Chapter 5. Trees

5.1 Introduction

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

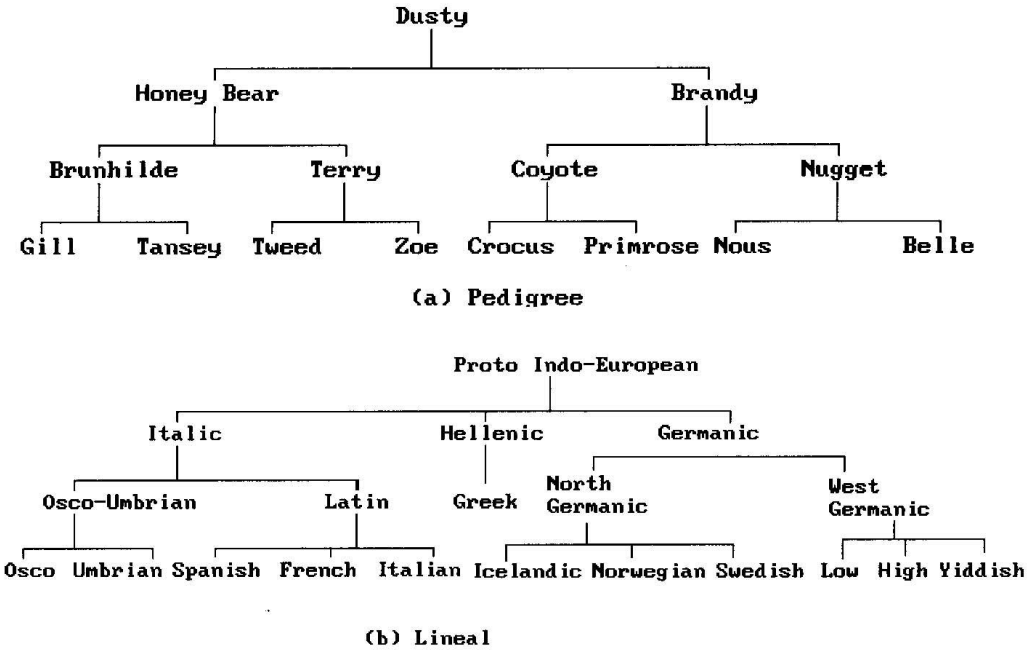


Figure 5.1: Two types of genealogical charts

- **Definition:** 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>=0 disjoint set $T_1,\ldots,T_n$, where each of these sets is a tree. $T_1,\ldots,T_n$ are called the *subtrees* of the root.

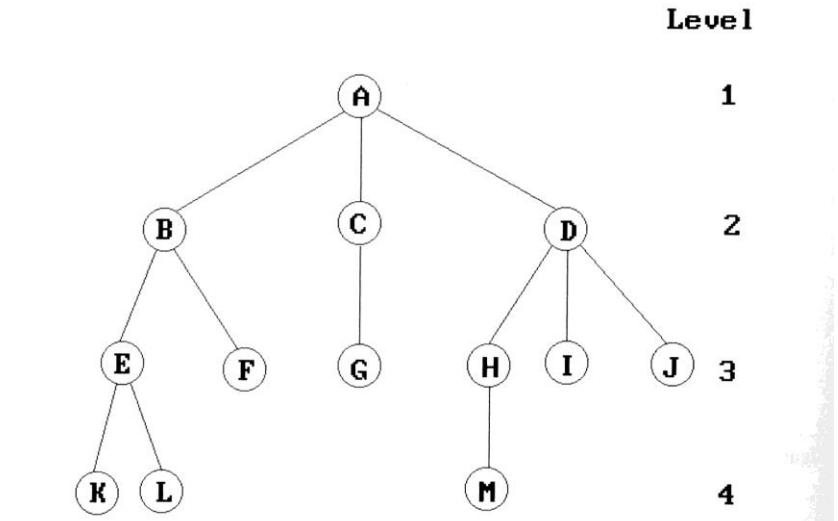


Figure 5.2: A sample tree

- **Some terminology**
  - *node*: the item of information plus the branches to each node.
  - The number of subtrees of a node is called its *degree*.
  - *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.

- The subtrees of a node X are the *children* of X. X is the *parent* of its children.
- *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 is defined by letting the root be at level one. If a node is at level *l*, then it children are at level *l+1*.
- *Height*(or *depth*): the maximum level of any node in the tree.

- **Representation of trees**
  - *List* Representation

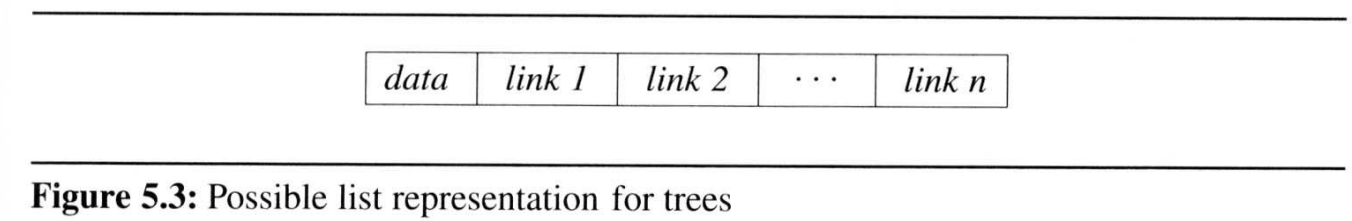


Figure 5.3: Possible list representation for trees

– *Left Child-Right Sibling* Representation

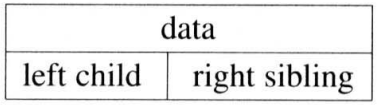


Figure 5.4: Left child-right sibling node structure

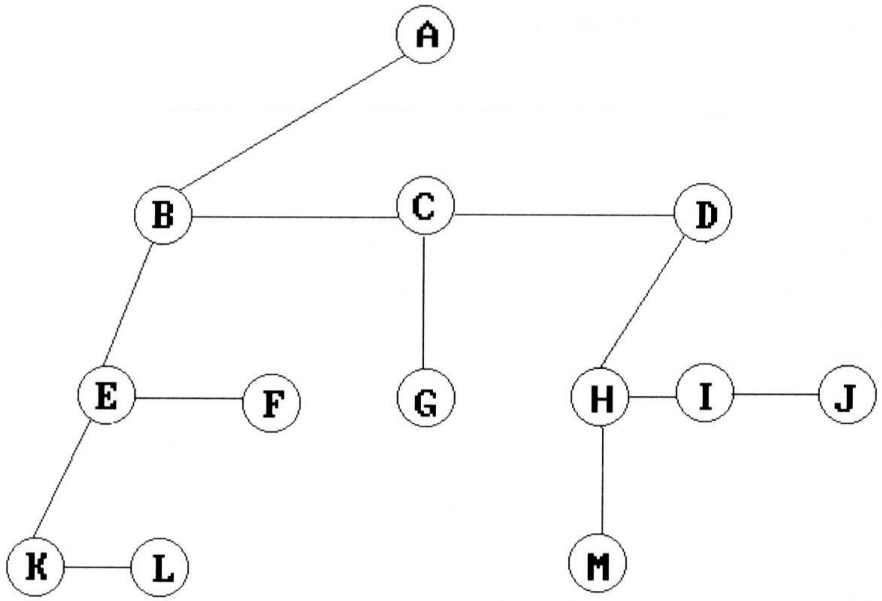


Figure 5.5: Left child-right sibling representation of a tree

Representation as a degree-two tree

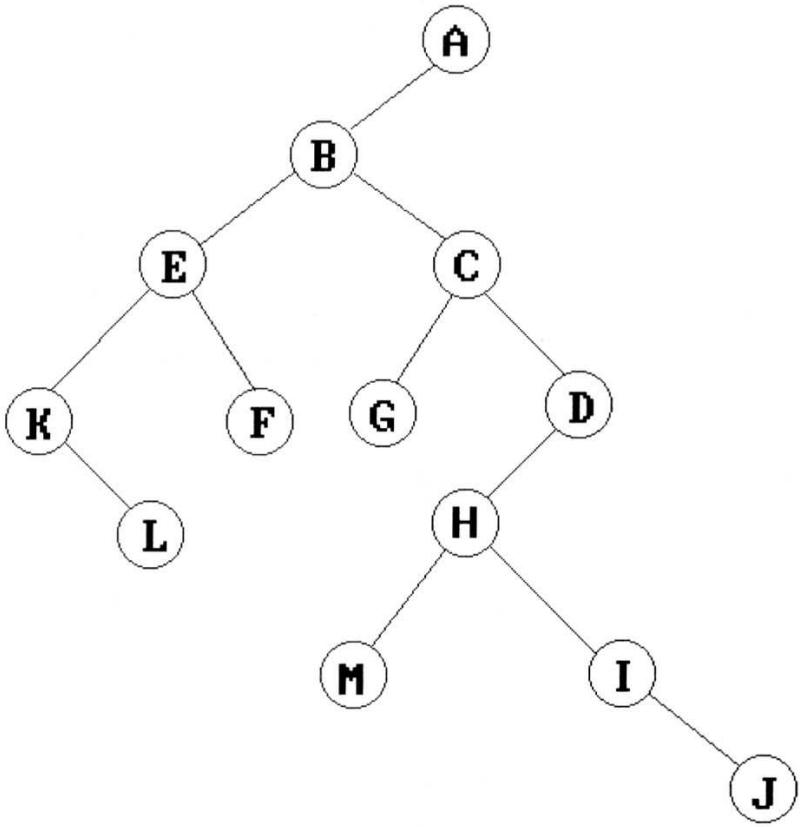


Figure 5.6: Left child-right child tree representation of a tree

5.2 Binary Trees

- Binary trees are characterized by the fact that any node can have at most two branches.
- For binary trees we distinguish between the subtrees on the left and that on the right, whereas for tree the order of subtrees is irrelevant.

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

---

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*,*bt*1,*bt*2 ∈ *BinTree*, *item* ∈ *element*

BinTree Create()	::=	creates an empty binary tree
Boolean IsEmpty(*bt*)	::=	**if** (*bt* == empty binary tree) **return** *TRUE* **else return** *FALSE*
BinTree MakeBT(*bt*1, *item*, *bt*2)	::=	**return** a binary tree whose left subtree is *bt*1, whose right subtree is *bt*2, 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*.

Structure 5.1: Abstract data type *Binary_Tree*

- Properties of binary trees
  - Lemma 5.1 [*Maximum number of nodes*]:

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

    (2) The maximum number of nodes in a binary tree of depth k is 2^k-1, $k \geq 1$.
  - Lemma 5.2 [*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 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Proof

n_0 is the number of nodes with 0 child

n_1 is the number of nodes with 1 child

n_2 is the number of nodes with 2 child

- n = total number of nodes = $n_0 + n_1 + n_2$

以B=branch個數來看 n = B+1 (as each node must have a link to its parent, except the root)

B = $n_1 + 2*n_2$

- Therefore, n = $n_1 + 2*n_2 + 1$

- 由以上兩式，,得 $n_0 = n_2 + 1$

– **Definition:** A *full binary tree* of depth *k* is a binary tree of depth k having 2^k-1 nodes, k ≥0.

