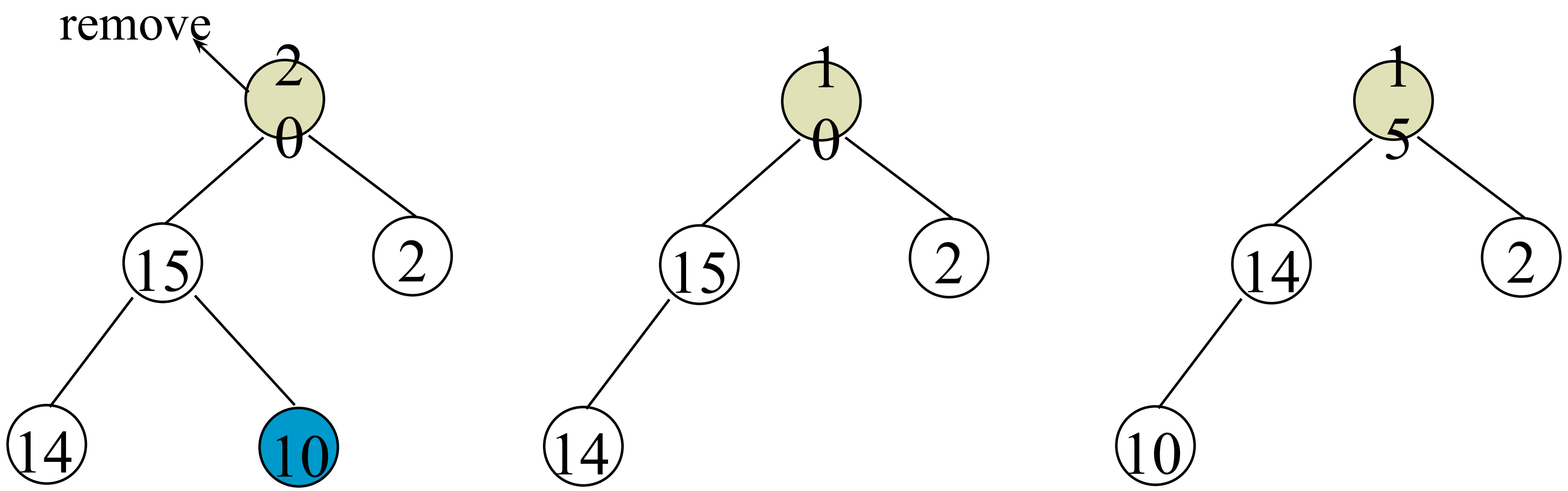
Insertion into a Max Heap

```
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;
}
```

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

$O(\log_2 n)$

Example of Deletion from Max Heap



Deletion from a Max Heap

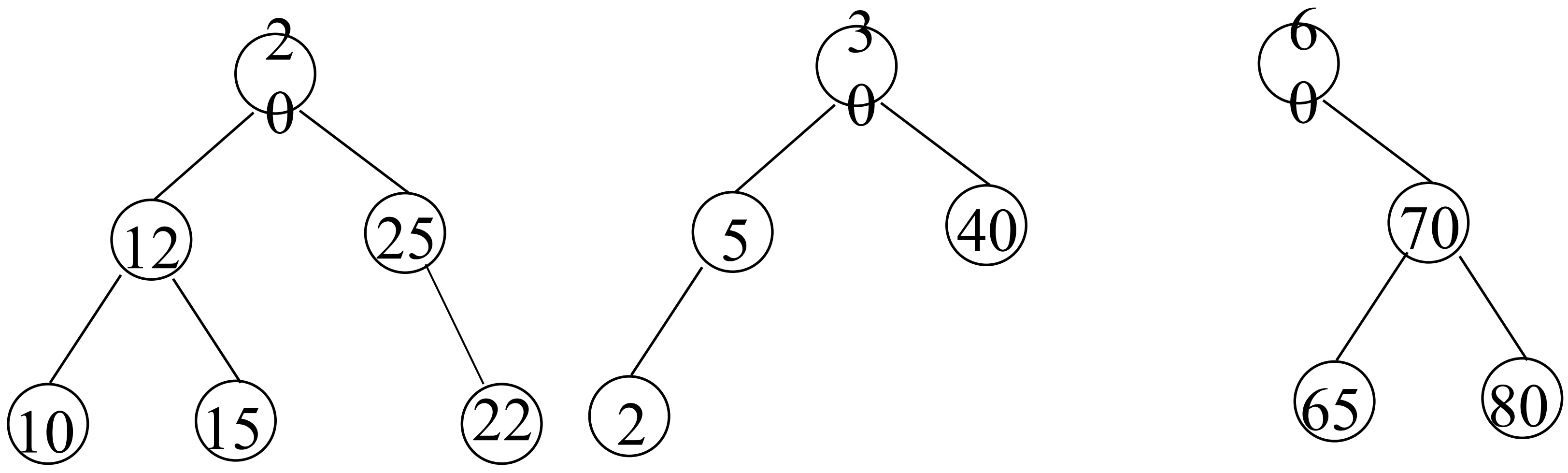
```
element delete_max_heap(int *n)
{
  int parent, child;
  element item, temp;
  if (HEAP_EMPTY(*n)) {
    fprintf(stderr, "The heap is empty\n");
    exit(1);
  }
  /* save value of the element with the
     highest key */
  item = heap[1];
  /* use last element in heap to adjust heap
  temp = heap[(*n)--];
  parent = 1;
  child = 2;
```

```
  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;
  }
```

Binary Search Tree

- Heap
  - a min (max) element is deleted. $O(\log_2 n)$
  - deletion of an arbitrary element $O(n)$
  - search for an arbitrary element $O(n)$
- Binary search tree
  - Every element has a unique key.
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
  - The left and right subtrees are also binary search trees.

Examples of Binary Search Trees



Searching a Binary Search Tree

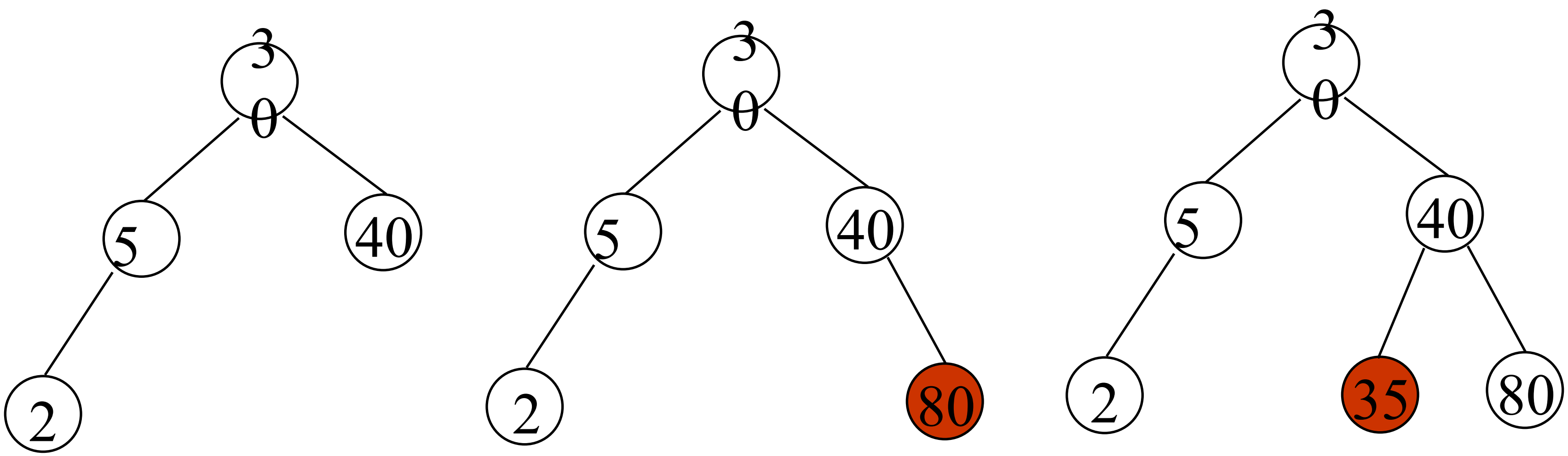
```
tree_pointer search(tree_pointer root,
                int key)
{
/* return a pointer to the node that
  contains key. If there is no such
  node, return NULL */

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

Another Searching Algorithm

