제5장 Trees

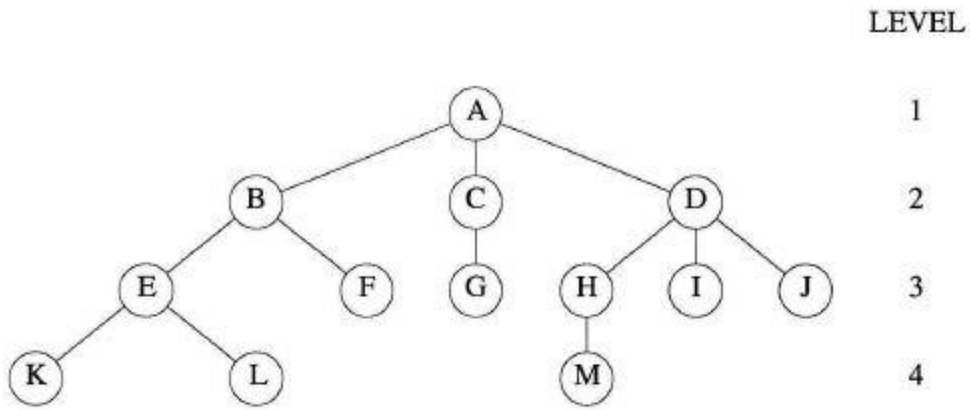
경북대학교 임경식 교수

5.1 Definition and terminologies

Definition: A *tree* is a finite set of one or more nodes such that

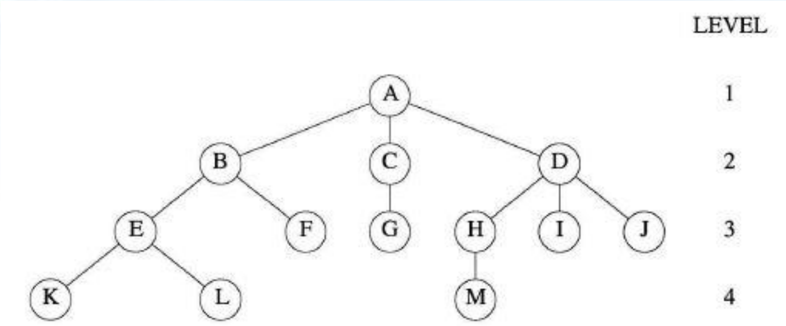
(1) There is a specially designated node called the *root*.

(2) The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \cdots, T_n$, where each of these sets is a tree. $T_1, \cdots, T_n$ are called the *subtrees* of the root. □



node degree, tree degree, leaf/terminal node, nonterminal node, children, parent, ancestors, siblings, level, tree height/depth

5.1 Definition and terminologies



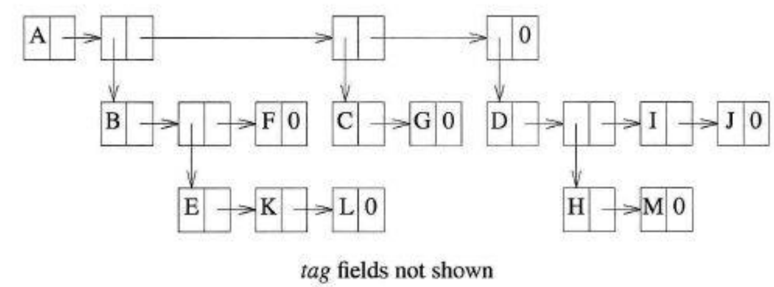


Figure 5.3: List representation of the tree of Figure 5.2

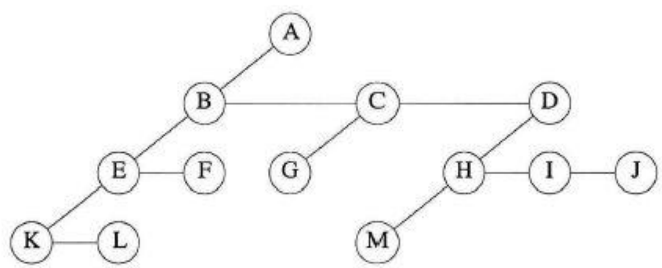


Figure 5.6: Left child-right sibling representation of tree of Figure 5.2

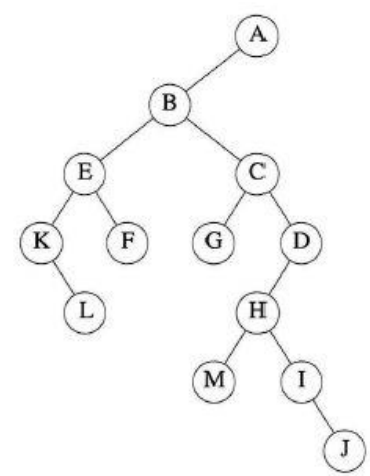


Figure 5.7: Left child-right child tree representation of tree of Figure 5.2

5.2 Binary trees

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

Figure 5.9: Two different binary trees

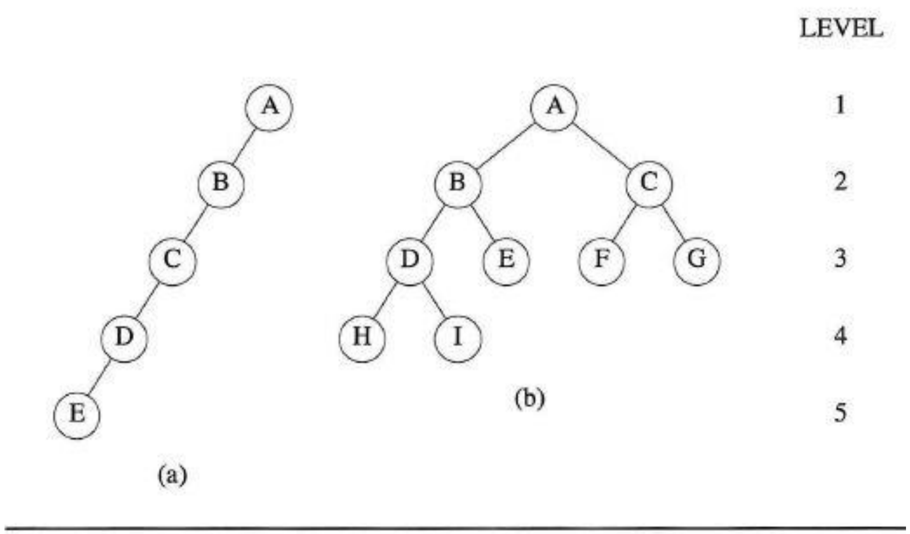


Figure 5.10: Skewed and complete binary trees

5.2 Binary trees

Lemma 5.2 [***Maximum number of nodes***]:

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

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

Proof:

(1) The proof is by induction on i.

Induction Base: The root is the only node on level $i = 1$. Hence, the maximum number of nodes on level $i = 1$ is $2^{i-1} = 2^0 = 1$.

Induction Hypothesis: Let i be an arbitrary positive integer greater than 1. Assume that the maximum number of nodes on level $i - 1$ is 2^{i-2}.

Induction Step: The maximum number of nodes on level $i - 1$ is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level $i-1$, or 2^{i-1}.

(2) The maximum number of nodes in a binary tree of depth k is

$$\sum_{i=1}^{k} (\text{maximum number of nodes on level } i) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1 \ \square$$

5.2 Binary trees

Lemma 5.3 [***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: Let n_1 be the number of nodes of degree one and n the total number of nodes. Since all nodes in T are at most of degree two, we have

$$n = n_0 + n_1 + n_2 \tag{5.1}$$

If we count the number of branches in a binary tree, we see that every node except the root has a branch leading into it. If B is the number of branches, then $n = B+1$. All branches stem from a node of degree one or two. Thus, $B = n_1 + 2n_2$. Hence, we obtain

$$n = B + 1 = n_1 + 2n_2 + 1 \tag{5.2}$$

Subtracting Eq. (5.2) from Eq. (5.1) and rearranging terms, we get

$$n_0 = n_2 + 1 \quad \square$$

5.2 Binary trees

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

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. □

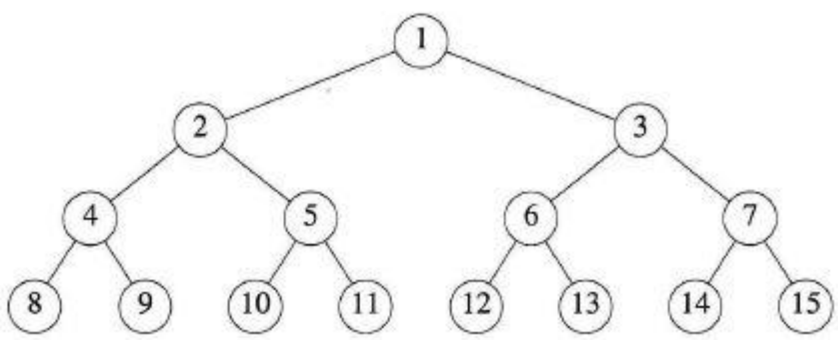


Figure 5.11: Full binary tree of depth 4 with sequential node numbers

Array representation of BT

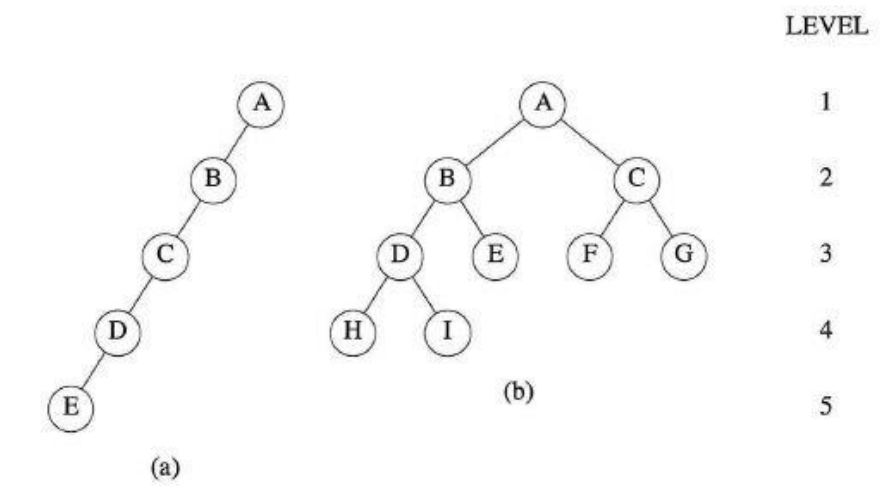


Figure 5.10: Skewed and complete binary trees

	tree
[0]	–
[1]	A
[2]	B
[3]	–
[4]	C
[5]	–
[6]	–
[7]	–
[8]	D
[9]	–
.	.
.	.
.	.
[16]	E

(a) Tree of Figure 5.10(a)

tree
–
A
B
C
D
E
F
G
H
I

(b) Tree of Figure 5.10(b)

Lemma 5.4: If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \leq i \leq n$, we have

(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 $2i \leq n$. If $2i > n$, then i has no left child.