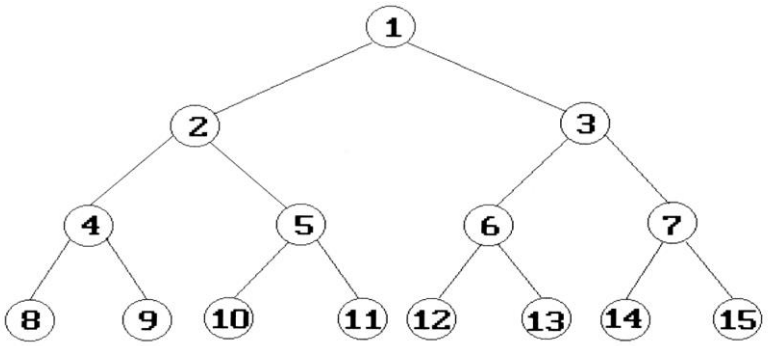


Figure 5.10: Full binary tree of depth 4 with sequential node numbers

– **Definition:** 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*.

– From Lemma 5.1, the height of a complete binary tree with *n* nodes is $\lceil \log_2(n+1) \rceil$.

- Two special kinds of binary trees: a *skewed tree* and a *complete binary tree*

```
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;
      }
   }
}
```

Program 5.9: *post_order_eval* function

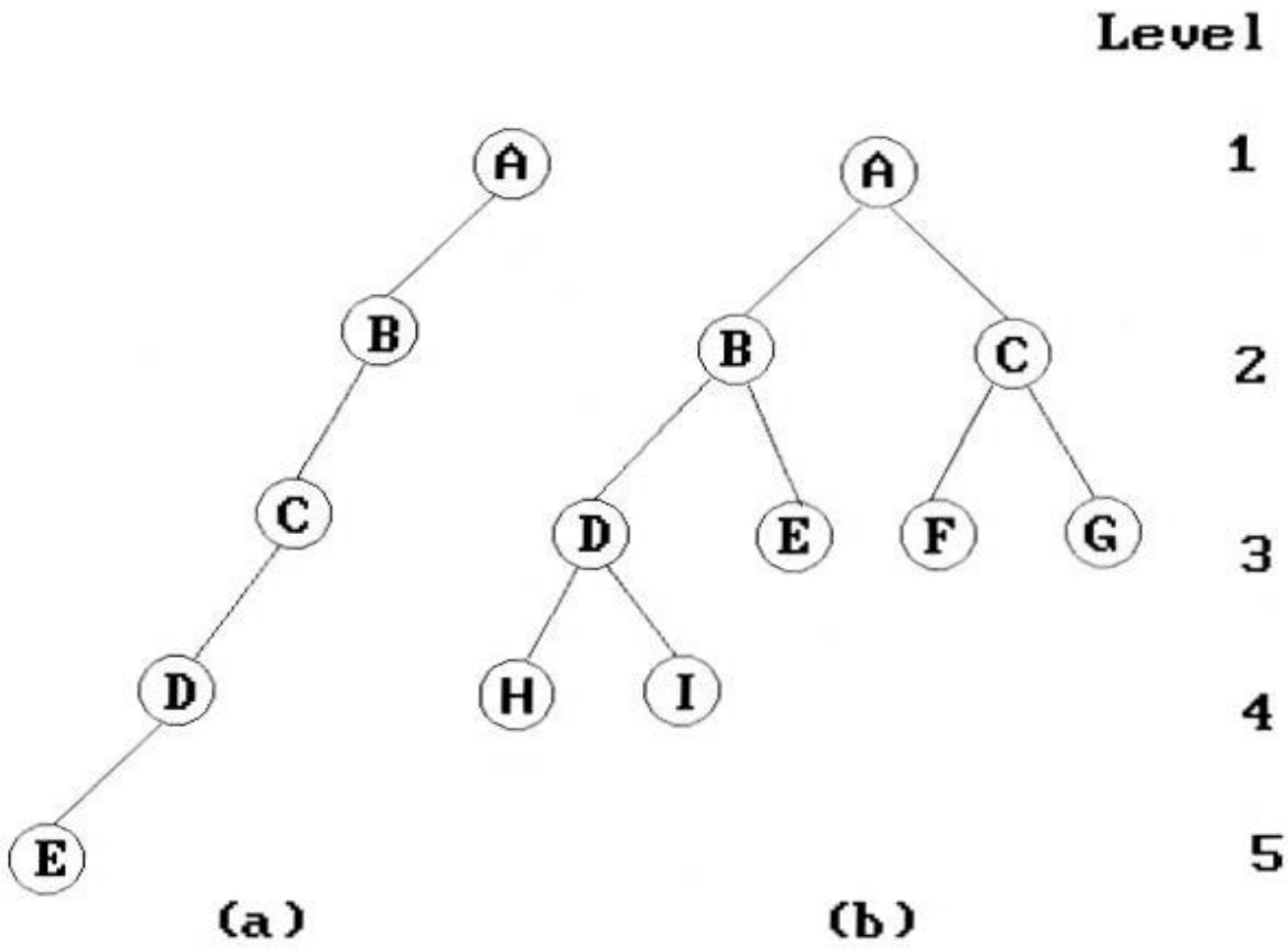


Figure 5.9: Skewed and complete binary trees

- Binary tree representations
  - Array representation
    - Lemma 5.3: If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \le i \le n$, we have

      (1) *parent(i)* is at $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i = 1$, i is at the root and has no parent.

      (2) *LeftChild(i)* is at $2i$ if $2i \le n$. If $2i > n$, then i has no left child.

      (3) *RightChild(i)* is at $2i+1$ if $2i+1 \le n$. If $2i+1 > n$, then i has no left child.

