

ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028



A T M E

College of Engineering

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

LECTURE NOTES



COURSE: DATA STRUCTURES AND APPLICATIONS

COURSE CODE: 1BCS305

SEMESTER: III

COURSE COORDINATOR: Mrs. PALLAVI R

(ACADEMIC YEAR 2026-227)

INSTITUTIONAL MISSION AND VISION

Vision

Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources

Mission

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torchbearers of tomorrow's society
- To strive to attain ever-higher benchmarks of educational excellence

DEPARTMENT MISSION AND VISION

Vision

To be a global leader in Computer Science and Design Engineering striving for design excellence, cultural awareness, a profound commitment to environmental stewardship, and societal responsibility.

Mission

1. To establish a technology-enabled experiential learning environment, prioritizing and cultivating problem-solving and design thinking skills among students.
2. To foster collaboration with industries, research and development organizations, jointly addressing socially relevant challenges in Computer Science with a core emphasis on design.

Programme Educational Objectives (PEOs)

PEO1: Demonstrate the proficiency and adaptability needed to seamlessly integrate with the latest advancements and changes in Computer Science and Design.

PEO2: Our graduates will possess the adaptability to explore various career paths, such as Research & Development, Manufacturing, and Entrepreneurship with independent design firms, Design Services, and Consultancies.

PEO3: Our students will be well-prepared to excel in collaborative, multidisciplinary environments, showcasing their versatility and skills.

Programme Specific Outcomes (PSOs)

PSO1: To develop stand-alone, embedded, and web-based solutions with easy-to-operate interfaces using software engineering practices and contemporary programming languages.

PSO2: Design and develop computer-based systems in various areas of Multimedia, Graphics data visualization and computer vision.

MODULE04: Trees and Graphs

MODULE04	TREES(Cont.): Binary Search trees, Selection Trees, Forests, Representation of Disjoint sets, Counting Binary Trees, GRAPHS: The Graph Abstract Data Types, Elementary Graph Operations
-----------------	---

Session 25

4.1 BINARY SEARCH TREES:

- Binary search trees (BST), sometimes called ordered or sorted binary trees, are a particular type of containers: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic set of items, or lookup table that allow finding an item by its key (e.g., finding the phone number of a person by name).
- Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.
- A binary search tree (Fig.4.1) T is a binary tree; either it is empty or each node in the tree contains an identifier and:
 - (i) all identifiers in the left subtree of T are less (numerically or alphabetically) than the identifier in the root node T ;
 - (ii) all identifiers in the right subtree of T are greater than the identifier in the root node T ;
 - (iii) the left and right subtrees of T are also binary search trees.

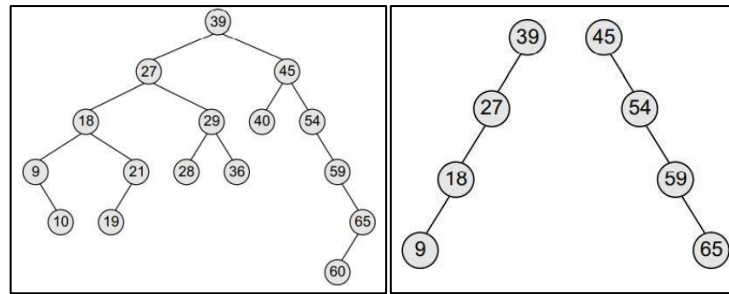


Fig.4.1: Binary Search tree

Structure:

```

structnode
{
    int info;
    struct node
    *llink; structnode *rli
    nk;
};
structnode *NODE;

```

4.1.1 CreateFunction

- First enter the data. If NODE=NULL, then create a new node using malloc function. Store the data in that new node and make its left and right link as NULL.

```

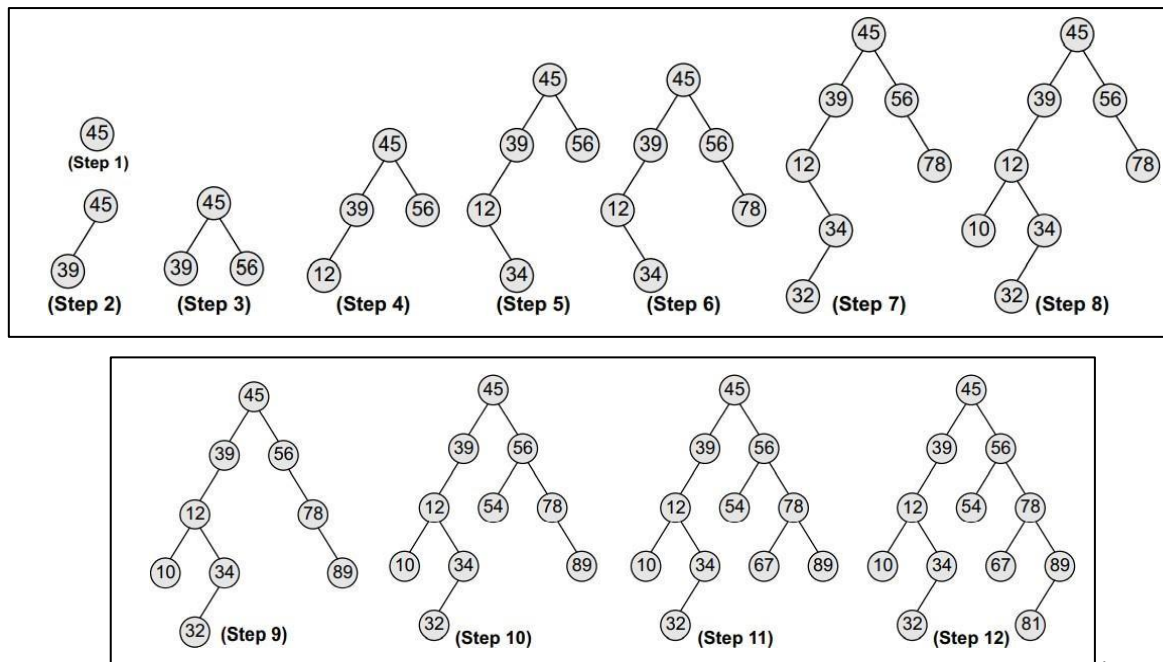
NODE createNode(int item)
{
    NODE newNode;
    newNode = (NODE) malloc(sizeof(struct
    node)); newNode->info=item;
    newNode->llink=
    NULL; newNode->rlink =
    NULL; return(newNode);
}

```

Example:

Create a binary search tree using the following data elements: 45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81

Solution is given in Figure 4.12



.Figure4.12:Creating a Binary Search tree

4.1.2 Insert Function

The insert function is used to add a new node with a given value at the correct position in the binary search tree. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree.

```

NODEinsert(intitem,NODEroot)
{
    NODE
    temp,cur,prev;inti;
    temp=createNod(
);
    temp->info=item;
    temp->llink=temp-
    >rlink=NULL;if(root==NULL)
    {
        root=temp;retur
        nroot;
    }
    else
    {
        prev=NULL;

```

```

    cur=root;
    while(cur!=NULL)
    {
        prev=cur;
        cur=(temp->info<cur->info)?cur->llink:cur->rlink;
    }
    if(temp->info<prev->info)prev->llink=temp;
    else if(temp->info>prev->info)prev->rlink=temp;
    returnroot;
}
}

```

4.1.3 SearchFunction:

The search function is used to find whether a given value is present in the tree or not. The searching process begins at the root node. The function first checks if the binary search tree is empty. If it is empty, then the value we are searching for is not present in the tree. So, the search algorithm terminates by displaying an appropriate message. However, if there are nodes in the tree, then the search function checks to see if the key value of the current node is equal to the value to be searched. If not, it checks if the value to be searched for is less than the value of the current node, in which case it should be recursively called on the left child node. In case the value is greater than the value of the current node, it should be recursively called on the right child node. The procedure to find the node with value 67 is illustrated in Fig.4.13.