(3) *rightChild* (i) is at $2i + 1$ if $2i + 1 \leq n$. If $2i + 1 > n$, then i has no right child.

Linked representation of BT

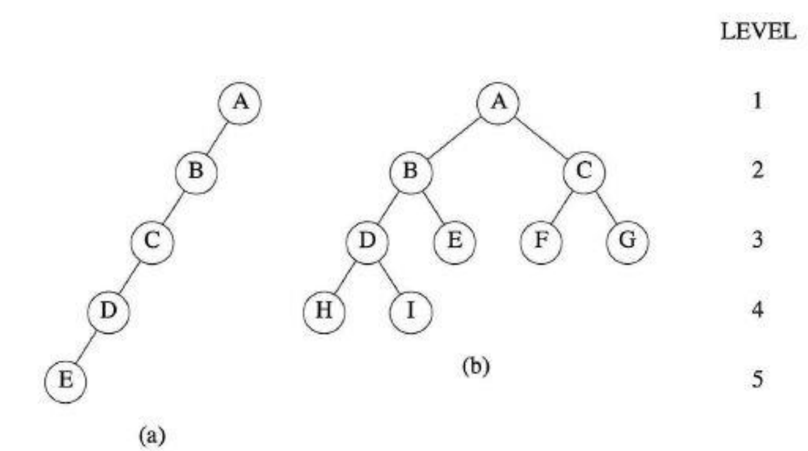
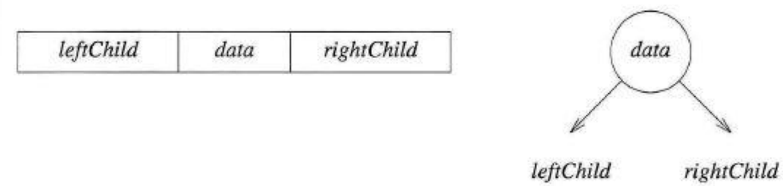
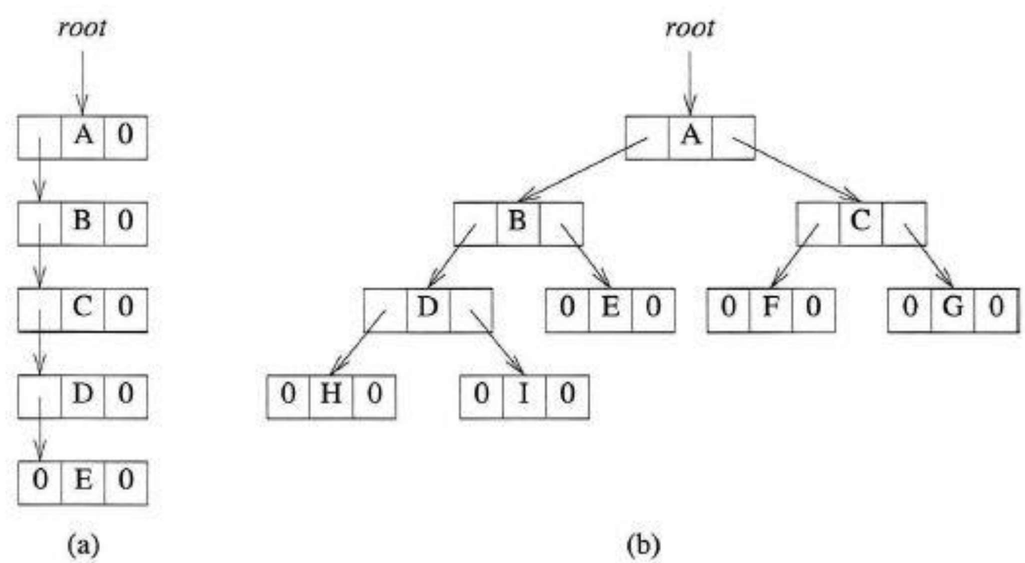


Figure 5.10: Skewed and complete binary trees

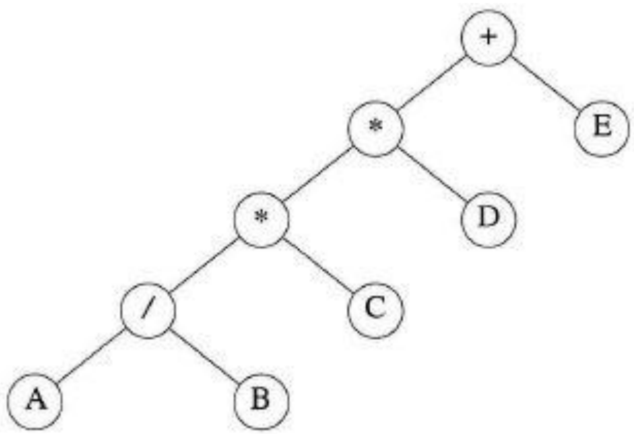
```
typedef struct node *treePointer;
typedef struct node {
        int data;
        treePointer leftChild, rightChild;
        };
```





5.3 Binary tree traversals

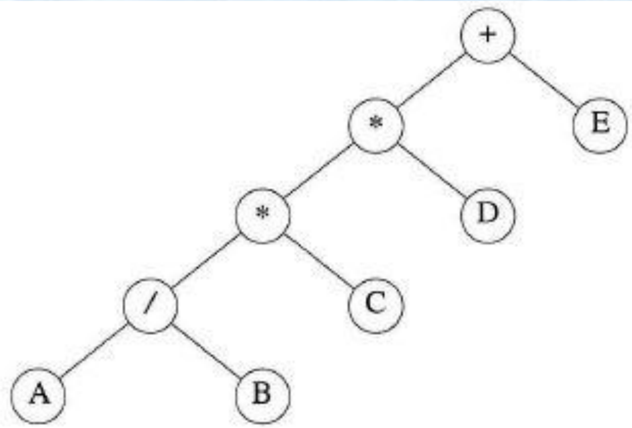
❖ If we let *L, V,* and *R* stand for moving left, visiting the node, and moving right when at a node, then there are six possible combinations of traversal: *LVR, LRV, VLR, VRL, RVL,* and *RLV.* If we adopt the convention that we traverse left before right, then only three traversals remain: *LVR, LRV,* and *VLR.* To these we assign the names *inorder, postorder,* and *preorder,* respectively, because of the position of the *V* with respect to the L and the *R.*




```
void inorder(treePointer ptr)
{/* inorder tree traversal */
   if (ptr) {
      inorder(ptr→leftChild);
      printf("%d",ptr→data);
      inorder(ptr→rightChild);
   }
}
```

$$A / B * C * D + E$$

5.3 Binary tree traversals



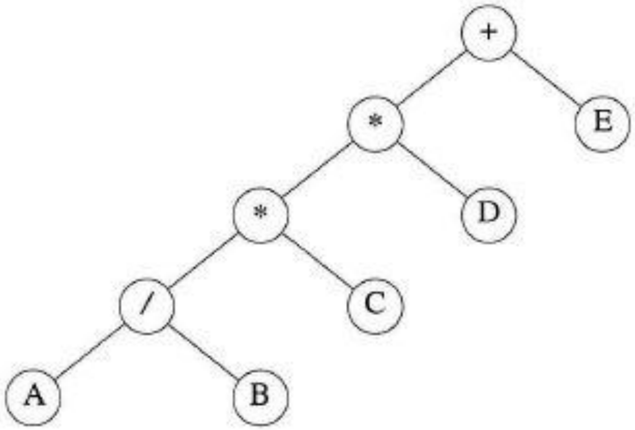

```
void preorder(treePointer ptr)
{/* preorder tree traversal */
   if (ptr) {
      printf("%d",ptr→data);
      preorder(ptr→leftChild);
      preorder(ptr→rightChild);
   }
}
```

$+ * * / A\ B\ C\ D\ E$

```
void postorder(treePointer ptr)
{/* postorder tree traversal */
   if (ptr) {
      postorder(ptr→leftChild);
      postorder(ptr→rightChild);
      printf("%d",ptr→data);
   }
}
```

$A\ B / C * D * E +$

5.3 Binary tree traversals




```
void iterInorder(treePointer node)
{
   int top = -1; /* initialize stack */
   treePointer stack[MAX_STACK_SIZE];
   for (;;) {
      for(; node; node = node→leftChild)
         push(node); /* add to stack */
      node = pop(); /* delete from stack */
      if (!node) break; /* empty stack */
      printf("%d", node→data);
      node = node→rightChild;
   }
}
```

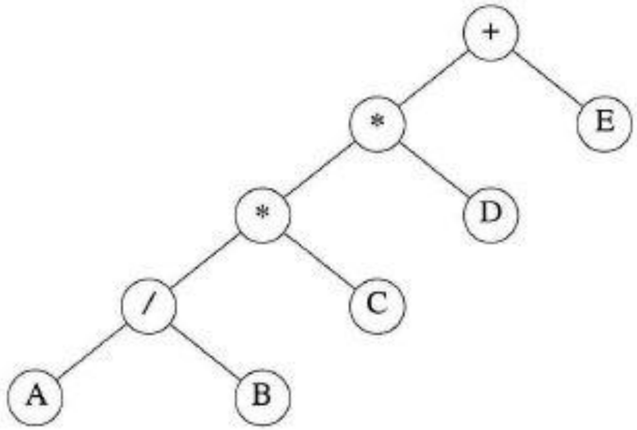
$A / B * C * D + E$

```
void levelOrder(treePointer ptr)
{/* level order tree traversal */
   int front = rear = 0;
   treePointer queue[MAX_QUEUE_SIZE];
   if (!ptr) return; /* empty tree */
   addq(ptr);
   for (;;) {
      ptr = deleteq();
      if (ptr) {
         printf("%d",ptr→data);
         if(ptr→leftChild)
            addq(ptr→leftChild);
         if (ptr→rightChild)
            addq(ptr→rightChild);
      }
      else break;
   }
}
```

$+ * E * D / C A B$

❖ Traversal without a stack => add a parent field to each node or use the threaded binary tree

5.4 Copying binary trees

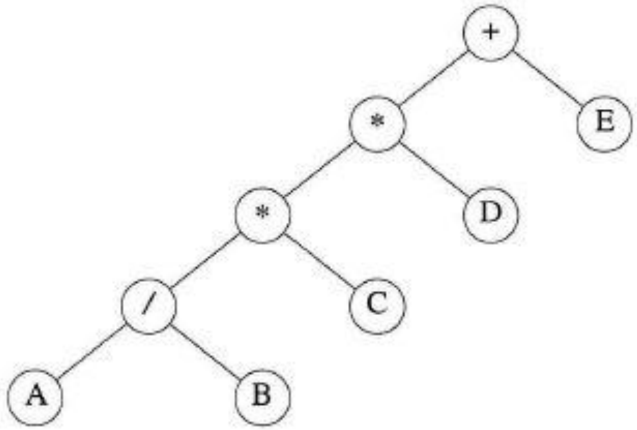



```
void postorder(treePointer ptr)
{/* postorder tree traversal */
   if (ptr) {
      postorder(ptr→leftChild);
      postorder(ptr→rightChild);
      printf("%d",ptr→data);
   }
}
```

