MODULE 04: TREE(CONTINUATION)

MODULE 04	**TREES(Cont..):** Binary Search trees, Selection Trees, Forests, Representation of Disjoint sets, Counting Binary Trees, **GRAPHS:** The Graph Abstract Data Types, Elementary Graph Operations

4.1 BINARY SEARCH TREES:

Definition: A binary search tree is a binary tree. It 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 keys 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 subtrees are also binary search trees.

4.1.1 Searching A Binary Search Tree

Suppose we wish to search for an element with a key. We begin at the root. If the root is NULL, the search tree contains no elements and the search is unsuccessful. Otherwise, we compare key with the key value in root. If key equals root's key value, then the search terminates successfully. If key is less than roofs key value, then no element in the right subtree can have a key value equal to key. Therefore, we search the left subtree of root. If key is larger than roofs key value, we search the right subtree of root.

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

Recursive search of a binary search tree

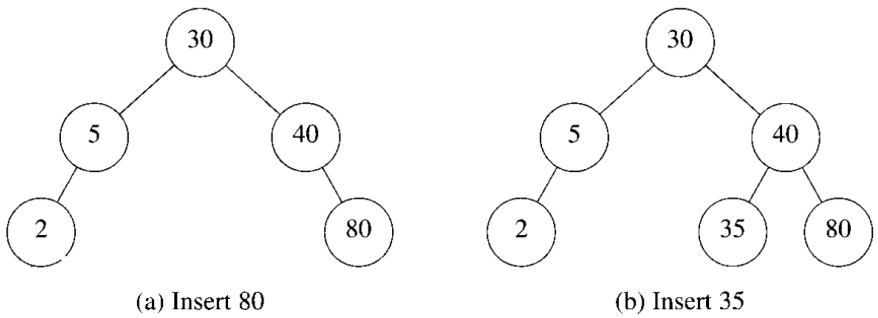
We can easily replace the recursive search function with a comparable iterative one. The function searches! accomplishes this by replacing the recursion with a while loop.

```
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;
}
```

Iterative search of a binary search tree

4.1.2 Inserting into A Binary Search Tree:

To insert a new element, key, we must first verify that the key is different from those of existing elements. To do this we search the tree. If the search is unsuccessful, then we insert the element at the point the search terminated. For instance, to insert an element with key 80 into the tree, we first search the tree for 80. This search terminates unsuccessfully, and the last node examined has value 40. We insert the new element as the right child of this node. The resulting search tree is shown in Figure 5.31(a). Figure 5.31(b) shows the result of inserting the key 35 into the search tree of Figure 5.31(a). This strategy is implemented by insert-node (Program 5.17). This uses the function modified -search which is a slightly modified version of function search 2 (Program 5.16). This function searches the binary search tree *node for the key num. If the tree is empty or if num is present, it returns NULL. Otherwise, it returns a pointer to the last node of the tree that was encountered during the search. The new element is to be inserted as a child of this node.



(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

4.1.3: Deletion from A Binary Search Tree:

Deletion of a leaf node is easy. For example, to delete 35 from the tree of Figure 5.31(b), we set the left child field of its parent to NULL and free the node. This gives us the tree of Figure 5.31(a). The deletion of a nonleaf node that has only a single child is also easy. We erase the node and then place the single child in the place of the erased node. For example, if we delete 40 from the tree of Figure 5.31 (a) we obtain the tree in Figure 5.32.

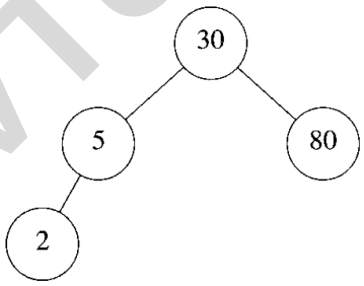


Figure 5.32: Deletion from a binary search tree

When we delete a nonleaf node with two children, we replace the node with either the largest element in its left subtree or the smallest element in its right subtree. Then we proceed by deleting this replacing element from the subtree from which it was taken. For instance, suppose that we wish to delete 60 from the tree of Figure 5.33(a). We may replace 60 with either the largest element (55) in its left subtree or the smallest element (70) in its right subtree. Suppose

we opt to replace it with the largest element in the left subtree. We move the 55 into the root of the subtree. We then make the left child of the node that previously contained the 55 the right child of the node containing 50, and we free the old node containing 55. Figure 5.33(b) shows the final result. One may verify that the largest and smallest elements in a subtree are always in a node of degree zero or one.