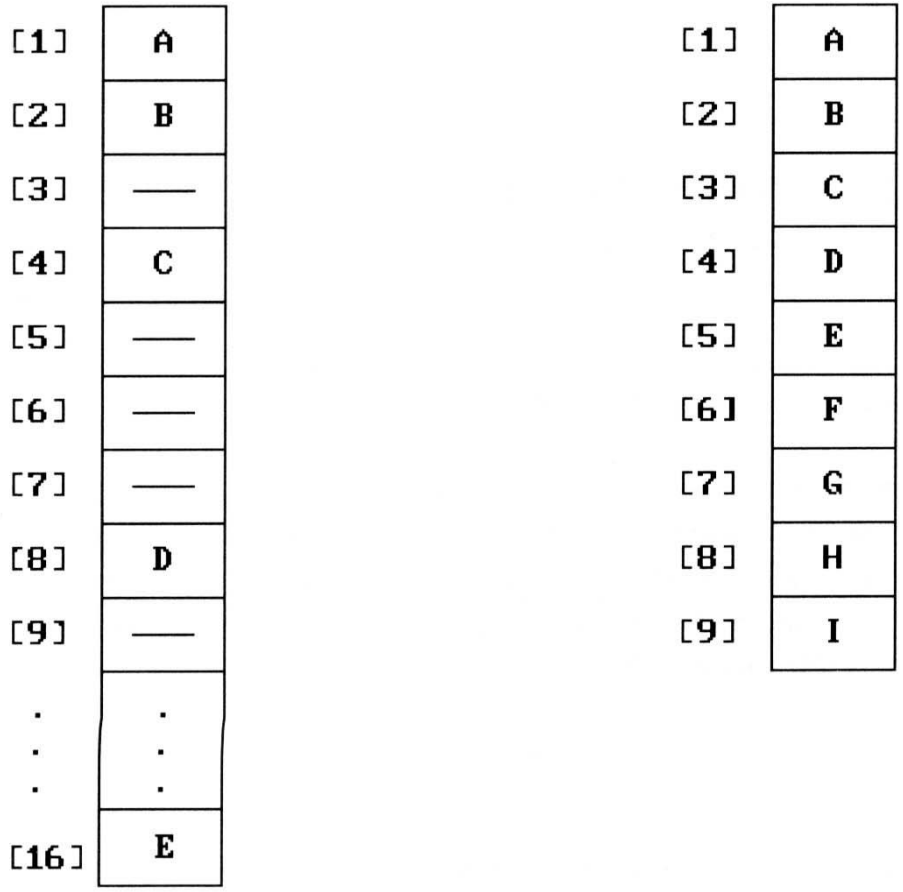


Figure 5.11: Array representation of binary trees of Figure 5.9

– Linked Representation

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

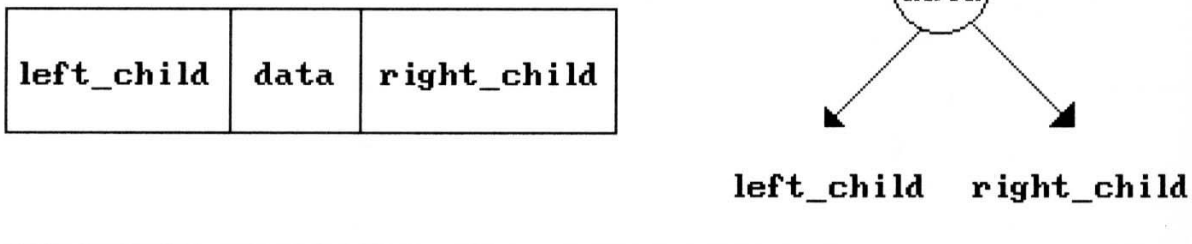


Figure 5.12: Node representation for binary trees

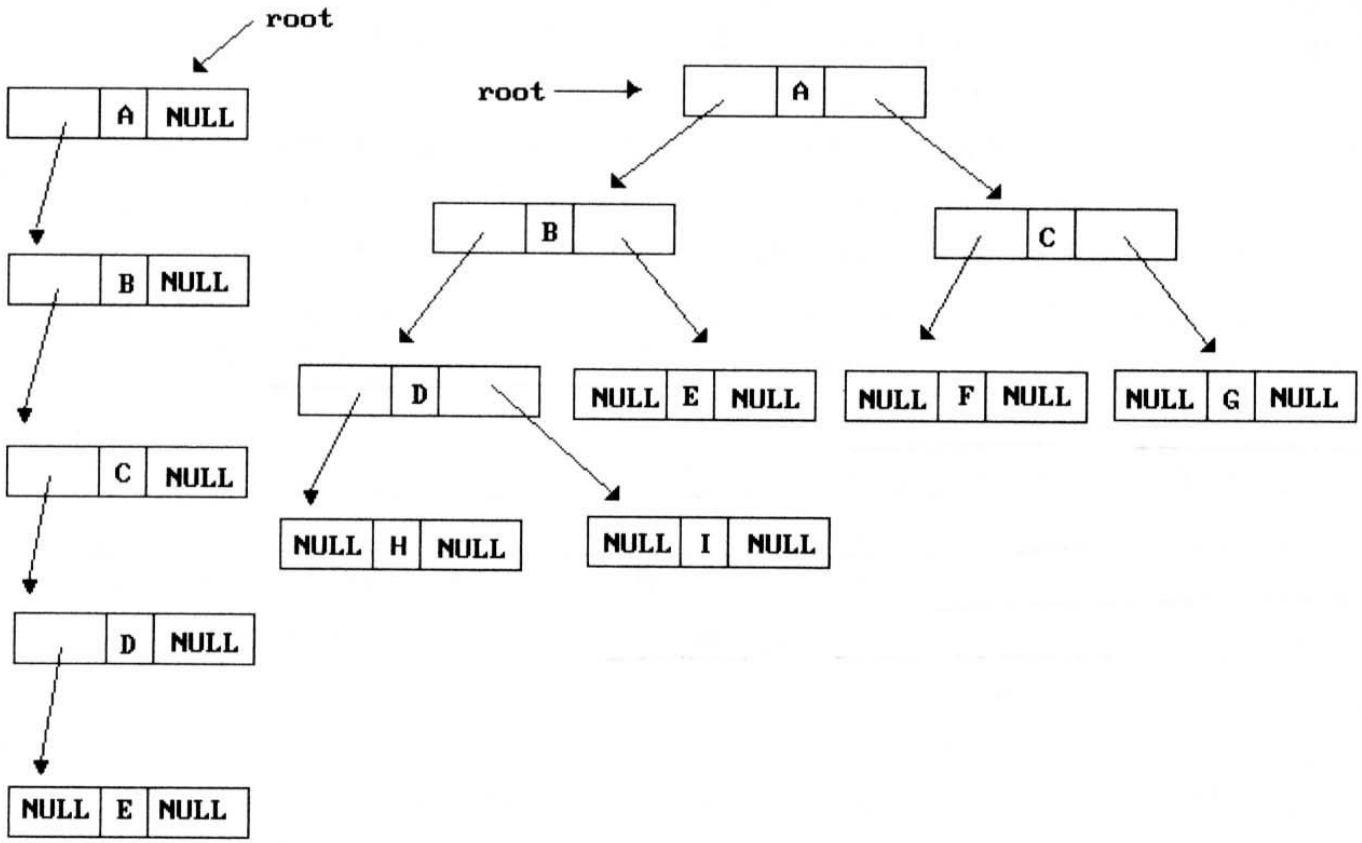


Figure 5.13: Linked representation for the binary trees of Figure 5.9

5.3 Binary Tree Traversal

- How to traverse a tree or visit each node in the tree exactly once?
  - If we let *L*, *V*, and *R* stand for moving left visiting the node, and moving right when at a node, then there are six possible combinations of traversal: *LVR*, *LRV*, *VLR*, *VRL*, *RVL*, and *RLV*.
  - If we adopt the convention that we traverse left before right, then only three traversals remain: *LVR*, *LRV*, and *VLR*. We assign these *inorder*, *postorder*, and *preorder*, respectively.

– We will use the following tree to illustrate each of the traversals.

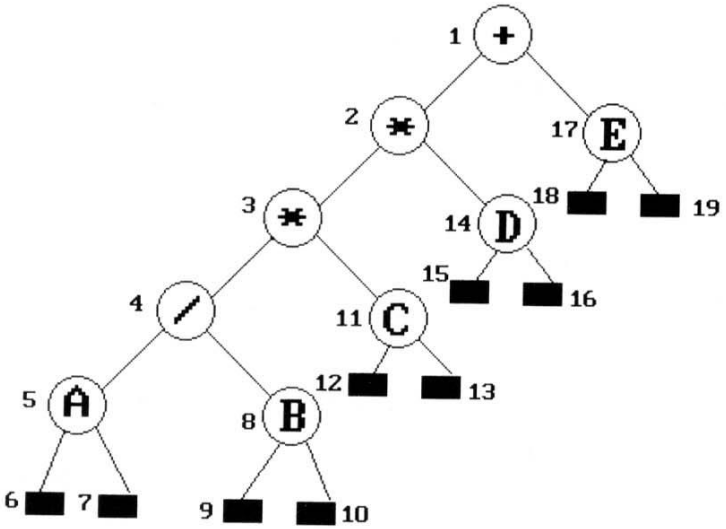


Figure 5.15: Binary tree with arithmetic expression

- Inorder traversal (*LVR*)

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

Program 5.1: Inorder traversal of a binary tree

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

Figure 5.16: Trace of Program 5.1

– The inorder traversal result of Figure 5.16

  A/B*C*D+E

- Preorder traversal (*VLR*)

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

Program 5.2: Preorder traversal of a binary tree

  - The preorder traversal result of Figure 5.16 +**/ABCDE

- Postorder traversal (*LRV*)

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

Program 5.3: Postorder traversal of a binary tree

  - The postorder traversal result of Figure 5.16

    AB/C*D*E+

- 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;
   }
}
```

Program 5.4: Iterative inorder traversal

– Analysis of inorder2 (Iter-Inorder)
  - Let n be the number of nodes in the tree.
  - Every node of the tree is placed on and removed from the stack exactly once. So, the time complexity is O(n).
  - The space requirement is equal to the depth of the tree which is O(n).

- 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.
  - The level-order traversal result of Figure 5.16

    +*E*D/CAB

```
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;
   }
}
```

Program 5.5: Level order traversal of a binary tree

5.4 Additional Binary Tree Operations

- Copying Binary Trees
  - For example, we can 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;
}
```

Program 5.6: Copying a binary tree

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

Program 5.7: Testing for equality of binary trees

- The satisfiability problem
  - Consider the set of formulas we can construct by taking variables x_1, x_2, x_3, …, and the operators $\wedge$(**and**), $\vee$ (**or**), $\neg$ (**not**). These variables can hold only one of two possible values, *true* or *false*.
  - Propositional calculus

    (1) a variable is an expression

    (2) if x and y are expressions then $x \wedge y$, $x \vee y$, $\neg x$ are expressions

    (3) parentheses can be used to alter the normal order of evaluation, which is **not** before **and** before **or**.

– Example: $(x_1 \wedge \neg x_2) \vee (\neg\, x_1 \wedge x_3) \vee \neg\, x_3$

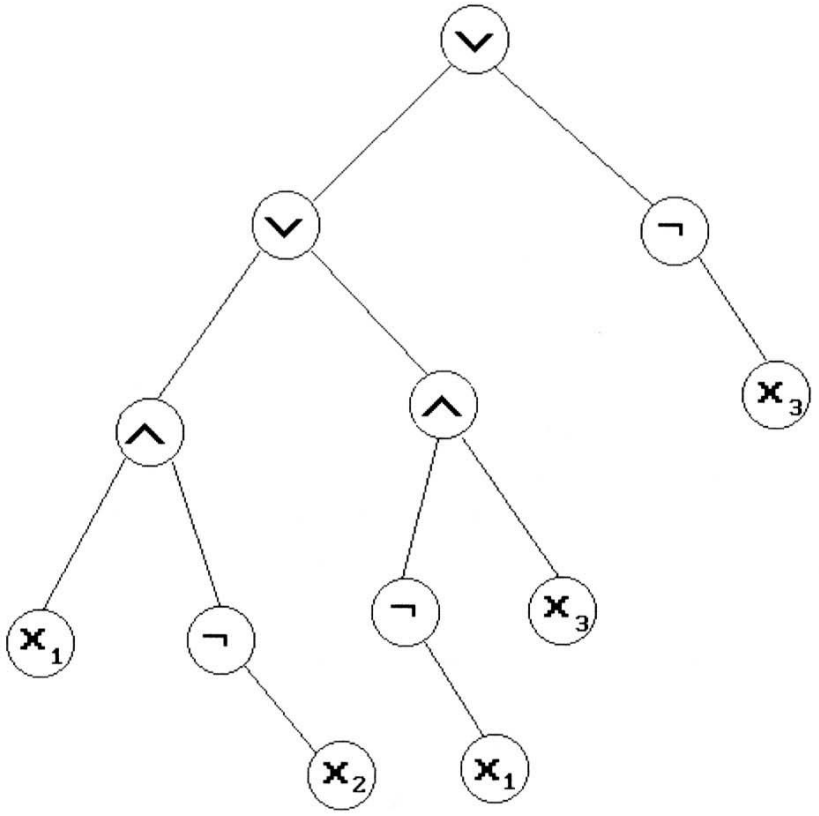


Figure 5.18: Propositional formula in a binary tree

– For the purpose of our evaluation algorithm, we assume each node has four fields:

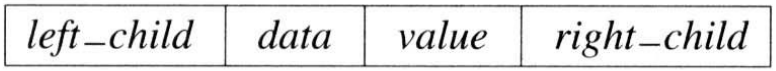


Figure 5.19: Node structure for the satisfiability problem

– The node structure may be defined in C as

```
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;
        } ;
```

– A 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");
```

Program 5.8: First version of satisfiability algorithm

– The C function that evaluates 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;
      }
   }
}
```

Program 5.9: *post_order_eval* function

5.5 Threaded Binary Trees

- Threads
  - Do you find any drawback of the above tree?
  - A. J. Perlis and C. Thornton criticize that too many null pointers are in the tree. There will be n+1 null links out of 2n total links. (Only n-1 links are truly used, wasting too much space.) How to make use of these null links?
  - They replace the null links by pointers, called threads, to other nodes in the tree.

– Rules for constructing the threads

(1) A null *RightChild* field in node *p* is replaced by a pointer to the node that would be visited after *p* when traversing the tree in inorder. That is , it is replaced by the inorder successor of *p*.