$A\ B\ /\ C * D * E\ +$

```
treePointer copy(treePointer original)
{/* this function returns a treePointer to an exact copy
    of the original tree */
  treePointer temp;
  if (original) {
    MALLOC(temp, sizeof(*temp));
    temp→leftChild = copy(original→leftChild);
    temp→rightChild = copy(original→rightChild);
    temp→data = original→data;
    return temp;
  }
  return NULL;
}
```

5.4 Testing equality

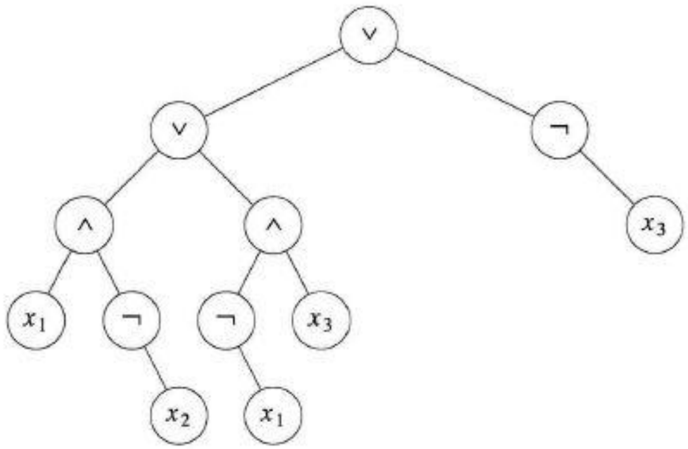



```
void preorder(treePointer ptr)
{/* preorder tree traversal */
   if (ptr) {
      printf("%d",ptr→data);
      preorder(ptr→leftChild);
      preorder(ptr→rightChild);
   }
}
```

$+ * * / A\ B\ C\ D\ E$

```
int equal(treePointer first, treePointer 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→leftChild,second→leftChild) &&
              equal(first→rightChild, second→rightChild))
}
```

5.4 The satisfiability problem



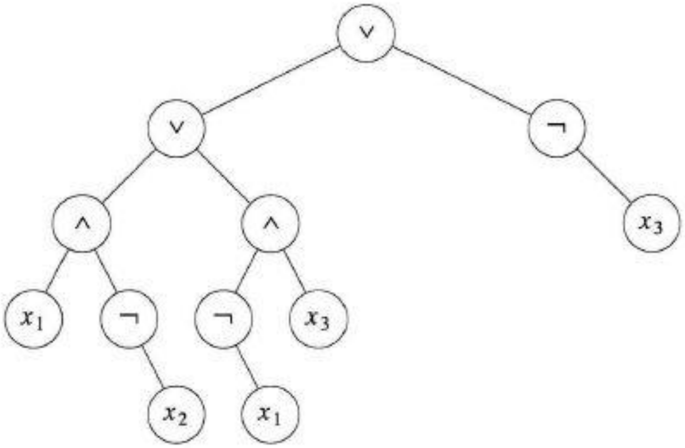
$$x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_3 \vee \neg x_3$$

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

```
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");
```

$O(g\,2^n)$

5.4 The satisfiability problem



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

leftChild	*data*	*value*	*rightChild*

```
typedef enum {not,and,or,true,false} logical;
typedef struct node *treePointer;
typedef struct node {
        treePointer leftChild;
        logical      data;
        short int    value;
        treePointer rightChild;
        } ;
```

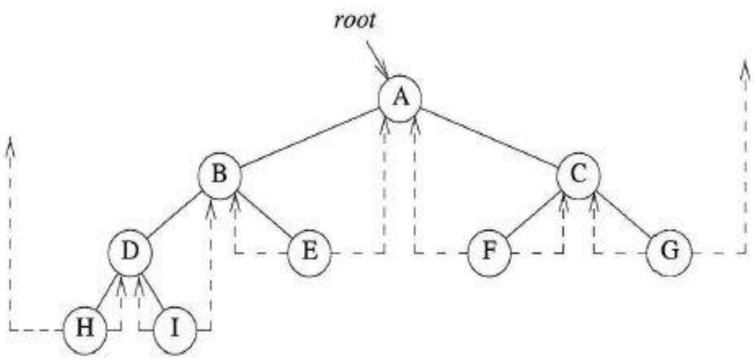
```
void postOrderEval(treePointer node)
{/* modified post order traversal to evaluate a
    propositional calculus tree */
   if (node) {
      postOrderEval(node→leftChild);
      postOrderEval(node→rightChild);
      switch(node→data) {
        case not:   node→value =
             !node→rightChild→value;
             break;
        case and:   node→value =
             node→rightChild→value &&
             node→leftChild→value;
             break;
        case or:    node→value =
             node→rightChild→value ||
             node→leftChild→value;
             break;
        case true:  node→value = TRUE;
             break;
        case false: node→value = FALSE;
      }
   }
}
```

5.5 Threaded binary trees

there are n + 1 null links out of 2 n total links. We replace the null links by pointers, called *threads*, to other nodes in the tree.

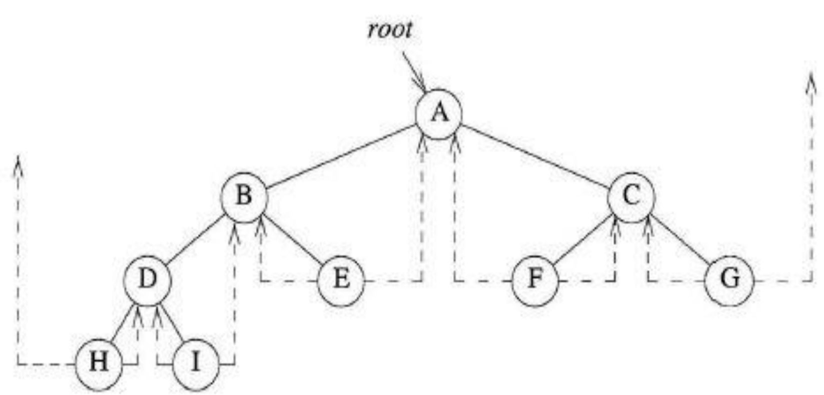
(1) If *ptr* → *leftChild* is null, replace *ptr* → *leftChild* with a pointer to the node that would be visited before *ptr* in an inorder traversal. That is we replace the null link with a pointer to the *inorder predecessor* of *ptr*.

(2) If *ptr* → *rightChild* is null, replace *ptr* → *rightChild* with a pointer to the node that would be visited after *ptr* in an inorder traversal. That is we replace the null link with a pointer to the *inorder successor* of *ptr*.

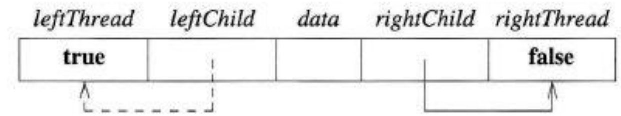


When we represent the tree in memory, we must be able to distinguish between threads and normal pointers. This is done by adding two additional fields to the node structure, *leftThread* and *rightThread.* Assume that *ptr* is an arbitrary node in a threaded tree. If *ptr ->leftThread = TRUE,* then *ptr ->leftChild* contains a thread; otherwise it contains a pointer to the left child. Similarly, if *ptr ->rightThread = TRUE,* then *ptr ->rightChild* contains a thread; otherwise it contains a pointer to the right child.

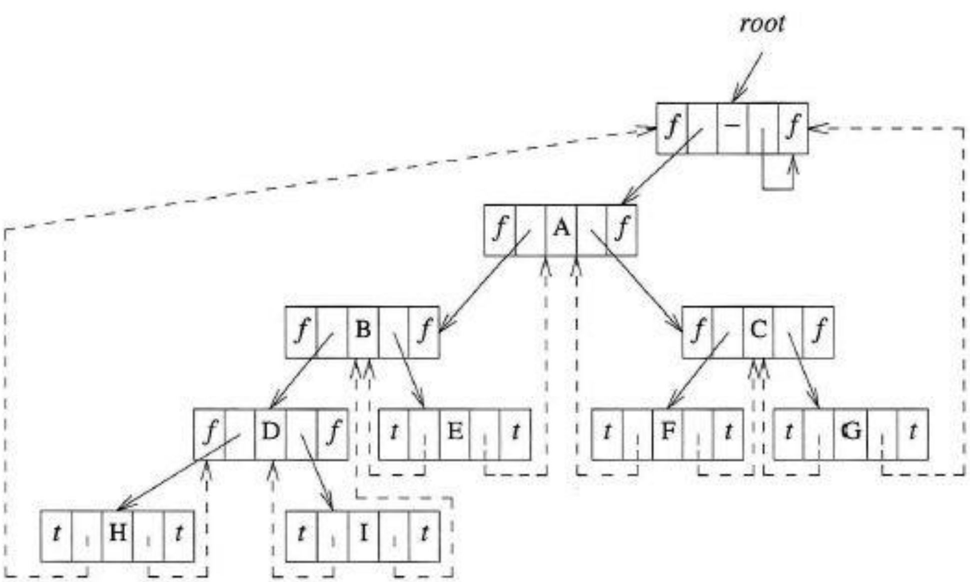
5.5 Threaded binary trees




```
typedef struct threadedTree *threadedPointer;
typedef struct threadedTree {
        short int leftThread;
        threadedPointer leftChild;
        char data;
        threadedPointer rightChild;
        short int rightThread;
        };
```

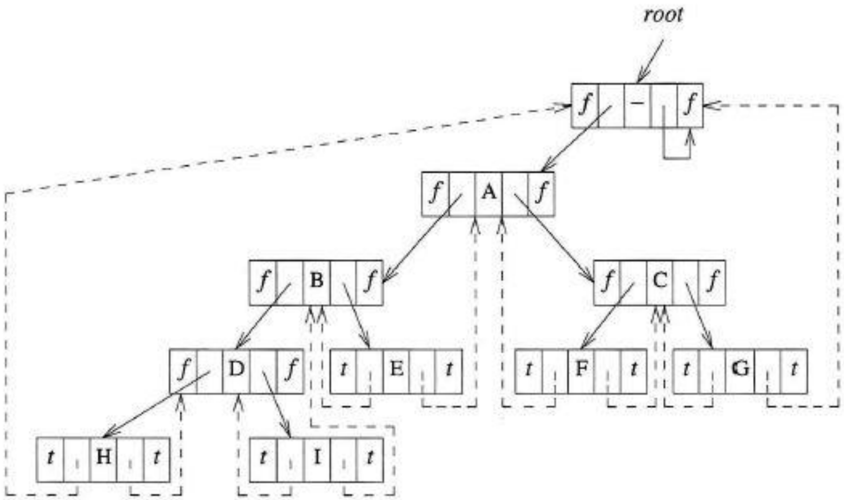


22: An empty threaded binary tree



f= **false**; *t* = **true**

5.5 Threaded binary trees



f = **false**; *t* = **true**

By using the threads, we can perform an inorder traversal without making use of a stack.