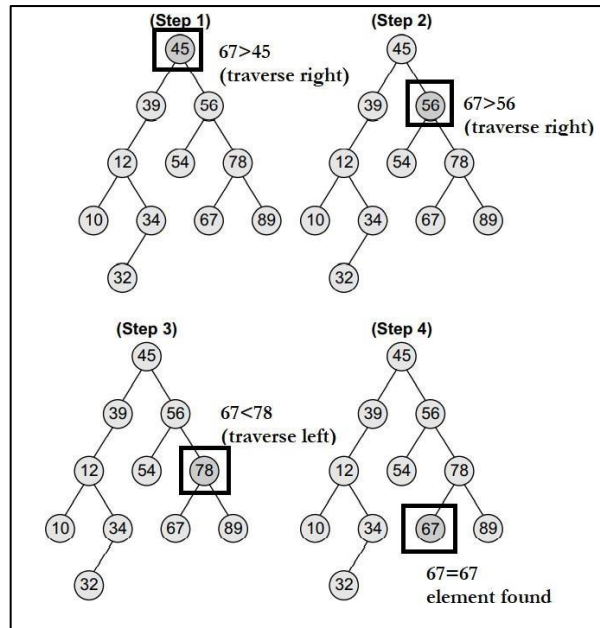


Fig4.13:Example for searching an element in a Binary Search tree

```

NODEsearch(NODEroot)/*for search function*/
{
    int
    item,i=0;N
    ODEcur;
    printf("enter the element to be
    searched\n");scanf("%d",&item);
    if(root==NULL)
    {
        printf("tree is
        empty\n");returnroot;
    }
    cur=root;while(cur
    !=NULL)
    {
        if(item==cur->info)
        {
            i++;
            printf("Found key%d in tree\n",cur->info);
        }
        if(item<cur->info)

```

```

        cur=cur->llink;cur=cur->rlink;
    }

    if(i==0)
        printf("Key not found\n");return root;
}

```

4.1.4 Delete Function:

The delete function deletes a node from the binary search tree. However, utmost care should be taken that the properties of the binary search tree are not violated and nodes are not lost in the process.

Case1: Deleting a Node that has No Children

Look at the binary search tree given in Fig. 4.14. If we have to delete node 78, we can simply remove this node without any issue. This is the simplest case of deletion.

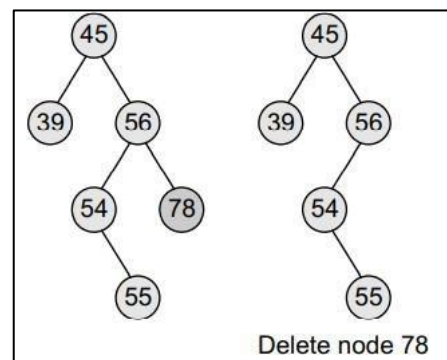


Fig.4.14: delete a node that has no children in a Binary Search tree

Case2: Deleting a Node with One Child

To handle this case, the node's child is set as the child of the node's parent. In other words, replace the node with its child. Now, if the node is the left child of its parent, the node's child becomes the left child of the node's parent. Correspondingly, if the node is the right child of its parent, the node's child becomes the right child of the node's parent. Look at the binary search tree shown in Fig. 4.15 and see how deletion of node 54 is handled.

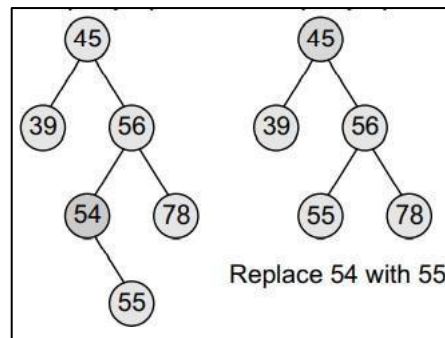


Fig.4.15: delete a node with one child in a Binary Search tree

Case 3: Deleting a Node with Two Children

To handle this case, replace the node's value with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree). The in-order predecessor or the successor can then be deleted using any of the above cases. Look at the binary search tree given in 4.16 and see how deletion of node with value 56 is handled.

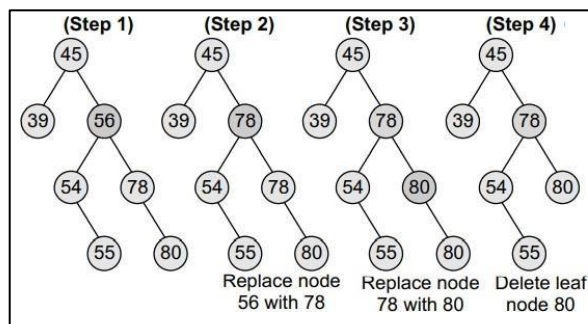


Fig.4.16: delete a node with 2 children in a Binary Search tree

4.1.5 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 is greater than mid. key. Following the join, both small and big are empty.

(b) twoWay 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 small 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).

- To perform a split ,we first make the following observation about splitting at the root. In this case small is left subtree of theTree, mid is the pair in the root, and big is the right subtree of theTree.
- If k is smaller than the key at the root, then the root together with its right subtree is to be in big.
- If k is larger than the key at the root, then the root together with its left subtree is to be in small.
- As we move down we construct the two search trees small and big. The snippet of split function is given below.

```

void split(nodePointer *theTree, int k, nodePointer *small,
          element *mid, nodePointer *big)
{
    /* split the binary search tree with respect to key k */
    if (!theTree) { *small = *big = 0;
                    (*mid).key = -1; return; } /* empty tree */
    nodePointer sHead, bHead, s, b, currentNode;
    /* create header nodes for small and big */
    MALLOC(sHead, sizeof(*sHead));
    MALLOC(bHead, sizeof(*bHead));
    s = sHead; b = bHead;

    /* do the split */
    currentNode = *theTree;
    while (currentNode)
        if (k < currentNode->data.key) { /* add to big */
            b->leftChild = currentNode;
            b = currentNode;
            currentNode = currentNode->leftChild;
        }
        else if (k > currentNode->data.key) { /* add to small */
            s->rightChild = currentNode;
            s = currentNode;
            currentNode = currentNode->rightChild;
        }
        else { /* split at currentNode */
            s->rightChild = currentNode->leftChild;
            b->leftChild = currentNode->rightChild;
            *small = sHead->rightChild; free(sHead);
            *big = bHead->leftChild; free(bHead);
            (*mid).item = currentNode->data.item;
            (*mid).key = currentNode->data.key;
            free(currentNode);
            return;
        }
    /* no pair with key k */
    s->rightChild = b->leftChild = 0;
    *small = sHead->rightChild; free(sHead);
    *big = bHead->leftChild; free(bHead);
    (*mid).key = -1;
    return;
}

```

4.1.6 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-nodes to insert the keys $1, 2, 3, \dots, n$, 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 $O(\log 2n)$, on the average.

Session 25 questions:

What is binary search tree?

How to perform create in BST?

How to perform Insert function in BST?

How to perform Search function in BST?

How to perform Delete function in BST?

How to perform Joining and splitting in BST?

How to find height of BST?