(2) A null *LeftChild* link at node *p* is replaced by a pointer to the node that immediately precedes node *p* in inorder (i.e., it is replaced by the inorder predecessor of *p*)

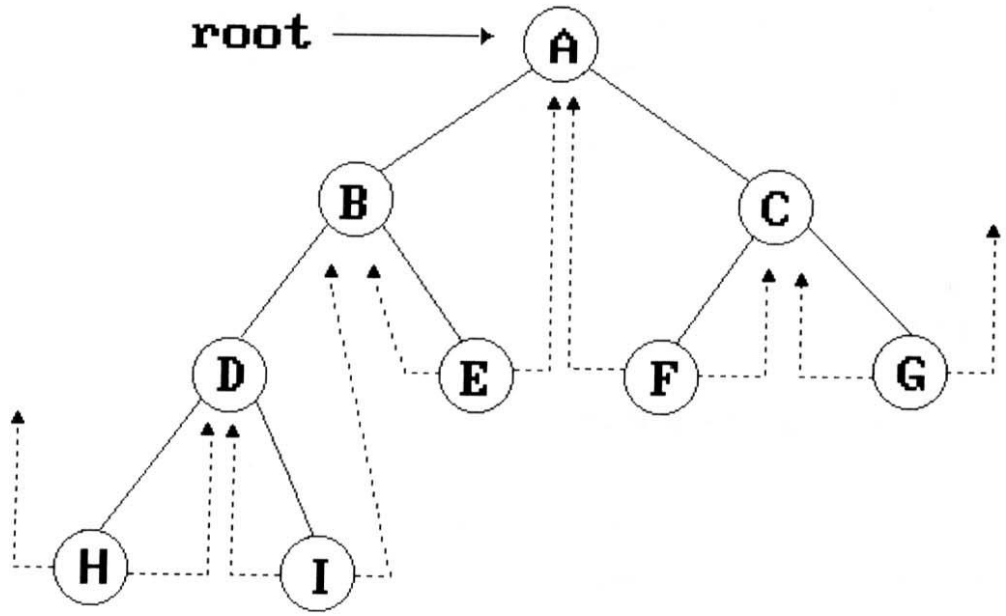


Figure 5.21: Threaded tree corresponding to Figure 5.9(b)

- In order to distinguish between threads and normal pointers, we add two additional fields to the node structure (on the next page), 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;
        };
```

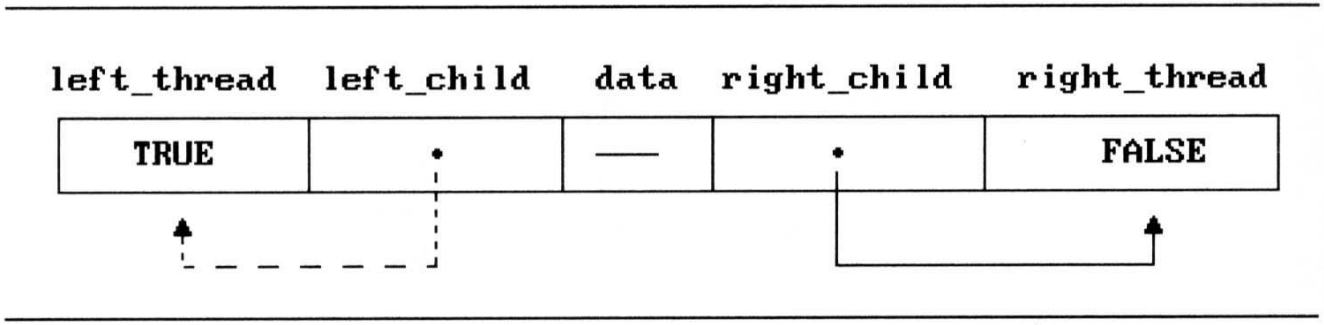


Figure 5.22: An empty threaded tree

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.

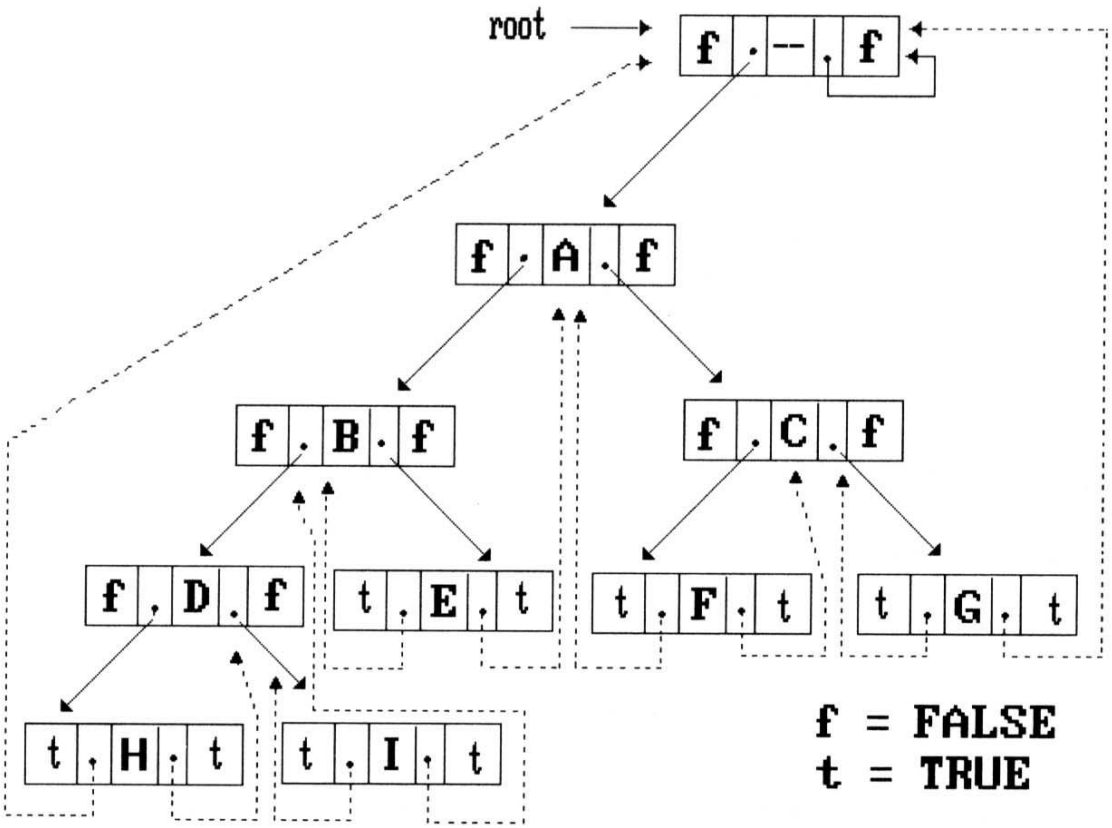


Figure 5.23: Memory representation of a threaded tree

- Inorder traversal of a threaded binary tree
  - we can perform an inorder traversal without making use of a stack (simplifying the task)
    - The reason is obvious: without having the threads, we will not know where to go when we reach H (in the previous slide). Hence, a stack has to be used. Now, we can follow the thread to the “next” node of inorder traversal.

- Inserting a node into a threaded binary tree
  - The operation is quite trivial. The next figure shows the results of inserting D (as a child of B), E and F (as children of D), and X (in between B and D), respectively.

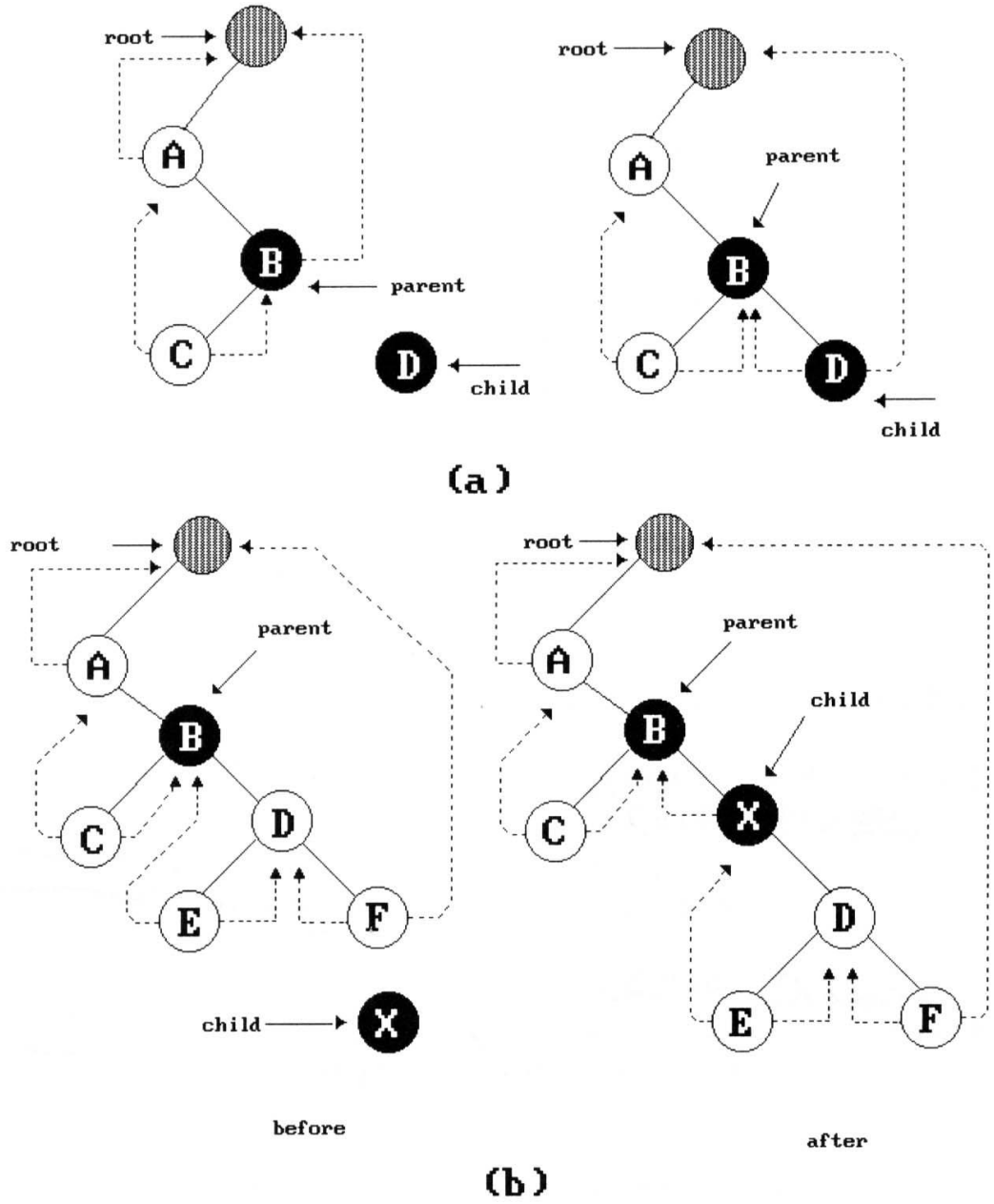


Figure 5.24: Insertion of child as a right child of parent in a threaded binary tree

5.6 Heaps

- The heap abstract data type
  - **Definition:** A *max tree* is a tree in which the key value in each node is no smaller than the key values in its children (if any). A *max heap* is a complete binary tree that is also a max tree.
  - **Definition:** A *min tree* is a tree in which the key value in each node is no larger than the key values in its children (if any). A *min heap* is a complete binary tree that is also a min tree.

– The examples of max heaps and min heaps

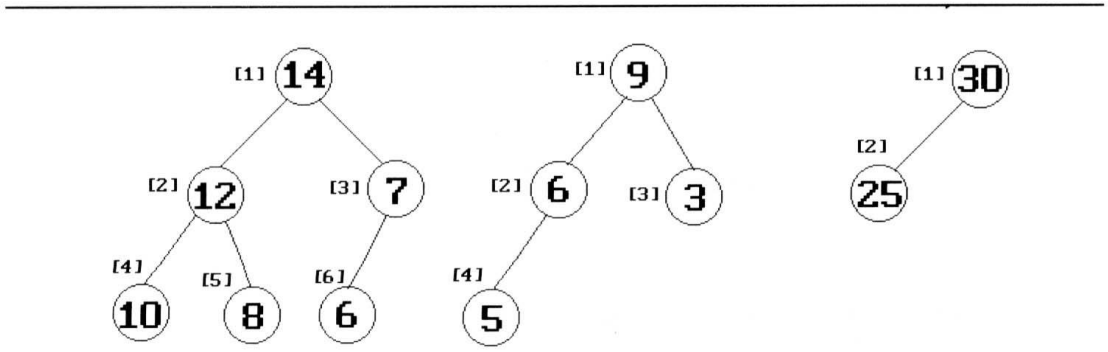


Figure 5.25: Sample max heaps

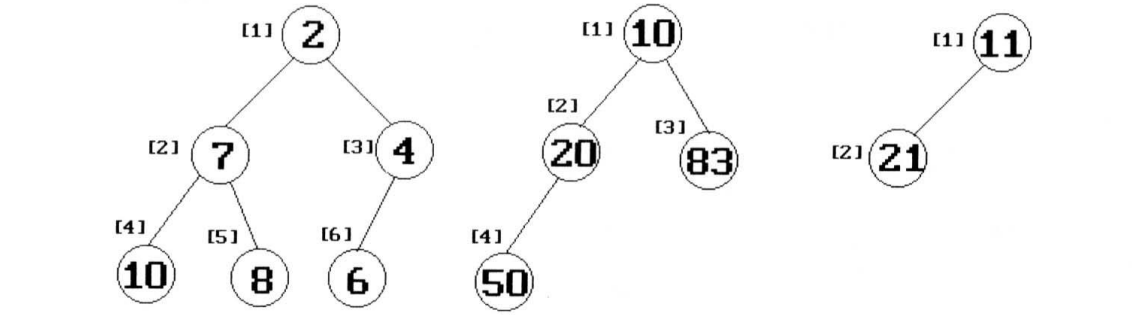


Figure 5.26: Sample min heaps

– The basic operations of heaps include
(1) creation (2) insertion (3) deletion

---

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* ∈ *MaxHeap*, *item* ∈ *Element*, *n*, *max_size* ∈ 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** *TRUE* **else return** *FALSE*
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.

Structure 5.2: Abstract data type *MaxHeap*

- Priority queues
  - Heaps are frequently used to implement *priority queues.*
  - The element to be deleted is the one with highest (or lowest) priority.
  - An element with arbitrary priority can be inserted into the 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	$O(\log_2 n)$	$O(\log_2 n)$

Figure 5.27: Priority queue representations

- Insertion into a max heap
  - After insertion, the heap is still a complete binary tree.

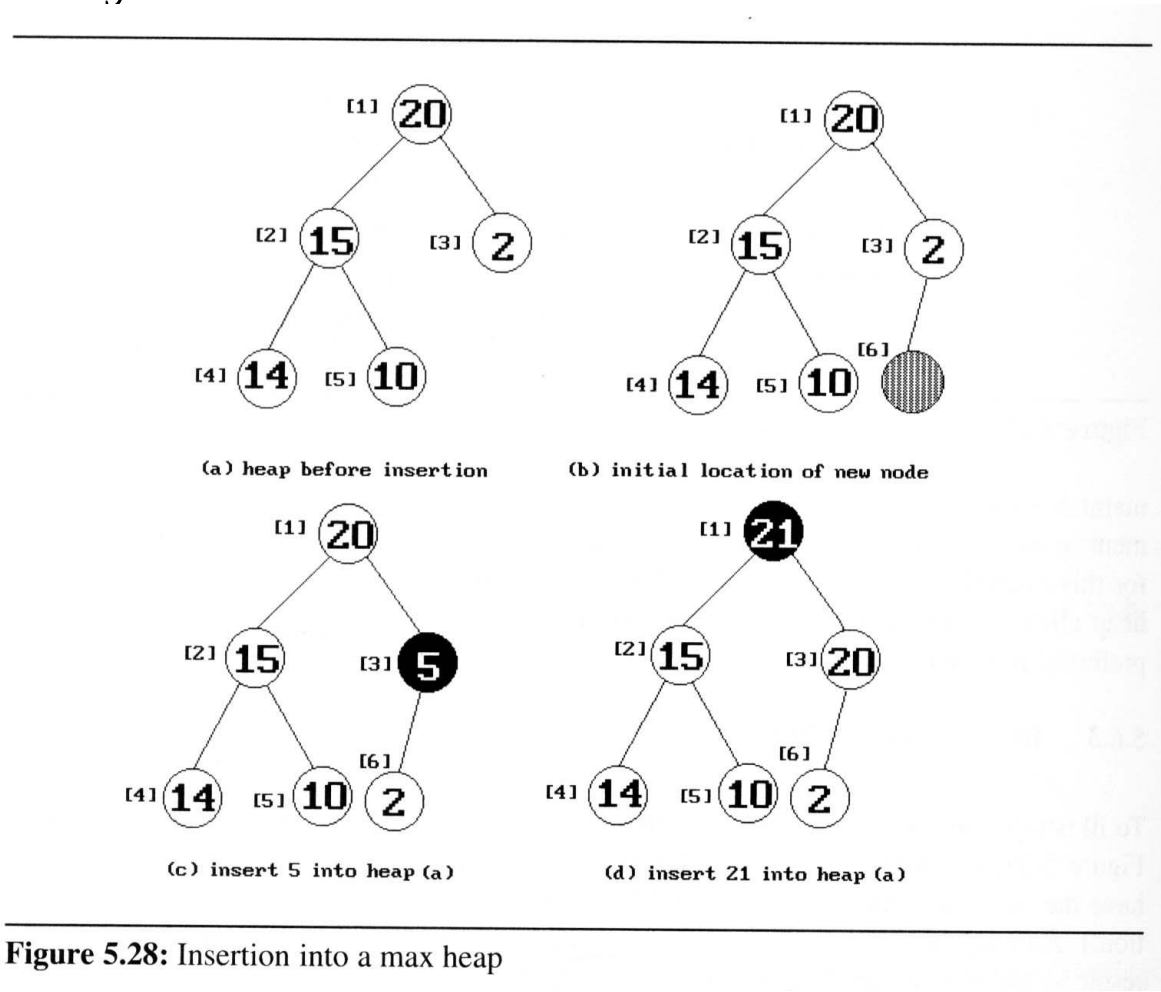


Figure 5.28: Insertion into a max heap