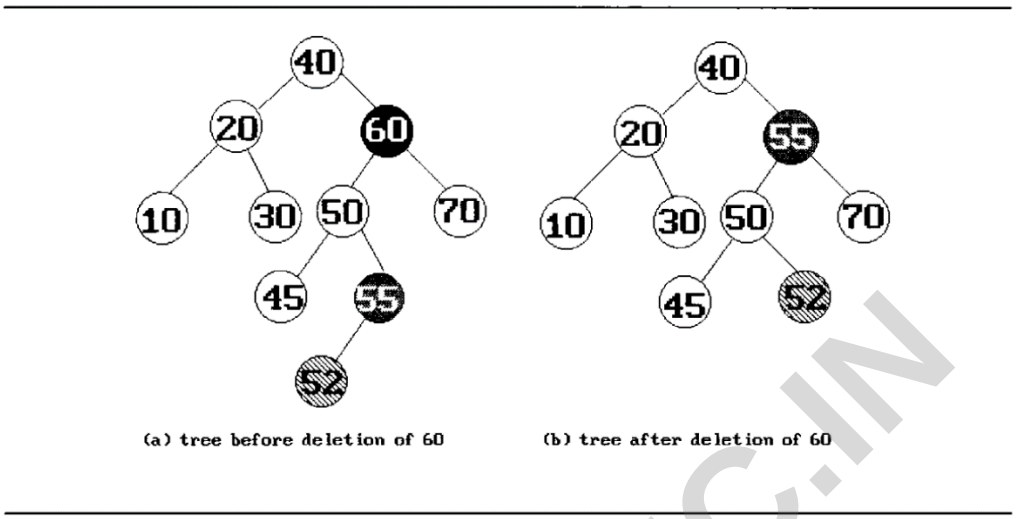


Figure 5.33: Deletion of a node with two children

4.1.3 Joining and splitting Binary search tree:

In a binary search tree, the following additional operations are useful in certain applications.

(a) three Way Join (small, mid, big): This creates a binary search tree consisting of the pairs initially in the binary search trees small and big, as well as the pair mid. It is assumed that each key in small is smaller than mid, key and that each key in big s greater than mid. key. Following the join, both small and big are empty.

(b) two Way Join (small, big): This joins the two binary search trees small and big to obtain a single binary search tree that contains all the pairs originally in smalt and big. It is assumed that all keys of small are smaller than all keys of big and that following the join both small and big are empty.

(c) split (theTree ,k, small, mid, big): The binary search tree the Tree is split into three parts: small is a binary search tree that contains all pairs of the Tree that have key less than k; mid is the pair (if any) in the Tree whose key is k, and big is a binary search tree that contains all pairs of the Tree that have key larger than k. Following the split operation theTree is empty. When the Tree has no pair whose key is k, mid.key is set to -1 (this assumes that-1 is not a valid key for a dictionary pair).

4.1.3 Height of a binary search tree:

Unless care is taken, the height of a binary search tree with n elements can become as large as n. This is the case, for instance, when we use insert-node to insert the keys 1, 2, 3, • • •, n, in that order, into an initially empty binary search tree. However, when insertion and deletions are made at random using the above functions, the height of the binary search tree is O(log2n), on the average.

4.2 SELECTION TREE:

4.2.1 Introduction:

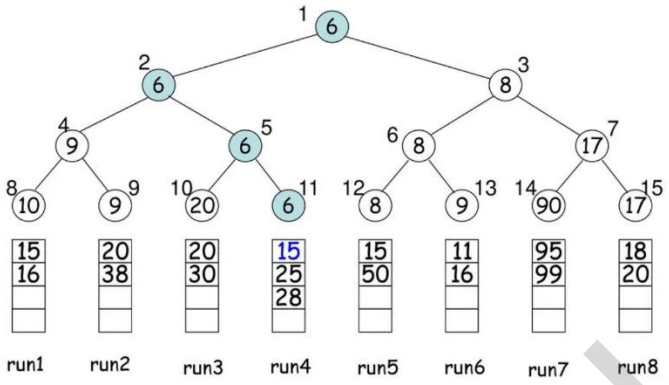
Suppose we have k ordered, called and is in merged into single ordered sequence. Each run consists of some records and is in non-decreasing order of a designated field called the key. Let n be the number of records in all k runs together task can be by the record with the smallest key. The smallest has to be found from k possibilities, and it could be the leading record in any of the k runs. The most direct way to merge k runs is to make k-1 to determine the next record to output. For k > 2, we can achieve a reduction in the number of 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.

4.2.1 Winner Tree:

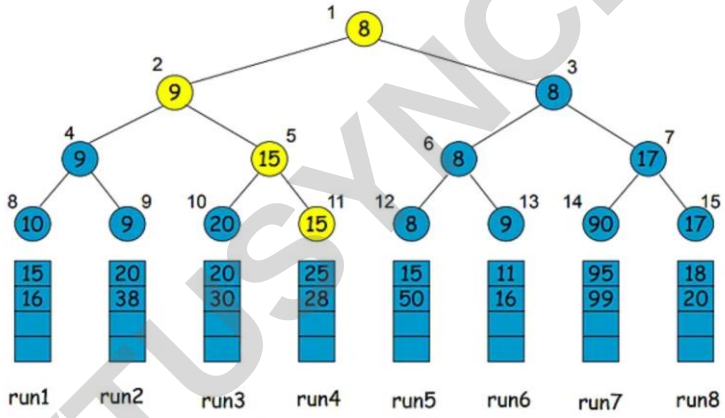
A winner tree is a binary tree in which each node represents the smaller of its two children. Thus, the root node the smallest node in the tree. Figure 5.32 illustrates a winner tree for the case k = 8.

The of this winner tree may be compared to the playing of a tournament in which the winner is the record with the smaller key. Then, each nonleaf node in the tree the winner of a , and the root node the overall winner, or the smallest key. Each leaf node the first record in the corresponding run. Since the records being merged are large, each node will contain only a pointer to the record it represents. Thus, the root node contains a pointer to the first record in run 4 A winner tree may be represented using the sequential allocation scheme for binary trees that results from Lemma 5.4. The number above each node in Figure 5.32 is the address of the node in this sequential. The record pointed to by the root has the smallest key and so may be output. Now, the next record from run 4 enters the winner tree. It has a key value of 15. To the tree, the tournament has to be replayed only along the path from node 11 to the root. Thus, the winner from nodes 10 and 11 is again node 11 (15 <20) The winner from nodes 4 and 5 is node 4 9 <15) The winner from 2 and 3 is node 3 (8 < 9) The new tree is shown in Figure 5.33. The

is played between sibling nodes and the result put in the parent node. Lemma 5.4 may be used to compute the address of sibling and parent nodes efficiently. Each new take place at the next higher level in the tree.



