MODULE 4: TREES

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. □

Searching A Binary Search Tree

Since the definition of a binary search tree is recursive, it is easiest to describe a recursive search method. 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 *root*'s 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 *root*'s key value, we search the right subtree of *root*. The function *search* (Program 5.15) recursively searches the subtrees.

```
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

We can easily replace the recursive search function with a comparable iterative one. The function *search* 2 (Program 5.16) 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;
}
```

Program 5.16: Iterative search of a binary search tree

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 of Figure 5.30(b), 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.

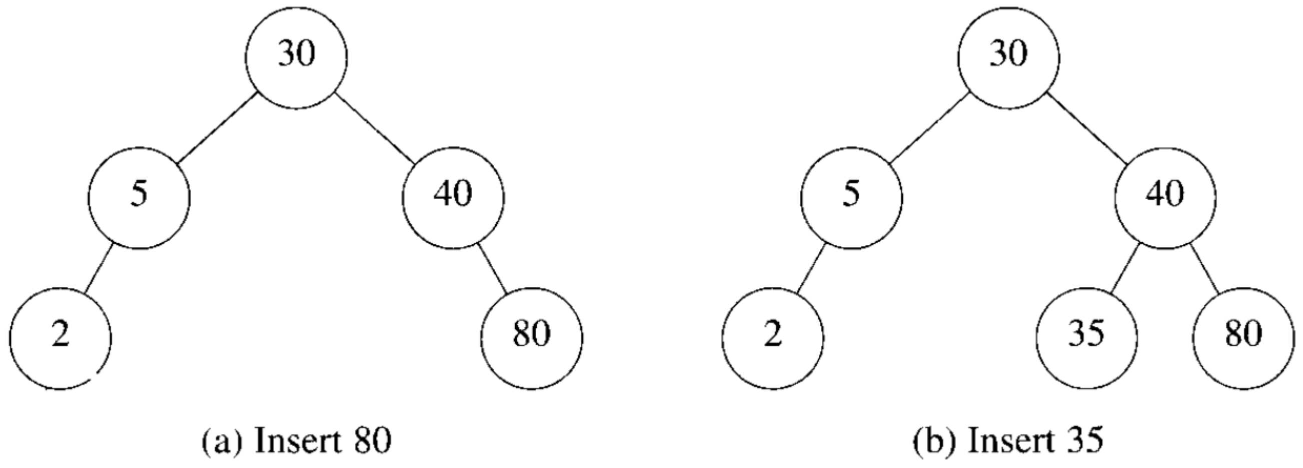


Figure 5.31: Inserting into a binary search tree

```
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

Analysis of *insert-node:* The time required to search the tree for *num* is O(/2) where *h* is its height. The remainder of the algorithm takes 0(1) time. So, the overall time needed by *insert-node* is O(/?).

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.

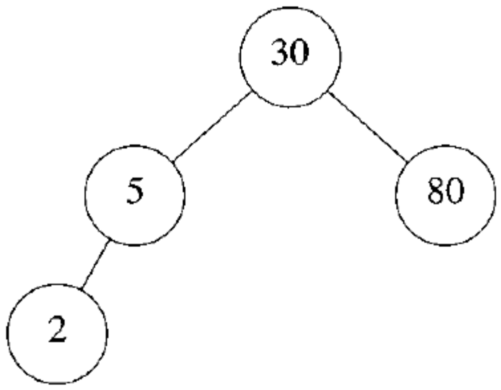


Figure 5.32: Deletion from a binary search tree

When we delete a non-leaf 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.

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.

SELECTION TREES

A *selection tree* is a binary tree where each node represents the smaller of its two children. Thus, the root node represents the smallest node in the tree.

If *k* ordered sequences that are to be merged into a single ordered sequence. Each sequence consists of some number of records and is in non-decreasing order of a designated field called the *key.* An ordered sequence is called a *run.* Let *n* be the number of records in the *k* runs together. The merging task can be accomplished by repeatedly outputting 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 would be to make *k* - 1 comparisons to determine the next record to output. For *k > 2,* we can achieve a reduction in the number of comparisons needed to find the next smallest element by using the idea of a selection tree.

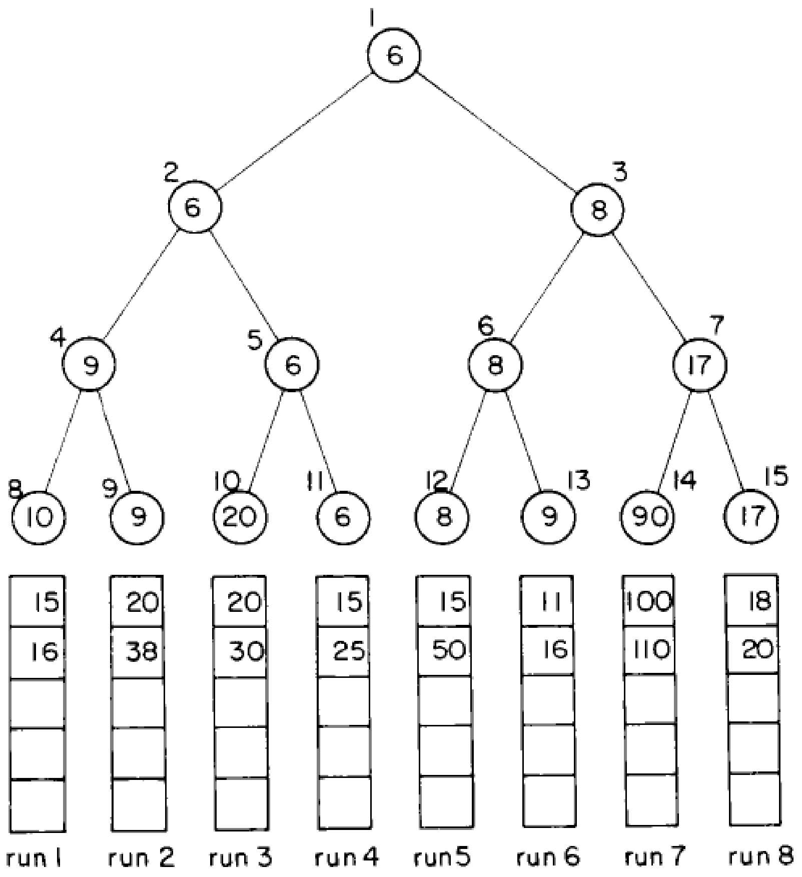


Figure 5.34: Selection tree for *k* =8 showing the first three keys in each of the eight runs

The construction of this selection 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 represents the winner of a tournament and the root node represents the overall winner or the smallest key. A leaf node here 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. Thus, the root node contains a pointer to the first record in run 4.

The number above each node in Figure 5.34 represents the address of the node in this sequential representation. 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 selection tree. It has a key value of 15. To restructure 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 5.35. The tournament is played between sibling nodes and the result put in the parent node.

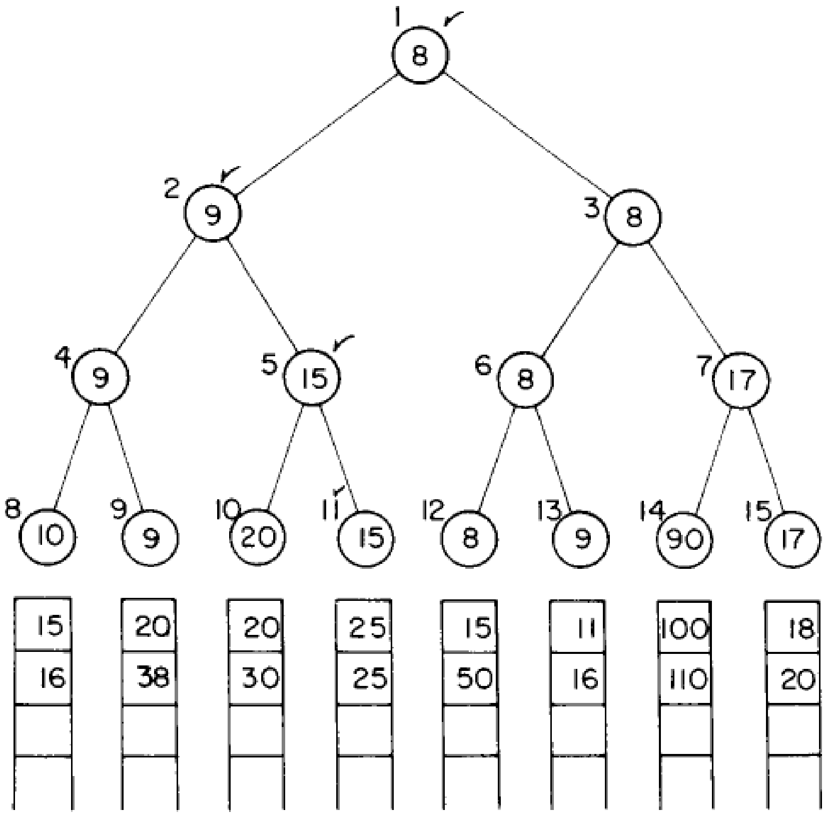


Figure 5.35: Selection tree of Figure 5.34 after one record has been output and the tree restructured (nodes that were changed are ticked)

A tournament tree in which each nonleaf node retains a pointer to the loser is called a *tree of losers*. Figure 5.36 shows the tree of losers corresponding to the selection tree of Figure 5.34. For convenience, each node contains the key value of a record rather than a pointer to the record represented. The leaf nodes represent the first record in each run. An additional node, node 0, has been added to represent the overall winner of the tournament. Following the output of the overall winner, the tree is restructured by playing tournaments along the path from node 11 to node 1. The records with which these tournaments are to be played are readily available from the parent nodes.

FORESTS

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

The concept of a forest is very close to that of a tree because if 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. In this section, we briefly consider several forest operations, including transforming a forest into a binary tree and forest traversals. In the next section, we use forests to represent disjoint sets.

Transforming A Forest into A Binary Tree

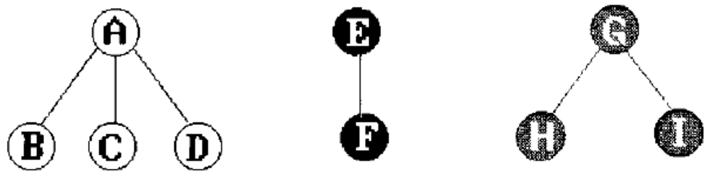


Figure 5.37: Forest with three trees

A forest of three trees as illustrated in Figure 5.37. To transform this forest into a single binary tree, we first obtain the binary tree representation for each of the trees in the forest. We then link all the binary trees together through the sibling field of the root node. Applying this transformation to the forest of Figure 5.37 gives us the tree of Figure 5.38.

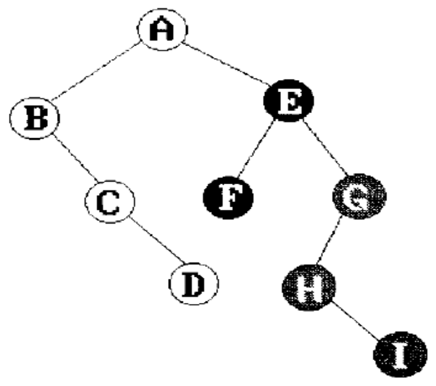


Figure 5.38: Binary tree representation of Figure 5.37

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 $T_{11}, T_{12}, \cdots, T_{1m}$ are the subtrees of root (T_1); and has right subtree $B(T_2, \cdots, T_n)$ □

Forest Traversals

Preorder, inorder, and postorder traversals of the corresponding binary tree T of a forest F have a natural correspondence with traversals of F. The preorder traversal of T is equivalent to visiting the nodes of F in 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 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.