```
tree_pointer 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)

Insert Node in Binary Search Tree



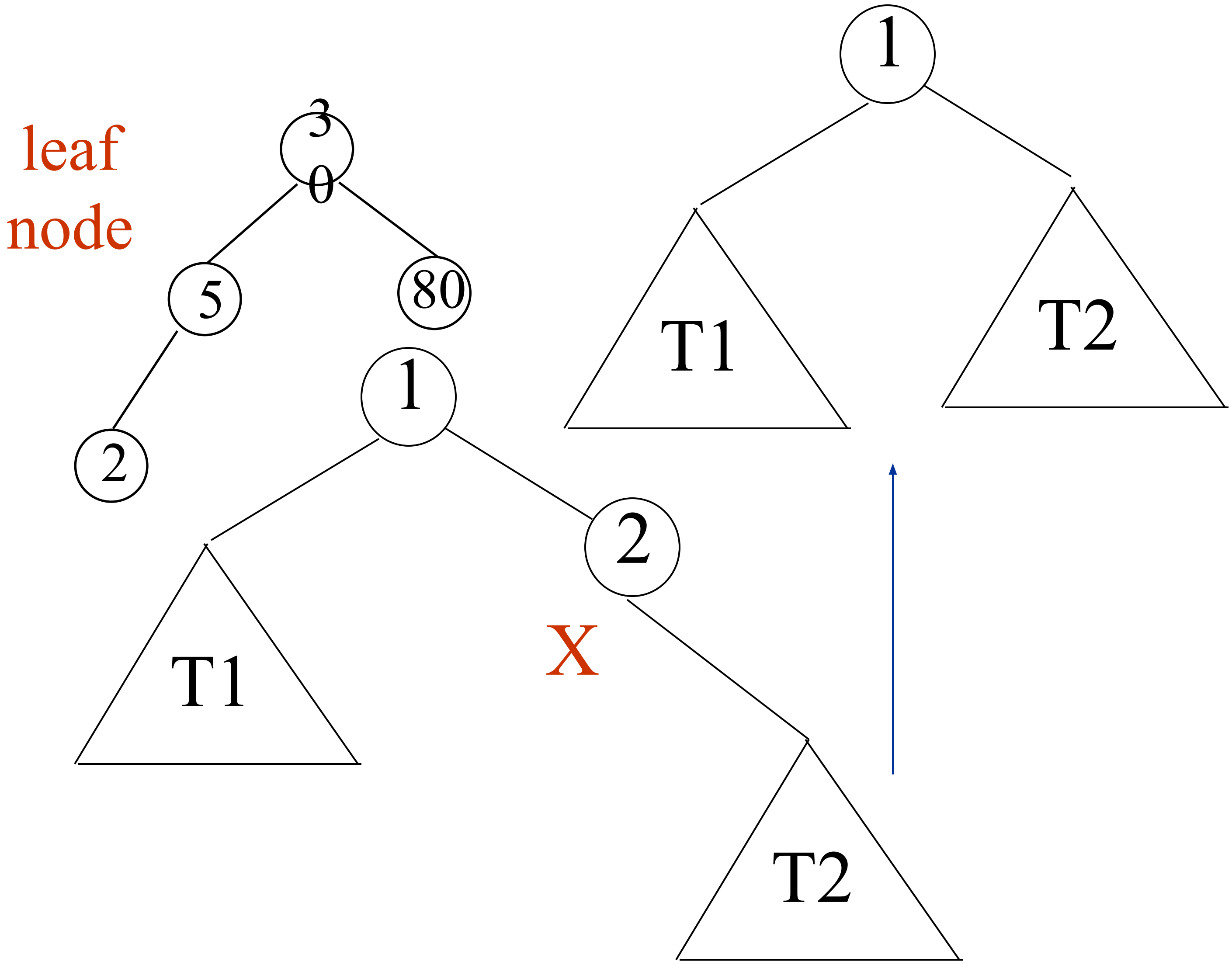
Insert 80

Insert 35

Insertion into A Binary Search Tree

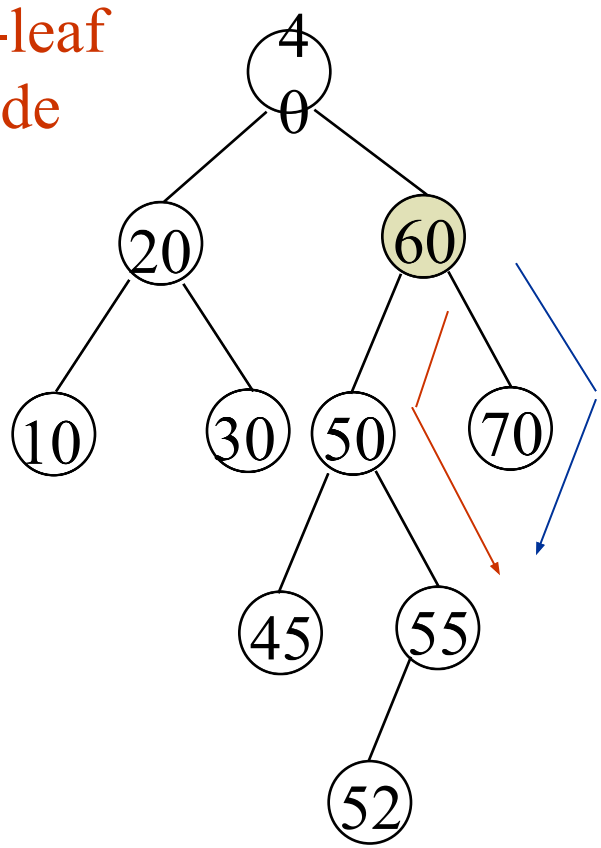
```
void insert_node(tree_pointer *node, int num)
{tree_pointer ptr,
      temp = modified_search(*node, num);
  if (temp || !(*node)) {
   ptr = (tree_pointer) malloc(sizeof(node));
   if (IS_FULL(ptr)) {
     fprintf(stderr, "The memory is full\n");
     exit(1);
   }
   ptr->data = num;
   ptr->left_child = ptr->right_child = NULL;
   if (*node)
     if (num<temp->data) temp->left_child=ptr;
        else temp->right_child = ptr;
   else *node = ptr;
  }
}
```