5.32: Winner tree for k=8

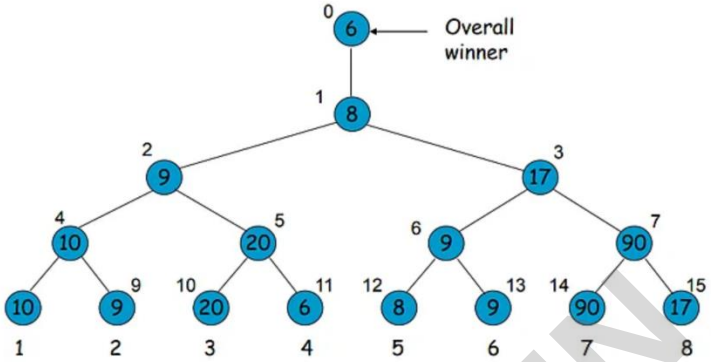


5.33: Winner tree of fig.5.32 after one record has been output and the tree restructured.

4.2.1 Loser Tree:

After the record with the smallest key value is output, the winner tree of Figure 5.32 is to be restructured. Since the record with the smallest key value is in run 4this re- involves inserting the next record from this run into the tree. The next record has key value 15. are played between sibling nodes along the path from node 11 to the root. Since these sibling nodes represent the losers of played earlier, we can simplify the process by placing in each nonleaf node a pointer to the record that loses the 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. Figure 5.34 shows the loser tree that to the winner tree of Figure 5.32. For, each node the key value of a record rather

than a pointer to the record represented. The leaf nodes the first record in each run. An additional node, node 0has been added to represent the overall winner of the tournament. Following the output of the overall winner, the tree is by playing along the path from node 11 to node 1The records with which these tournaments are to be played are readily available from the parent nodes. As a result, sibling nodes along the path from 11 to 1 are not accessed.

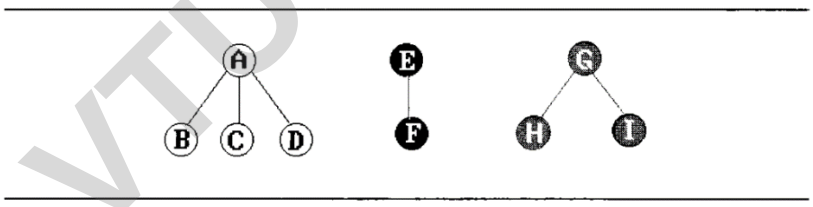


5.34: Loser tree corresponding to winner tree of fig.5.32

4.3: Forest

Definition: A forest is a set of n> 0 disjoint trees. When we remove the root of a

tree we obtain a forest. For example, removing the root of any binary tree produces a forest of two trees.



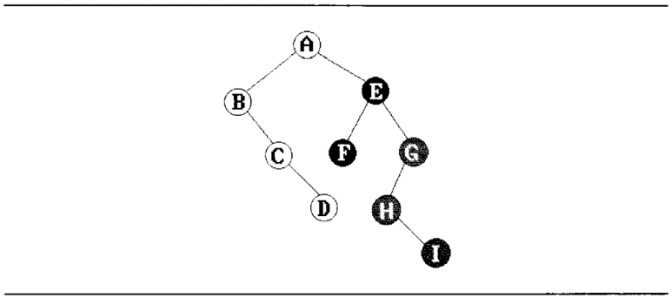
Three-tree forest

4.3.1: 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 $T_{11}, \ldots, T1m$ are the subtrees of root (T_1); and has right subtree $B(T_2, \ldots, T_n)$



Binary tree representation of forest

4.3.2 Forest Traversal

Preorder Traversal:

The preorder traversal of T is equivalent to visiting the nodes of Fin tree preorder. We define this as:

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 trees of F in preorder.

Inorder Traversal:

Inorder traversal of T is equivalent to visiting the nodes of F in tree inorder, which is defined as:

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.
4. Traverse the remaining trees in tree inorder.

Postorder Traversal:

There is no natural analog for the postorder traversal of the corresponding binary tree of a forest. Nevertheless, we can define the postorder traversal of a forest, F, as:

1. If F is empty, then return.
2. Traverse the subtrees of the first tree of F in tree postorder.
3. Traverse the remaining trees of F in tree postorder.
4. Visit the root of the first tree of F.

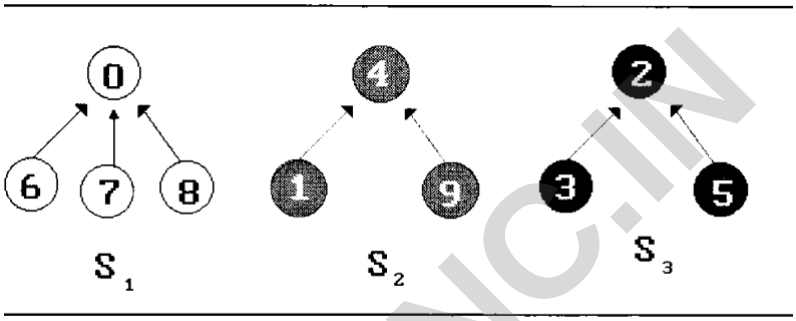
4.4 Representation of Disjoint Sets

4.4.1 Introduction

The use of trees in the representation of sets. assume that the elements of the sets are the numbers 0, 1, 2,. . .n-1. In practice, these numbers might be indices into a symbol table that stores the actual names of the elements.