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.

Representation of Disjoint sets

The elements of the sets are the numbers 0, 1, 2,… n-1, numbers might be indices into a symbol table that stores the actual names of the elements. We also assume that the sets being represented are pairwise disjoint, that is, if S_i and S_j are two sets and $i \neq j$, then there is no element that is in both S_i and S_j. For example, if we have 10 elements numbered 0 through 9, we may partition them into three disjoint sets, $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$, and $S_3 = \{2, 3, 5\}$. Figure 5.39 shows one possible representation for these sets. Notice that for each set we have linked the nodes from the children to the parent, rather than our usual method of linking from the parent to the children. The reason for this change in linkage will become apparent when we discuss the implementation of set operations.

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

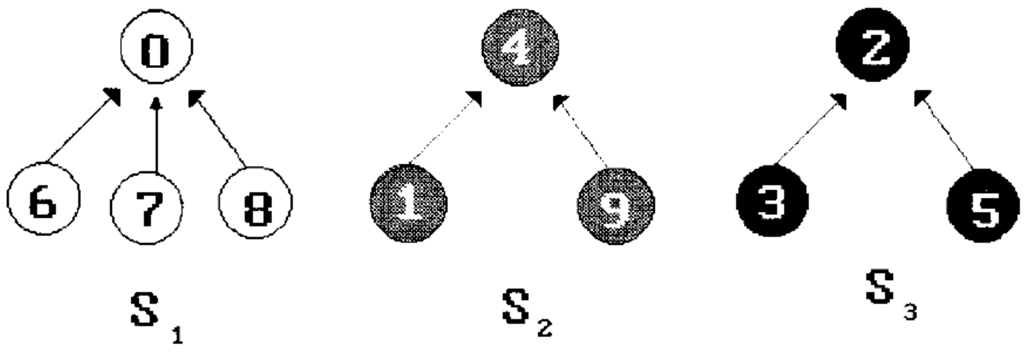


Figure 5.39: Possible forest representation of sets

(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.

Union and Find Operations

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. If, in addition, each root has a pointer to the set name, we can find which set an element is in by following the parent links to the root of its tree and then returning the pointer to the set name. Figure 5.41 shows this representation of Si, S2, and S3.

Let us consider the union operation first. Suppose that we wish 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. $S_1 \cup S_2$ could have either of the representations of Figure 5.40.

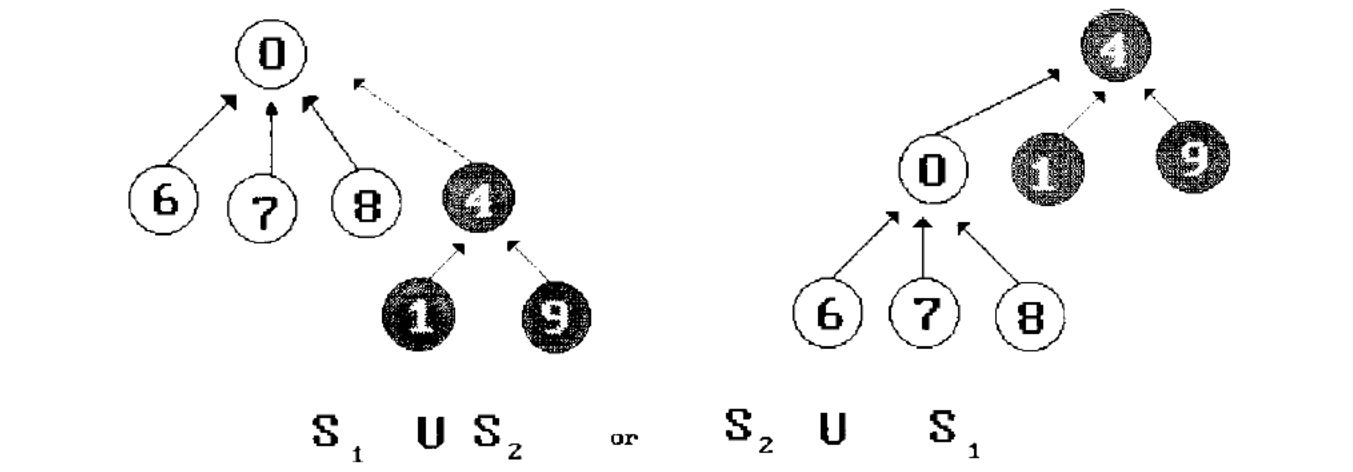


Figure 5.40: Possible representation of $S_1 \cup S_2$

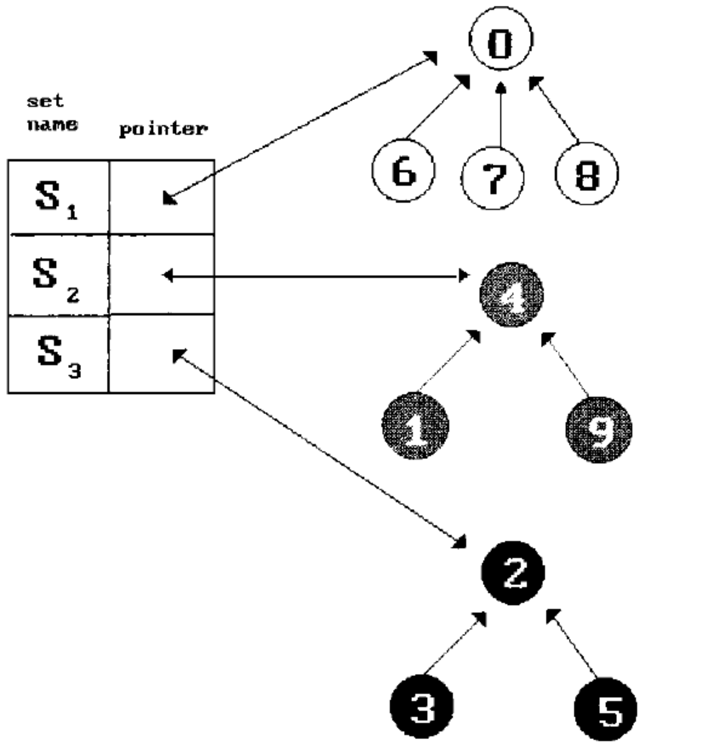


Figure 5.41: Data representation of S_1, S_2, and S_3

To simplify the discussion of the union and find algorithms, we will ignore the set names and identify the sets by the roots of the trees representing them. For example, 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 j, and j has a pointer to entry k in the set name table, then the set name is just *name*[k].

Since the nodes in the trees are numbered 0 through $n - 1$ we can use the node's number as an index. This means that each node needs only one field, the index of its parent, to link to its parent. Thus, the only data structure that we need is an array, *int parent*[*MAX_ELEMENTS*], where *MAX_ELEMENTS* is the maximum number of elements. Figure 5.42 shows this representation of the sets, S_1, S_2, and S_3. Notice that root nodes have a parent of −1.

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

We can now implement *find*(i) by simply following the indices starting at i and continuing until we reach a negative parent index. For example, *find*(5), starts at 5, and then moves to 5's parent, 2. Since this node has a negative index we have reached the root. The operation *union*(i,j) is equally simple. We pass in two trees with roots i and j. Assuming that we adopt the convention that the first tree becomes a subtree of the second, the statement *parent* [i] = j accomplishes the union. Program 5.18 implements the union and find operations as just discussed.

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

Program 5.18: Initial attempt at union-find functions

Analysis of *union*1 and *find*1: Although *union*1 and *find*1 are easy to implement, their performance characteristics are not very good. For instance, if we start with p elements, each in a set of its own, that is, $S_i = \{i\}$, $0 \le i < p$, then the initial configuration is a forest with p nodes and $parent[i] = -1$, $0 \le i < p$. Now let us process the following sequence of union-find operations:

$$union(0, 1), find(0)$$
$$union(1, 2), find(0)$$
$$\vdots$$
$$union(n-2, n-1), find(0)$$

This sequence produces the degenerate tree of Figure 5.43. Since the time taken for a union is constant, we can process all the $n-1$ unions in time $O(n)$. However, for each *find*, we must follow a chain of parent links from 0 to the root. If the element is at level i, then the time required to find its root is $O(i)$. Hence, the total time needed to process the $n-1$ finds is:

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

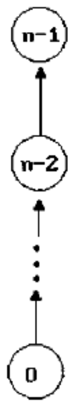


Figure 5.43: Degenerate tree

By avoiding the creation of degenerate trees, we can attain far more efficient implementations of the union and find operations. We accomplish this by adopting the following *Weighting rule* for *union(i, j)*.

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*. □

When we use this rule on the sequence of set unions described above, we obtain the trees of Figure 5.44. To implement the weighting rule, we need to know how many nodes there are in every tree. To do this easily, we maintain a count field in the root of every tree. If *i* is a root node, then *count*[*i*] equals the number of nodes in that tree. Since all nodes but the roots of trees have a nonnegative number in the parent field, we can maintain the count in the parent field of the roots as a negative number. When we incorporate the weighting rule, the union operation takes the form given in *union*2 (Program 5.19). Remember that the arguments passed into *union*2 must be roots of trees.

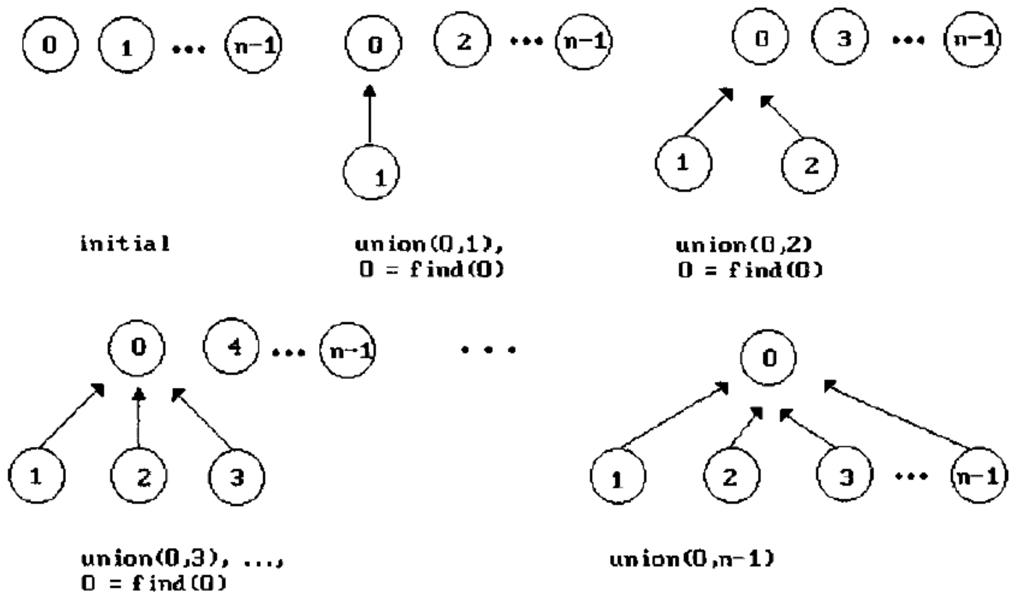


Figure 5.44: Trees obtained using the weighting rule