Deletion for A Binary Search Tree

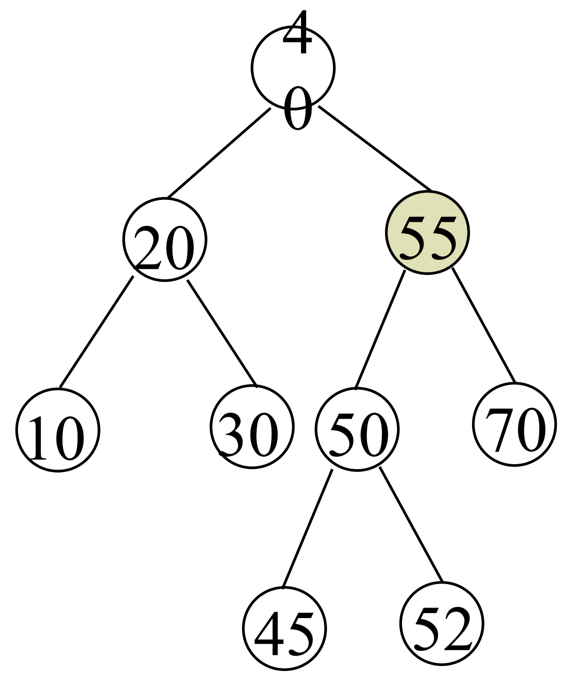


Deletion for A Binary Search Tree

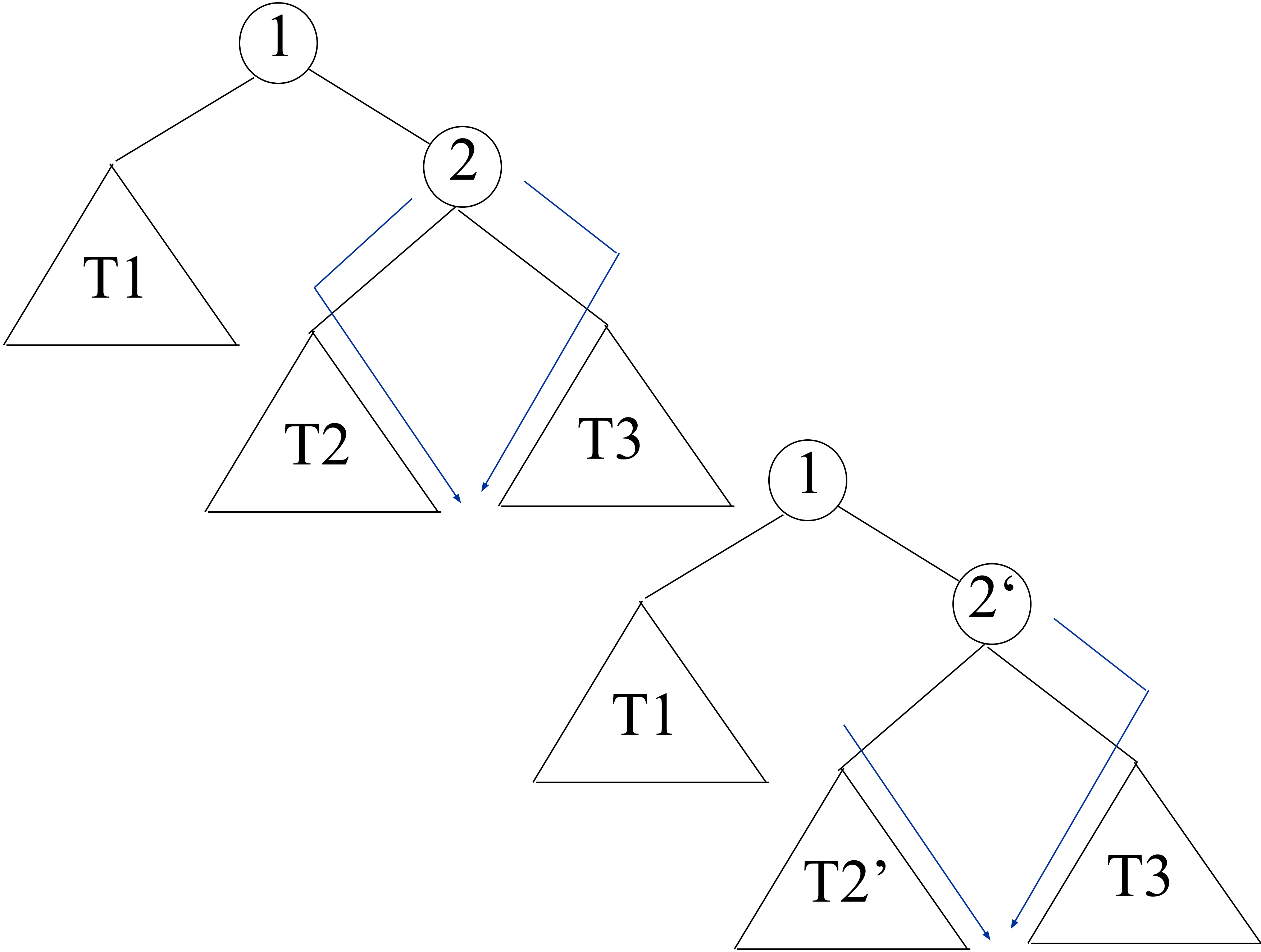
non-leaf node



Before deleting 60



After deleting 60


1
2
T1
T2
T3
1
2‘
T1
T2’
T3

Selection Trees

(1) winner tree
(2) loser tree

winner tree

sequential allocation scheme (complete binary tree)

Each node represents the smaller of its two children.