```
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;
}
```

Program 5.13: Insertion into a max heap

– Analysis of *insert_max_heap*

The complexity of the insertion function is $O(\log_2 n)$

- Deletion from a max heap
  - After insertion, the heap is still a complete binary tree.

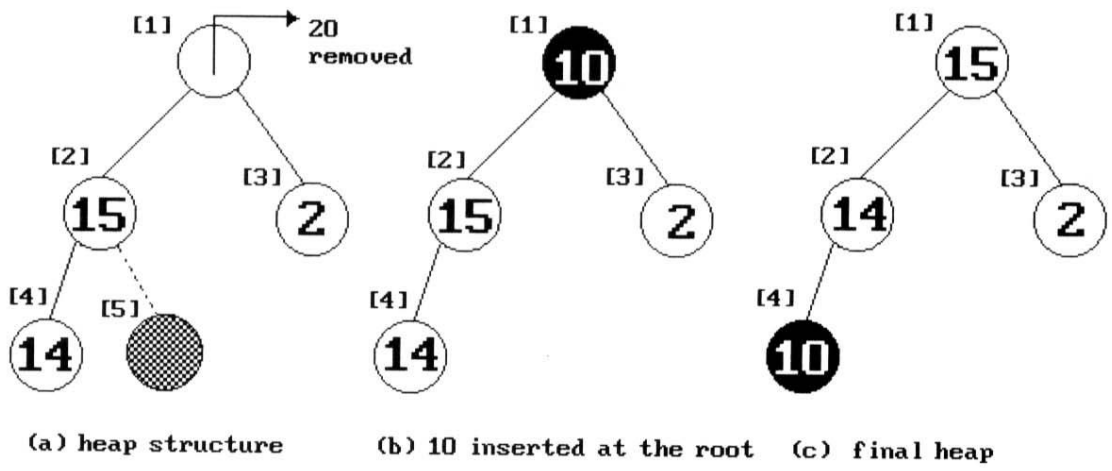


Figure 5.29: Deletion from a max heap

```
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;
}
```

Program 5.14: Deletion from a max heap

– Analysis of *delete_max_heap*

The complexity of the insertion function is $O(\log_2 n)$

5.7 Binary search trees

- Why do binary search trees need?
  - Heap is not suited for applications in which arbitrary elements are to be deleted from the element list. (complexity: O(*n*))
- **Definition:**

  A *binary search tree* is a binary tree. I may be empty. If it is not empty, it satisfies the following properties:

  (1) Every element has a key, and no two elements have the same key, that is, the key are unique.

(2) The keys in a nonempty left subtree must be smaller than the key in the root of the subtree.

(3) The keys in a nonempty right subtree must be larger than the key in the root of the subtree.

(4) The left and right subtree are also binary search tree.

– Example: (b) and (c) are binary search trees.

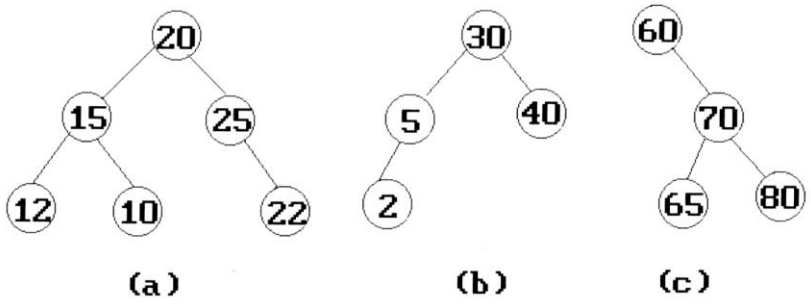


Figure 5.30: Binary trees

- Searching a binary search tree

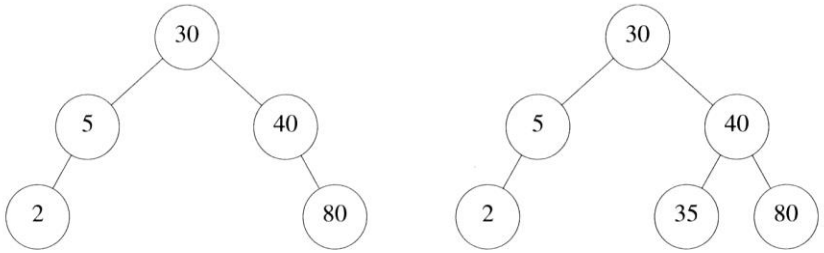
```
tree_pointer search(tree_pointer root, int key)
{
/* return a pointer to the node that contains key.  If
there is no such node, return NULL. */
   if (!root) return NULL;
   if (key == root->data) return root;
   if (key < root->data)
      return search(root->left_child, key);
   return search(root->right_child,key);
}
```

Program 5.15: Recursive search of a binary search tree

```
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;
}
```

Program 5.16: Iterative search of a binary search tree

- Inserting into a binary search tree



(a) Insert 80 (b) Insert 35

Figure 5.31: 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)) {
      /* 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;
   }
}
```

Program 5.17: Inserting an element into a binary search tree

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

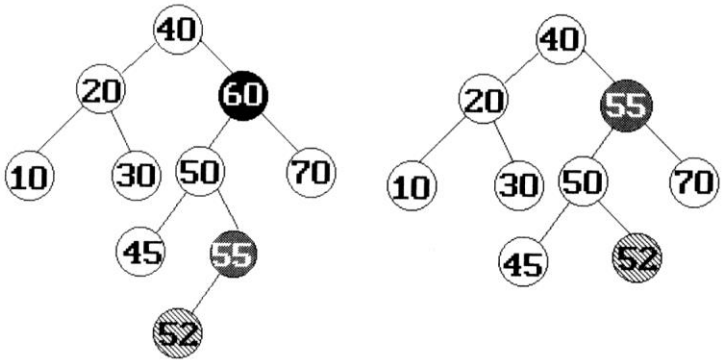


Figure 5.33: Deletion of a node with two children

- 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*.