```
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

Example 5.1: Consider the behavior of *union*2 on the following sequence of unions starting from the initial configuration of $parent[i] = -count[i] = -1, 0 \leq i < n = 2^3$:

$$\begin{array}{llll} union(0, 1) & union(2, 3) & union(4, 5) & union(6, 7) \\ union(0, 2) & union(4, 6) & union(0, 4) & \end{array}$$

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 5.45 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. □

As a result of Lemma 5.4, the time to process a find in an n element tree is $O(\log_2 n)$. If we must process an intermixed sequence of $n - 1$ union and m find operations, then the time becomes $O(n + m \log_2 n)$. Surprisingly, further improvement is possible if we add a collapsing rule to the find operation.

Definition [Collapsing rule]: If j is a node on the path from i to its root then make j a child of the root. □

Program 5.20 incorporates the collapsing rule into the find operation. The new function roughly doubles the time for an individual find. However, it reduces the worst case time over a sequence of finds.

Example 5.2: Consider the tree created by *union*2 on the sequence of unions of Example 5.1. Now process the following 8 finds:

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

Using the old version of *find*, *find(7)* requires going up three parent link fields for a total of 24 moves to process all eight finds. In the new version of *find*, the first *find(7)* requires going up three links and then resetting two links. Each of the remaining seven finds requires going up only one link field. The total cost is now only 13 moves (note that even though only two links need to be changed, function *find*2 sets three including the one from node four). □

The worst case behavior of the union-find algorithms while processing a sequence of unions and finds is stated in Lemma 5.5. Before stating this lemma, let us introduce a very slowly growing function, $\alpha(m, n)$, which is related to a functional inverse of Ackermann's function $A(p,q)$. We have the following definition for $\alpha(m, n)$:

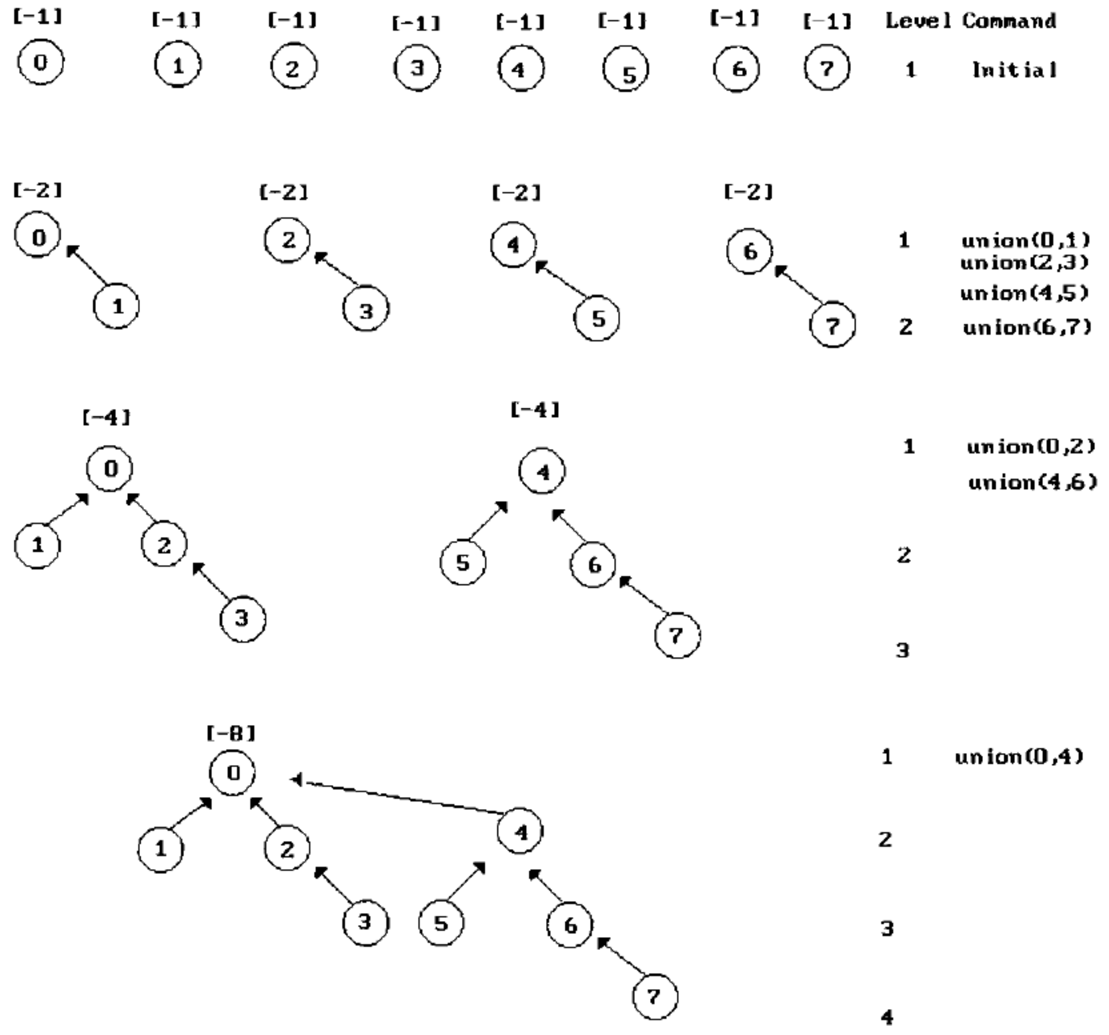


Figure 5.45: Trees achieving worst case bound

$$\alpha(m, n) = \min\{z \geq 1 \mid A(z, 4\lceil m/n \rceil) > \log_2 n\}$$

The definition of Ackermann's function used here is:

$$A(p, q) = \begin{cases} 2q & p = 0 \\ 0 & q = 0 \text{ and } p \geq 1 \\ 0 & p \geq 1 \text{ and } p = 1 \\ A(p-1, A(p, q-1)) & p \geq 1 \text{ and } q \geq 2 \end{cases}$$

```
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;
}
```

Program 5.20: Find function

COUNTING BINARY TREES

To determine the number of distinct binary trees having *n* nodes, the number of distinct permutations of the numbers from 1 to *n* obtainable by a stack, and the number of distinct ways of multiplying *n* + 1 matrices.