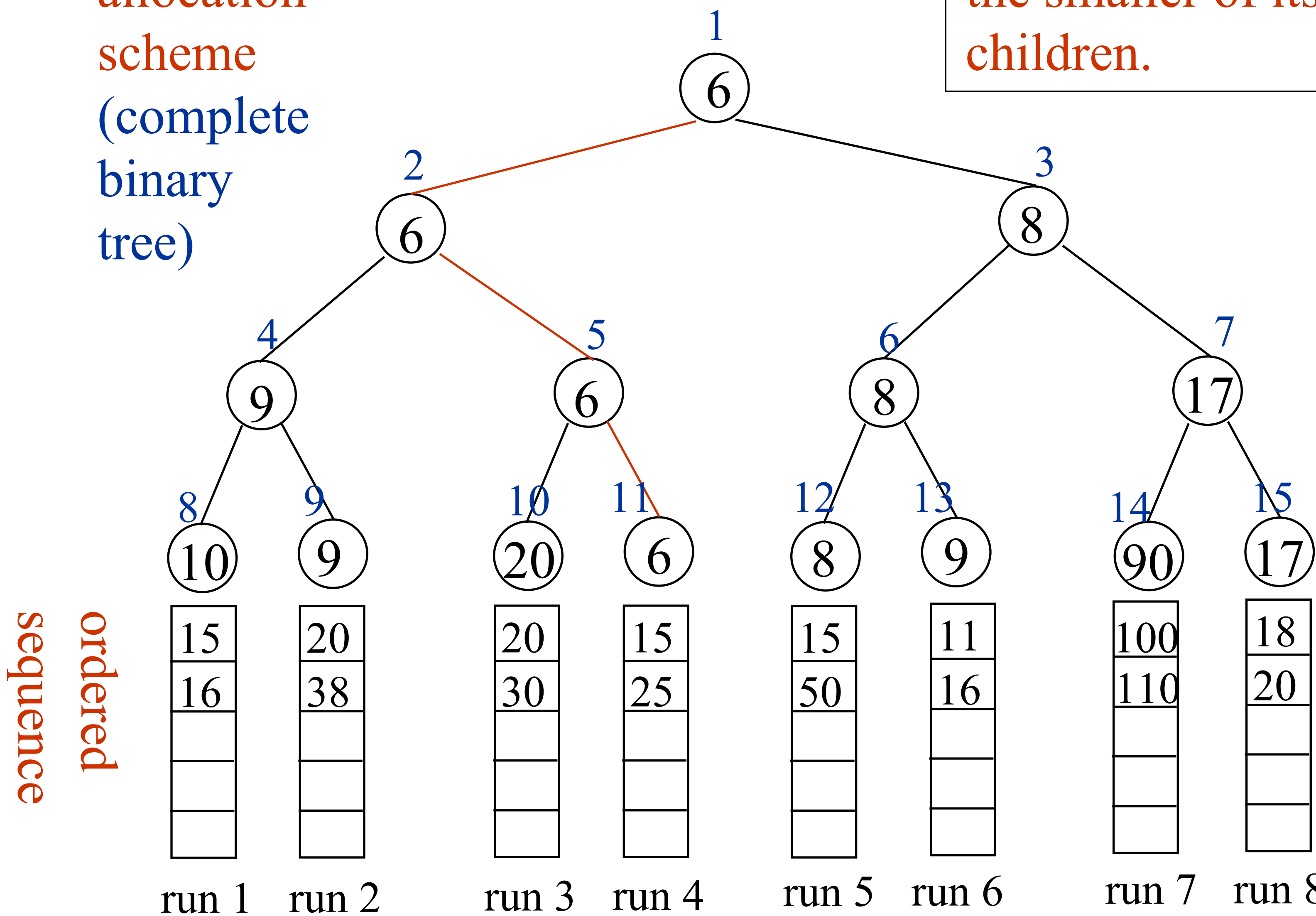
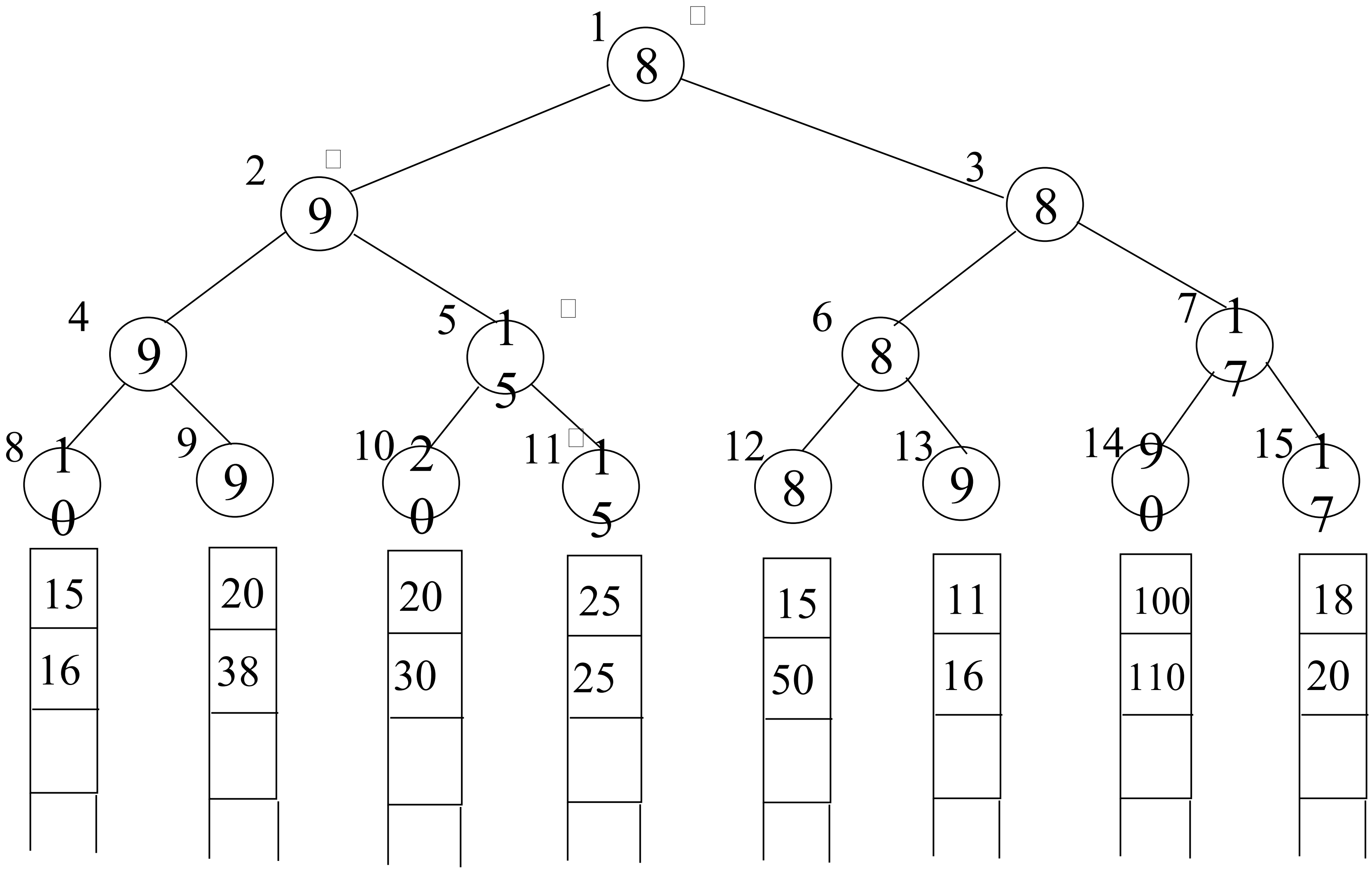


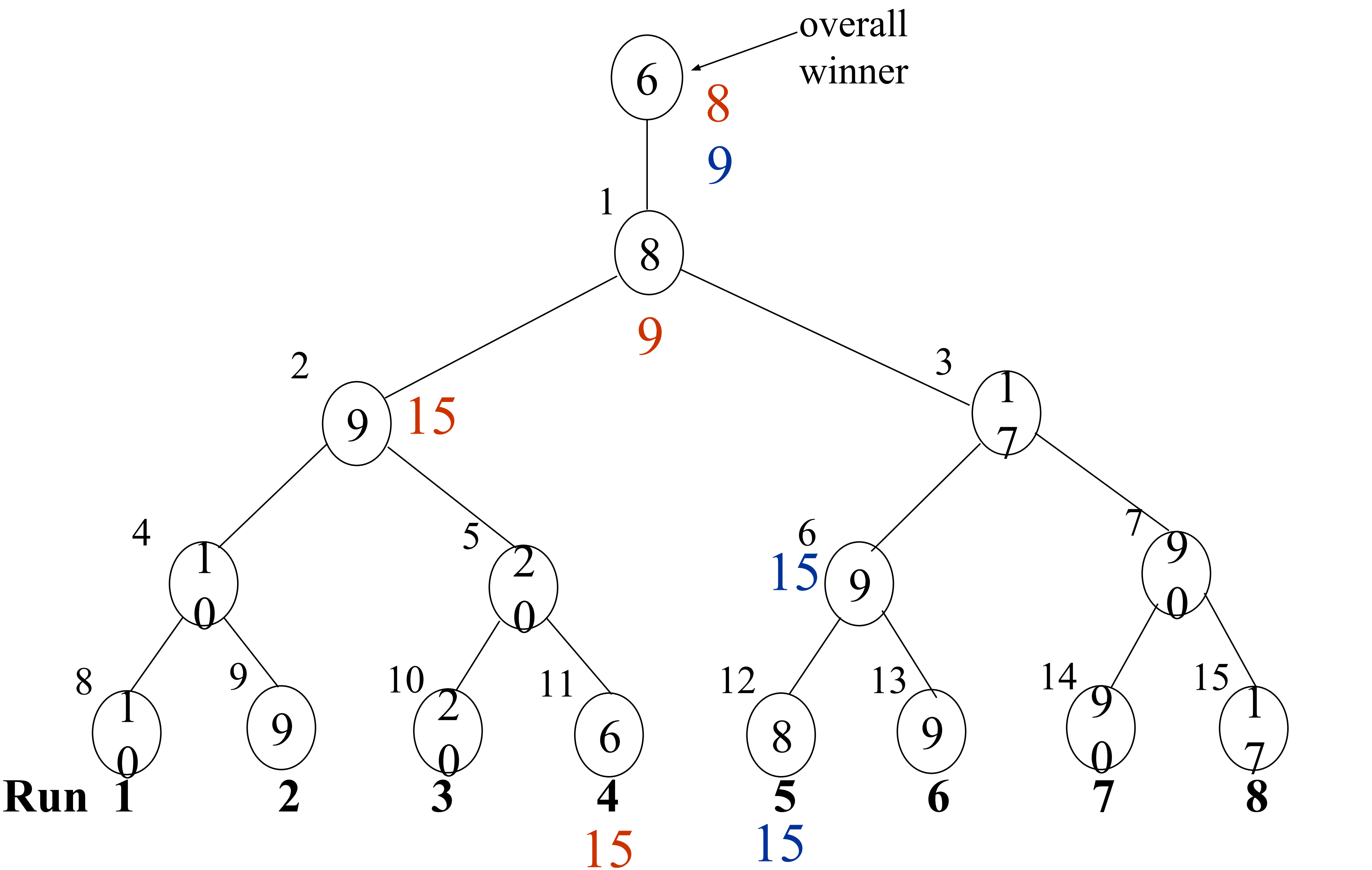
***Figure 5.35:** Selection tree of Figure 5.34 after one record has been output and the tree restructured(nodes that were changed are ticked)*



