UNIT – 6 : TREES – 2, GRAPHS

Introduction

The first recorded evidence of the use of graphs dates back to 1736 when Euler used them to solve the now classical Koenigsberg bridge problem.Some of the applications of graphs are: analysis of electrical circuits, finding shortest routes, analysis of project planning, identification of chemical compounds, statistical mechanics, genetics, cybernetics, linguistics, social sciences, etc. Indeed, it might well be said that of all mathematical structures, graphs are the most widely used.

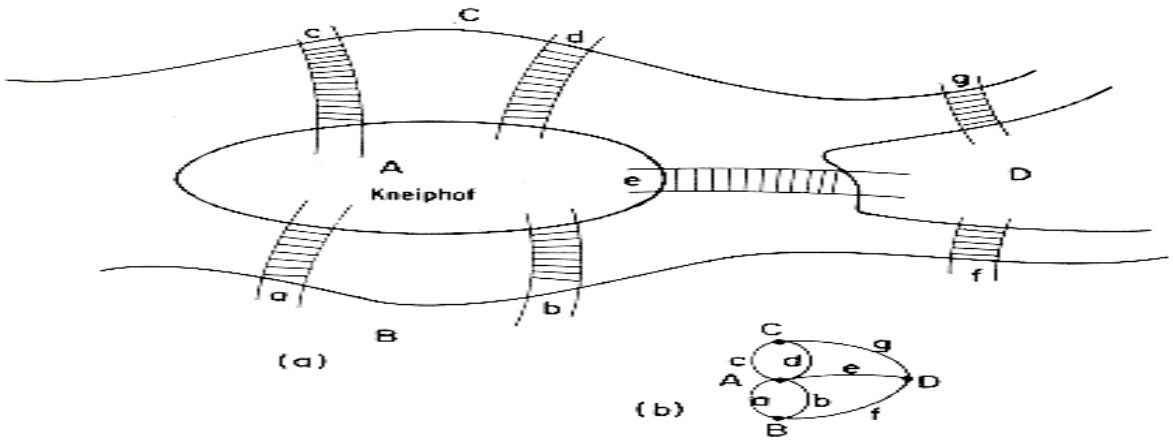


Figure 6.1 Section of the river Pregal in Koenigsberg and Euler's graph.

Definitions and Terminology

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 <v_2, v_1> and <v_1, v_2> represent two different edges. Figure 6.2 shows three graphs G_1, G_2 and G_3.

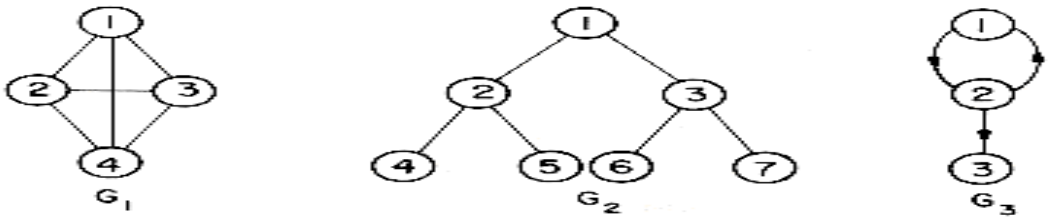


Figure 6.2 Three sample graphs.

The graphs G_1 and G_2 are undirected. G_3 is a directed graph.

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

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

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

Note that the edges of a directed graph are drawn with an arrow from the tail to the head. The graph G_2 is also a tree while the graphs G_1 and G_3 are not. Trees can be defined as a special case of graphs,

In addition, since $E(G)$ is a set, a graph may not have multiple occurrences of the same edge. When this restriction is removed from a graph, the resulting data object is referred to as a multigraph. The data object of figure 6.3 is a multigraph which is not a graph.

The number of distinct unordered pairs (v_i,v_j) with v_i v_j in a graph with n vertices is $n(n - 1)/2$. This is the maximum number of edges in any n vertex undirected graph.

An n vertex undirected graph with exactly $n(n - 1)/2$ edges is said to be *complete*. G_1 is the complete graph on 4 vertices while G_2 and G_3 are not complete graphs. In the case of a directed graph on n vertices the maximum number of edges is $n(n - 1)$.