Distinct Binary Trees

We know that if $n = 0$ or $n = 1$, there is only one binary tree. If $n = 2$, then there are two distinct trees (Figure 5.47) and if $n = 3$, there are five such trees (Figure 5.48). How many distinct trees are there with *n* nodes? Before deriving a solution, we will examine

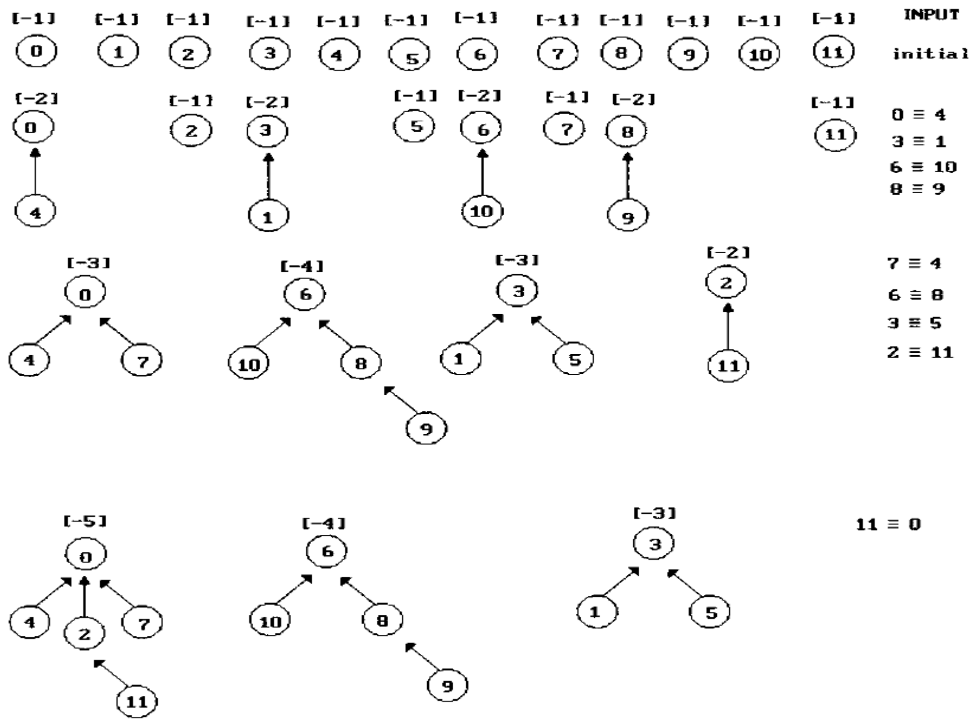


Figure 5.46: Trees for equivalence example

the two remaining problems. You might attempt to sketch out a solution of your own before reading further.

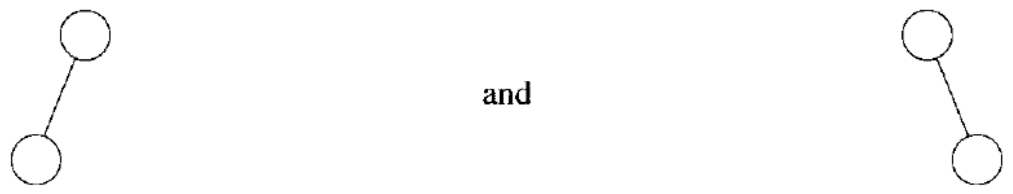


Figure 5.47: Distinct binary trees with $n = 2$

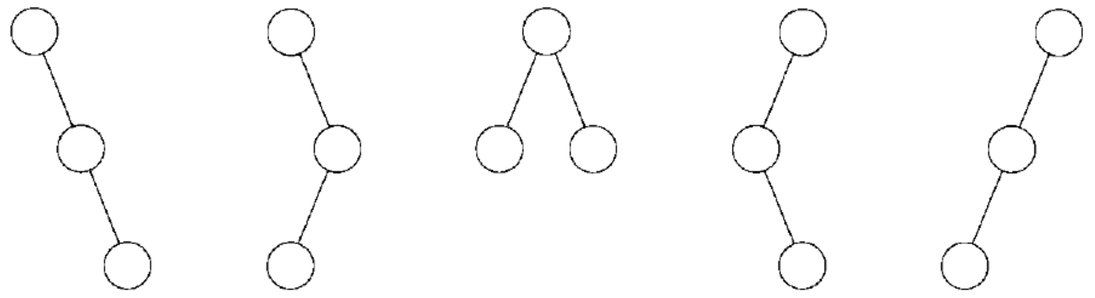


Figure 5.48: Distinct binary trees with $n = 3$

5.11.2 Stack Permutations

In Section 5.3, we introduced preorder, inorder, and postorder traversals and indicated that each traversal required a stack. Suppose we have the preorder sequence:

$$A\ B\ C\ D\ E\ F\ G\ H\ I$$

and the inorder sequence:

$$B\ C\ A\ E\ D\ G\ H\ F\ I$$

of the same binary tree. Does such a pair of sequences uniquely define a binary tree? Put another way, can this pair of sequences come from more than one 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 C) are in the left subtree, while the remaining nodes (E D G H F I) 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.

Let the nodes of an *n* node binary tree be numbered from 1 to *n*. The inorder permutation defined by such a binary tree is the order in which its nodes are visited during an inorder traversal of the tree. A preorder permutation is similarly defined.

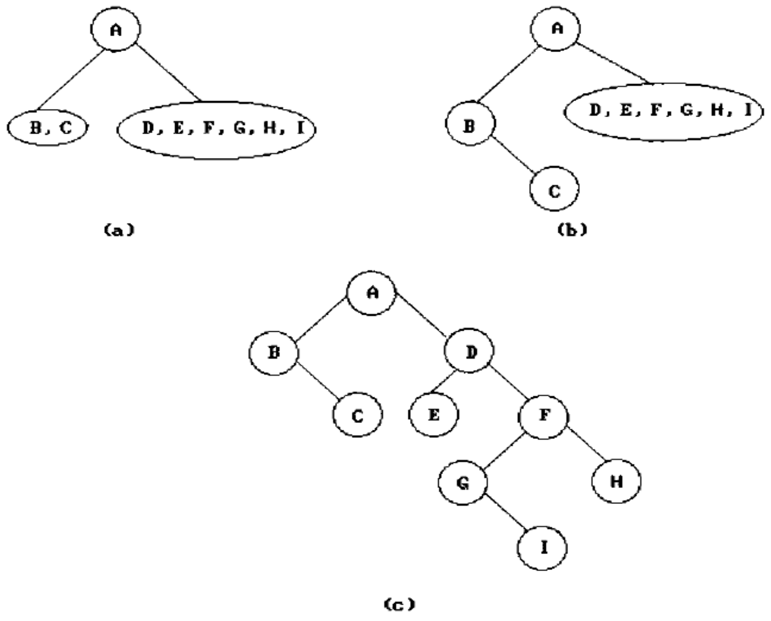


Figure 5.49: Constructing a binary tree from its inorder and preorder sequences

As an example, consider the binary tree of Figure 5.49(c) with the node numbering of Figure 5.50. Its preorder permutation is 1, 2, · · · , 9, and its inorder permutation is 2, 3, 1, 5, 4, 7, 8, 6, 9.

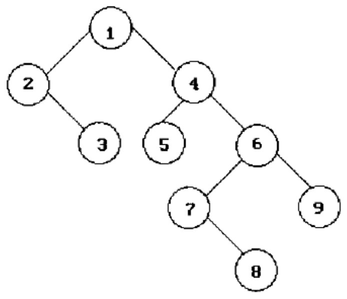


Figure 5.50: Binary tree of Figure 5.49(c) with its nodes numbered

If the nodes of the tree are numbered such that its preorder permutation is 1, 2, $\cdots$, n, then from our earlier discussion it follows that distinct binary trees define distinct inorder permutations. Thus, 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, $\cdots$, n.

Using the concept of an inorder permutation, we can show that the number of distinct permutations obtainable by passing the numbers 1 to n through a stack and deleting in all possible ways is equal to the number of distinct binary trees with n nodes (see the exercises). If we start with the numbers 1, 2, 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 (Figure 5.51).