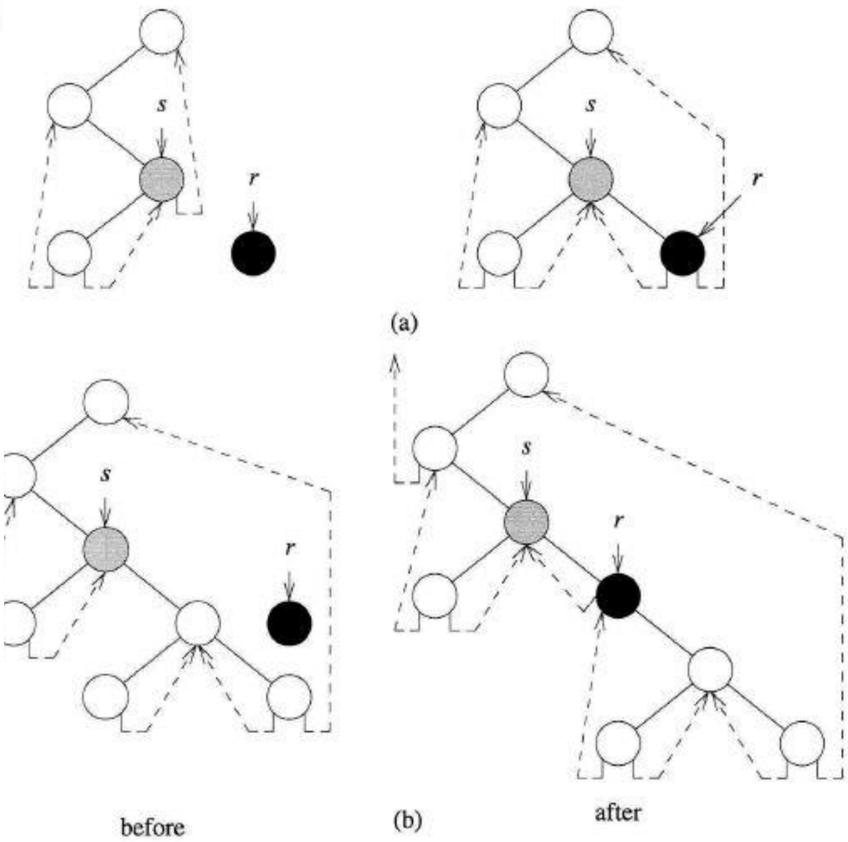
O(*n*)

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

```
threadedPointer insucc(threadedPointer tree)
{/* find the inorder sucessor of tree in a threaded binary
    tree */
   threadedPointer temp;
   temp = tree→rightChild;
   if (!tree→rightThread)
      while (!temp→leftThread)
         temp = temp→leftChild;
   return temp;
}
```

5.5 Threaded binary trees

```
void insertRight(threadedPointer s, threadedPointer r)
{/* insert r as the right child of s */
  threadedPointer temp;
  r→rightChild = parent→rightChild;
  r→rightThread = parent→rightThread;
  r→leftChild = parent;
  r→leftThread = TRUE;
  s→rightChild = child;
  s→rightThread = FALSE;
  if (!r→rightThread) {
    temp = insucc(r);
    temp→leftChild = r;
  }
}
```



5.6 Heaps

Heaps are frequently used to implement *priority queues.* In this kind of queue, the element to be deleted is the one with highest (or lowest) priority. At any time, an element with arbitrary priority can be inserted into the queue.

The simplest way to represent a priority queue is as an unordered linear list. Regardless of whether this list is represented sequentially or as a chain, the *isEmpty* function takes O(1) time; the *top* () function takes $\Theta(n)$ time, where n is the number of elements in the priority queue; a push can be done in O(1) time as it doesn't matter where in the list the new element is inserted; and a *pop* takes $\Theta(n)$ time as me must first find the element with max priority and then delete it. As we shall see shortly, when a max heap is used, the complexity of *isEmpty* and *top* is O(1) and that of *push* and *pop* is O(log n).

5.6 Heaps

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

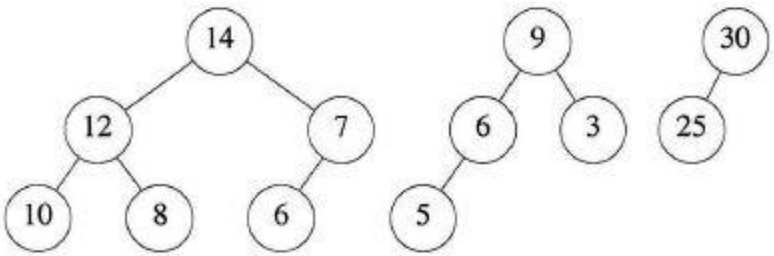


Figure 5.25: Max heaps

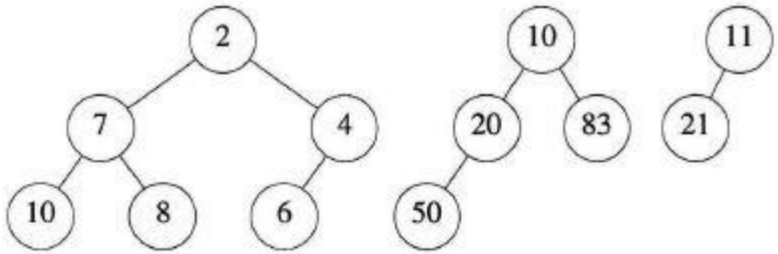
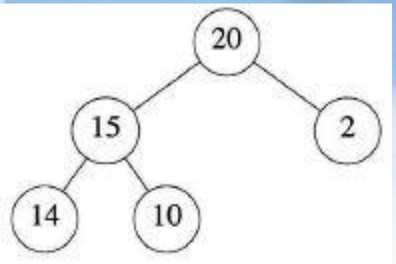


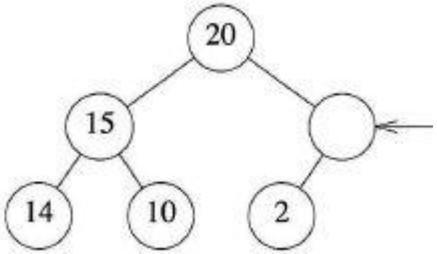
Figure 5.26: Min heaps

5.6 Heaps

```
#define MAX_ELEMENTS 200 /* maximum heap size+1 */
#define HEAP_FULL(n) (n == MAX_ELEMENTS-1)
#define HEAP_EMPTY(n) (!n)
typedef struct {
        int key;
        /* other fields */
        } element;
element heap[MAX_ELEMENTS];
int n = 0;
```

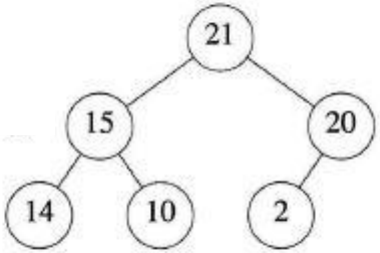


push (21, &n);




```
void push(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(EXIT_FAILURE);
   }
   i = ++(*n);
   while ((i != 1) && (item.key > heap[i/2].key)) {
      heap[i] = heap[i/2];
      i /= 2;
   }
   heap[i] = item;
}
```

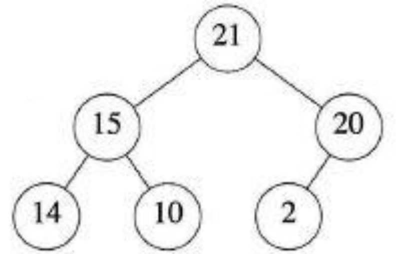
Program 5.13: Insertion into a max heap

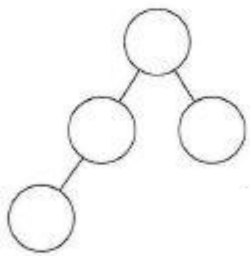


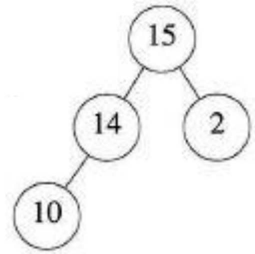
$O(\log_2 n)$

5.6 Heaps

```
element pop(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(EXIT_FAILURE);
   }
   /* 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;
}
```







$O(\log_2 n)$

5.6 Heaps

Heap sort

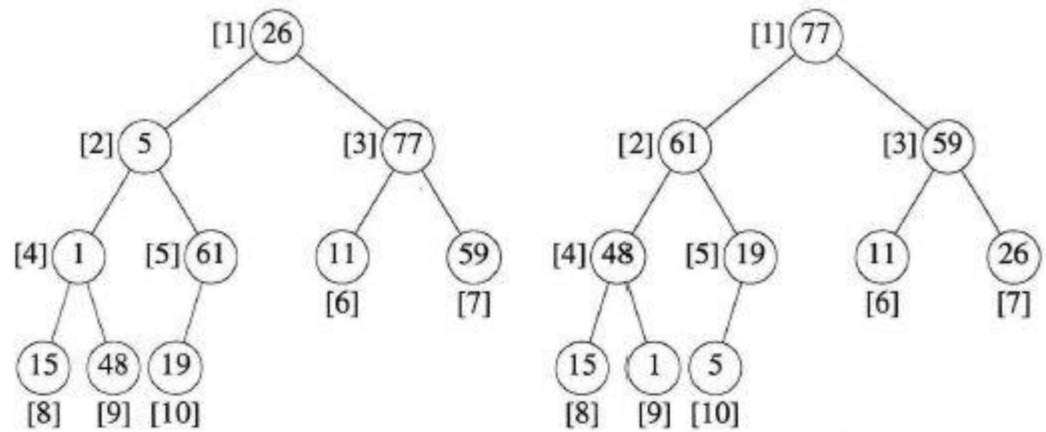
```
void heapSort(element a[], int n)
{/* perform a heap sort on a[1:n] */
   int i,j;
   element temp;

   for (i = n/2; i > 0; i--)
      adjust(a,i,n);
   for (i = n-1; i > 0; i--) {
      SWAP(a[1],a[i+1],temp);
      adjust(a,1,i);
   }
}
```