5.8 Selection Trees

- Suppose we have k order sequences, called runs, that are to be merged into a single ordered sequence.
  - Solution: *Selection tree*.
  - There are two kinds of selection trees: *winner trees* and *loser trees*

– Winner trees

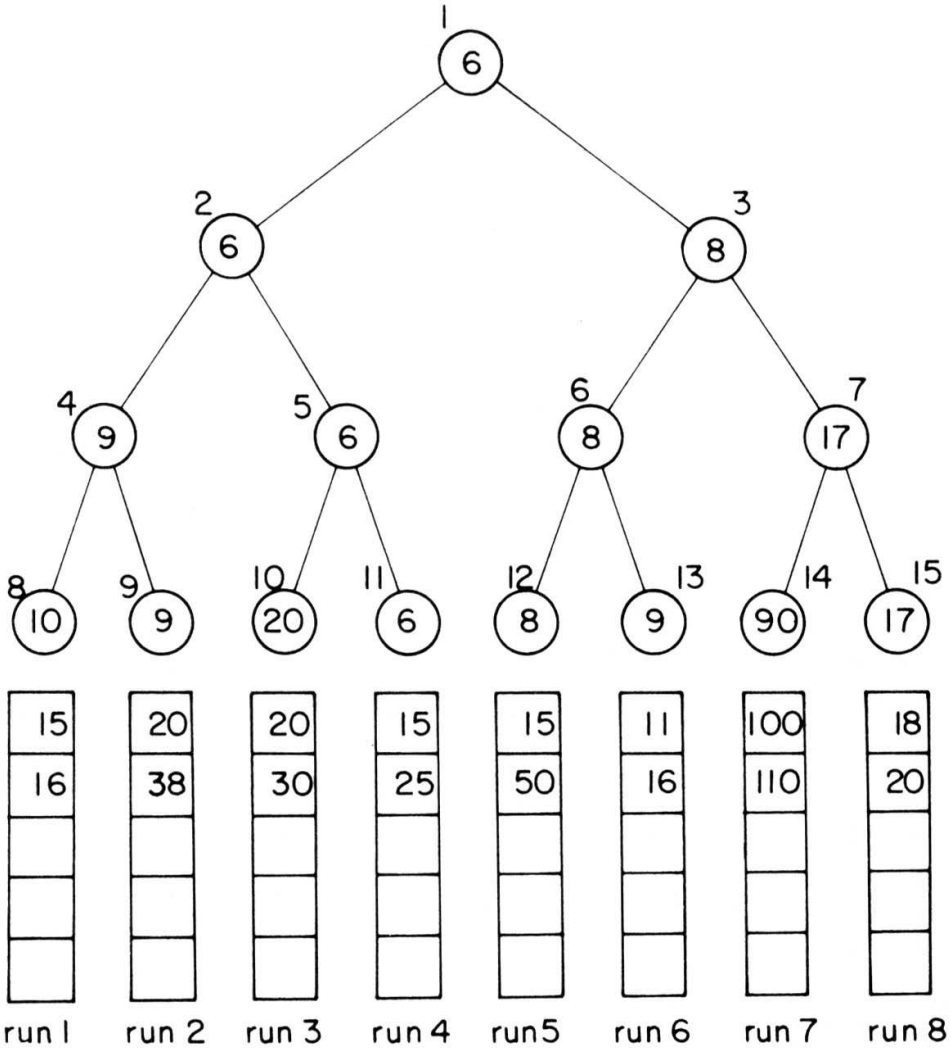


Figure 5.34: Selection tree for $k = 8$ showing the first three keys in each of the eight runs

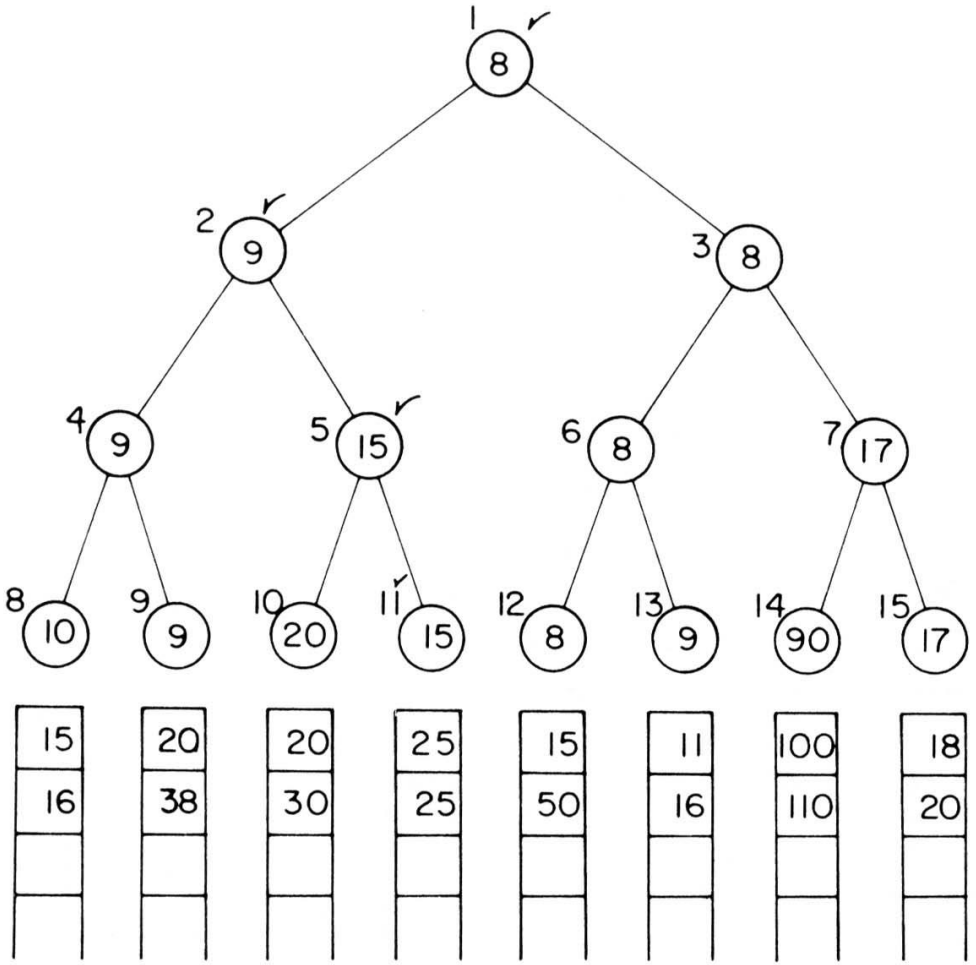


Figure 5.35: Selection tree of Figure 5.34 after one record has been output and the tree restructured (nodes that were changed are ticked)

- Analysis of merging runs using winner trees

  1. The number of levels in the tree is $\log_2 (k)+1$, so the time to restructure the tree is $O(\log_2 k)$.
  2. The time required to merge all n records is $O(n \log_2 k)$.
  3. The time required to set up the selection tree the first time is $O(k)$.
  4. The total time needed to merge the k runs is $O(n \log_2 k)$.

5.9 Forests

- Definition:

  A *forest* is a set of $n \geq 0$ disjoint trees.

- Transforming a forest into a binary tree
  - Definition: If $T_1,\ldots,T_n$ is a forest of trees, then the binary tree corresponding to this forest, denoted by $B(T_1,\ldots,T_n)$:

    (1) is empty, if $n = 0$.

    (2) has root equal to root (T_1); has left subtree equal to $B(T_{11}, T_{12}, \ldots, T_{1m})$, where $T_{11}, T_{12}, \ldots, T_{1m}$ are the subtrees of root (T_1); and has right subtree.

– Example:

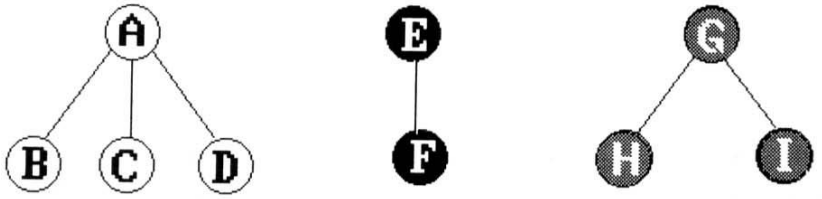


Figure 5.37: Forest with three trees

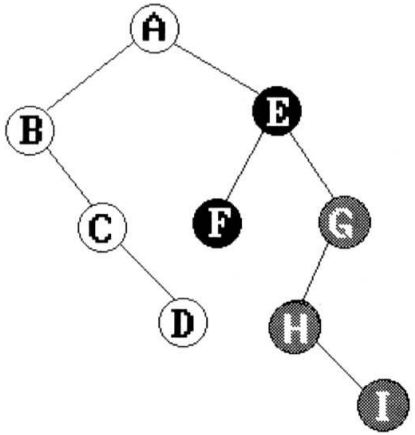


Figure 5.38: Binary tree representation of Figure 5.37

- Forest traversals
  - Forest preorder traversal

    (1) If *F* is empty, then return.

    (2) Visit the root of the first tree of *F*.

    (3) Traverse the subtrees of the first tree in tree preorder.

    (4) Traverse the remaining tree of *F* in preorder.
  - Forest inorder traversal

    (1) If *F* is empty, then return.

    (2) Traverse the subtrees of the first tree in tree inorder.

    (3) Visit the root of the first tree of *F*.

    (4) Traverse the remaining tree of *F* in inorder.

– Forest postorder traversal

(1) If *F* is empty, then return.