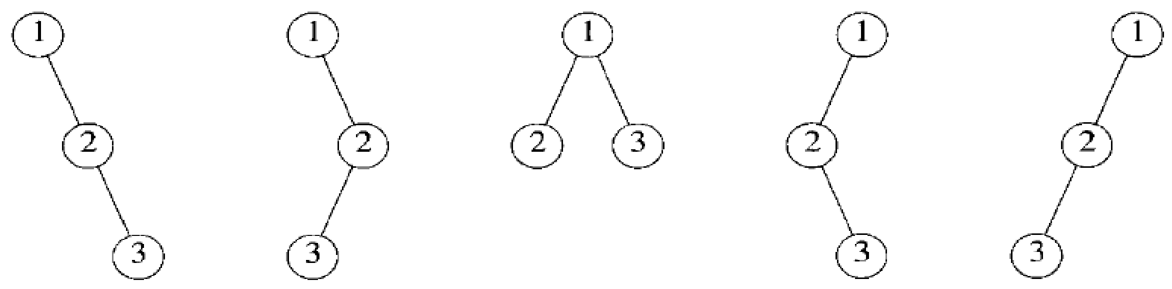


Figure 5.51: Binary trees corresponding to five permutations

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:

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

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

Let b_n be the number of different ways to compute the product of n matrices. Then $b_2 = 1$, $b_3 = 2$, and $b_4 = 5$. Let M_{ij}, $i \le j$, be the product $M_i * M_{i+1} * \cdots * M_j$. The product we wish to compute is M_{1n}. We may compute M_{1n} by computing any one of the products $M_{1i} * M_{i+1,n}$, $1 \le i \le n$. The number of distinct ways to obtain M_{1i} and $M_{i+1,n}$ are b_i and b_{n-i}, respectively. Therefore, letting $b_1 = 1$, we have:

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

If we can determine the expression for b_n only in terms of n, then we have a solution to our problem.

Now instead let b_n be the number of distinct binary trees with n nodes. Again an expression for b_n in terms of n is what we want. 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 $0 \le i < n$ (Figure 5.52). This explanation says that

$$b_n = \sum_{i=0}^{n-1} b_i\, b_{n-i-1}\ ,\ n\ge 1\ ,\text{and } b_0 = 1 \qquad (5.3)$$

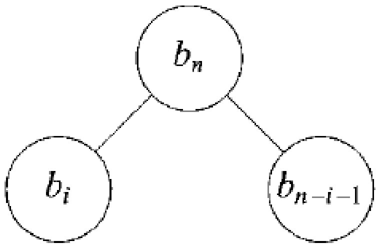


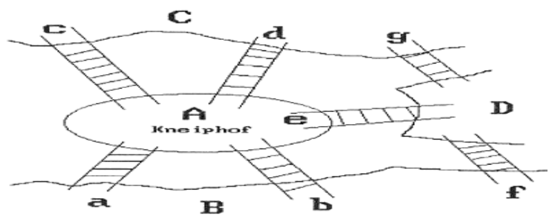
Figure 5.52: Decomposing b_n

This formula and the previous one are essentially the same. 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.

GRAPHS

Introduction to Graphs

Graph is a non linear data structure; A map is a well-known example of a graph. In a map various connections are made between the cities. The cities are connected via roads, railway lines and aerial network. We can assume that the graph is the interconnection of cities by roads. Euler used graph theory to solve Seven Bridges of Königsberg problem. Is there a possible way to traverse every bridge exactly once – Euler Tour



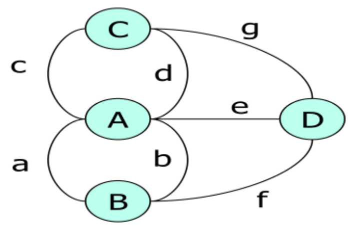


Figure: Section of the river Pregal in Koenigsberg and Euler's graph.

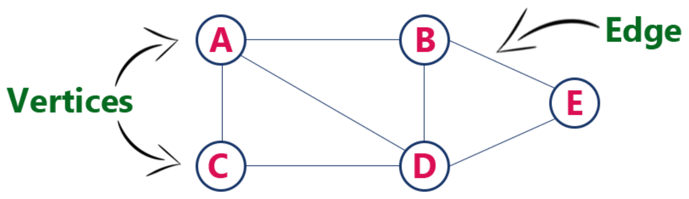
Defining the degree of a vertex to be the number of edges incident to it, Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex iff the degree of each, vertex is even. A walk which does this is called Eulerian. There is no Eulerian walk for the Koenigsberg bridge problem as all four vertices are of odd degree.

A graph contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices.

A graph is defined as Graph is a collection of vertices and arcs which connects vertices in the graph. A graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

Example: graph G can be defined as G = (V , E) Where V = {A,B,C,D,E} and

E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.   This is a graph with 5 vertices and 6 edges.



Graph Terminology

1.**Vertex** : An individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

2.**Edge** : An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex).

In above graph, the link between vertices A and B is represented as (A,B).

Edges are three types:

1.Undirected Edge - An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

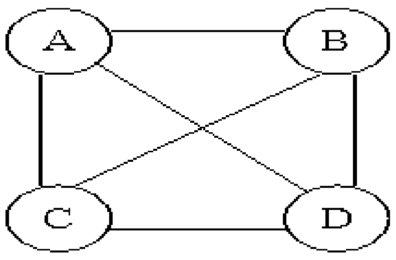
2.Directed Edge - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

3.Weighted Edge - A weighted edge is an edge with cost on it.

Types of Graphs

1.Undirected Graph

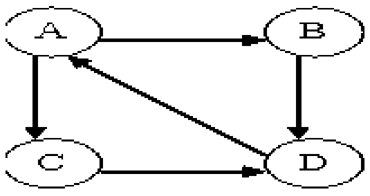
A graph with only undirected edges is said to be undirected graph.



Undirected Graph.

2.Directed Graph

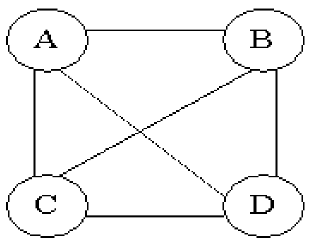
A graph with only directed edges is said to be directed graph.



Directed Graph.

3.Complete Graph

A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in the graph. The following figure shows a complete graph.

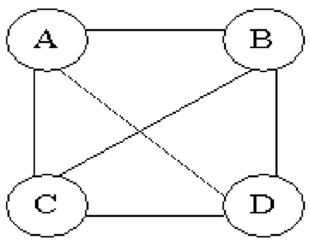


Type your text

A complete graph.

4.Regular Graph

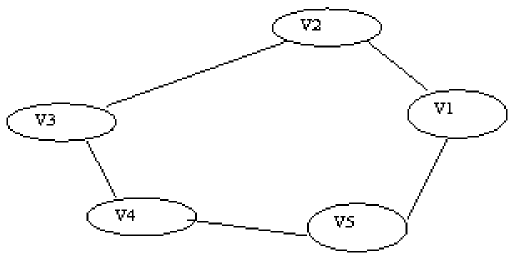
Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.



A regular graph

5.Cycle Graph

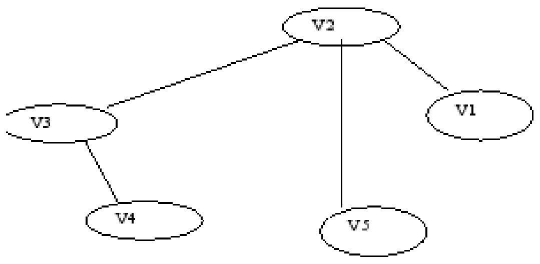
A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.



A cycle graph

6.Acyclic Graph

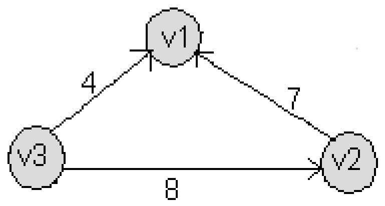
A graph without cycle is called acyclic graphs.



A acyclic graph

7. Weighted Graph

A graph is said to be weighted if there are some non negative value assigned to each edges of the graph. The value is equal to the length between two vertices. Weighted graph is also called a network.



A weighted graph

Outgoing Edge

A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Adjacent nodes

When there is an edge from one node to another then these nodes are called adjacent nodes.

Incidence

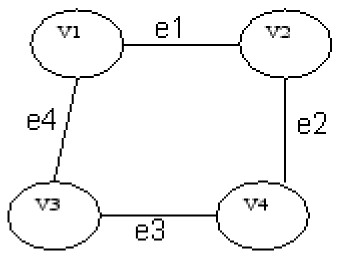
In an undirected graph the edge between v1 and v2 is incident on node v1 and v2.

Walk

A walk is defined as a finite alternating sequence of vertices and edges, beginning and ending with vertices, such that each edge is incident with the vertices preceding and following it.

Closed walk

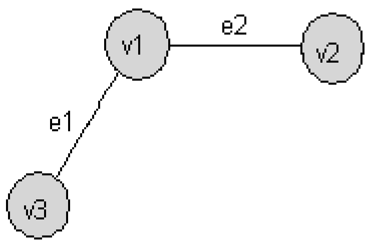
A walk which is to begin and end at the same vertex is called close walk. Otherwise it is an open walk.



If e1,e2,e3,and e4 be the edges of pair of vertices (v1,v2),(v2,v4),(v4,v3) and (v3,v1) respectively ,then v1 e1 v2 e2 v4 e3 v3 e4 v1 be its closed walk or circuit.

Path

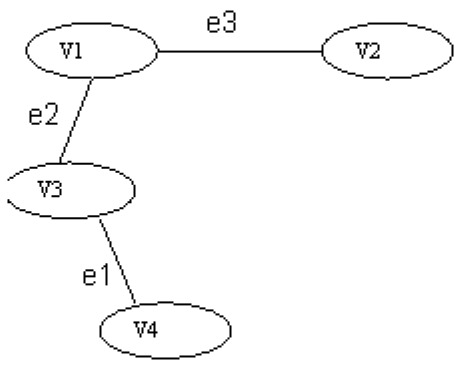
A open walk in which no vertex appears more than once is called a path.