For example, if we have 10 elements numbered 0 through 9, we may partition them into three disjoint sets, 51 = {0, 6, 7, 8), S2 = {1, 4, 9}, and S3 = {2, 3, 5}.

Figure shows one possible representation for these sets.



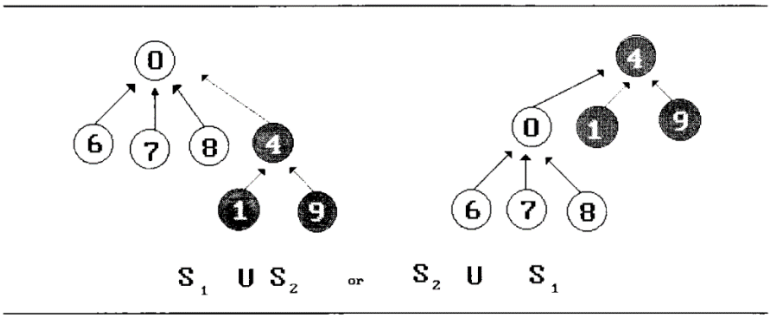
Possible forest representation of sets

The minimal operations that we wish to perform on these sets are:

Disjoint set union and Find(i).

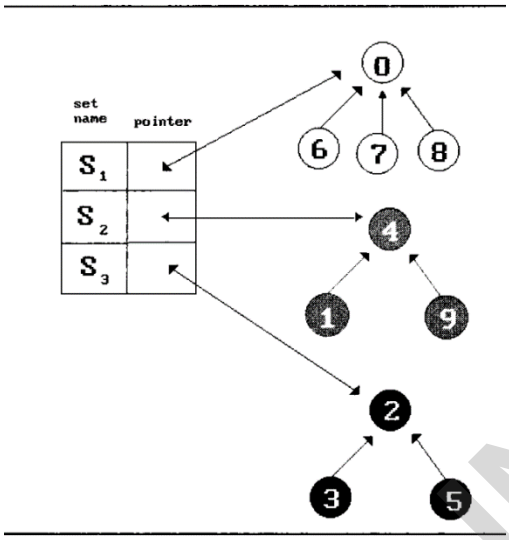
4.4.2 Union and Find operations:

to obtain the union of S_1 and S_2 Since we have linked the nodes from children to parent, we simply make one of the trees a subtree of the other.



Possible representation of $S_1 \cup S_2$

To implement the set union operation, we simply set the parent field of one of the roots to the other root. We can accomplish this easily if, with each set name, we keep a pointer to the root of the tree representing that set.



Data representation of S_1, S_2, and S_3

rather than using the set name S_1 we refer to this set as 0. The transition to set names is easy. We assume that a table, name [ ], holds the set names. If i is an element in a tree with root 7, and j has a pointer to entry k in the set name table, then the set name is just name[k].

```
int find1(int i)
{
   for(; parent[i] >= 0; i = parent[i])
      ;
   return i;
}
void union1(int i, int j)
{
   parent[i] = j;
}
```

Initial attempt at union-find functions

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 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;
   }
}
```

Union function

Fig. Union function using weighting rule

Definition [collapsing rule] : If j is a node on the path from i to its root and parent[i] != root(i), then set parent [j] to root(i).