Analysis

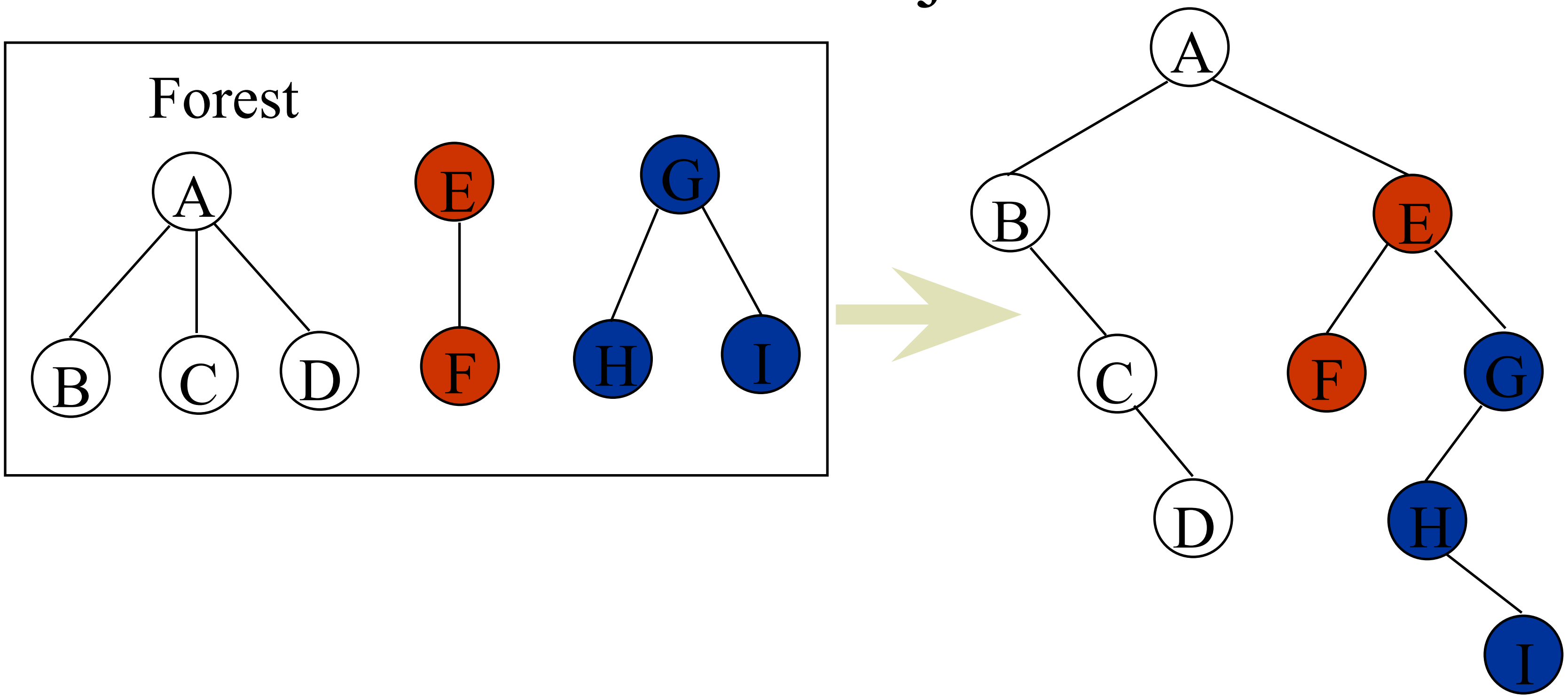
- K: # of runs
- n: # of records
- setup time: $O(K)$ (K-1)
- restructure time: $O(\log_2 K)$ $\lceil \log_2(K+1) \rceil$
- merge time: $O(n\log_2 K)$
- slight modification: tree of loser
  - consider the parent node only (vs. sibling nodes)