Session 26

4.2 SELECTION TREE:

4.2.1 Introduction:

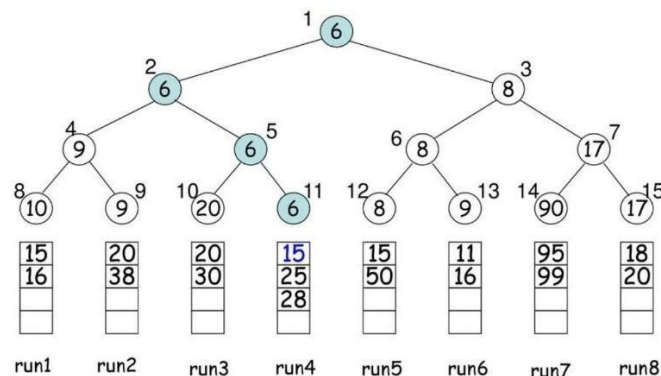
Suppose we have k ordered, called and is in merged into single ordered sequence. Each run consists of some records and is in non-decreasing order of a designated field called the key. Let n be the number of records in all k runs together task can be by the record with the smallest key. The smallest has 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:

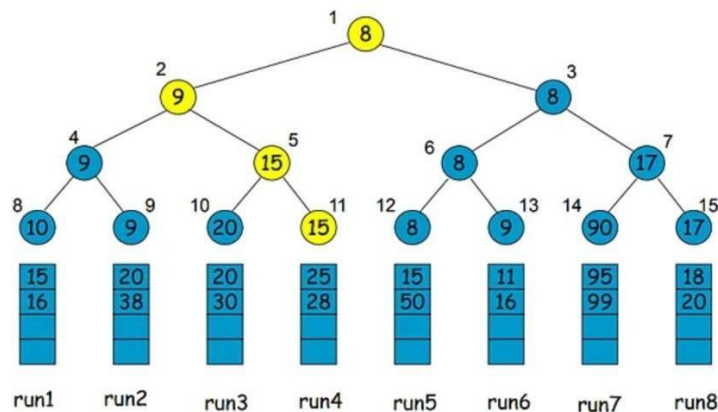
A winner tree is a binary tree in which each node represents the smaller of its two children. Thus, the root node the smallest node in the tree. Figure 5.32 illustrates a winner tree for the case $k=8$.

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 non-leaf 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 : Winnertreefork=8

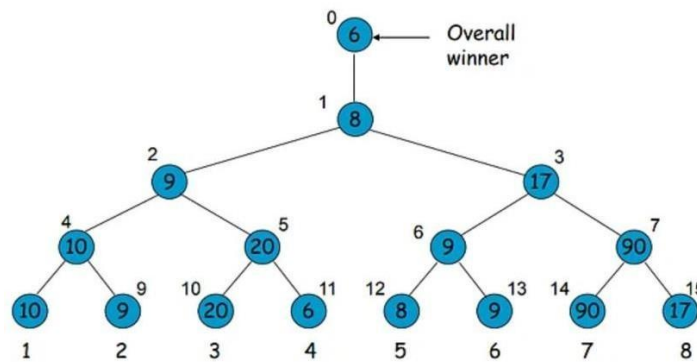


5.33 : Winnertreeoffig.5.32afterone recordhasbeenoutput andthetreerestructured.

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 4 this 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 keyvalue of a record rather than a pointer to the record represented. The leaf nodes 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 by playing 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. 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

Session 26:

What do you mean by selection?

What is winner tree, how to find it?

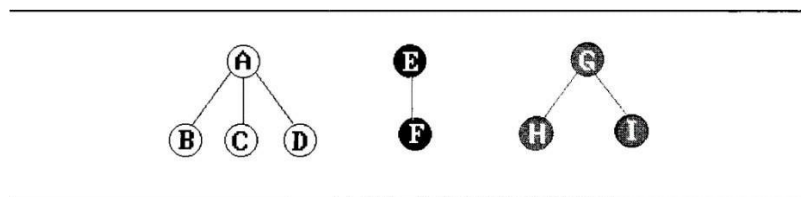
What is loser tree how to find it?

Session 27

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.



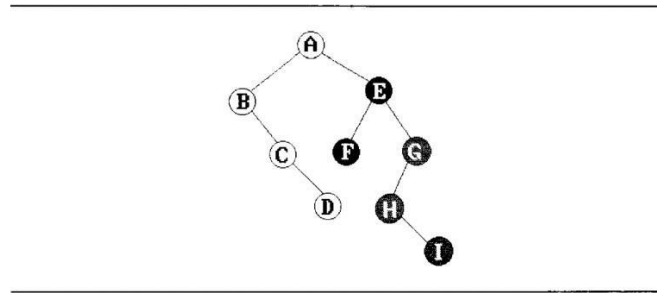
Three-tree

forest 4.3.1: Transforming a forest into a binary tree

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

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

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



Binary tree representation of forest

4.3.2 Forest Traversal

Preorder Traversal:

The preorder traversal of T is equivalent to visiting the nodes of 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:

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 .

Session 27:

What is forest in data structure?

How to represent Binary tree representation of forest?

How to perform preorder traversal in forest?

How to perform post order traversal in forest?

How to perform in order traversal in forest?

Session 28

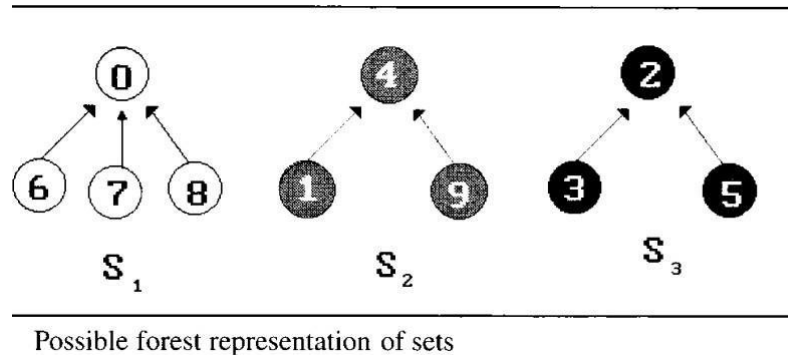
Representation of Disjoint Sets

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

Figure shows one possible representation for these sets.

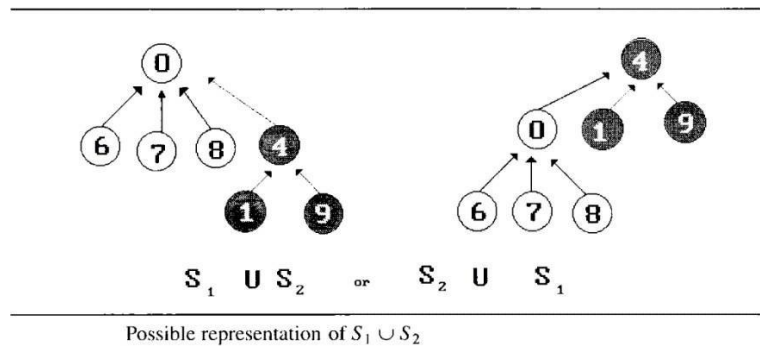


The minimal operations that we wish to perform on these sets

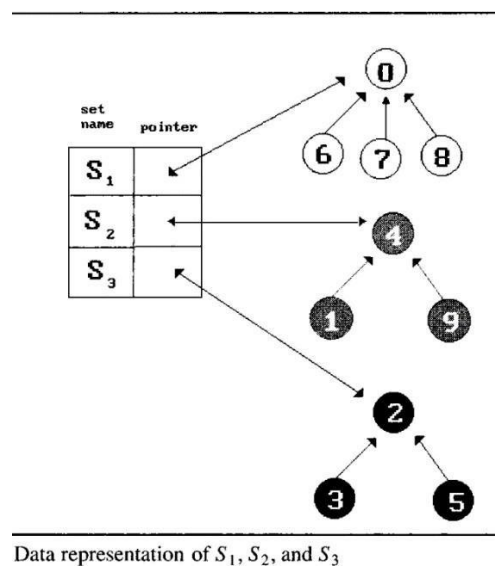
are: Disjoint set union and Find(i).

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.



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.



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

thenumberintree jthenmakejthe parentofi;otherwisemakeitthe parentofj.

```

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 $\text{parent}[i] \neq \text{root}(i)$, then set $\text{parent}[j]$ to $\text{root}(i)$.

```

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

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

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

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

The time to process a find is $O(\log m)$ if there are m elements in a tree. If an intermixed sequence of u-1 union and f find operations is to be processed, the time becomes $O(u + f \log u)$, as no tree has more than u nodes in it.