```
int find2(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;
}
```

Fig: Collapsing rule

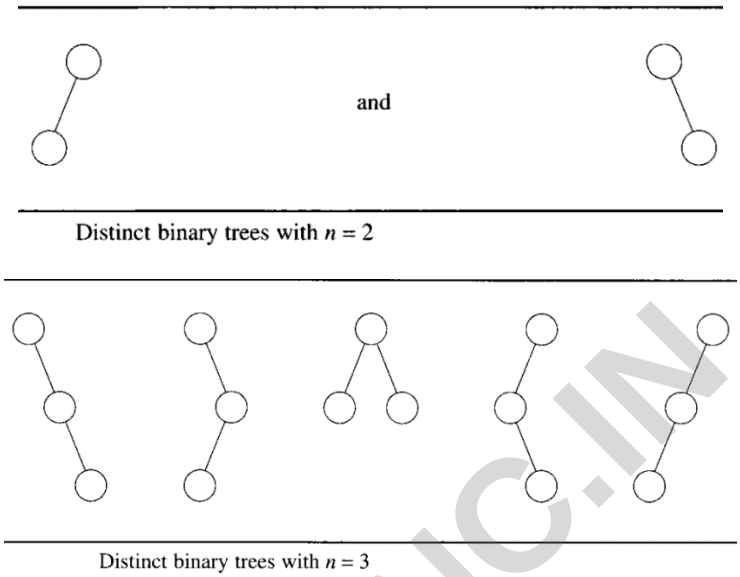
4.4.3 Application to equivalence classes

The equivalence classes to be generated may be regarded as set. These sets are disjoint since no polygon can be in two equivalence classes. Initially, all n polygons are in an equivalence class of their own; thus parent{i} = -1, 0 < =i < n. If an equivalence pair, i = j, is to be processed, we must first determine the sets containing i and j. If they are different, then we replace the two sets by their union. If the two sets are the same, then we do nothing since the relation i = j is redundant: I and j are already in the same equivalence class. To process each equivalence pair, we need to perform two finds and at most one union.

4.5 Counting Binary trees

4.5.1: Distinct Binary tree

if n = 0 or n = 1, there is only one binary tree. If n = 2, then there are two distinct trees and if n = 3.

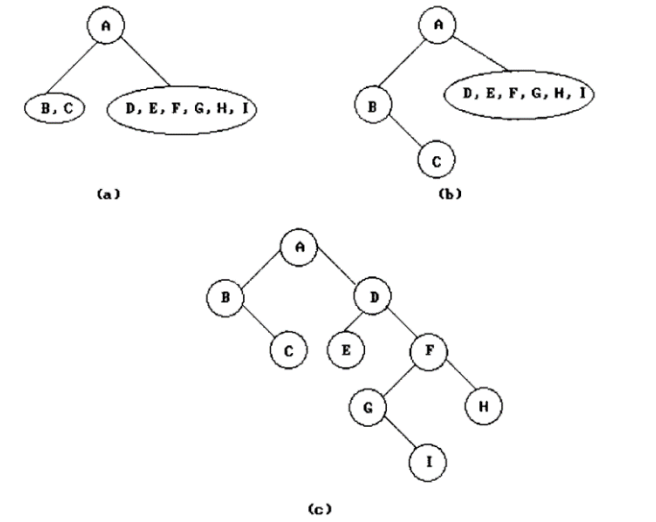


Distinct binary trees with $n = 2$

Distinct binary trees with $n = 3$

4.5.2 Stack permutations:

Suppose we have the preorder sequence: ABCDEFGHI and the inorder sequence: BCAEDGHFI of the binary tree. To construct the binary tree from these sequences, we look at the first letter in the preorder sequence, A. This letter must be the root of the tree by definition of the preorder traversal (VLR.}. We also know by definition of the inorder traversal {LVR} that all nodes preceding A in the inorder sequence (B Q are in the left subtree, while the remaining nodes {ED GHFI) are in the right subtree. Figure 5.49(a) is our first approximation to the correct tree. Moving right in the preorder sequence, we find B as the next root. Since no node precedes B in the inorder sequence, B has an empty left subtree, which means that C is in its right subtree. Figure 5.49(b) is the next approximation. Continuing in this way, we arrive at the binary tree of Figure 5.49(c). By formalizing this argument (see the exercises for this section), we can verify that every binary tree has a unique pair of preorder inorder sequences.



Constructing a binary tree from its inorder and preorder sequences

4.5.3 Matrix multiplication

Suppose that we wish to compute the product of n matrices: $M_1 * M_2 * \bullet \bullet \bullet * M_n$. 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:

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

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

The number of distinct ways to obtain M_{1i} and M_{1+I} are b_i and b_{n-i}, respectively. Therefore, letting

$b_1 = 1$, we have

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

Then we see that b_n is the sum of all the possible binary trees formed in the following way: a root and two subtrees with b_i and b_{n-i-1} nodes, for $0 < i < n$. This explanation says that

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

Therefore, 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.

4.5.4 Number of Distinct binary trees

To obtain the number of distinct binary trees with n nodes, we must solve the recurrence

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

To begin we let:

$$B(x) = \sum_{i \geq 0} b_i \, x^i$$

which is the generating function for the number of binary trees. Next observe that by the recurrence relation we get the identity:

$$xB^2(x) = B(x) - 1$$

Using the formula to solve quadratics and the recurrence that B (0) = b_0= 1 we get:

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

We can use the binomial theorem to expand (1 - 4x) $^{1/2}$ to obtain:

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

$$= \sum_{m \geq 0} \binom{1/2}{m+1} (-1)^m \, 2^{2m+1} \, x^m$$

we see that b_n which is the coefficient of x" in B(x), is:

$$\binom{1/2}{n+1} (-1)^n \, 2^{2n+1}$$

Some simplification yields the more compact form

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

which is approximately

$$b_n = O(4^n/n^{3/2})$$

5. GRAPHS

5.1 The graph Abstract Data Type

5.1.1 Introduction

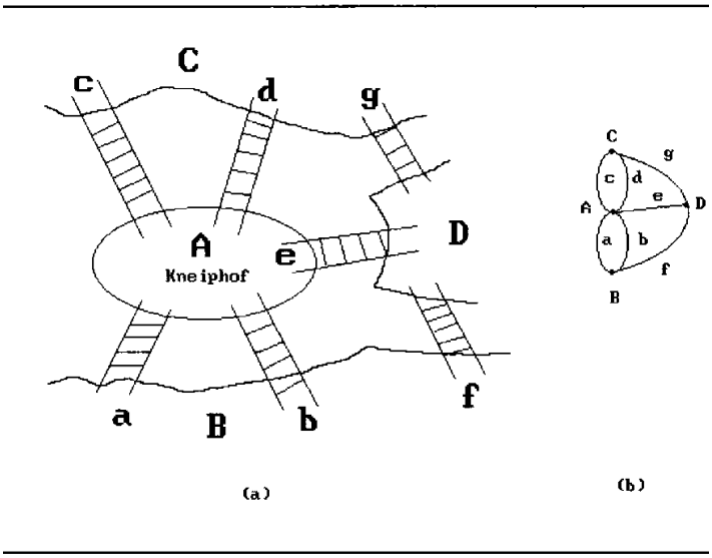
In Koenigsberg, the Pregal river flows around the island of Kneiphof. There are four land areas, labelled A through D in Figure 6.1, that have this river on their border. Seven bridges, labelled a through g, connect the land areas. The Koenigsberg bridge problem is as follows: Starting at some land area, is it possible to return to our starting location after walking across each of the bridges exactly once?

A possible walk might be:

- start from land area B
- walk across bridge a to island A
- take bridge e to area D
- take bridge g to C
- take bridge d to A
- take bridge b to B
- take bridge f to D

This walk does not cross all bridges exactly once, nor does it return to the starting land area B.