***Figure 5.36:** Tree of losers corresponding to Figure 5.34 (p.235)



Forest

- A forest is a set of n >= 0 disjoint trees



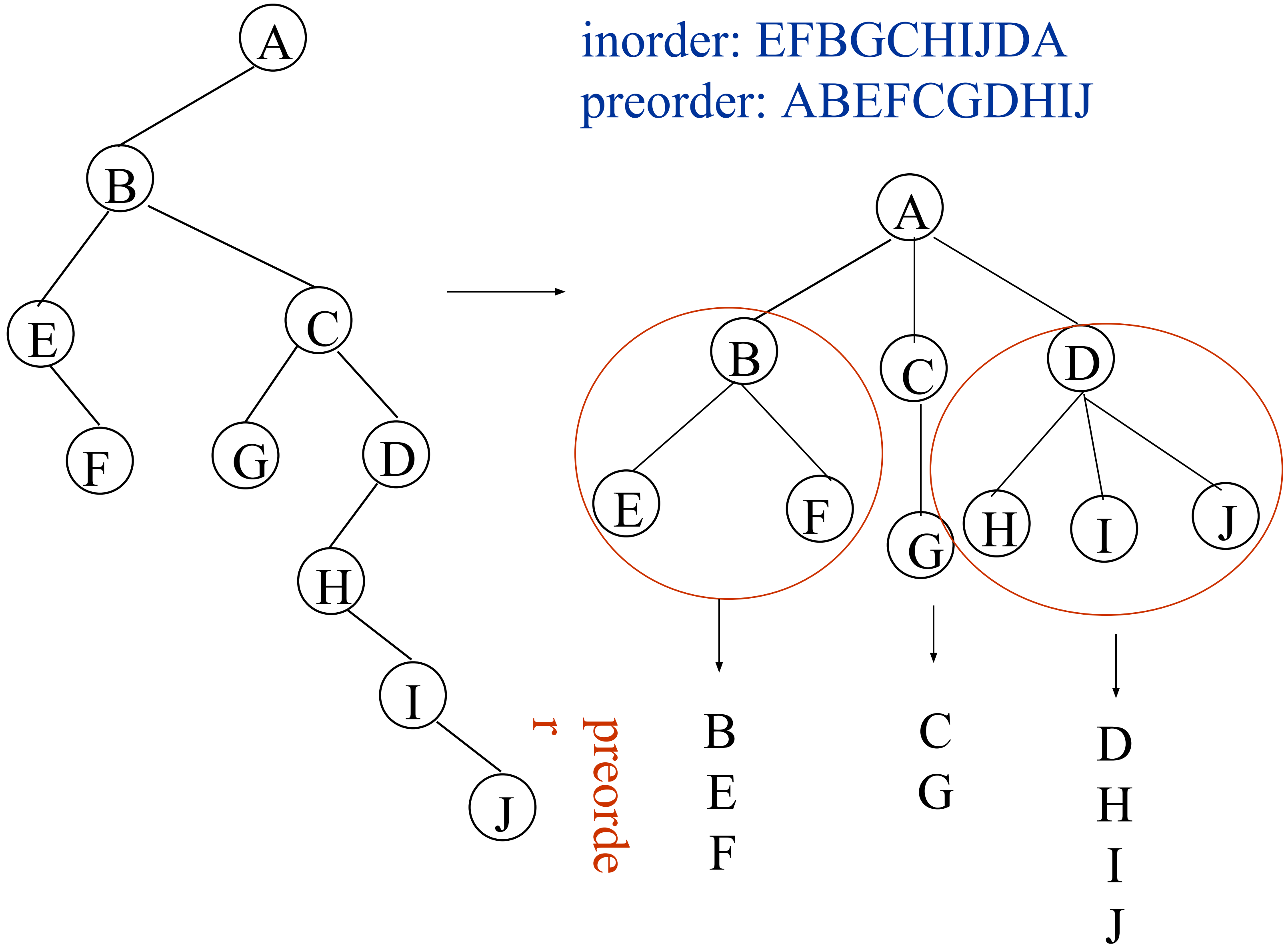
Transform a forest into a binary tree

- $T1, T2, \ldots, Tn$: a forest of trees
  $B(T1, T2, \ldots, Tn)$: a binary tree corresponding to this forest
- algorithm
  (1) empty, if $n = 0$
  (2) has root equal to root(T1)
  has left subtree equal to $B(T11,T12,\ldots,T1m)$
  has right subtree equal to $B(T2,T3,\ldots,Tn)$

Forest Traversals

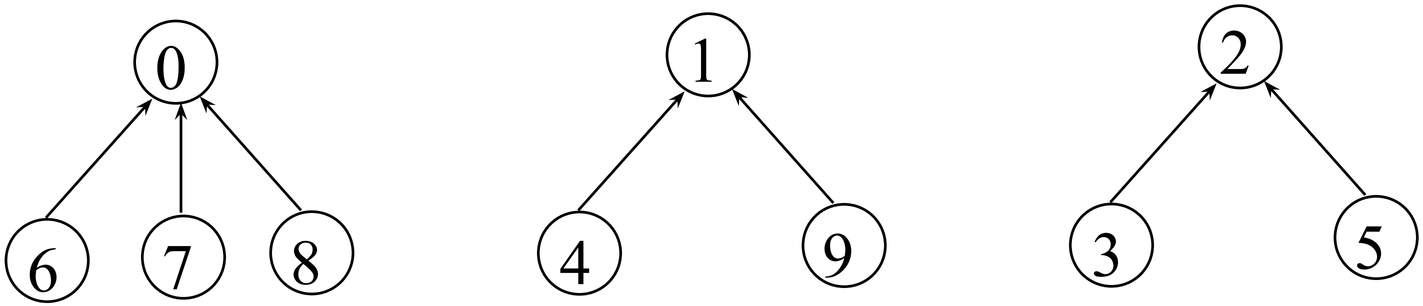
- Preorder
  - If F is empty, then return
  - Visit the root of the first tree of F
  - Taverse the subtrees of the first tree in tree preorder
  - Traverse the remaining trees of F in preorder
- Inorder
  - If F is empty, then return
  - Traverse the subtrees of the first tree in tree inorder
  - Visit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

inorder: EFBGCHIJDA
preorder: ABEFCGDHIJ



Set Representation

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



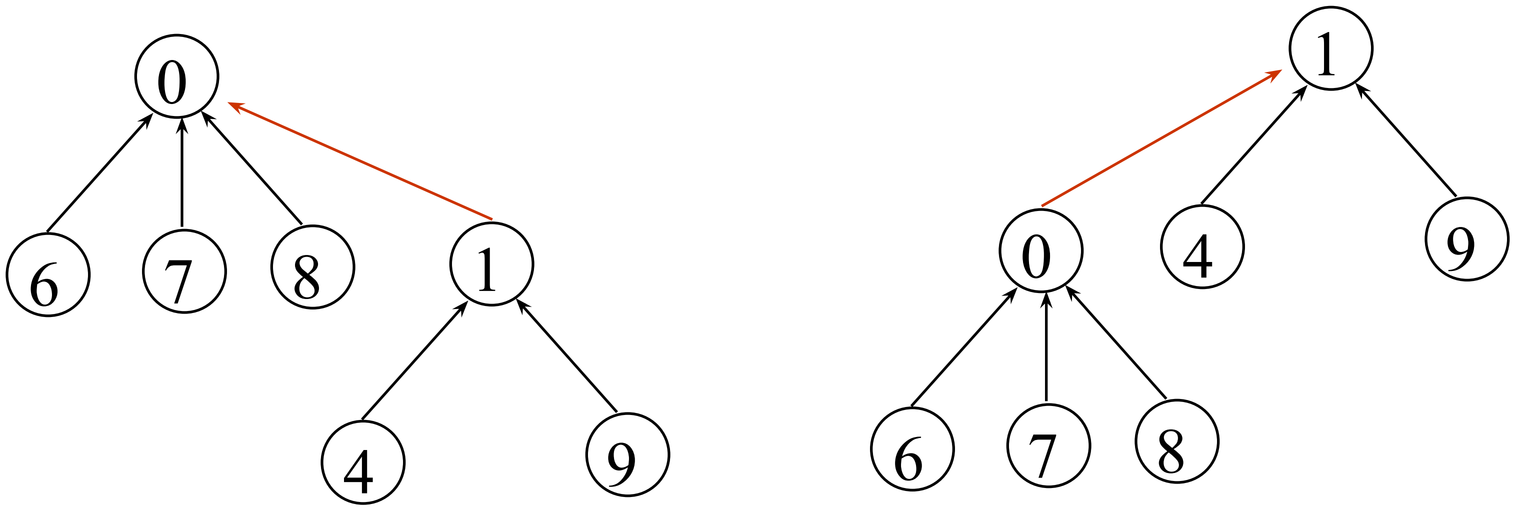
$S_i \cap S_j = \varphi$

- 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*.