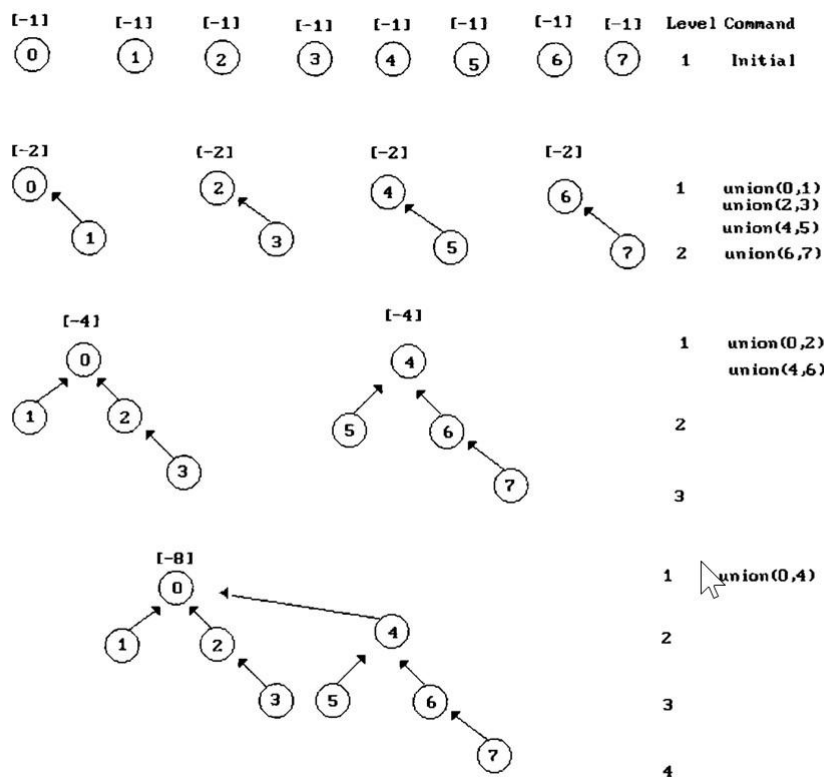


Figure 5.45: Trees achieving worst case bound

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 $\text{parent}[i] = -1$, $0 \leq i < n$. 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.

Example: We use the union-find algorithms to process the set of equivalence pairs. Initially, there are 12 trees, one for each variable and $\text{parent}[i] = -1$, $0 < i < 11$. The forest configuration following the processing of each equivalence pair is shown in Figure 5.46. Each tree represents an equivalence class. It is possible to determine if two elements are currently in the same equivalence class at each stage of the processing by simply making two finds.

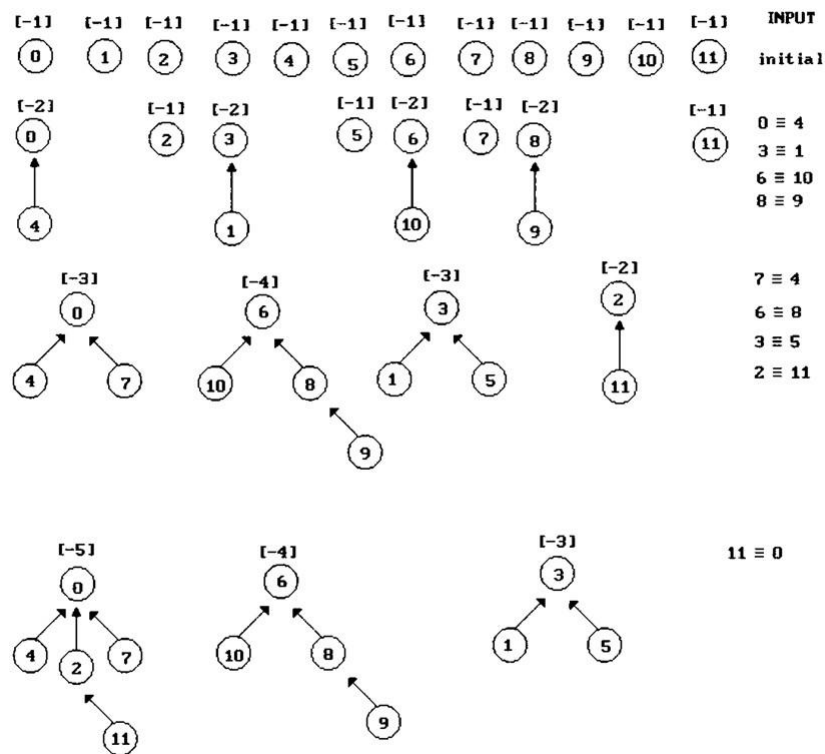


Figure 5.46: Trees for equivalence example

Session 28:

What is the difference between union and find function?

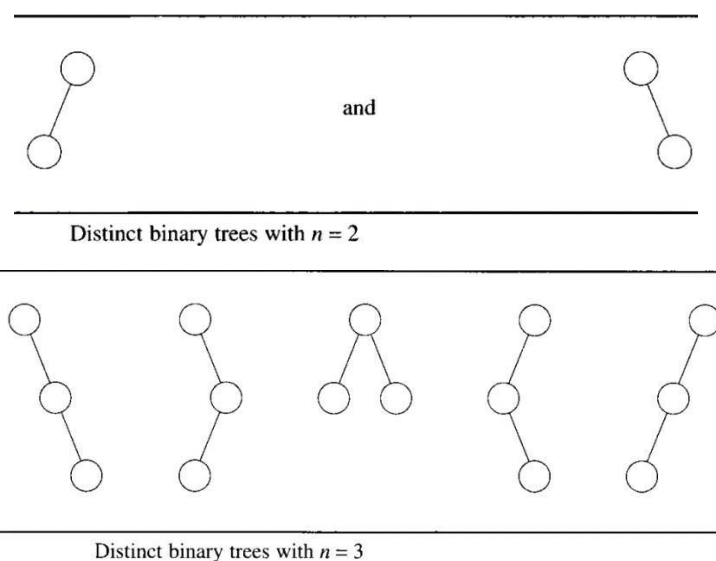
How to implement union using weighting rule?

How to implement find function?

4.4 Counting Binary

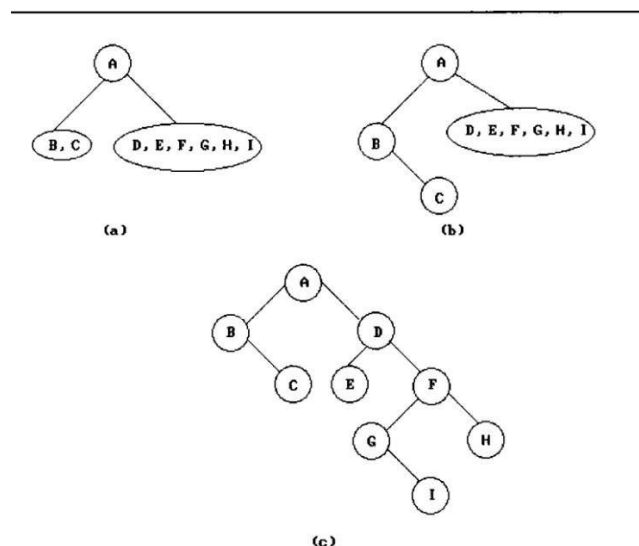
trees4.5.1:DistinctBinarytree

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



4.5.2 Stack permutations:

Suppose we have the preorder sequence: ABCDEFGHI and the inorder sequence: BCAEDGHI 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, C) are in the left subtree, while the remaining nodes {D, E, F, G, H, 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 and inorder sequences.



Constructing a binary tree from its inorder and preorder sequences

- If the nodes of the tree are numbered such that its preorder permutation is $1, 2, \dots, n$, 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 $1, 2, \dots, n$.
- 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.