If e1 and e2 be the two edges between the pair of vertices (v1,v3) and (v1,v2) respectively, then v3 e1 v1 e2 v2 be its path.

Length of a path

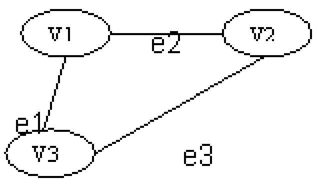
The number of edges in a path is called the length of that path. In the following, the length of the path is 3.



An open walk Graph

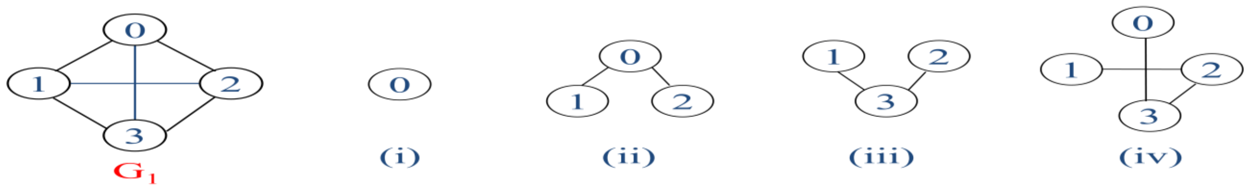
Circuit

A closed walk in which no vertex (except the initial and the final vertex) appears more than once is called a circuit.
A circuit having three vertices and three edges.



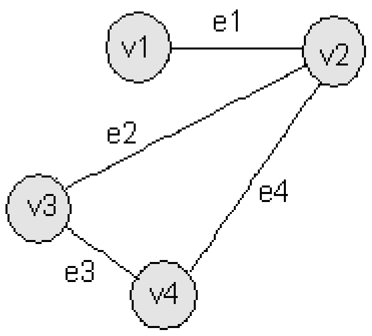
Sub Graph

A graph S is said to be a sub graph of a graph G if all the vertices and all the edges of S are in G, and each edge of S has the same end vertices in S as in G. A subgraph of G is a graph G' such that V(G') ⊆ V(G) and E(G') ⊆ E(G)

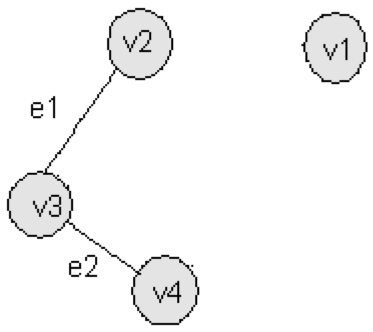


Connected Graph

A graph G is said to be connected if there is at least one path between every pair of vertices in G. Otherwise,G is disconnected.



A connected graph G



A disconnected graph G

This graph is disconnected because the vertex v1 is not connected with the other vertices of the graph.

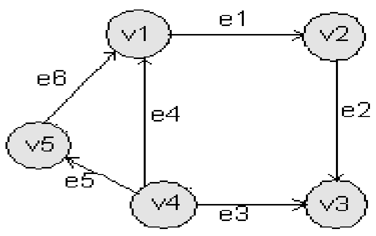
Degree

In an undirected graph, the number of edges connected to a node is called the degree of that node or the degree of a node is the number of edges incident on it.

In the above graph, degree of vertex v1 is 1, degree of vertex v2 is 3, degree of v3 and v4 is 2 in a connected graph.

Indegree

The indegree of a node is the number of edges connecting to that node or in other words edges incident to it.



In the above graph,the indegree of vertices v1, v3 is 2, indegree of vertices v2, v5 is 1 and indegree of v4 is zero.

Outdegree

The outdegree of a node (or vertex) is the number of edges going outside from that node or in other words the

ADT of Graph:

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 edge.

Graph InsertEdge(*graph*, *v1*,*v2*)::= return a graph with new edge between *v1* and *v2*

Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed

Graph DeleteEdge(*graph*, *v1*, *v2*)::=return a graph in which the edge (*v1*, *v2*) is removed

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*

Graph Representations

Graph data structure is represented using following representations

1. **Adjacency Matrix**
2. **Adjacency List**
3. **Adjacency Multilists**

1.Adjacency Matrix

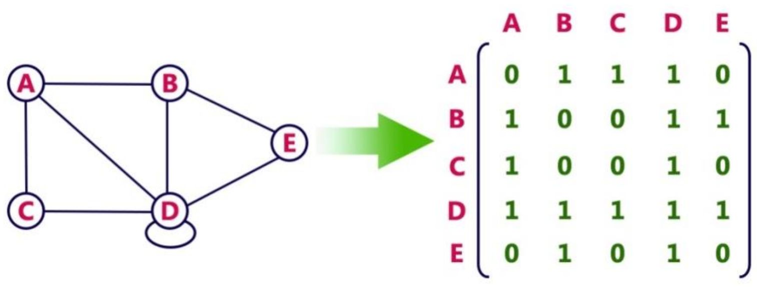
In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size.

In this matrix, rows and columns both represent vertices.

This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

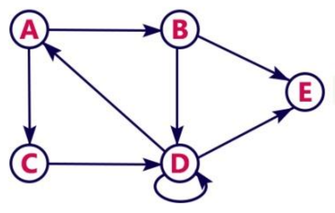
Adjacency Matrix : let G = (V, E) with n vertices, n ≥ 1. The adjacency matrix of G is a 2-dimensional n × n matrix, A, A(i, j) = 1 iff (v_i, v_j) ∈E(G) (⟨v_i, v_j⟩ for a diagraph), A(i, j) = 0 otherwise.

example : for undirected graph



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

For a Directed graph



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric.

Merits of Adjacency Matrix:

From the adjacency matrix, to determine the connection of vertices is easy

The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$

For a digraph, the row sum is the out_degree, while the column sum is the in_degree

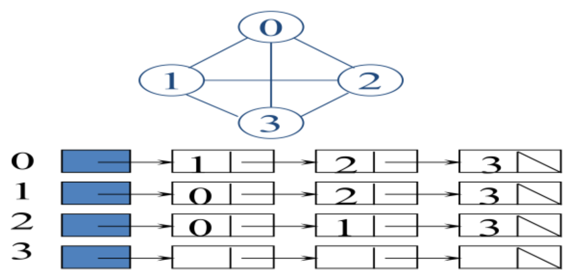
$ind(vi) = \sum_{j=0}^{n-1} A[j,i]$ $\qquad$ $outd(vi) = \sum_{j=0}^{n-1} A[i,j]$

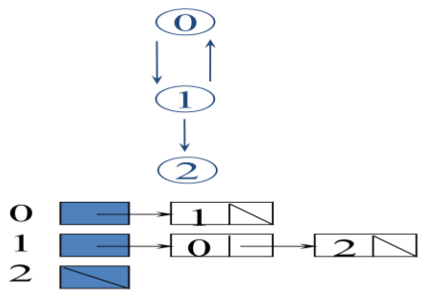
The space needed to represent a graph using adjacency matrix is n^2 bits. To identify the edges in a graph, adjacency matrices will require at least $O(n^2)$ time.

2. Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency matrix are represented as n chains. The nodes in chain I represent the vertices that are adjacent to vertex i.

It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store its adjacencies. Example:



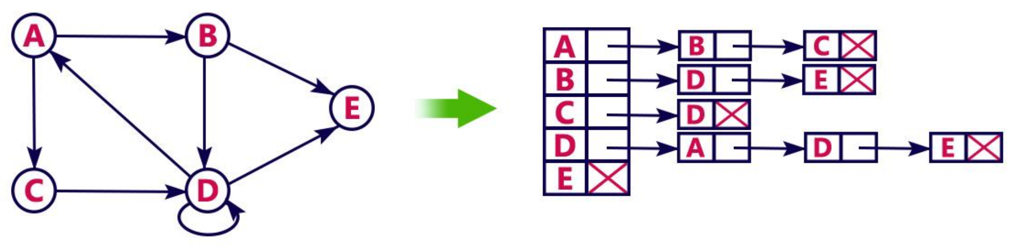


So that we can access the adjacency list for any vertex in O(1) time. Adjlist[i] is a pointer to to first node in the adjacency list for vertex i. Structure is