Program 7.13: Heap sort

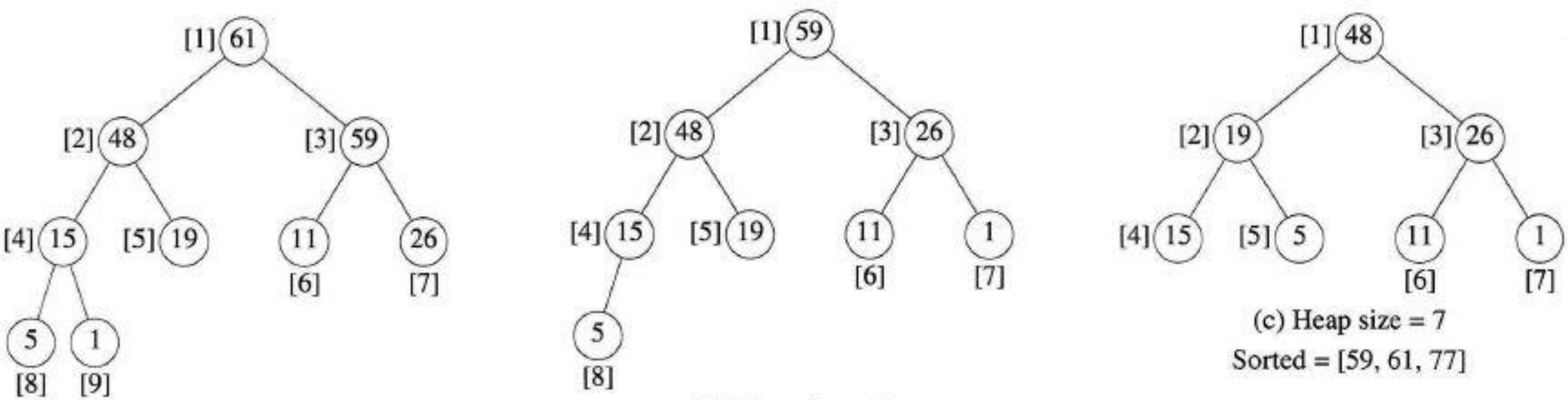
```
void adjust(element a[], int root, int n)
{/* adjust the binary tree to establish the heap */
   int child,rootkey;
   element temp;
   temp = a[root];
   rootkey = a[root].key;
   child = 2 * root;                          /* left child */
   while (child <= n) {
      if ((child < n) &&
      (a[child].key < a[child+1].key))
         child++;
      if (rootkey > a[child].key) /* compare root and
                                     max. child */
         break;
      else {
         a[child / 2] = a[child]; /* move to parent */
         child *= 2;
      }
   }
   a[child/2] = temp;
}
```

5.6 Heaps



(a) Input array

(b) Initial heap

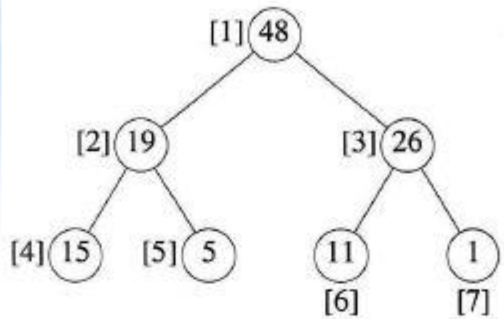


(a) Heap size = 9
Sorted = [77]

(b) Heap size = 8
Sorted = [61, 77]

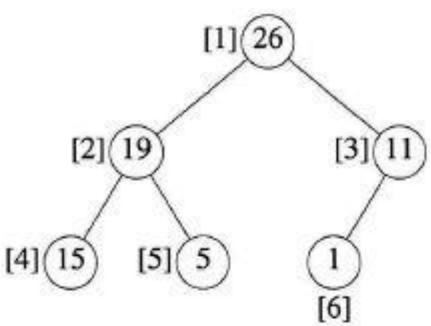
(c) Heap size = 7
Sorted = [59, 61, 77]

5.6 Heaps



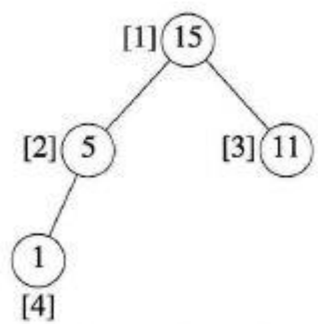
(c) Heap size = 7

Sorted = [59, 61, 77]



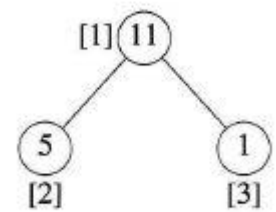
(d) Heap size = 6

Sorted = [48, 59, 61, 77]



(f) Heap size = 4

[19, 26, 48, 59, 61, 77]



(g) Heap size = 3

[15, 19, 26, 48, 59, 61, 77]

worst-case and average computing time $O(n \log n)$

5.7 Binary search trees

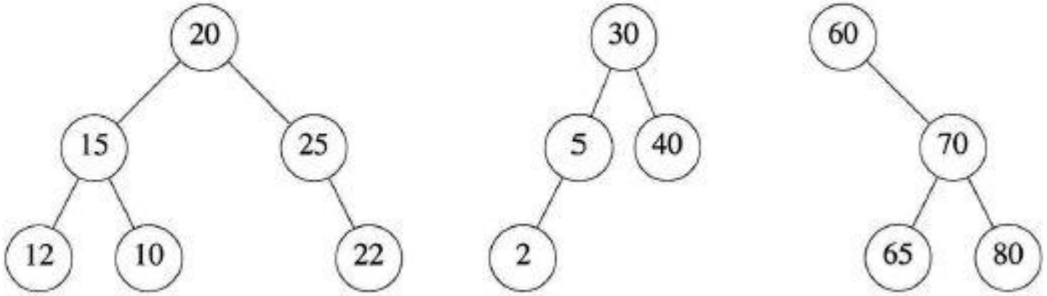
Definition: A *binary search tree* is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

(1) Each node has exactly one key and the keys in the tree are distinct.

(2) The keys (if any) in the left subtree are smaller than the key in the root.

(3) The keys (if any) in the right subtree are larger than the key in the root.

(4) The left and right subtrees are also binary search trees. □

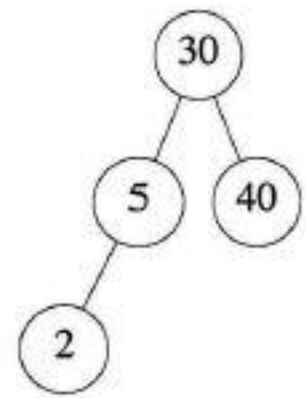


no yes yes

5.7 Binary search trees

```
element* search(treePointer root, int key)
{/* return a pointer to the element whose key is k, if
    there is no such element, return NULL. */
   if (!root) return NULL;
   if (k == root→data.key) return &(root→data);
   if (k < root→data.key)
     return search(root→leftChild, k);
   return search(root→rightChild, k);
}
```

Program 5.15: Recursive search of a binary search tree



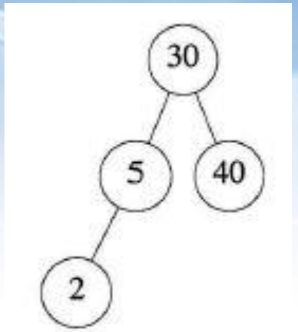

```
element* iterSearch(treePointer tree, int k)
{/* return a pointer to the element whose key is k,  if
    there is no such element, return NULL. */
   while (tree) {
      if (k == tree→data.key) return &(tree→data);
      if (k < tree→data.key)
         tree = tree→leftChild;
      else
         tree = tree→rightChild;
      }
   return NULL;
}
```

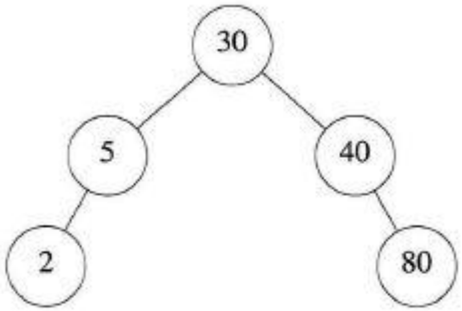
Program 5.16: Iterative search of a binary search tree

If *h* is the height of the binary search tree, then we can perform the search using either *search* or *iterSearch* in *O(h).* However, *search* has an additional stack space requirement which is O(*h*).

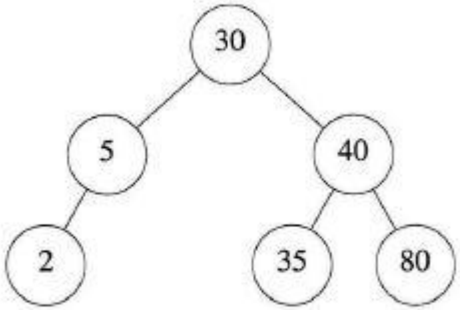
5.7 Binary search trees

To insert a dictionary pair whose key is *k,* we must first verify that the key is different from those of existing pairs. To do this we search the tree. If the search is unsuccessful, then we insert the pair at the point the search terminated.





(a) Insert 80



(b) Insert 35

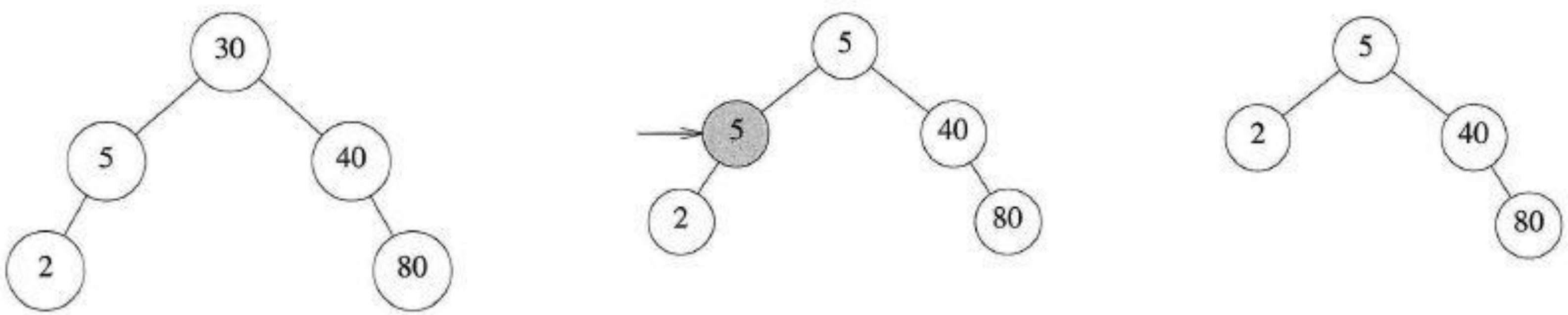
```
void insert(treePointer *node, int k, iType theItem)
{/* if k is in the tree pointed at by node do nothing;
    otherwise add a new node with data = (k, theItem) */
   treePointer ptr, temp = modifiedSearch(*node, k);
   if (temp || !(*node)) {
      /* k is not in the tree */
      MALLOC(ptr, sizeof(*ptr));
      ptr→data.key = k;
      ptr→data.item = theItem;
      ptr→leftChild = ptr→rightChild = NULL;
      if (*node) /* insert as child of temp */
         if (k < temp→data.key) temp→leftChild = ptr;
         else temp→rightChild = ptr;
      else *node = ptr;
   }
}
```

O(h)

5.7 Binary search trees

Deletion of a leaf is quite easy. The deletion of a nonleaf that has only one child is also easy. The node containing the dictionary pair to be deleted is freed, and its single-child takes the place of the freed node. When the pair to be deleted is in a nonleaf node that has two children, the pair to be deleted is replaced by either the largest pair in its left subtree or the smallest one in its right subtree. Then we proceed to delete this replacing pair from the subtree from which it was taken.