Euler solved the problem by using a graph (actually a multigraph) in which the land areas are vertices and the bridges are edges. His solution is not only elegant, it applies to all graphs.



The bridges of Koenigsberg

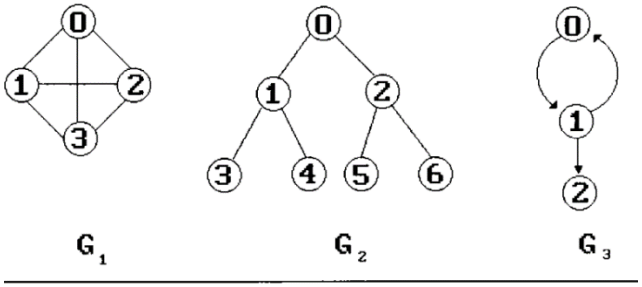
Euler defined the degree of a vertex as the number of edges incident on it. He then showed that there is a walk starting at any vertex, going through each edge exactly once, and terminating at the starting vertex iff the degree of each vertex is even. We now call a walk that does this an Eulerian walk. Since this first application, graphs have been used in a wide variety of applications, including analysis of electrical circuits, finding shortest routes, project planning, and the identification of chemical compounds. Indeed graphs may be the most widely used of all mathematical structures.

5.1.2 Definitions

A graph, G, consists of two sets: a finite, nonempty set of vertices, and a finite, possibly empty set of edges. V(G) and E(G) represent the sets of vertices and edges of G, respectively. Alternately, we may write G = (V, E) to represent a graph.

An undirected graph is one in which the pair of vertices representing any edge is unordered. For example, the pairs (v_0, v_1) and (v_0, v_1) represent the same edge.

A directed graph is one in which we represent each edge as a directed pair of vertices. For example, the pair $<v_0, v_1>$ represents an edge in which v_0 is the tail and v_1 is the head. Therefore, $<v_0, v_1>$ and $<v_1, v_0>$ represent two different edges in a directed graph.



Three sample graphs

A *complete graph* is a graph that has the maximum number of edges. For an undirected graph with n vertices, the maximum number of edges is the number of distinct, unordered pairs, (v_i, v_j), $i \neq j$. This number is:

$$n(n-1)/2$$

A *subgraph* of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.

A *path* from vertex v_p to vertex v_q in a graph, G, is a sequence of vertices, $v_p, v_{i_1}, v_{i_2}, \cdots, v_{i_n}, v_q$ such that $(v_p, v_{i_1}), (v_{i_1}, v_{i_2}), \cdots, (v_{i_n}, v_q)$ are edges in an undirected graph. If G' is a directed graph, then the path consists of $<v_p, v_{i_1}>, <v_{i_1}, v_{i_2}>, \cdots, <v_{i_n}, v_q>$. The *length* of a path is the number of edges on it.

A *cycle* is a simple path in which the first and the last vertices are the same. For example, 0, 1, 2, 0 is a cycle in G_1, and 0, 1, 0 is a cycle in G_3. For directed graphs, we usually add the prefix "directed" to the terms cycle and path.

A *connected component*, or simply a *component*, of an undirected graph is a maximal connected subgraph. For example, G_4 has two components, H_1 and H_2. A *tree* is a graph that is connected and acyclic (it has no cycles).

The degree of a vertex is the number of edges incident to that vertex.

structure *Graph* is
objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices.
functions:
for all *graph* ∈ *Graph*, *v*, v_1, and v_2 ∈ *Vertices*

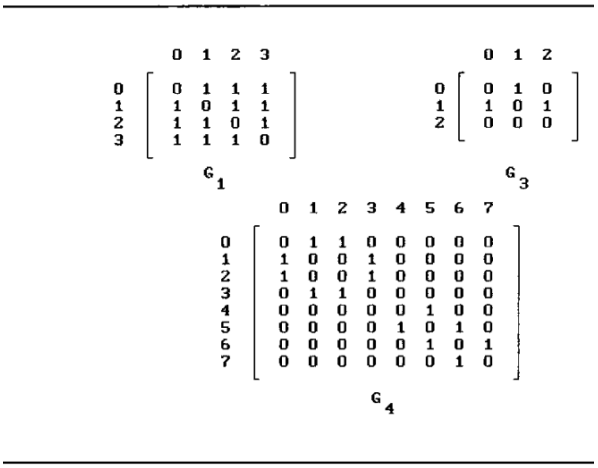
Graph Create()	::=	**return** an empty graph.
Graph InsertVertex(*graph*, *v*)	::=	**return** a graph with *v* inserted. *v* has no incident edges.
Graph InsertEdge(*graph*, v_1, v_2)	::=	**return** a graph with a new edge between v_1 and v_2.
Graph DeleteVertex(*graph*, *v*)	::=	**return** a graph in which *v* and all edges incident to it are removed.
Graph DeleteEdge(*graph*, v_1, v_2)	::=	**return** a graph in which the edge (v_1, v_2) is removed. Leave the incident nodes in the graph.
Boolean IsEmpty(*graph*)	::=	**if** (*graph* == empty graph) **return** *TRUE* **else return** *FALSE*.
List Adjacent(*graph*, *v*)	::=	**return** a list of all vertices that are adjacent to *v*.

Fig: Abstract data type graph

5.1.3 Graph representations

Adjacency Matrix:

Let G = (V, E) be a graph with n vertices, n>= 1. The adjacency matrix of G is a two-dimensional n x n array, say adj-mat. If the edge (Vi, Vj) (<V_i,V_j>for a digraph) is in E(G), adj-mat[i][j] = 1. If there is no such edge in E(G), adj~mat[i][j] = 0. The adjacency matrices for graphs Gi, G3, and G4 are shown in Figure. The adjacency matrix for an undirected graph is symmetric since the edge (v_i, v_j) is in E(G) iff the edge (v_i, V_j) is also in E(G). In contrast, the adjacency matrix for a digraph need not be symmetric. (This is true of G3.)) For undirected graphs, we can save space by storing only the upper or lower triangle of the matrix.



Adjacency matrices for G_1, G_3, and G_4

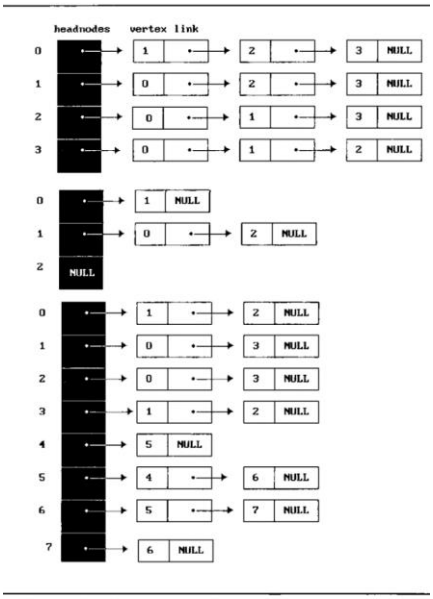
Adjacency Lists:

For any given list, i, the nodes in the list contain the vertices that are adjacent from vertex i. Figure 6.8 shows the adjacency lists for G_1, G_3, and G_4.