(2) Traverse the subtrees of the first tree in tree postorder.

(4) Traverse the remaining tree of *F* in postorder.

(3) Visit the root of the first tree of *F*.

5.10 Set Representation

- Example:
  - When n = 10, the elements may be partitioned into three disjoint set, $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$, $S_3 = \{2, 3, 5\}$. Figure 5.39 shows one possible representation for these sets.

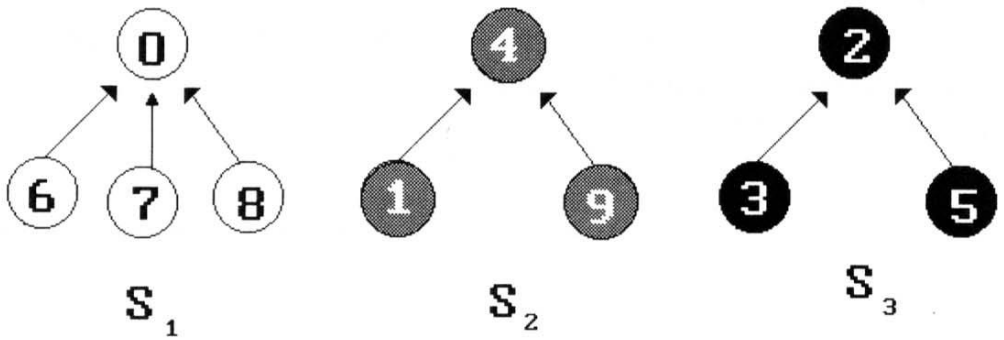


Figure 5.39: Possible forest representation of sets

- The operations we wish to perform on these sets are

  (1) disjoint set union

  (2) Find(*i*). Find the set containing element *i*.

- Union and find operations
  - To obtain the union of two sets, all that has to do is to set the parent field of one of the roots to the other root.

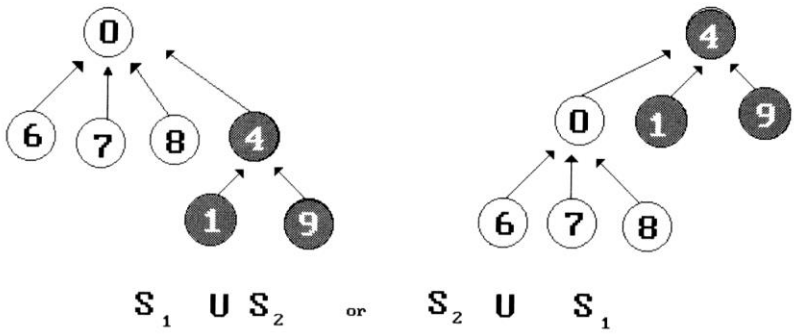


Figure 5.40: Possible representation of $S_1 \cup S_2$

– Data representation for S_1, S_2, and S_3 may then take the form shown in Figure 5.41.

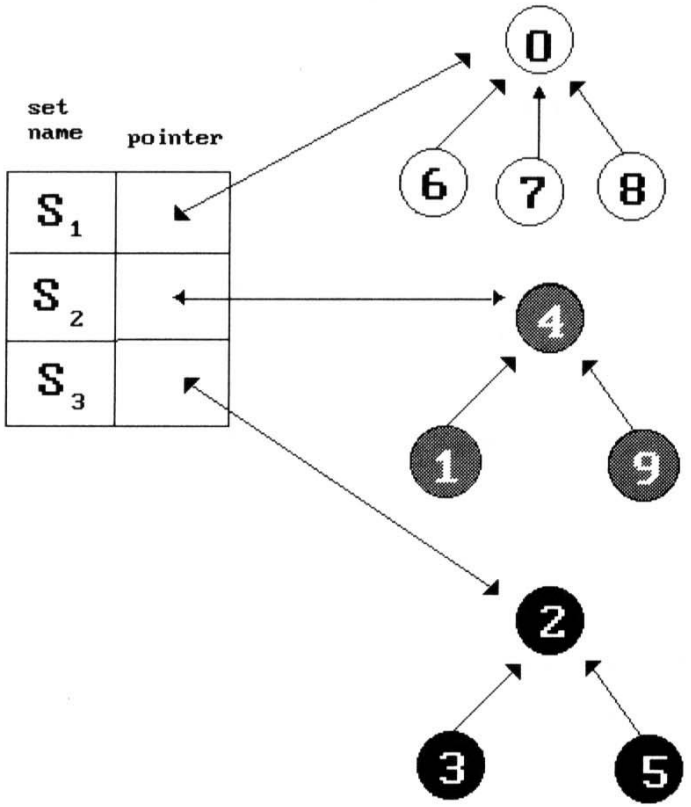


Figure 5.41: Data representation of S_1, S_2, and S_3

– Another representation for S_1, S_2, and S_3 . (array representation)

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

Figure 5.42: Array representation of S_1, S_2, and S_3

– Complexity
  - union1: O(n)
  - find1: O(n^2)
– Worst case:
  - *union*(0, 1),*find*(0), *union*(1, 2),*find*(0), …, *union*(n-2, n-1), *find*(0)

Figure 5.43: Degenerate tree

– To avoid the creation of degenerate trees, we use *weighting rule* for *union(i, j)* to improve the efficiency.

– **Definition:** *Weighting rule for union(i, j).*

  • 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*.

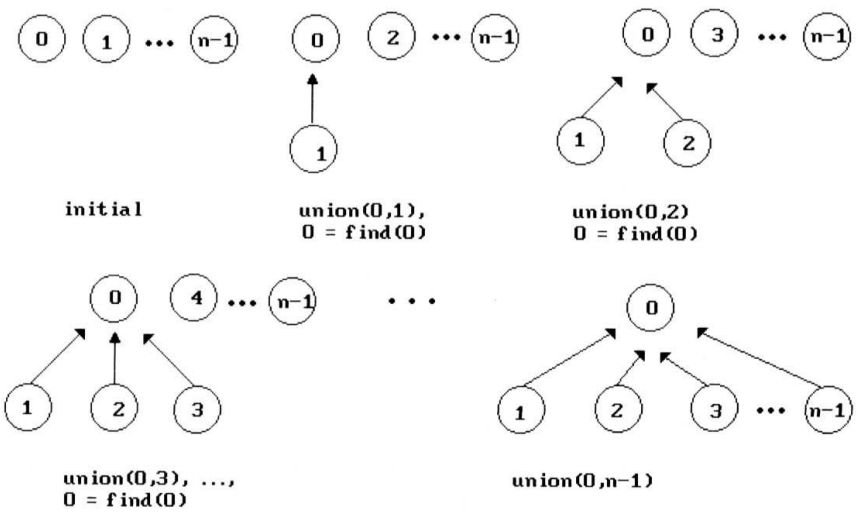


Figure 5.44: Trees obtained using the weighting rule

– Algorithm(*weighting rule for union(i, j)*)

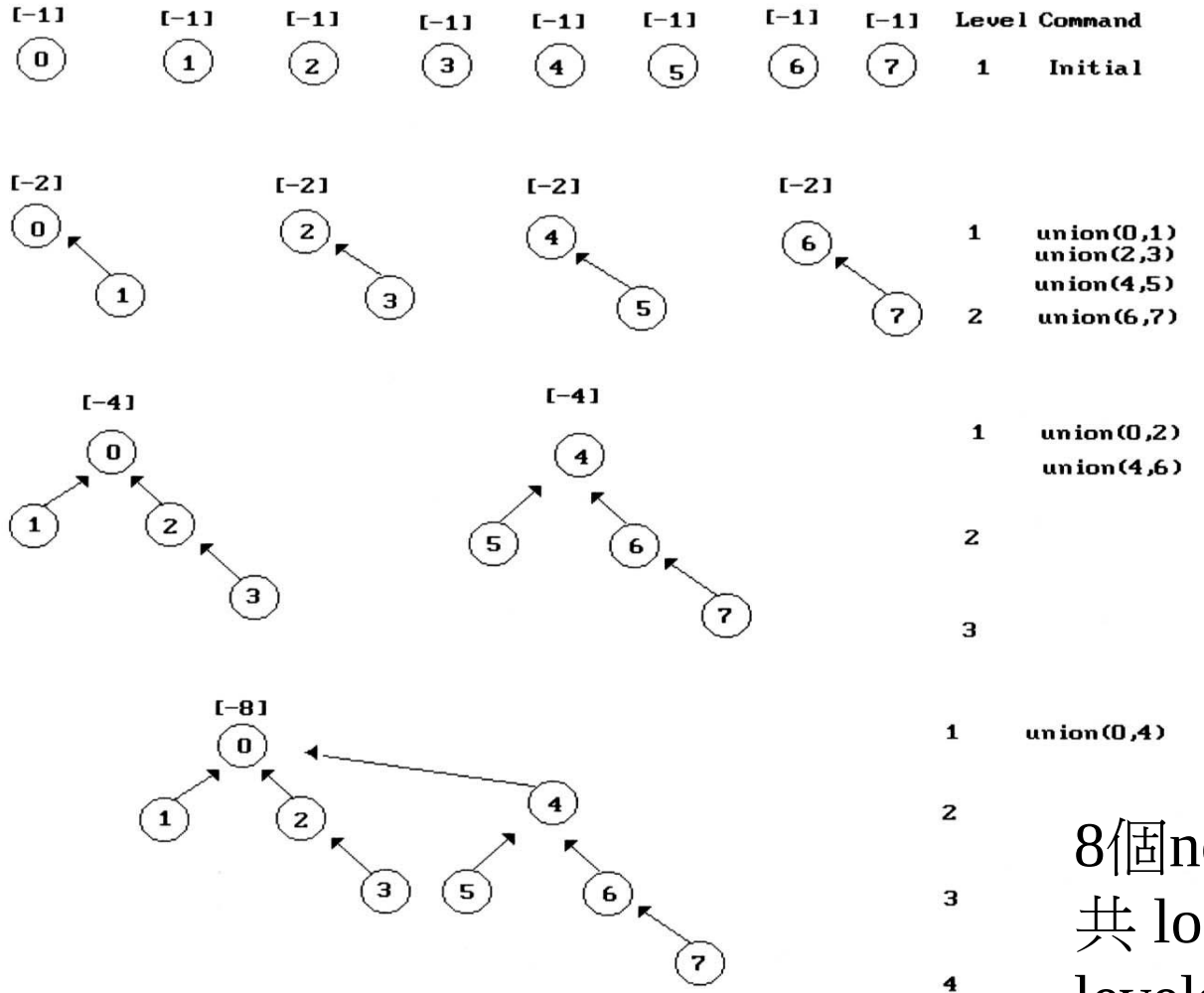
```
void union2(int i, int j)
{
/* union the sets with roots i and j, i != j, using
the weighting rule. parent[i] = -count[i] and
parent[j] = -count[j] */
   int temp = parent[i] + parent[j];
   if (parent[i] > parent[j]) {
      parent[i] = j; /* make j the new root */
      parent[j] = temp;
   }
   else {
      parent[j] = i; /*make i the new root */
      parent[i] = temp;
   }
}
```

Program 5.19: Union function

– Time complexity of find(*i*): $O(\log_2 n)$

- **Lemma 5.4:** Let *T* be a tree with *n* nodes created as a result of *union2*. No node in *T* has level greater than $\lfloor \log_2 n \rfloor + 1$.