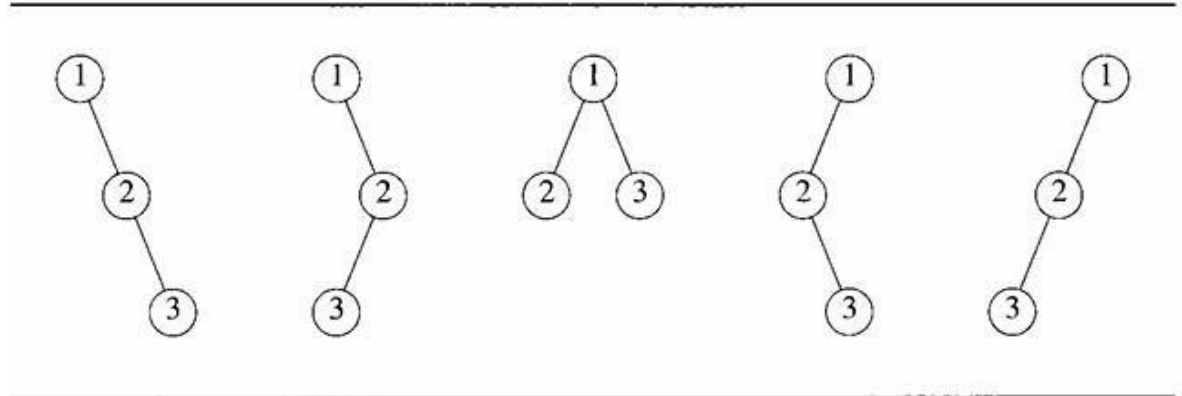


Figure 5.51: Binary trees corresponding to five permutations

4.5.3 Matrix multiplication

Suppose that we wish to compute the product of n matrices: $M_1 * M_2 * \dots * M_n$. 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_i and M_{i+1} are b_i and b_{i+1} , respectively. Therefore, letting

$$b_1 = 1, \text{ we have}$$

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

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

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

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

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

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 to 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:

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

we see that b_n which is the coefficient of x^n 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

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

Session 29:

What is counting binary tree?

What is distinct binary tree?

How to perform stack permutation?

How to perform matrix multiplication?

5. GRAPHS

5.1 The graph Abstract Data Type

5.1.1 Introduction

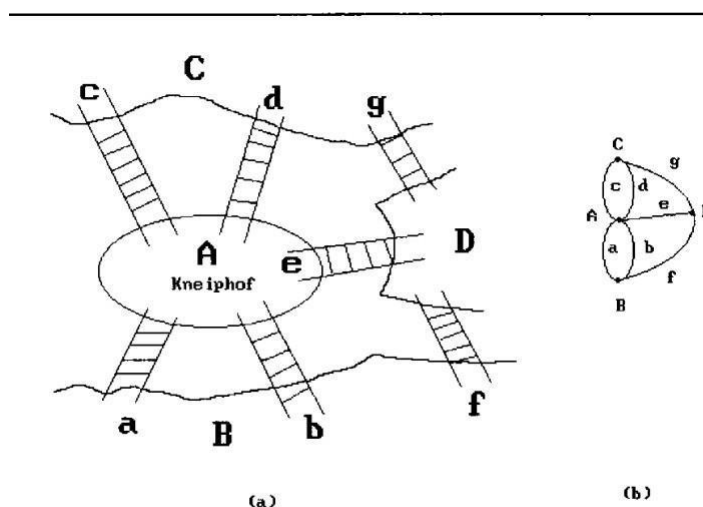
In Königsberg, the Pregal river flows around the island of Kneiphof. There are four land areas, labelled A through D in Figure 6.1, that have this river on their border. Seven bridges, labelled a through g, connect the land areas. The Königsberg bridge problem is as follows: Starting at some land area, 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.



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 is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices.
- A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.
- A graph, G , consists of two sets V and E . V is a finite non-empty set of vertices. E is a set of pairs of vertices, these pairs are called edges. $V(G)$ and $E(G)$ will represent the sets of vertices and edges of graph G . We will also write $G = (V, E)$ to represent a graph.
- In an undirected graph the pair of vertices representing any edge is unordered. Thus, the pairs (v_1, v_2) and (v_2, v_1) represent the same edge. In a directed graph each edge is represented by a directed pair (v_1, v_2) . v_1 is the tail and v_2 the head of the edge. Therefore, and represent two different edges.
- Figure 5.5 shows three graphs G_1 , G_2 and G_3 . The graphs G_1 and G_2 are undirected. G_3 is a directed graph.

$$V(G_1) = \{1, 2, 3, 4\}; E(G_1) = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$$

$$V(G_2) = \{1, 2, 3, 4, 5, 6, 7\}; E(G_2) =$$

$$\{(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7)\}$$

$$V(G_3) = \{1, 2, 3\}; E(G_3) = \{<1, 2>, <2, 1>, <2, 3>\}$$

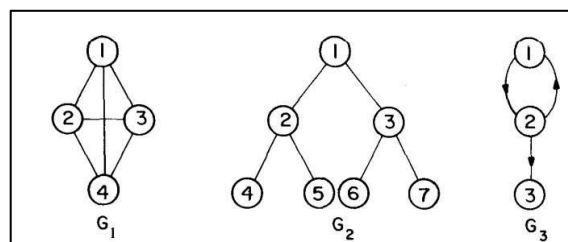
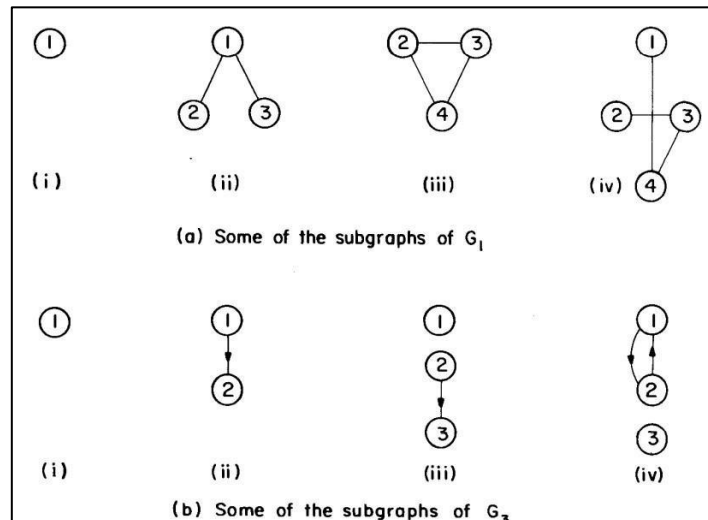


Fig.5.5. Sample graphs

5.1.3 Terminologies

- **Complete graph:** A graph is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has $n(n-1)/2$ edges, where n is the number of nodes in G . Example is G_1 in Fig.5.5.
- **Adjacent and incident:** If (v_1, v_2) is an edge in $E(G)$, then we shall say the vertices v_1 and v_2 are adjacent and that the edge (v_1, v_2) is incident on vertices v_1 and v_2 . The vertices adjacent to vertex 2 in G_2 are 4, 5 and 1. The edges incident on vertex 3 in G_2 are $(1,3)$, $(3,6)$ and $(3,7)$. If $\langle v_1, v_2 \rangle$ is a directed edge, then vertex v_1 will be said to be adjacent to v_2 while v_2 is adjacent from v_1 . The edge $\langle v_1, v_2 \rangle$ is incident to v_1 and v_2 . In G_3 the edges incident to vertex 2 are $\langle 1, 2 \rangle$, $\langle 2, 1 \rangle$ and $\langle 2, 3 \rangle$.
- **A subgraph:** A subgraph of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. Figure 5.6 shows some of the subgraphs of G_1 and G_3 in the figure 5.5.

Fig.5.6.(a) Subgraphs of G_1 and (b) Subgraphs of G_3

- **Path:** a path from vertex v_p to vertex v_q in graph G is a sequence of vertices $v_p, v_1, v_2, \dots, v_n, v_q$ such that $(v_p, v_1), (v_1, v_2), \dots, (v_n, v_q)$ are edges in $E(G)$. If G' is directed then the path consists of $\langle v_p, v_1 \rangle, \langle v_1, v_2 \rangle, \dots, \langle v_n, v_q \rangle$, edges in $E(G')$.
- **Length:** length of a path is the number of edges on it.
- **Simple Path:** A simple path is a path in which all vertices except possibly the first and last are distinct. A path such as $(1,2) (2,4) (4,3)$ we write as $1,2,4,3$. Paths $1,2,4,3$ and $1,2,4,2$ are both of length 3 in G_1 . The first is a simple path while the second is not. $1,2,3$ is a simple directed path in G_3 . $1,2,3,2$ is not a path in G_3 as the edge is not in $E(G_3)$.