```
#define MAX_VERTICES 50 /*maximum number of vertices*/
typedef struct node *node_pointer;
typedef struct node {
        int vertex;
        struct node *link;
        };
node_pointer graph[MAX_VERTICES];
int n = 0; /* vertices currently in use */
```

C declarations for the adjacency list representation

sequential representation for the graph G4 of Figure 6.5. We can determine the degree of any vertex in an undirected graph by simply counting the number of nodes in its adjacency list. This also gives us the number of edges incident on the vertex.



Adjacency lists for G_1, G_3, and G_4

Adjacency Multilists:

In the adjacency list representation of an undirected graph, we represent each edge, (v_i, V_j), by two entries. One entry is on the list for v_i, and the other is on the list for V_j.

For each edge there is exactly one node, but this node is on the adjacency list for each of the two vertices it is incident to the new node structure.

marked	vertex1	vertex2	path1	path2

Node structure for adjacency multilists

```
typedef struct edge *edge_pointer;
typedef struct edge {
        short int marked;
        int vertex1;
        int vertex2;
        edge_pointer path1;
        edge_pointer path2;
        };
edge_pointer graph[MAX_VERTICES];
```

C declaration for multilists

Weighted Edges:

The edges of a graph are assigned weights. These weights may represent the distance from one vertex to another or the cost of going from one vertex to an adjacent vertex. A graph with weighted edges is called a network.

5.2 Elementary Graph Operations

Given an undirected graph, G = (V, E), and a vertex, v, in V(G) we wish to visit all vertices in G that are reachable from v, that is, ail vertices that are connected to v. There are two ways of doing this: depth first search and breadth first search.

5.2.1 Depth First Search

Depth first search is similar to a preorder tree traversal. We begin the search by visiting the start vertex, v. visiting consists of printing the node's vertex field. Next, we select an unvisited vertex, w, from v's adjacency list and carry out a depth first search on w. Eventually our search reaches a vertex, M, that has no unvisited vertices on its adjacency list. At this point, we remove a vertex from the stack and continue processing its adjacency list. Previously visited vertices are discarded; unvisited vertices are visited and placed on the stack. The search terminates when the stack is empty.

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

```
void dfs(int v)
{
/* depth first search of a graph beginning with vertex v.*/
   node_pointer w;
   visited[v] = TRUE;
   printf("%5d",v);
   for (w = graph[v]; w; w = w->link)
      if (!visited[w->vertex])
         dfs(w->vertex);
}
```

Depth first search

5.2.2 Breadth First Search

breadth first search resembles a level order tree traversal. Breadth first search starts at vertex v and marks it as visited. It then visits each of the vertices on v's adjacency list. When we have visited all the vertices on v's adjacency list, we visit all the unvisited vertices that are adjacent

to the first vertex on v's adjacency list. To implement this scheme, as we visit each vertex, we place the vertex in a queue. When we have exhausted an adjacency list, we remove a vertex from the queue and proceed by examining each of the vertices on its adjacency list. Unvisited vertices are visited and then placed on the queue; visited vertices are ignored. We have finished the search when the queue is empty.

The queue definition and the function prototypes used by BFS are:

```
typedef struct queue *queue_pointer;
typedef struct queue {
        int vertex;

        queue_pointer link;
        };
void addq(queue_pointer *, queue_pointer *, int);
int deleteq(queue_pointer *);
```