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

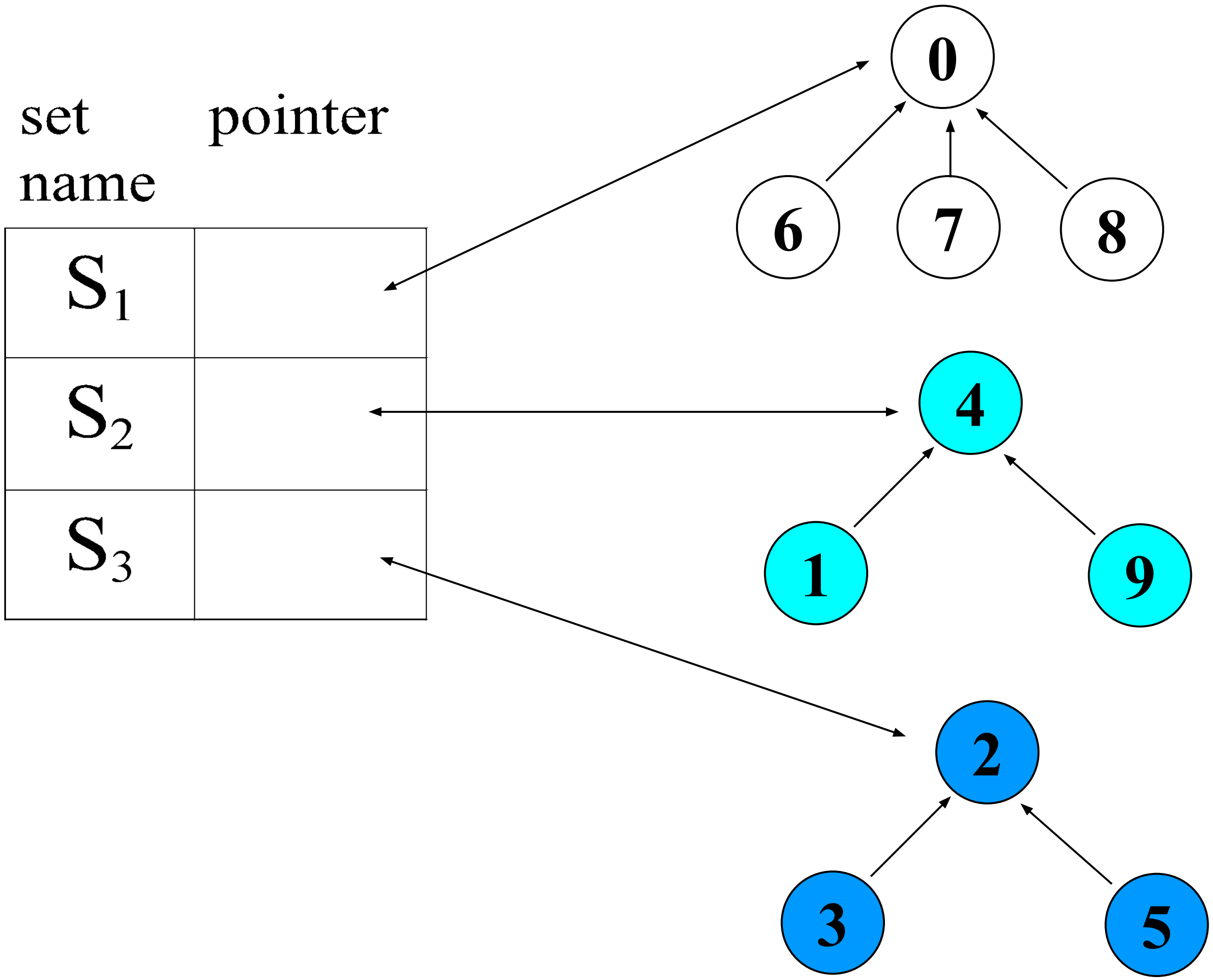
Disjoint Set Union

Make one of trees a subtree of the other



Possible representation for S_1 *union* S_2

***Figure 5.41:** Data Representation of S_1 S_2 and S_3 (p.240)



Array Representation for Set

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

```
int find1(int i)
{
    for (; parent[i]>=0; i=parent[i])
    return i;
}

void union1(int i, int j)
{
    parent[i]= j;
}
```

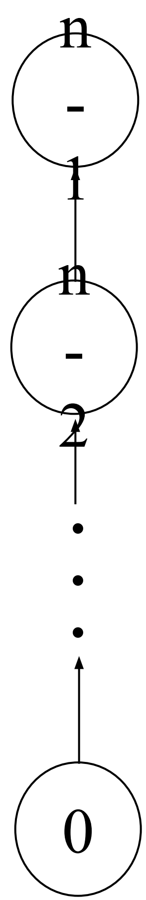
***Figure 5.43:**Degenerate tree (p.242)

union operation

O(n)   n-1

find operation

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



degenerate tree

union(0,1), find(0)

union(1,2), find(0)

.

.

.

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

***Figure 5.44:** Trees obtained using the weighting rule(p.243)

weighting rule for union(i,j): if # of nodes in i < # in j then j the parent of i