When deleting 30,



O(h)

5.8 Selection trees

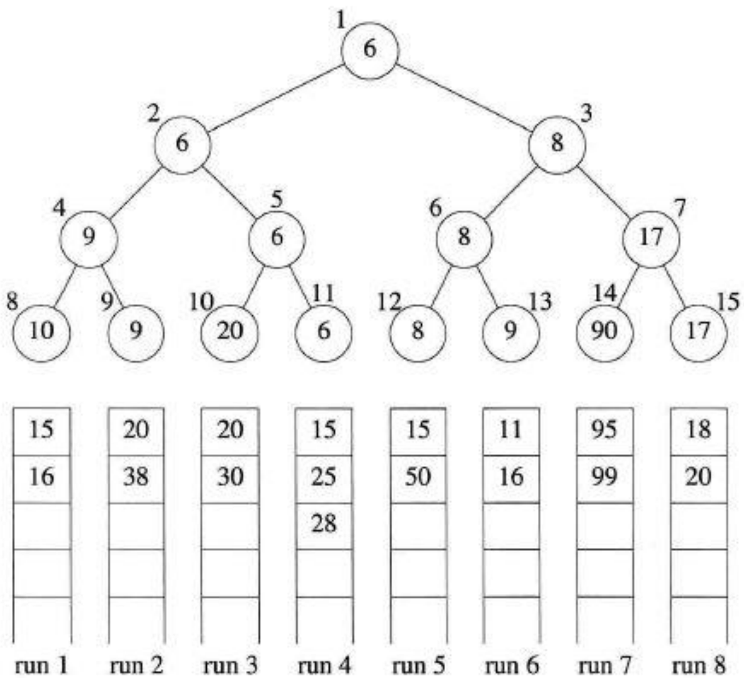
Suppose we have *k* ordered sequences, called *runs*, that are to be merged into a single ordered sequence. Each run consists of some records and is in nondecreasing order of a designated field called the *key.* Let *n* be the number of records in all *k* runs together.

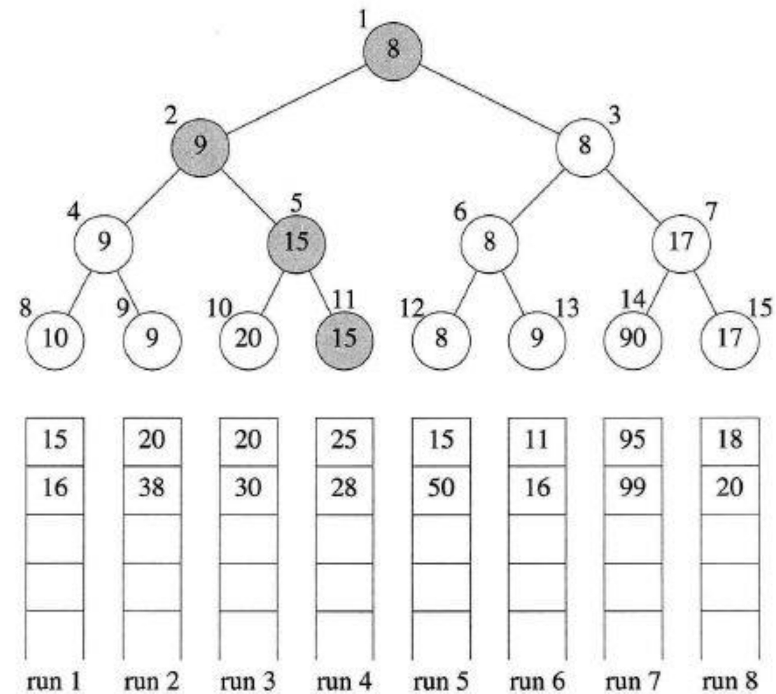
The most direct way to merge *k* runs is to make *k-1* comparisons to determine the next record to output. Hence O(nk).

For *k* > 2, we can achieve a reduction in the number of comparisons needed to find the next smallest element by using the *selection tree* data structure. There are two kinds of selection trees: *winner trees* and *loser trees.*

5.8 Selection trees

A *winner tree* is a complete binary tree in which each node represents the smaller of its two children. Thus, the root node represents the smallest node in the tree. Then, each nonleaf node in the tree represents the winner of a tournament, and the root node represents the overall winner, or the smallest key. Each leaf node represents the first record in the corresponding run. Since the records being merged are generally large, each node will contain only a pointer to the record it represents.



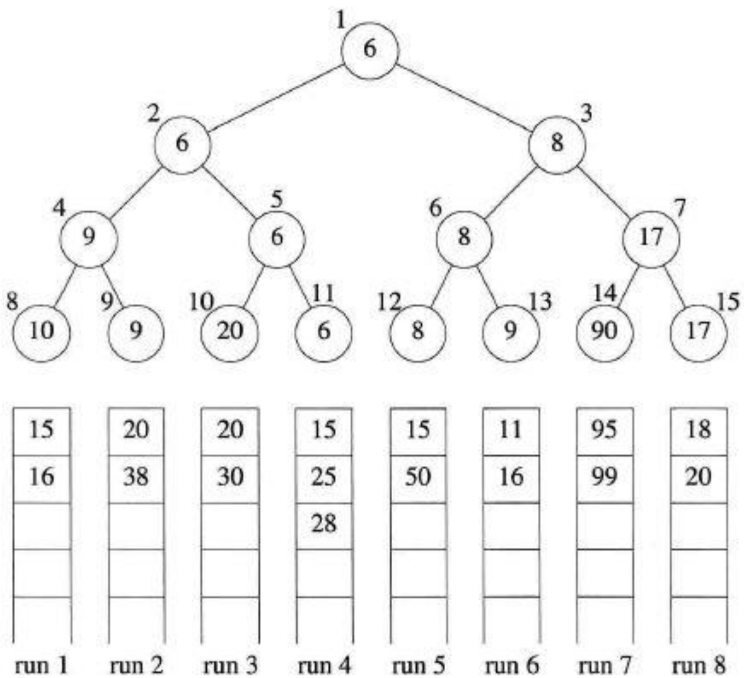


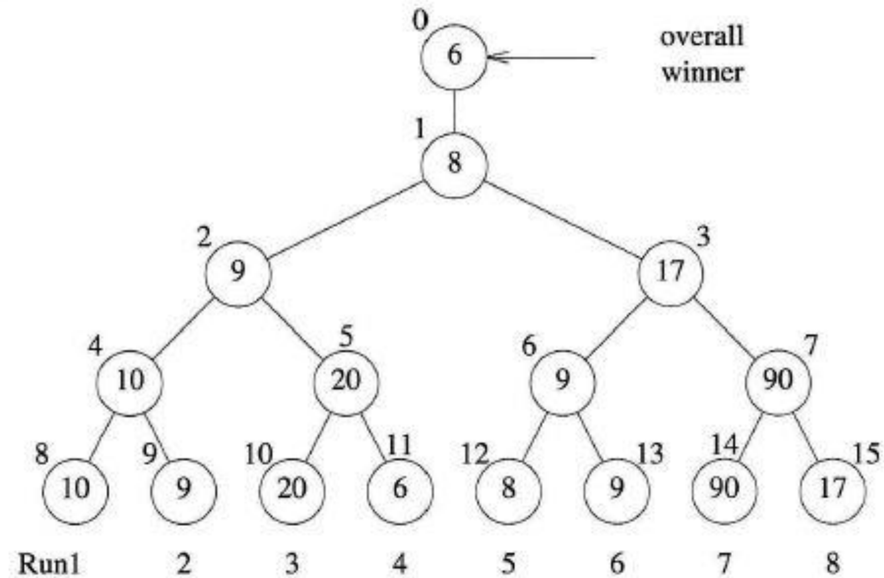
Time complexity $O(n \log_2 k)$

Space complexity $O(n)$

5.8 Selection trees

After the record with the smallest key value is output, the tree is to be restructured by placing in each nonleaf node a pointer to the record that loses the tournament rather than to the winner of the tournament. A selection tree in which each nonleaf node retains a pointer to the loser is called a *loser tree*.



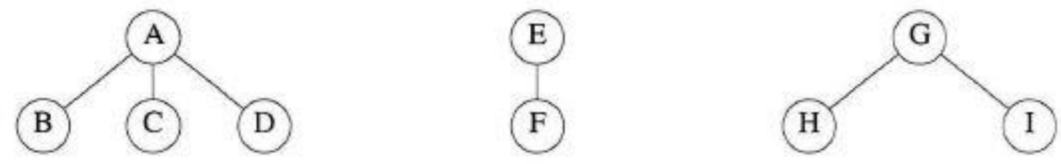


Time complexity $O(n \log_2 k)$

Space complexity $O(n)$

5.9 Forests

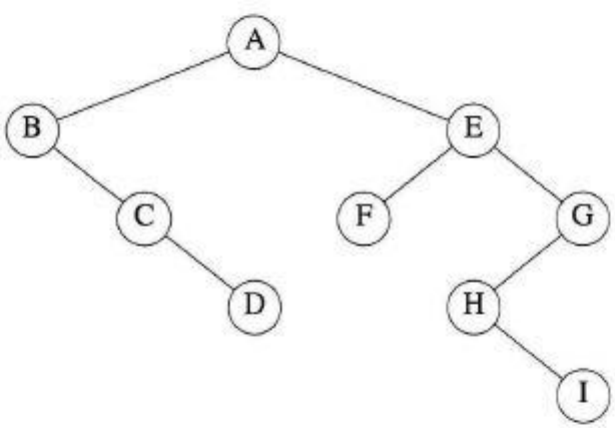
Definition: A *forest* is a set of $n \geq 0$ disjoint trees. □



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

(1) is empty if $n = 0$

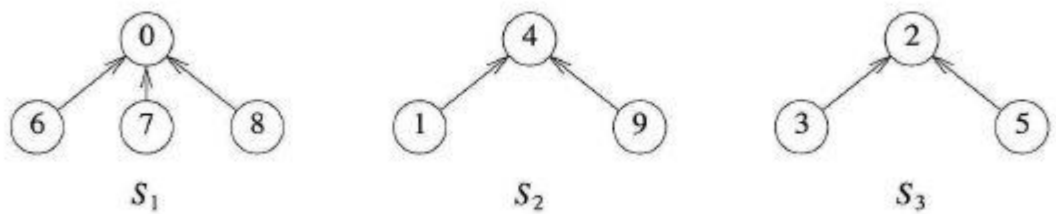
(2) has root equal to root (T_1); has left subtree equal to $B(T_{11}, T_{12}, \cdots, T_{1m})$, where $T_{11}, \cdots, T_{1m}$ are the subtrees of root(T_1); and has right subtree $B(T_2, \cdots, T_n)$.