– Example 5.1



8個node，結果共 $\log_2 8+1=4$個levels

Figure 5.45: Trees achieving worst case bound

– Further improvement is possible

- **Definition [Collapsing rule]:** If *j* is a node on the path from *i* to its root then make *j* a child of the root.
- **Example 5.2:** Consider the tree created by union2 on the sequence of unions of Example 5.1. Now process the following 8 finds: *find*(*7*), *find*(*7*), …, *find*(*7*)

  The first *find*(*7*) requires going up three links and resetting two links. Each of the remaining seven finds requires going up only one links field.

- Equivalence Classes (previous example in Ch.4)

Let $0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, $8 \equiv 9$, $7 \equiv 4$, $6 \equiv 8$, $3 \equiv 5$, $2 \equiv 11$, $11 \equiv 0$.

  - We can regard the equivalence classes to be generated as sets.
  - Example 5.3

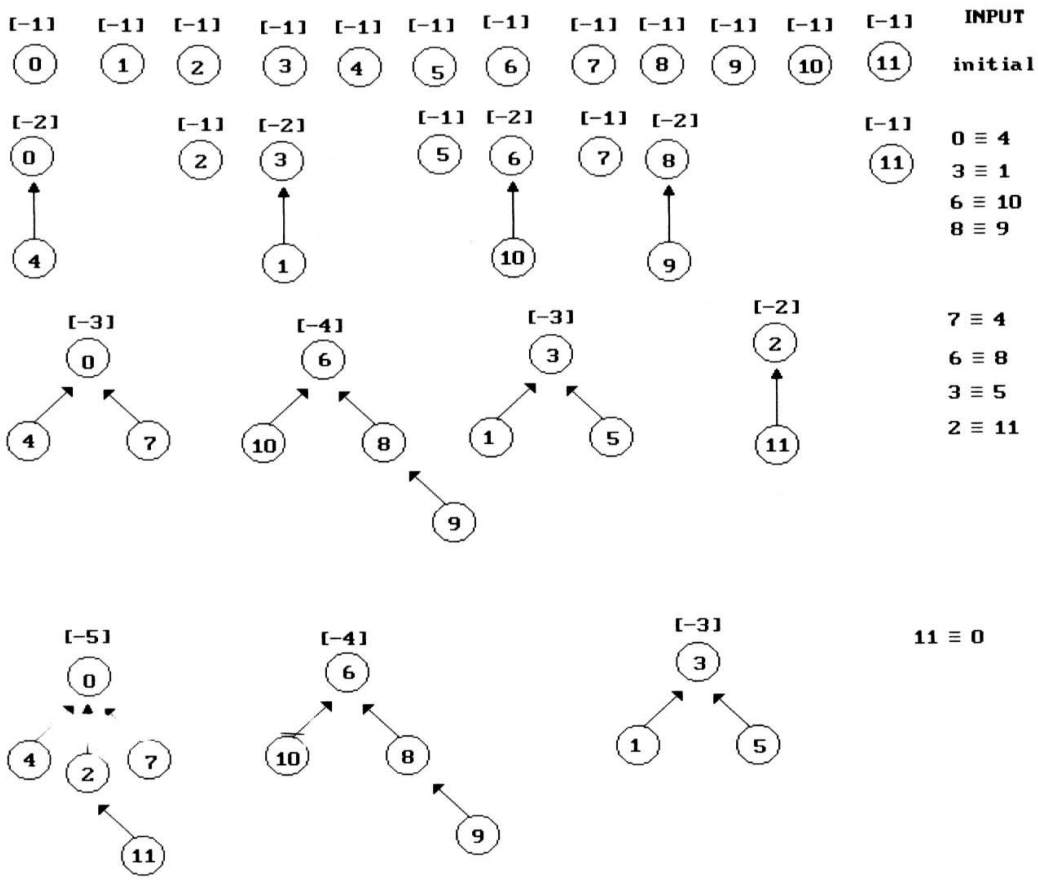


Figure 5.46: Trees for equivalence example

5.11 Counting Binary trees

- Distinct binary trees

Figure 5.47: Distinct binary trees with $n = 2$

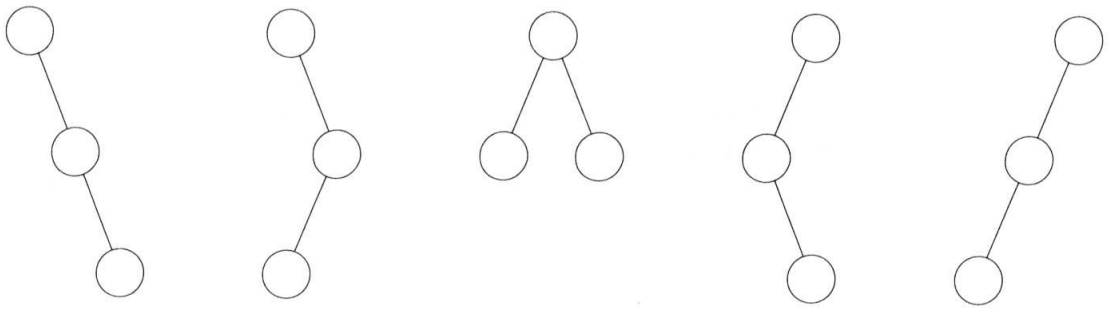


Figure 5.48: Distinct binary trees with $n = 3$

- Stack permutations
  - we can verify that every binary tree has a unique pair of preorder-inorder sequence.
  - Example
    - preorder: ABCDEFGHI
    - inorder: BCAEDGHFI

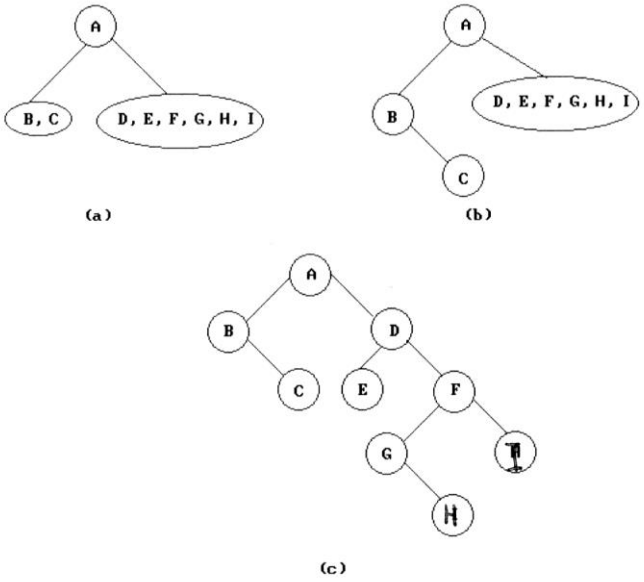


Figure 5.49: Constructing a binary tree from its inorder and preorder sequences

– If the nodes of the tree are numbered such that its preorder permutation is 1, 2, …, *n*, then from our earlier discussion it follows that distinct binary trees define distinct inorder permutations.
  - Example: the nodes of the tree in Figure 5.49(c) are numbered.

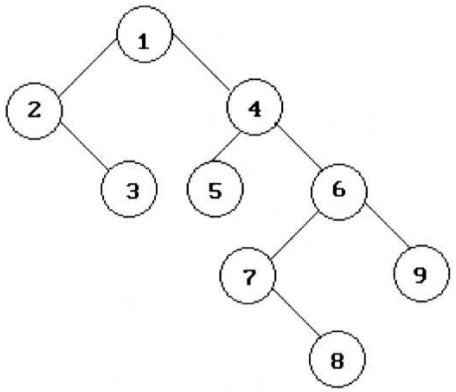


Figure 5.50: Binary tree of Figure 5.49(c) with its nodes numbered

- Example:

  preorder: 1, 2, 3

  possible permutation:

  (1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 2, 1)

  · obtaining (3, 2, 1) is impossible.

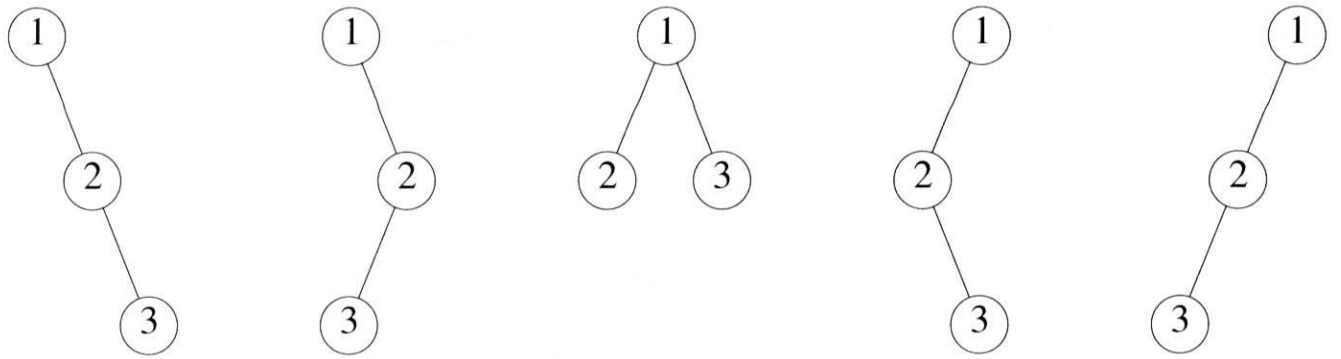


Figure 5.51: Binary trees corresponding to five permutations

- Matrix multiplication
  - How many different ways we can perform these multiplication?

$$M_1 * M_2 * \ldots * M_n$$

    - if $n = 3$, there are two possibilities.
      - $(M_1 * M_2) * M_3$
      - $M_1 * (M_2 * M_3)$
    - if $n = 4$, there are five possibilities.
      - $((M_1 * M_2) * M_3) * M_4$
      - $(M_1 * (M_2 * M_3)) * M_4$
      - $M_1 * ((M_2 * M_3) * M_4)$
      - $(M_1 * (M_2 * (M_3 * M_4)))$
      - $((M_1 * M_2) * (M_3 * M_4))$

- Let b_n be the number of different way to compute the product of n matrices.

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

– The number of binary trees with n nodes, the number of permutations of 1 to n obtainable with a stack, and the number of ways to multiply (n +1) matrices are all equal.

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

** Sum 裡 i 由0 開始

- Number of distinct binary trees
  - let b_n be the number of binary tree with distinct shape that can be formed each containing n nodes.

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$