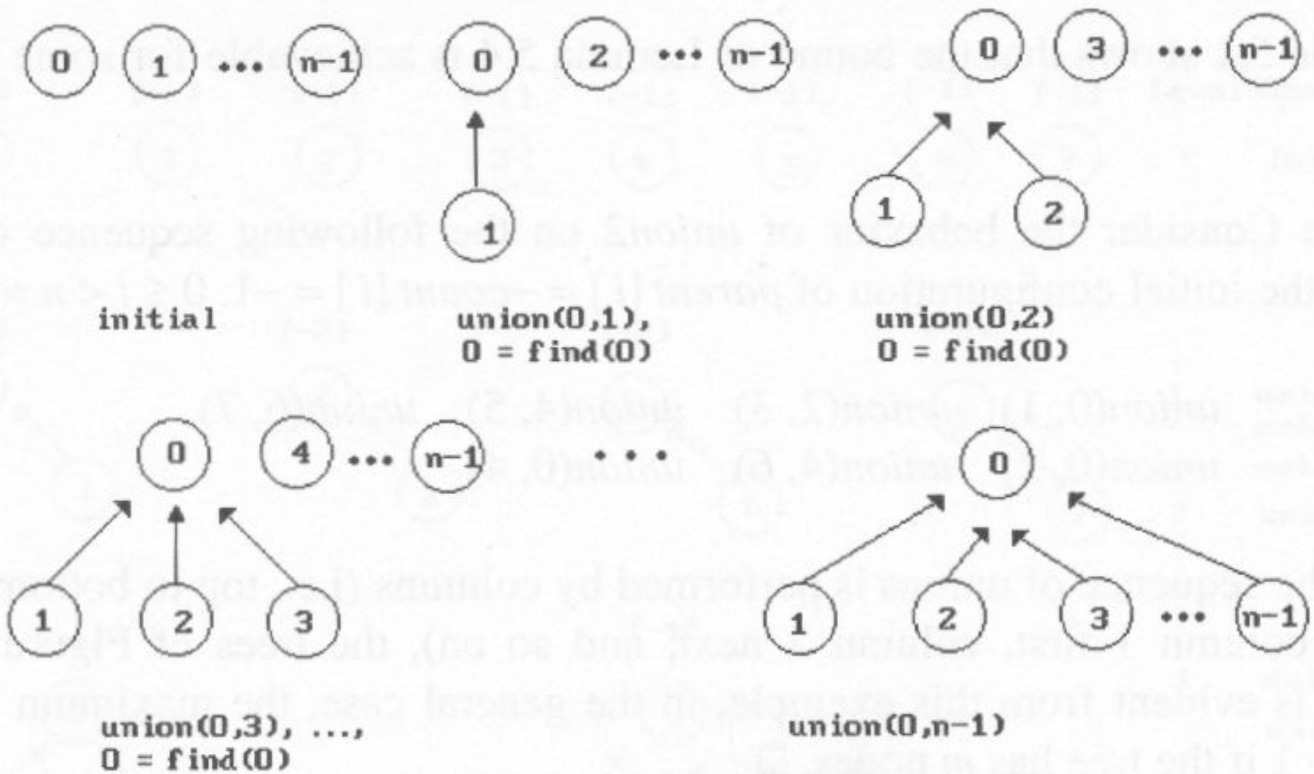


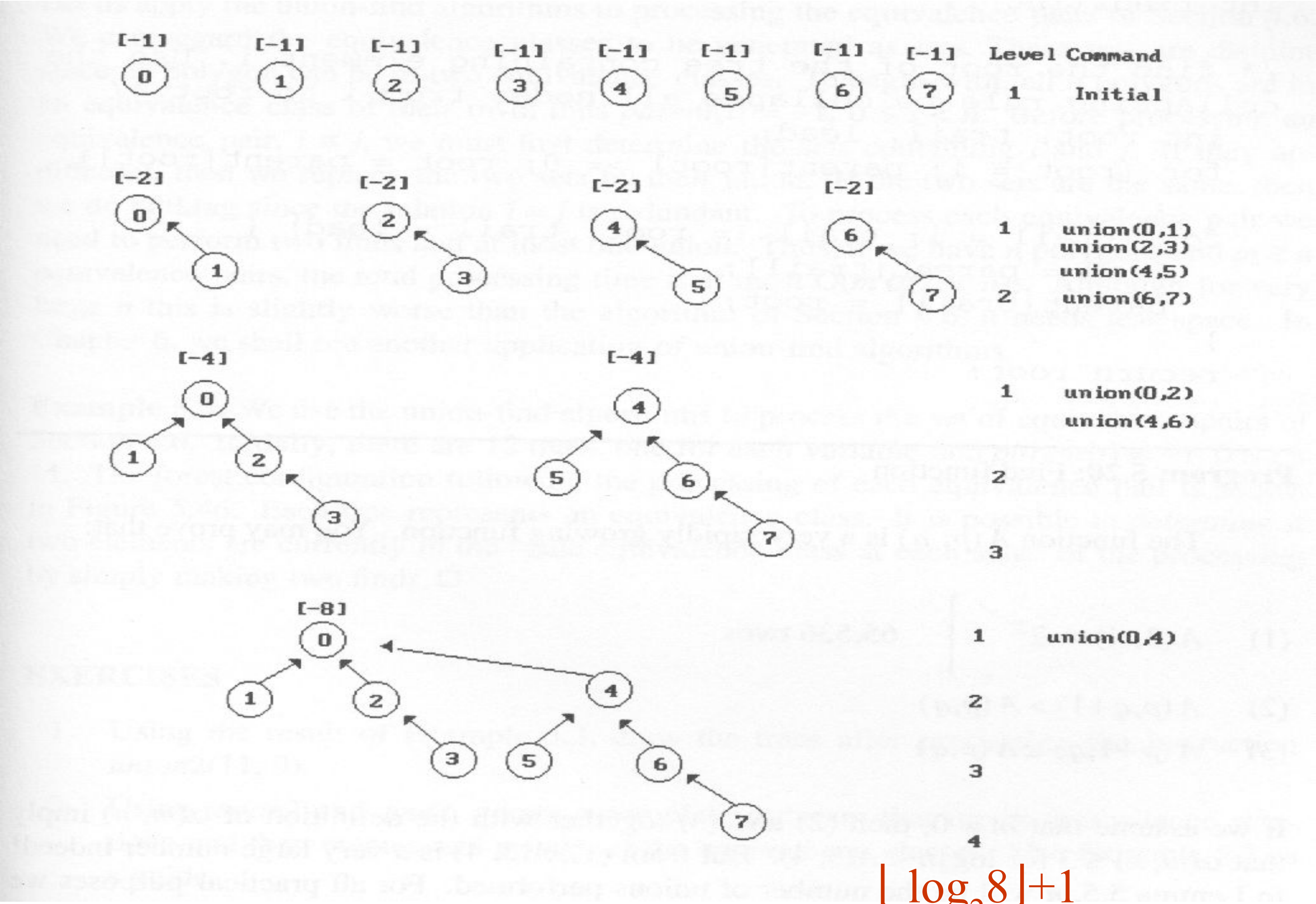
Figure 5.44: Trees obtained using the weighting rule

Modified Union Operation

```
void union2(int i, int j)
{       Keep a count in the root of tree
        int temp = parent[i]+parent[j];
        if (parent[i]>parent[j]) {
            parent[i]=j;      i has fewer nodes.
            parent[j]=temp;
        }
        else { j has fewer nodes
            parent[j]=i;
            parent[i]=temp;
        }
}
```

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

Figure 5.45: Trees achieving worst case bound (p.245)

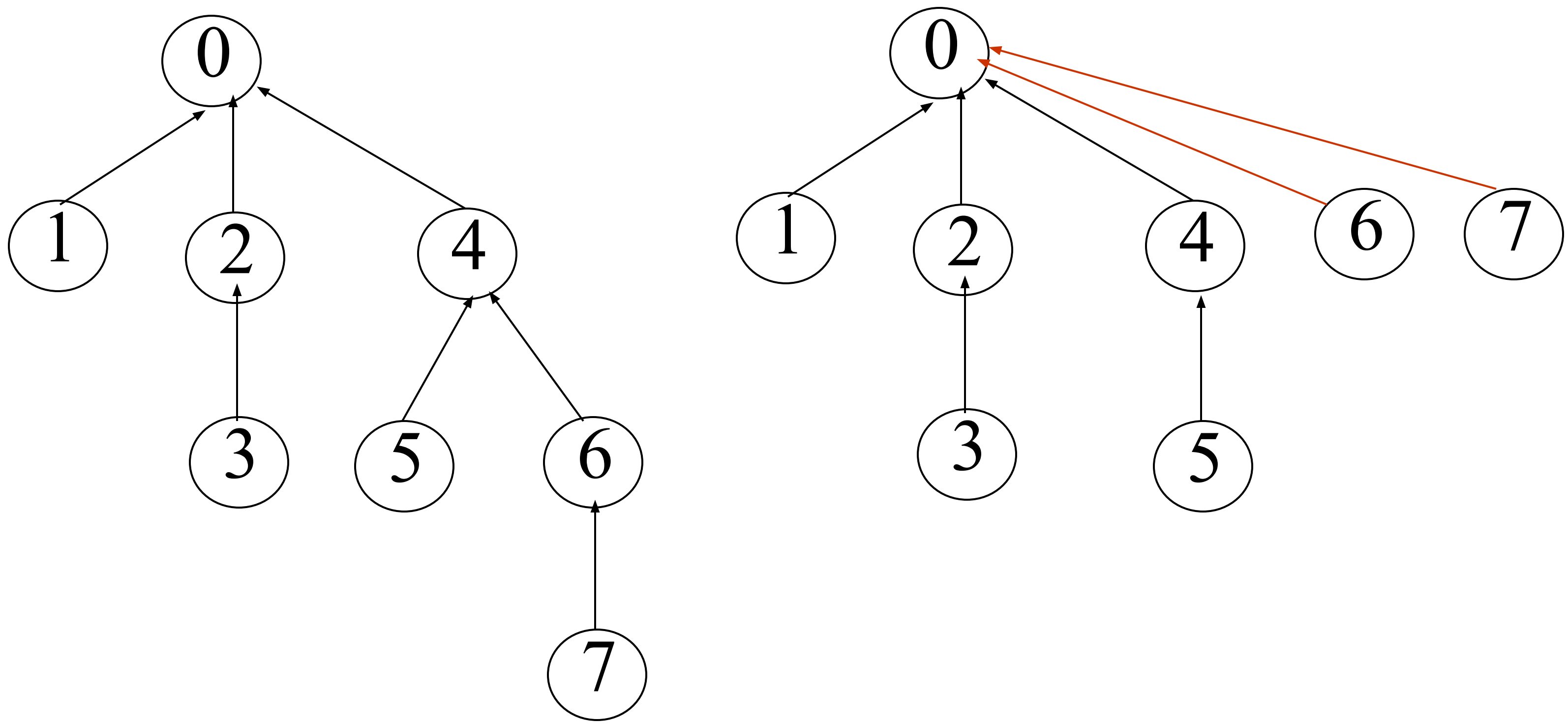


$\lfloor \log_2 8 \rfloor + 1$

Modified *Find*(*i*) Operation

```
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:
}
```

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



	find(7)	find(7)	find(7)	find(7)	find(7)	find(7)	find(7)	find(7)
go up	3	1	1	1	1	1	1	1
reset	2							

12 moves (vs. 24 moves)

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

inorder: B C A E D G H F I