5.10 Disjoint sets

We study the use of trees in the representation of sets. For simplicity, we assume that the elements of the sets are the numbers 0, 1, 2, · · · , n-1. We also assume that the sets being represented are pairwise disjoint, that is, if *Si* and *Sj* are two sets and *i != j,* then there is no element that is in both *Si* and *Sj.*

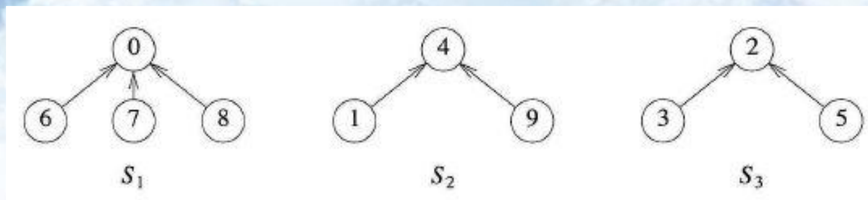
If we have 10 elements numbered 0 through 9, we may partition them into three disjoint sets, S1 = {0, 6, 7, 8}, S2 = {1, 4, 9}, and S3 = {2, 3, 5}.



(1) *Disjoint set union.* If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = {all elements, x, such that x is in S_i or S_j}. Thus, $S_1 \cup S_2 = \{0, 6, 7, 8, 1, 4, 9\}$. Since we have assumed that all sets are disjoint, following the union of S_i and S_j we can assume that the sets S_i and S_j no longer exist independently. That is, we replace them by $S_i \cup S_j$.

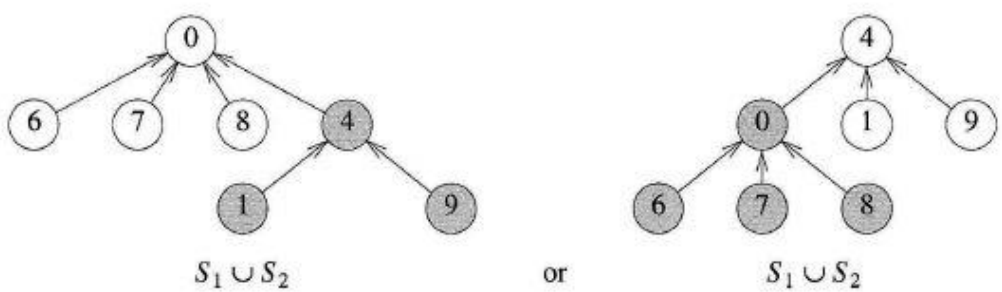
(2) *Find(i).* Find the set containing the element, i. For example, 3 is in set S_3 and 8 is in set S_1.

5.10 Disjoint sets



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

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



5.10 Disjoint sets

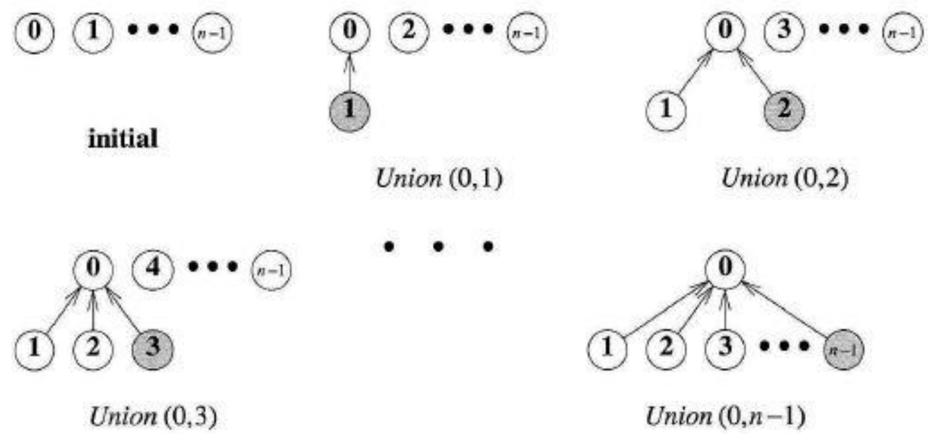
union(0, 1), *find*(0)
union(1, 2), *find*(0)
.
.
.
union(*n*−2, *n*−1), *find*(0)

Total time to process the *n-1* unions $O(n)$

Total time to process the *n-1* finds $\sum_{i=2}^{n} i = O(n^2)$

5.10 Disjoint sets

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. □




```
void weightedUnion(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;
   }
}
```

5.10 Disjoint sets

Lemma 5.5: Let T be a tree with n nodes created as a result of *weightedUnion*. No node in T has level greater than $\lfloor \log_2 n \rfloor + 1$.

Proof: The lemma is clearly true for $n = 1$. Assume that it is true for all trees with i nodes, $i \le n - 1$. We show that it is also true for $i = n$. Let T be a tree with n nodes created by *weightedUnion*. Consider the last union operation performed, *union*(k, j). Let m be the number of nodes in tree j and $n-m$, the number of nodes in k. Without loss of generality, we may assume that $1 \le m \le n/2$. Then the maximum level of any node in T is either the same as k or is one more than in j. If the former is the case, then the maximum level in T is $\le \lfloor \log_2(n-m) \rfloor + 1 \le \lfloor \log_2 n \rfloor + 1$. If the latter is the case, then the maximum level is $\le \lfloor \log_2 m \rfloor + 2 \le \lfloor \log_2 n/2 \rfloor + 2 \le \lfloor \log_2 n \rfloor + 1$. □

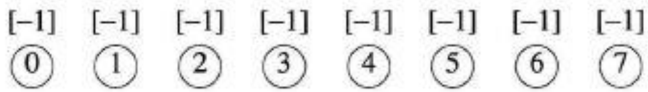
Example 5.3: Consider the behavior of *weightedUnion* on the following sequence of unions starting from the initial configuration of $parent[i] = -count[i] = -1, 0 \le i < n = 2^3$:

union(0, 1)	*union*(2, 3)	*union*(4, 5)	*union*(6, 7)
union(0, 2)	*union*(4, 6)	*union*(0, 4)	

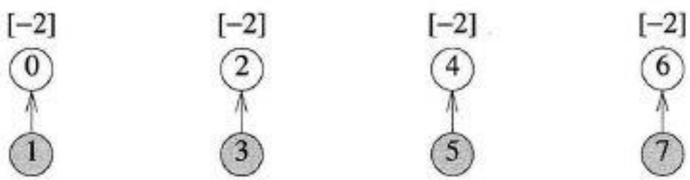
When the sequence of unions is performed by columns (i.e., top to bottom within a column with column 1 first, column 2 next, and so on), the trees of Figure 5.43 are obtained. As is evident from this example, in the general case, the maximum level can be $\lfloor \log_2 m \rfloor + 1$ if the tree has m nodes. □

From Lemma 5.5, it follows that the time to process a find is O(log m) if there are m elements in a tree. If an intermixed sequence of $u - 1$ union and f find operations is to be processed, the time becomes $O(u + f \log u)$, as no tree has more than u nodes in it. Of course, we need $O(n)$ additional time to initialize the n-tree forest.

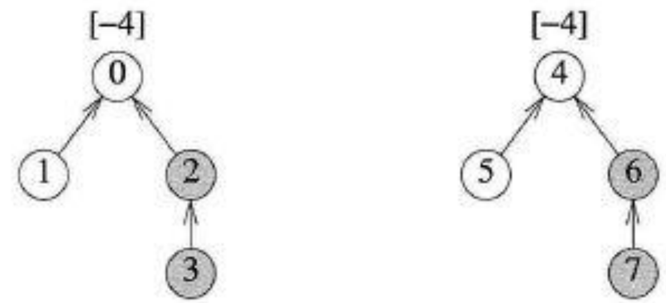
5.10 Disjoint sets



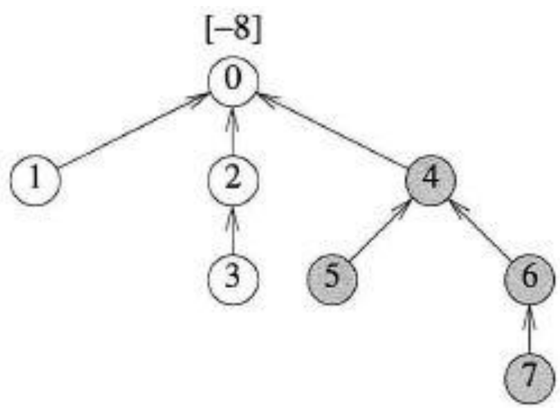
(a) Initial height-1 trees



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

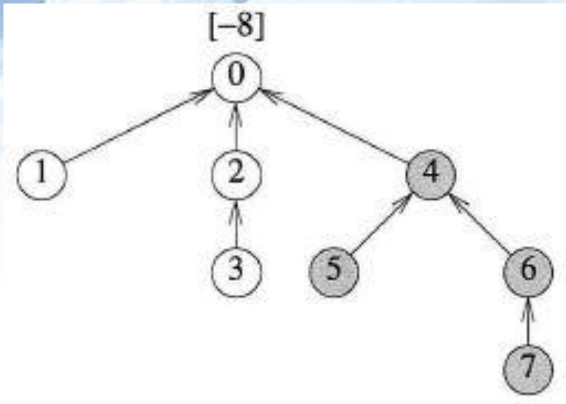


(c) Height-3 trees following *Union* (0,2) and (4,6)



(d) Height-4 tree following *Union* (0,4)

5.10 Disjoint sets



(d) Height-4 tree following *Union* (0,4)

```
int collapsingFind(int i)
{/* find the root of the tree containing element i. Use the
    collapsing rule to collapse all nodes from i to root */
   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;
}
```