- **Cycle:** A cycle is a simple path in which the first and last vertices are the same. 1,2,3,1 is a cycle in G_1 . 1,2,1 is a cycle in G_3 . For the case of directed graphs we normally add on the prefix "directed" to the terms cycle and path.
- **Connected graph:** In an undirected graph, G , two vertices v_1 and v_2 are said to be connected if there is a path in G from v_1 to v_2 (since G is undirected, this means there must also be a path from v_2 to v_1). An undirected graph is said to be connected if for every pair of distinct vertices v_i, v_j in $V(G)$ there is a path from v_i to v_j in G . Graphs G_1 and G_2 are connected while G_4 of figure 5.7 is not. A connected component or simply a component of an undirected graph is a maximal connected subgraph. G_4 has two components H_1 and H_2 .

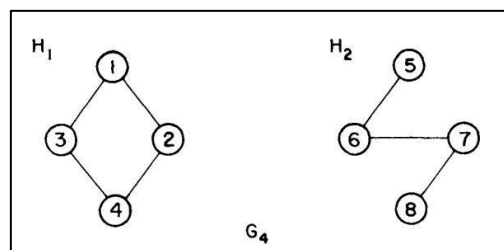


Fig.5.7. A graph with 2 connected components

- **Connected acyclic graph:** A tree is a connected acyclic (i.e., has no cycles) graph.
- **Strongly connected:** A directed graph G is said to be strongly connected if for every pair of distinct vertices v_i, v_j in $V(G)$ there is a directed path from v_i to v_j and also from v_j to v_i . The graph G_3 is not strongly connected as there is no path from v_3 to v_2 . A strongly connected component is a maximal subgraph that is strongly connected. G_3 has two strongly connected components.

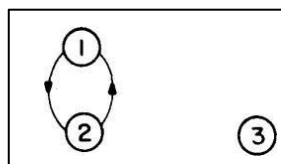


Fig.5.8. Strongly connected components of G_3 in 5.5.

- **Degree:** The degree of a vertex is the number of edges incident to that vertex. The degree of vertex 1 in G_1 is 3. In case G is a directed graph, we define the *in-degree* of a vertex v to be the number of edges for which v is the head. The *out-degree* is defined to be the number of edges for which v is the tail. Vertex 2 of G_3 has in-degree 1, out-degree 2 and degree 3. If d_i is the degree of vertex i in a graph G with n vertices and e edges, then it is easy to see that $\sum d_i = 2e$.

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* $\in$ *Graph*, *v*, *v*₁, and *v*₂ $\in$ *Vertices*

<i>Graph</i> Create()	::=	return an empty graph.
<i>Graph</i> InsertVertex(<i>graph</i> , <i>v</i>)	::=	return a graph with <i>v</i> inserted. <i>v</i> has no incident edges.
<i>Graph</i> InsertEdge(<i>graph</i> , <i>v</i> ₁ , <i>v</i> ₂)	::=	return a graph with a new edge between <i>v</i> ₁ and <i>v</i> ₂ .
<i>Graph</i> DeleteVertex(<i>graph</i> , <i>v</i>)	::=	return a graph in which <i>v</i> and all edges incident to it are removed.
<i>Graph</i> DeleteEdge(<i>graph</i> , <i>v</i> ₁ , <i>v</i> ₂)	::=	return a graph in which the edge (<i>v</i> ₁ , <i>v</i> ₂) is removed. Leave the incident nodes in the graph.
<i>Boolean</i> IsEmpty(<i>graph</i>)	::=	if (<i>graph</i> == empty graph) return <i>TRUE</i> else return <i>FALSE</i> .
<i>List</i> Adjacent(<i>graph</i> , <i>v</i>)	::=	return a list of all vertices that are adjacent to <i>v</i> .

Fig: Abstractdatatypegraph

Session 30:

What is Graph?

How to implement complete graph?

What is path?

What is cycle?

What is connected graph?

How to find degree?

What is ADT graph?

5.1.3 Graphrepresentations

Therearethreewaysofrepresentation

1. AdjacencyMatrixrepresentation
2. AdjacencyListrepresentation
3. AdjacencyMulti-listrepresentation
4. WeightedEdges

1.AdjacencyMatrix:

Let $G = (V, E)$ be a graph with n vertices, $n \geq 1$. The adjacency matrix of G is a 2-dimensional $n \times n$ array, say A , with the property that $A(i, j) = 1$ iff the edge (v_i, v_j) (for a directed graph) is in $E(G)$. $A(i, j) = 0$ if there is no such edge in G . examples forthe adjacencymatricesforthe areshown infigure5.9.

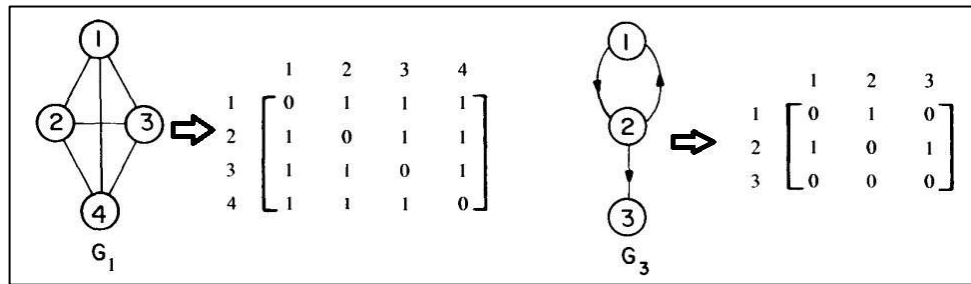


Fig.5.9.Adjacencymatrixrepresentationofagraph.

2. Adjacency Lists:

In this representation the n rows of the adjacency matrix are represented as n linked lists. There is one list for each vertex in G . The nodes in list i represent the vertices that are adjacent from vertex i . Each node has at least two fields: VERTEX and LINK. The VERTEX fields contain the indices of the vertices adjacent to vertex i . The adjacency lists for G_1 and G_3 are shown in figure 5.10. Each list has a head node. The head nodes are sequential providing easy random access to the adjacency list for any particular vertex. In the case of an undirected graph with n vertices and e edges, this representation requires n head nodes and $2e$ list nodes. Each list node has 2 fields.

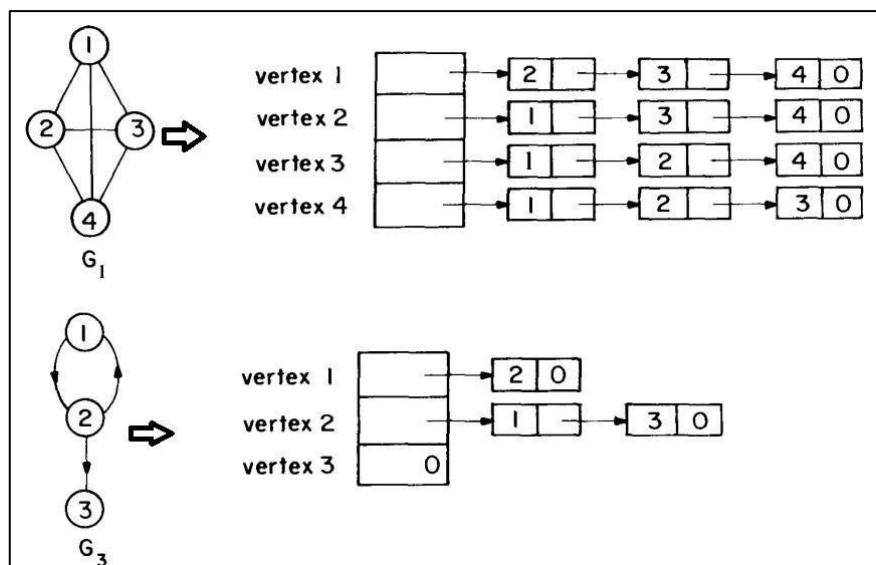


Fig.5.10.AdjacencyListrepresentationofagraph.