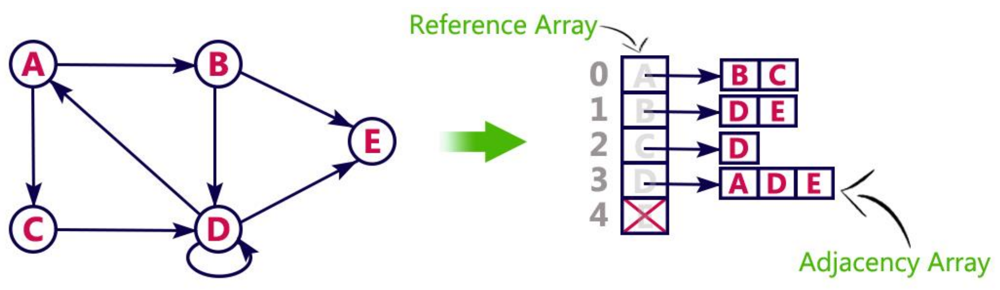
```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct node {
    int vertex;
    struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```

Another type of representation is given below.

example: consider the following directed graph representation implemented using linked list

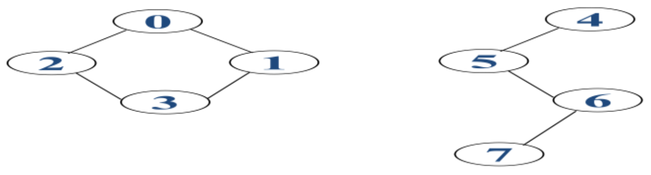


This representation can also be implemented using array



Sequential representation of adjacency list is

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6

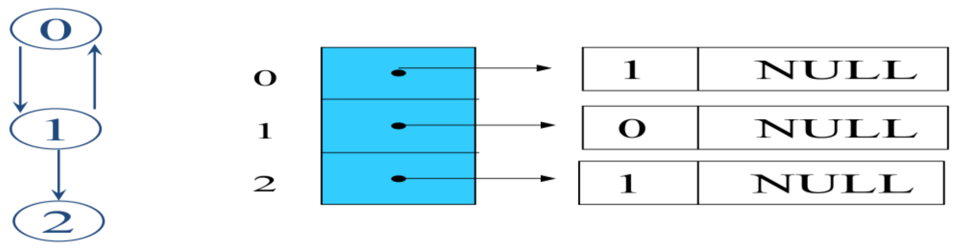


Graph

Instead of chains, we can use sequential representation into an integer array with size n+2e+1. For 0<=i<n, Array[i] gives starting point of the list for vertex I, and array[n] is set to n+2e+1. The adjacent vertices of node I are stored sequentially from array[i].

For an undirected graph with n vertices and e edges, linked adjacency list requires an array of size n and 2e chain nodes. For a directed graph, the number of list nodes is only e. the out degree of any vertex may be determined by counting the number of nodes in its adjacency list. To find in-degree of vertex v, we have to traverse complete list.

To avoid this, inverse adjacency list is used which contain in-degree.



Determine in-degree of a vertex in a fast way.

3.Adjacency Multilists

In the adjacency-list representation of an undirected graph each edge (u, v) is represented by two entries one on the list for u and the other on tht list for v. As we shall see in some situations it is necessary to be able to determin ie ~ nd enty for a particular edge and mark that edg as having been examined. This can be accomplished easily if the adjacency lists are actually maintained as multilists (i.e., lists in which nodes may be shared among several lists). For each edge there will be exactly one node but this node will be in two lists (i.e. the adjacency lists for each of the two nodes to which it is incident).

For adjacency multilists, node structure is

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

marked	vertex1	vertex2	path1	path2

Lists: vertex 0: N0->N1->N2, vertex 1: N0->N3->N4
vertex 2: N1->N3->N5, vertex 3: N2->N4->N5

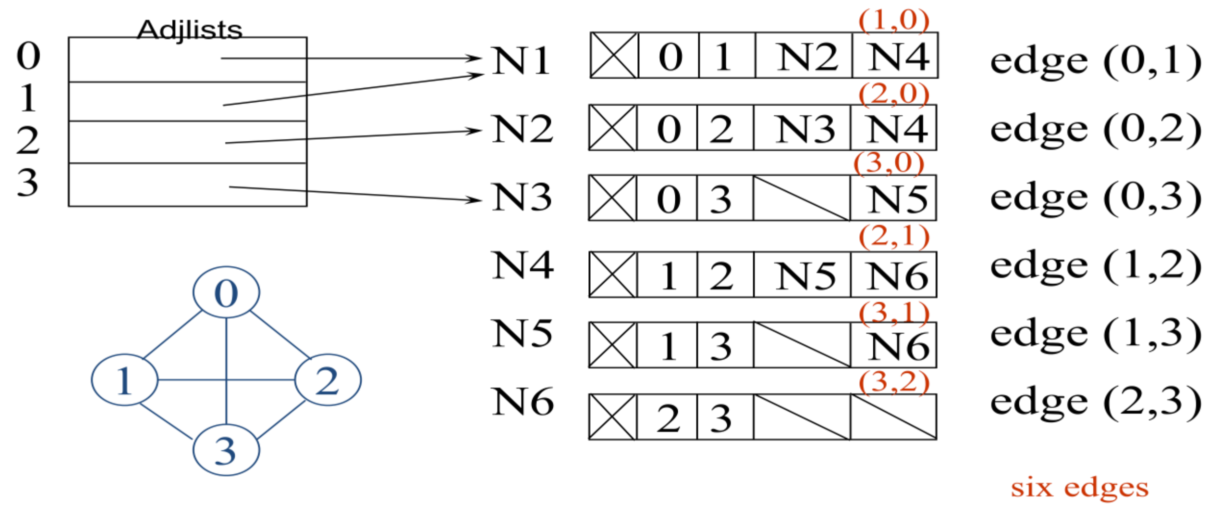
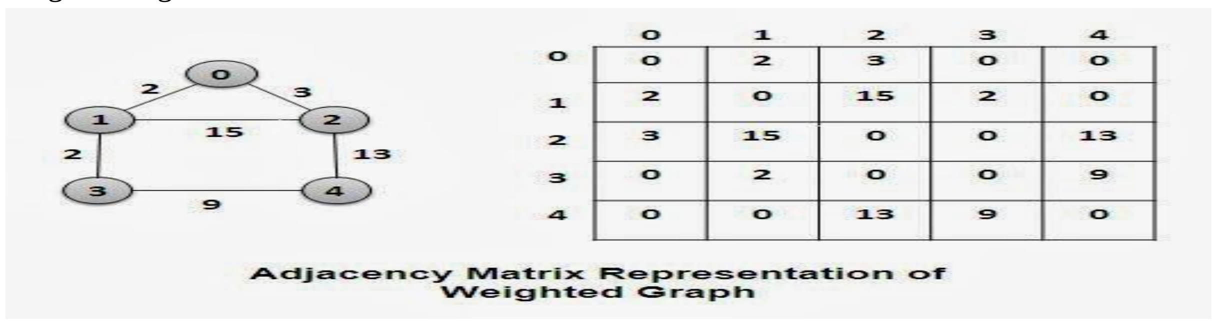

Adjlists
0
1
2
3
N1 | 0 | 1 | N2 | N4 (1,0) edge (0,1)
N2 | 0 | 2 | N3 | N4 (2,0) edge (0,2)
N3 | 0 | 3 | | N5 (3,0) edge (0,3)
N4 | 1 | 2 | N5 | N6 (2,1) edge (1,2)
N5 | 1 | 3 | | N6 (3,1) edge (1,3)
N6 | 2 | 3 | | (3,2) edge (2,3)
six edges

Figure: Adjacency multilists for given graph

4. Weighted edges

In many applications the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one; vertex to an adjacent vertex In these applications the adjacency matrix entries A [i][j] would keep this information too. When adjacency lists are used the weight information may be kept in the list'nodes by including an additional field weight. A graph with weighted edges is called a network.



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Adjacency Matrix Representation of Weighted Graph

ELEMENTARY GRAPH OPERATIONS

Given a graph G = (V E) and a vertex v in V(G) we wish to visit all vertices in G that are reachable from v (i.e., all vertices that are connected to v). We shall look at two ways of doing this: depth-first search and breadth-first search. Although these methods work on both directed and undirected graphs the following discussion assumes that the graphs are undirected.

Depth-First Search

- Begin the search by visiting the start vertex v
  - If v has an unvisited neighbor, traverse it recursively
  - Otherwise, backtrack
- Time complexity
  - Adjacency list: O(|E|)
  - Adjacency matrix: O($|V|^2$)

We begin by visiting the start vertex v. Next an unvisited vertex w adjacent to v is selected, and a depth-first search from w is initiated. When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited vertex w adjacent to it and initiate a depth-first search from w. The search terminates when no unvisited vertex can be reached from any of the visited vertices.

DFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

We use the following steps to implement DFS traversal...

Step 1: Define a Stack of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3: Visit any one of the adjacent vertex of the verex which is at top of the stack which is not visited and push it on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.

Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

This function is best described recursively as in Program.

```
#define FALSE 0
#define TRUE 1
 int visited[MAX_VERTICES];
void dfs(int v)
{
  node_pointer w;
  visited[v]= TRUE;
  printf("%d", v);
  for (w=graph[v]; w; w=w->link)
    if (!visited[w->vertex])
      dfs(w->vertex);
}
```

Consider the graph G of Figure 6.16(a), which is represented by its adjacency lists as in Figure 6.16(b). If a depth-first search is initiated from vertex 0 then the vertices of G are visited in the following order: **0 1 3 7 4 5 2 6.** Since DFS(O) visits all vertices that can be reached from 0 the vertices visited, together with all edges in G incident to these vertices form a connected component of G.

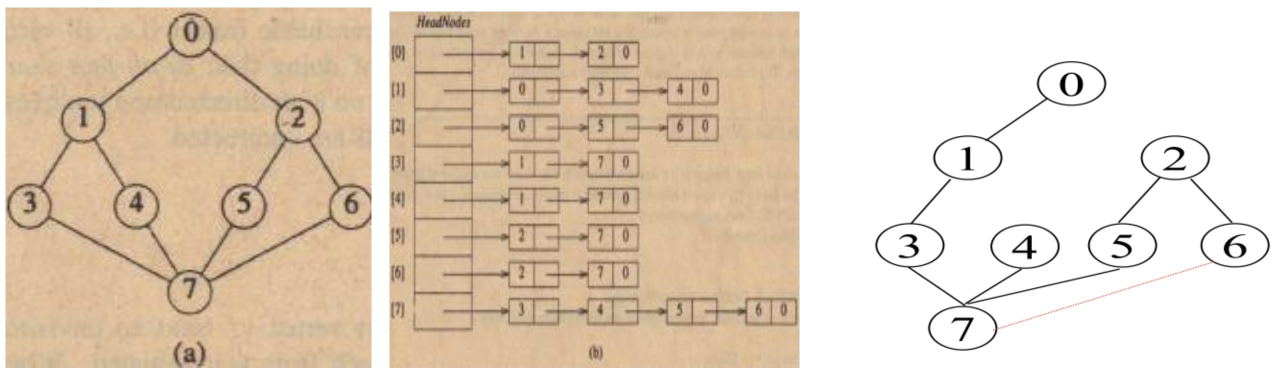


Figure: Graph and its adjacency list representation, DFS spanning tree

Analysis or DFS:
When G is represented by its adjacency lists, the vertices w adjacent to v can be determined by following a chain of links. Since DFS examines each node in the adjacency lists at most once and there are 2e list nodes the time to complete the search is O(e). If G is represented by its adjacency matrix then the time to determine all vertices adjacent to v is O(n). Since at most n vertices are visited the total time is O(n^2).

Breadth-First Search
In a breadth-first search, we begin by visiting the start vertex v. Next all unvisited vertices adjacent to v are visited. Unvisited vertices adjacent to these newly visited vertices are then visited and so on. Algorithm BFS (Program 6.2) gives the details.