Example 5.4: Consider the tree created by function *weightedUnion* on the sequence of unions of Example 5.5. Now process the following eight finds:

$$find(7), find(7), \cdots, find(7)$$

If *simpleFind* is used, each *find*(7) requires going up three parent link fields for a total of 24 moves to process all eight finds. When *collapsingFind* is used, the first *find*(7) requires going up three links and then resetting two links. Note that even though only two parent links need to be reset, function *collapsingFind* will actually reset three (the parent of 4 is reset to 0). Each of the remaining seven finds requires going up only one link field. The total cost is now only 13 moves. □

5.10 Disjoint sets

Application to equivalence classes

Definition: A relation, $\equiv$, over a set, S, is said to be an *equivalence relation* over S *iff* it is symmetric, reflexive, and transitive over S. □

Examples of equivalence relations are numerous. For example, the "equal to" (=) relationship is an equivalence relation since

(1) $x = x$

(2) $x = y$ implies $y = x$

(3) $x = y$ and $y = z$ implies that $x = z$

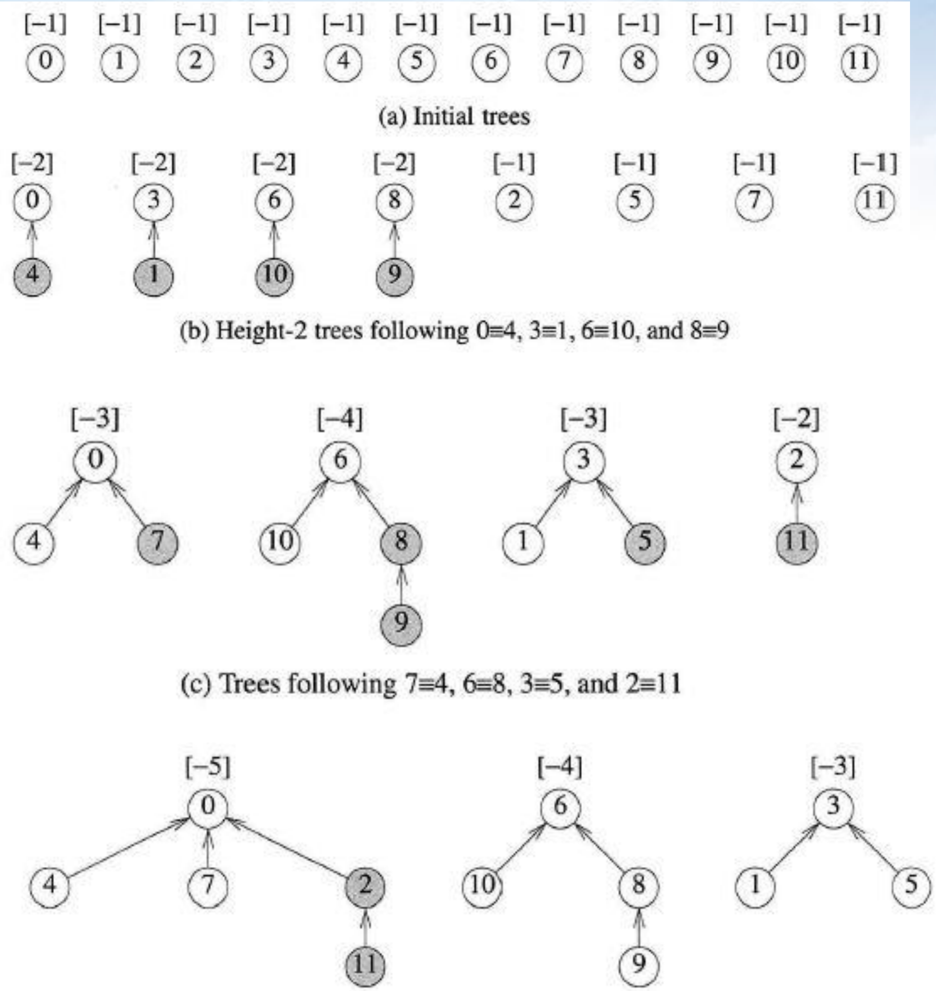
$$0 \equiv 4,\ 3 \equiv 1,\ 6 \equiv 10,\ 8 \equiv 9,\ 7 \equiv 4,\ 6 \equiv 8,\ 3 \equiv 5,\ 2 \equiv 11,\ 11 \equiv 0$$

use an equivalence relation to partition a set *S* into equivalence classes such that two members *x* and y of *S* are in the same equivalence class *iff* $x \equiv y$.

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

5.10 Disjoint sets

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



(a) Initial trees

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

(c) Trees following 7≡4, 6≡8, 3≡5, and 2≡11

(d) Trees following 11≡0

5.11 Counting binary trees

the number of distinct binary trees having *n* nodes,
the number of distinct permutations of the numbers from 1 through *n* obtainable by a stack,
the number of distinct ways of multiplying *n* + 1 matrices

Distinct binary trees

Figure 5.45: Distinct binary trees with $n = 2$

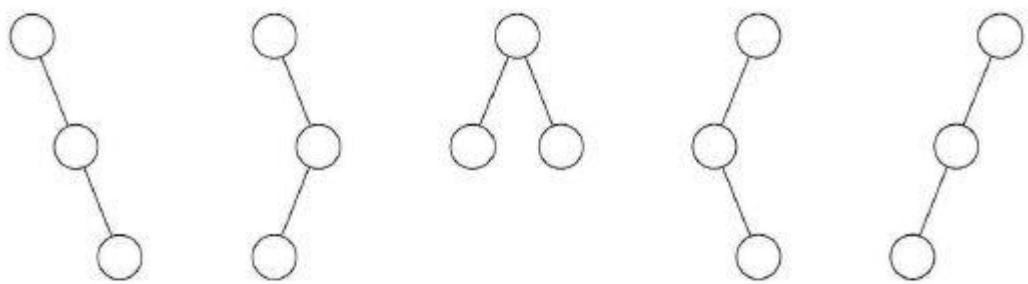
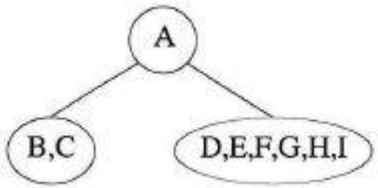


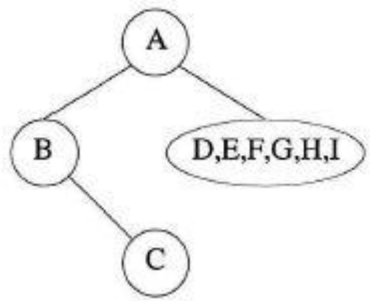
Figure 5.46: Distinct binary trees with $n = 3$

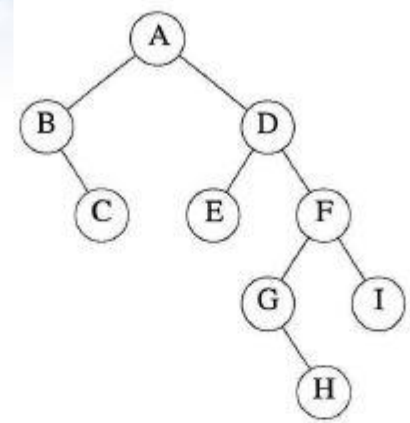
5.11 Counting binary trees

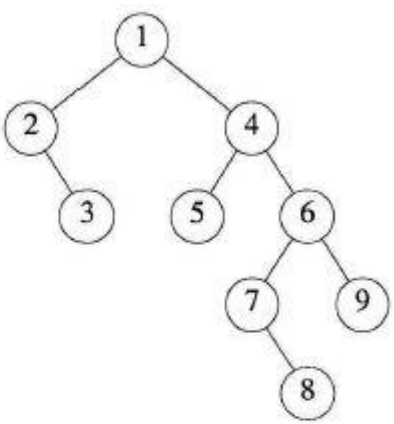
Stack permutations

Suppose we have the preorder sequence *A B CD E F G H I* and the inorder sequence *B* C *A ED G H F I* of the same binary tree. Does such a pair of sequences uniquely define a binary tree?









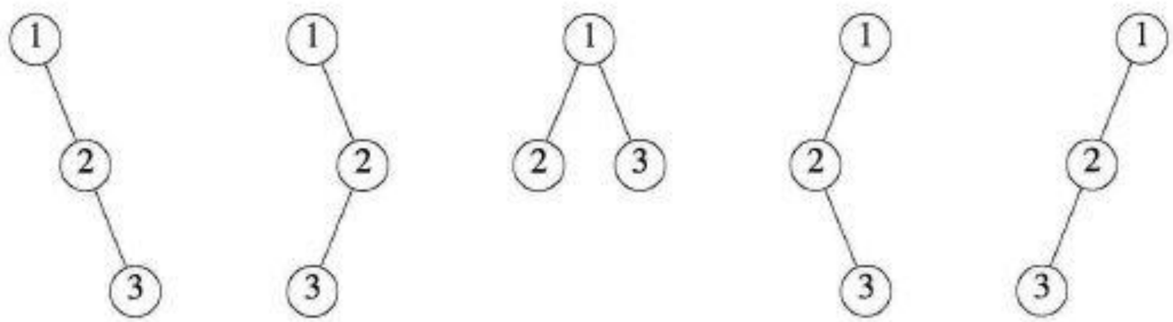
The number of distinct binary trees is equal to the number of distinct inorder permutations obtainable from binary trees having the preorder permutation, 1, 2, · · ·, *n.*

The number of distinct permutations obtainable by passing the numbers 1 through *n* through a stack and deleting in all possible ways is equal to the number of distinct binary trees with *n* nodes

5.11 Counting binary trees

Stack permutations

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).** Obtaining (3, 1, 2) is impossible. Each of these five permutations corresponds to one of the five distinct binary trees with three nodes.



preorder sequence (1, 2, 3)
inorder sequence (1, 2, 3) (1, 3, 2) (2, **1,** 3) (2, 3, 1) (3, 2, **1)**

5.11 Counting binary trees

Matrix multiplication

Another problem that surprisingly has a connection with the previous two involves the product of n matrices. Suppose that we wish to compute the product of n matrices:

$$M_1 * M_2 * \cdots * M_n$$

Since matrix multiplication is associative, we can perform these multiplications in any order. We would like to know how many different ways we can perform these multiplications. For example, if $n = 3$, there are two possibilities:

$$(M_1 * M_2) * M_3$$
$$M_1 * (M_2 * M_3)$$

and if $n = 4$, there are five:

$$((M_1 * M_2) * M_3) * M_4$$
$$(M_1 * (M_2 * M_3)) * M_4$$
$$M_1 * ((M_2 * M_3) * M_4)$$
$$(M_1 * (M_2 * (M_3 * M_4)))$$
$$((M_1 * M_2) * (M_3 * M_4))$$

Thank You !

▌ 노력 없이 이룰 수 있는 것 아무것도 없다.