3. 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. Multi-lists are the lists in which nodes may be shared among several lists. The node structure is given in the Figure 5.11.

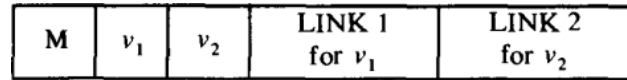


Fig.5.11. Node structure of Adjacency Multi-list.

M is a one-bit mark field that may be used to indicate whether or not the edge has been examined. Figure 5.12 shows the adjacency multi-lists for G_1

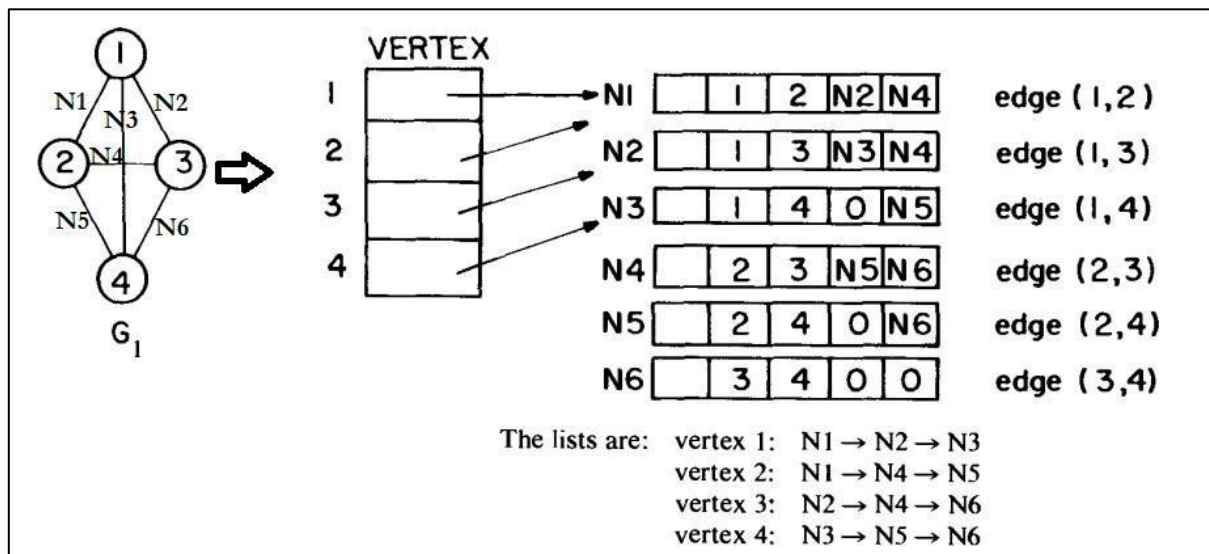


Figure 5.12 Adjacency Multi-lists for G_1

4. 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.

Session 31:

How to represent graph?

What is adjacency matrix?

What is adjacency list?

What is adjacency multi list?

What is weighted edge?

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, all 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.

Program to Check whether a given graph is connected or not using DFS method

```
#include <stdio.h>
int a[20][20], n, i, j, visited[20], count;
/*****Connectivity using DFS*****/
// to insert the vertices which are visited void
dfs(int v)
{
    int i;
    visited[v] = 1;
    for (i = 1; i <= n; i++) {
        if (a[v][i] && !visited[i]) {
            printf("\n %d->%d", v, i);
            count++;
            dfs(i);
        }
    }
}

void main() {
    int v, choice;
    printf("\n Enter the number of cities: ");
    scanf("%d", &n);
    printf("\n Enter graph data in matrix form:\n");
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            scanf("%d", &a[i][j]);
    while (1) {
        printf("\n1.Test for connectivity");
        printf("\n2.Exit");
        printf("\nEnter Your Choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                for (i = 1; i <= n; i++) {
                    visited[i] = 0;
                }
            }
    }
}
```

```

printf("ENTER the source vertex");
scanf("%d", &v);

scanf("%d", &v);

dfs(v);

        if (count == n - 1)
printf("\nGraph is connected\n");
        else
printf("\n Graph is not connected");

        count = 0;
        break;

        case 2:
            return;
        default:
printf("\nEnter proper Choice");
        }
    }
}

```

Output

```

Enter the number of cities: 4

Enter graph data in matrix form:
0 1 0 1
0 0 1 0
0 0 0 0
0 0 1 0

1.Test for connectivity
2.Exit
Enter Your Choice: 1
ENTER the source vertex1

1->2
2->3
1->4
Graph is connected

1.Test for connectivity
2.Exit
Enter Your Choice:

```

5.2.2 BreadthFirstSearch

breadthfirstsearchresemblesalevelordertreetraversal.Breadthfirstsearchstarts 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.

Program to print all the nodes reachable from a given starting node in a digraph using BFS method

```
#include<stdio.h>
int a[20][20], q[20],
visited[20]; int n, i, j, f=0, r=-1;

void create_graph() // Create the Digraph using Adjacency matrix
{
    printf("\nEnter the number of cities:"); scanf("%d", &n);
```

```

printf("\nEnter graph data in matrix form:\n"); for(i=1; i<=n; i++)
    for(j=1; j<=n; j++)
        scanf("%d", &a[i][j]); // read adjacency matrix

return;
}

void bfs(int v) // Reachability using Breadth First Search
{
    for(i=1; i<=n; i++)
        if(a[v][i] && !visited[i]) // check weather node
            isvisited[q[++r]=i]; // if not add it Queue
    if(f<=r) // check for non empty Queue
    {
        visited[q[f]]=1; // set visited status for front node of
        Queue bfs(q[f++]); // recursive call BSF
    }
} // end of BSF

void main()
{
    int v,
    choice; while
    (1)
    {
        printf("\n1. Create a Digraph of N cities using Adjacency Matrix");
        printf("\n2. Print all the nodes reachable from a given starting node in a digraph using BFS method");
        printf("\n3. Exit");
        printf("\nEnter Your Choice:"); scanf("%d", &choice); switch(choice)
        {
            case 1: create_graph();

```

```

break;

case 2: printf("Enter the source vertex:"); scanf("%d", &v);
        if((v < 1) || (v > n)) // check for valid source
            entry printf("\nEnter a valid source vertex");
        else // if valid begin test for reachability
        {
            for(i = 1; i <= n; i++) // begin with assuming all cities are not visited
                visited[i] = 0;
            visited[v] = 1; // source is visited
            calBFSTo check reachability
            printf("The reachable nodes from node %d:\n", v);
            for(i = 1; i <= n; i++) // display reachable cities from the source city
                if(visited[i] && i != v)
                    printf("node %d\n", i);
        }
        break;
case 3: return;
default: printf("\nInvalid Choice");
} // end of switch
} // end of while
} // end of main

```

output:

```

Enter the number of cities: 4

Enter graph data in matrix form:
0 1 0 1
0 0 1 0
0 0 0 0
0 0 1 0

1. Create a Digraph of N cities using Adjacency Matrix
2. Print all the nodes reachable from a given starting node in a digraph using BFS method
3. Exit
Enter Your Choice: 2
Enter the source vertex: 1
The reachable nodes from node 1:
node 2
node 3
node 4

1. Create a Digraph of N cities using Adjacency Matrix
2. Print all the nodes reachable from a given starting node in a digraph using BFS method
3. Exit
Enter Your Choice: 2
Enter the source vertex: 2
The reachable nodes from node 2:
node 3

1. Create a Digraph of N cities using Adjacency Matrix
2. Print all the nodes reachable from a given starting node in a digraph using BFS method
3. Exit
Enter Your Choice:

```

5.2.3 Connected Components:

```

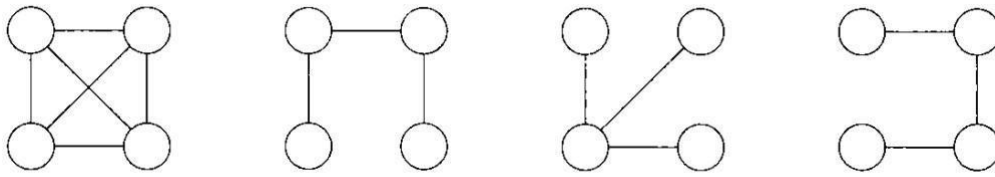
void connected(void)
{
    /* determine the connected components of a graph */
    int i;
    for (i = 0; i < n; i++)
        if(!visited[i]) {
            dfs(i);
            printf("\n");
        }
}

```

Connected components

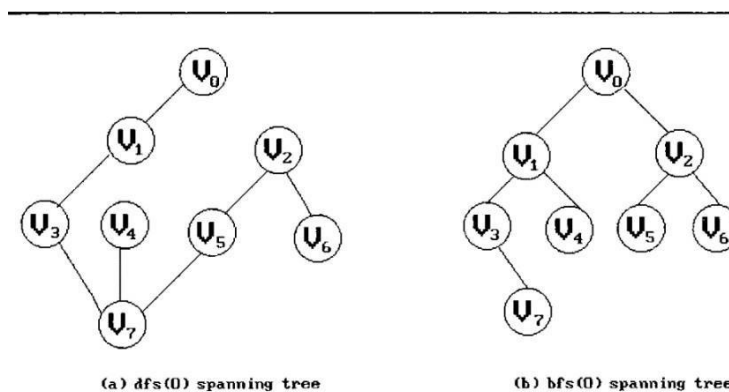
5.2.4 Spanning Trees

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.

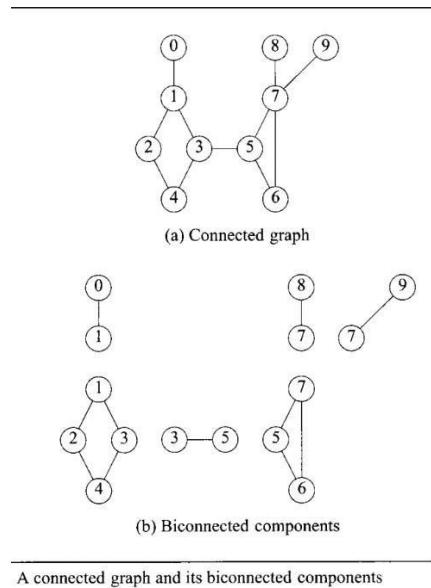


dfs and bfs spanning trees for graph of Figure 6.19

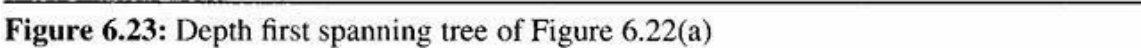
A spanning tree is a minimal subgraph G' of G such that $V(G') = V(G)$ and G' is connected. We define a minimal subgraph as one with the fewest number of edges. Any connected graph with n vertices must have at least $n - 1$ edges, and all connected graphs with $n - 1$ edges are trees. Therefore, we conclude 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 . For example, the graph of Figure (a) contains the six biconnected components shown in Figure (b).



- We can find the biconnected components of a connected undirected graph, G , by using any depth first spanning tree of G . For example, the function call `dfs(3)` applied to the graph of Figure (a) above produces the spanning tree of Figure 6.23(a). We have redrawn the tree in Figure 6.23(b) to better reveal its tree structure.
- The numbers outside the vertices in either figure give the sequence in which the vertices are visited during the depth first search. We call this number the depth first number, or dfn , of the vertex. For example, $dfn(3) = 0$, $dfn(0) = 4$, and $dfn(9) = 8$. Notice that vertex 3, which is an ancestor of both vertices 0 and 9, has a lower dfn than either of these vertices. Generally, if u and v are two vertices, and u is an ancestor of v in the depth first spanning tree, then $dfn(u) < dfn(v)$.



- $$low(u) = \min\{dfn(u), \min\{low(w) \mid w \text{ is a child of } u\}, \min\{dfn(w) \mid (u, w) \text{ is a back edge}\}\}$$

38

Vertex	0	1	2	3	4	5	6	7	8	9
<i>dfn</i>	4	3	2	0	1	5	6	7	9	8
<i>low</i>	4	3	0	0	0	5	5	7	9	8

figure 6.24: *dfn* and *low* values for *dfs* spanning tree with *root* = 3

We can easily modify *dfs* to compute *dfn* and *low* for each vertex of a connected undirected graph. The program is given below.

```
#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*

```

void bicon(int u, int v)
{
    /* compute dfn and low, and output the edges of G by their
    biconnected components, v is the parent (if any) of the u
    (if any) in the resulting spanning tree. It is assumed that
    all entries of dfn[] have been initialized to -1, num has
    been initialized to 0, and the stack has been set to empty
    */
    node_pointer ptr;
    int w,x,y;
    dfn[u] = low[u] = num++;
    for (ptr = graph[u]; ptr; ptr = ptr->link) {
        w = ptr->vertex;
        if (v != w && dfn[w] < dfn[u])
            add(&top,u,w); /* add edge to stack */
        if (dfn[w] < 0) { /* w has not been visited */
            bicon(w,u);
            low[u] = MIN2(low[u],low[w]);
            if (low[w] >= dfn[u]) {
                printf("New biconnected component: ");
                do { /* delete edge from stack */
                    delete(&top, &x, &y);
                    printf(" <%d,%d>",x,y);
                } while (!(x == u) && (y == w));
                printf("\n");
            }
        }
        else if (w != v) low[u] = MIN2(low[u],dfn[w]);
    }
}

```

Program 6.6: Biconnected components of a graph

Session 32:

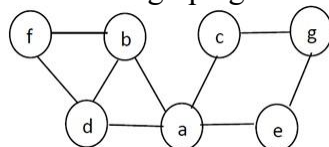
What is DFS?

What is BFS?

What is spanning tree?

Question bank

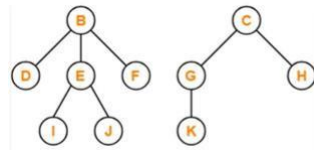
1. Design an algorithm to traverse a graph using Depth First Search (DFS). Apply DFS for the graph given below.



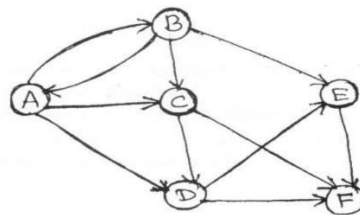
2. Construct a binary tree from the Post-order and In-order sequence given below In-order: GDHBAEICF Post-order: GHDBIEFCA
3. Define selection tree. Construct min winner tree for the runs of a game given below. Each run consists of values of players. Find the first 5 winners.

10	9	20	6	8	9	90	17
15	20	20	15	15	11	95	18
16	38	30	25	50	16	99	20
			28				

- Define Binary Search tree. Construct a binary search tree (BST) for the following elements: 100, 85, 45, 55, 120, 20, 70, 90, 115, 65, 130, 145. Traverse using in-order, pre-order, and post-order traversal techniques. Write recursive C functions for the same.
- Define Forest. Transform the given forest into a Binary tree and traverse using inorder, preorder and postorder traversal.



- Define the Disjoint set. Consider the tree created by the weighted union function on the sequence of unions: union(0,1), union(2,3), union(4,5), union(6,7), union(0,2), union(4,6), and union(0,4). Process the simple find and collapsing find on eight finds and compare which find is efficient.
- What is a graph? Give the matrix and adjacency list representation of graphs.
- Write an algorithm for breadth first search and depth first search.
- Define graph. Write the difference between graph and trees. For the given graph, show the adjacency matrix and adjacency list representation of the graph given below.



- What are the methods used for traversing a graph? Explain any one with example.
- What is a binary search tree? Write algorithm to implement for recursive search or iterative search for a binary search tree.
- Write a function to insert an item into an ordered binary search tree. For the given data, draw a binary search tree and show the array and linked representation of the same 100, 85, 45, 55, 110, 20, 70, 65.
- Define binary search tree. Write the recursive search and iterative search algorithm for a binary search tree.
- What is selection tree explain with an example?