```
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)
{
  node_pointer w;
  queue_pointer front, rear;
  front = rear = NULL;
  printf("%d", 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("%d", w->vertex);
        addq(&front, &rear, w->vertex);
        visited[w->vertex] = TRUE;
      }
  }
}
```

Steps:

BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal...

Step 1: Define a Queue of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3: Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.

Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

Step 5: Repeat step 3 and 4 until queue becomes empty.

Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

Analysis Of BFS:

Each visited vertex enters the queue exactly once. So the while loop is iterated at most n times If an adjacency matrix is used the loop takes O(n) time for each vertex visited. The total time is therefore, $O(n^2)$. If adjacency lists are used the loop has a total cost of $d_0 + \ldots + d_{n-1} = O(e)$, where d is the degree of vertex i. As in the case of DFS all visited vertices together with all edges incident to them, form a connected component of G.

3.Connected Components

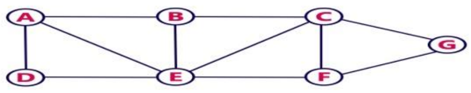
If G is an undirected graph, then one can determine whether or not it is connected by simply making a call to either DFS or BFS and then determining if there is any unvisited vertex. The connected components of a graph may be obtained by making repeated calls to either DFS(v) or BFS(v); where v is a vertex that has not yet been visited. This leads to function Connected(Program 6.3), which determines the connected components of G. The algorithm uses DFS (BFS may be used instead if desired). The computing time is not affected. Function connected –Output outputs all vertices visited in the most recent invocation of DFS together with all edges incident on these vertices.

```
void connected(void){
   for (i=0; i<n; i++) {
       if (!visited[i]) {
           dfs(i);
  printf("\n");        }     } }
```

Analysis of Components:

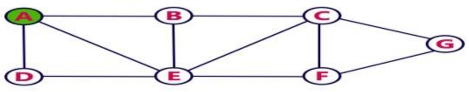
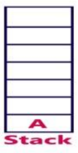
If G is represented by its adjacency lists, then the total time taken by dfs is O(e). Since the for loops take O(n) time, the total time to generate all the Connected components is O(n+e). If adjacency matrices are used,then the time required is $O(n^2)$

Consider the following example graph to perform DFS traversal



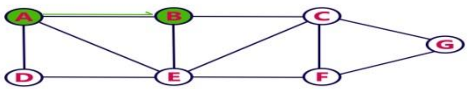
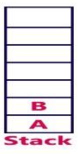
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

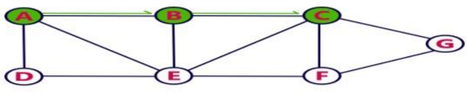
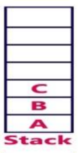
Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

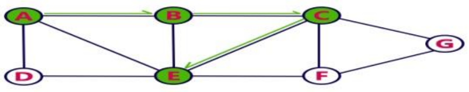
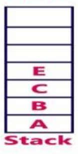
Step 3:

- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

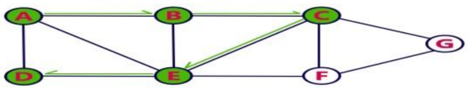
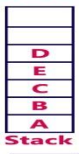
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

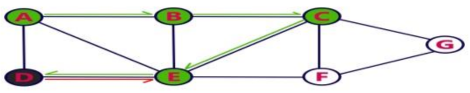
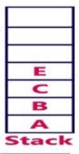
Step 5:

- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

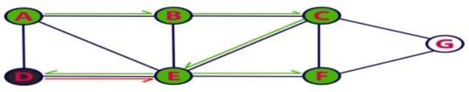
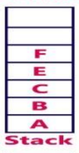
Step 6:

- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

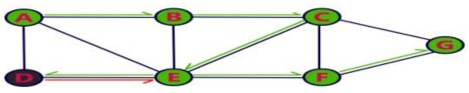
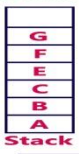
Step 7:

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

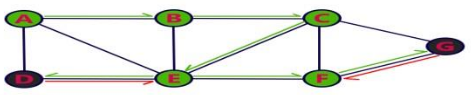
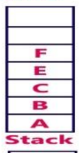
Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

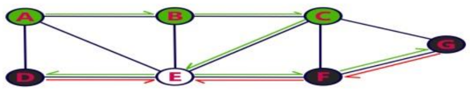
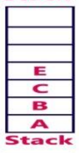
Step 9:

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

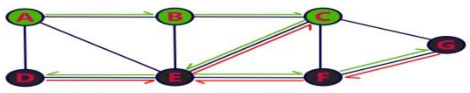
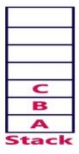
Step 10:

- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

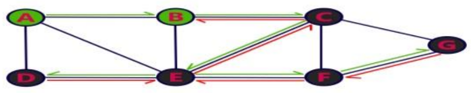
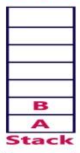
Step 11:

- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

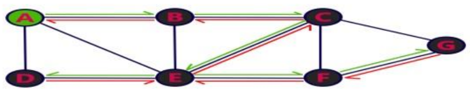
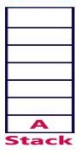
Step 12:

- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

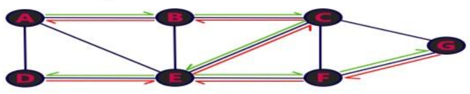
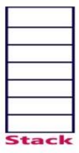
Step 13:

- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

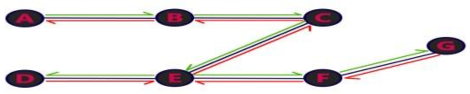



Step 14:

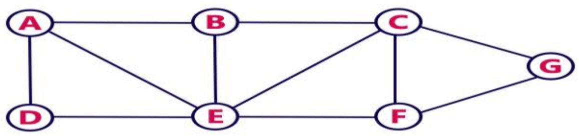
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.

- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.

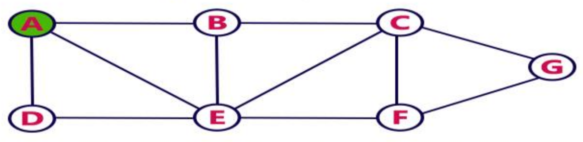


Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

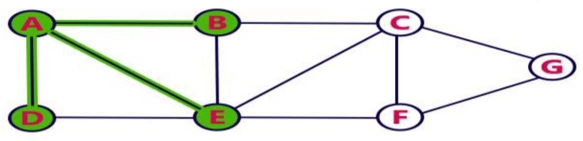


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

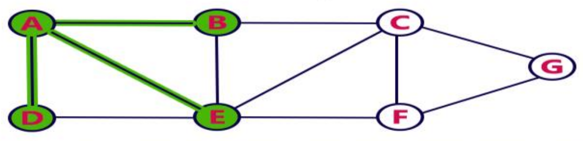


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

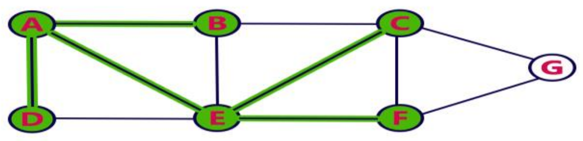


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

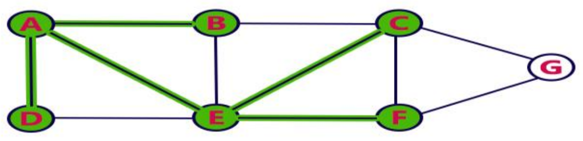


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

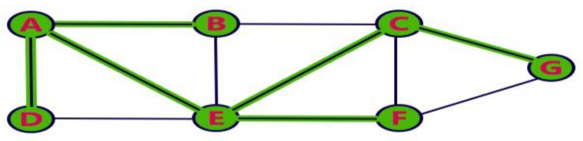


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

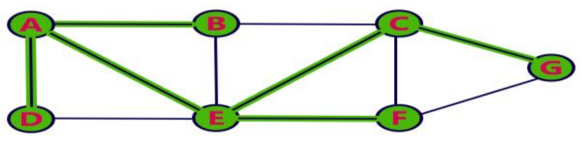


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

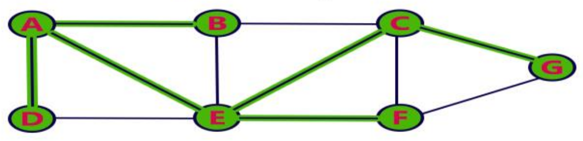


Queue

						G

Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...