```
void bfs(int v)
{
/* breadth first traversal of a graph, starting with node v
the global array visited is initialized to 0, the queue
operations are similar to those described in
Chapter 4. */
   node_pointer w;
   queue_pointer front,rear;
   front = rear = NULL; /* initialize queue */
   printf("%5d",v);
   visited[v] = TRUE;
   addq(&front, &rear, v);
   while (front) {
      v = deleteq(&front);
      for (w = graph[v]; w; w = w->link)
         if (!visited[w->vertex]) {
            printf("%5d", w->vertex);
            addq(&front,&rear,w->vertex);
            visited[w->vertex] = TRUE;
         }
      }
}
```

Breadth first search of a graph

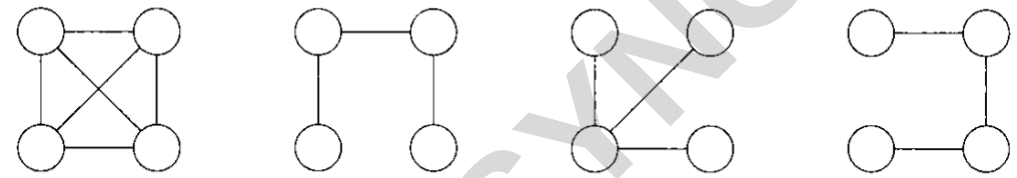
5.2.3 Connected Components:

```
void connected(void)
{
/* determine the connected components of a graph */
int i;
for (i = 0; i < n; i++)
   if(!visited[i]) {
      dfs(i);
      printf("\n");
   }
}
```

Connected components

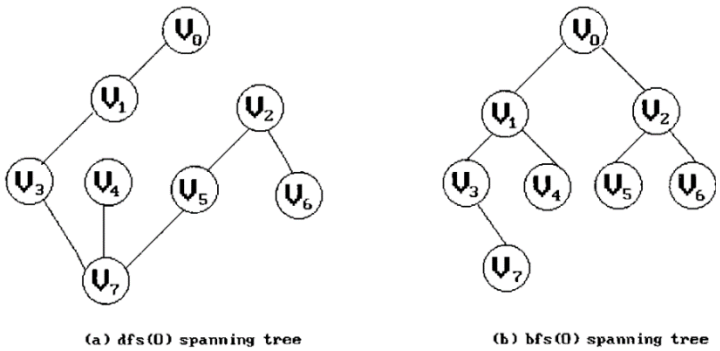
5.2.4 Spanning Trees

A spanning tree is any tree that consists solely of edges in G and that includes all the vertices in G.



A complete graph and three of its spanning trees

we may use either dfs or bfs to create a spanning tree. When dfs is used, the resulting spanning tree is known as a depth first spanning tree. When bfs is used, the resulting spanning tree is called a breadth first spanning tree.



dfs and *bfs* spanning trees for graph of Figure 6.19

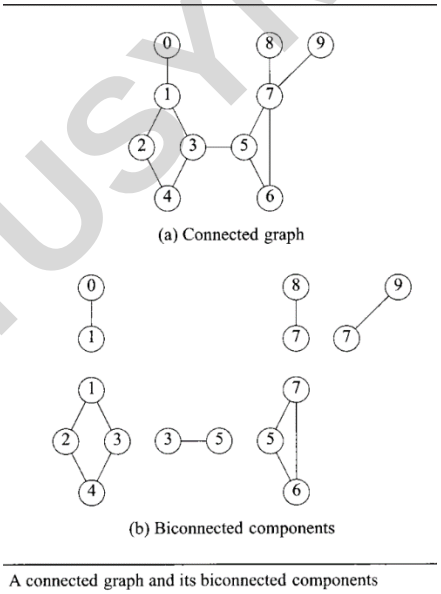
A spanning tree is a minimal subgraph G' of G such that V (G) = V(G) and G' is connected. We define a minimal subgraph as one with the fewest number of edges. Any connected graph with n vertices must have at least n - 1 edges, and all connected graphs with n - 1 edges are trees. Therefore, we conclude that a spanning tree has n - 1 edges.

5.2.5 Biconnected Components

An articulation point is a vertex v of G such that the deletion of v, together with all edges incident on v, produces a graph, G', that has at least two connected components. For example, the connected graph of Figure has four articulation points, vertices 1, 3, 5, and 7.

A biconnected component of a connected undirected graph is a maximal biconnected subgraph, H, of G. By maximal, we mean that G contains no other subgraph that is both biconnected and properly contains H. F

We can find the biconnected components of a connected undirected graph, G, by using any depth first spanning tree of G. F



A connected graph and its biconnected components

```
#define MIN2(x,y) ((x) < (y) ? (x) : (y))
short int dfn[MAX_VERTICES];
short int low[MAX_VERTICES];
int num;
```

The global declaration

```
void init(void)
{
   int i;
   for (i = 0; i < n; i++) {
      visited[i] = FALSE;
      dfn[i] = low[i] = -1;
   }
   num = 0;
}
```

Initialization of *dfn* and *low*

```
void dfnlow(int u, int v)
{
/* compute dfn and low while performing a dfs search
beginning at vertex u, v is the parent of u (if any) */
   node_pointer ptr;
   int w;
   dfn[u] = low[u] = num++;
   for (ptr = graph[u]; ptr; ptr = ptr->link) {
      w = ptr->vertex;
      if (dfn[w] < 0) { /* w is an unvisited vertex */
         dfnlow(w,u);
         low[u] = MIN2(low[u],low[w]);
      }
      else if (w != v)
         low[u] = MIN2(low[u],dfn[w]);
   }
}
```

Determining *dfn* and *low*

****