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 $<v_1,v_2>$ 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 $<v_1,v_2>$ is incident to v_1 and v_2. In G_3 the edges incident to vertex 2 are <1,2>, <2,1> and <2,3>.

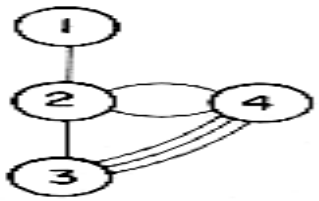


Figure 6.3 Example of a multigraph that is not a graph.

A *subgraph* of G is a graph G' such that $V(G')$ $V(G)$ and $E(G')$ $E(G)$. Figure 6.4 shows some of the subgraphs of G_1 and G_3.

A *path* from vertex v_p to vertex v_q in graph G is a sequence of vertices $v_p,v_{i1},v_{i2}, ...,v_{in},v_q$ such that $(v_p,v_{i1}),(v_{i1},v_{i2}), ...,(v_{in},v_q)$ are edges in $E(G)$. If G' is directed then the path consists of $<v_p,v_{i1}>,<v_i,v_{i2}>, ...,<v_{in},v_q>$, edges in $E(G')$.

The *length* of a path is the number of edges on it.

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 <3,2> is not in $E(G_3)$.

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.

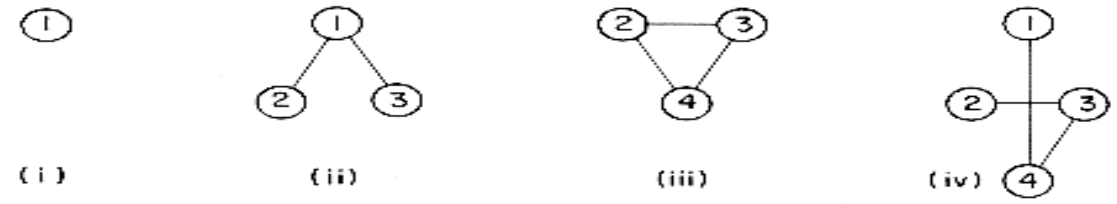
In an undirected graph, G, two vertices v_1 and v_2 are said to be *connected* if there is a path in G from v_1 to v_2 (since G is undirected, this means there must also be a path from v_2 to v_1). An undirected graph is said to be connected if for every pair of distinct vertices v_i, v_i in $V(G)$ there is a path from v_i to v_j in G.

Graphs G_1 and G_2 are connected while G_4 of figure 6.5 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 (see figure 6.5).

A *tree* is a connected acyclic (i.e., has no cycles) graph . 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.



(a) Some of the subgraphs of G_1

(b) Some of the subgraphs of G_3

Figure 6.4 (a) Subgraphs of G_1 and (b) Subgraphs of G_3

Figure 6.5 A graph with two connected components.

Figure 6.6 strongly connected components of G_3.

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 $e = (1/2) \sum_{i=1}^{n} d_i$.

6.1 Binary Search Trees

Characteristics

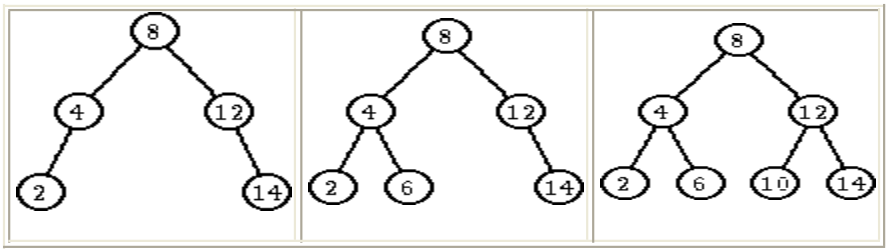
Trees in which the key of an internal node is greater than the keys in its left subtree and is smaller than the keys in its right subtree.

Search

search (tree,key)
IF empty tree THEN return not-found
IF key == value in root THEN return found
IF key > value in root THEN search (left-subtree, key)
search (right-subtree, key)
Time: O(depth of tree)

Insertion

	insert 6	insert 10

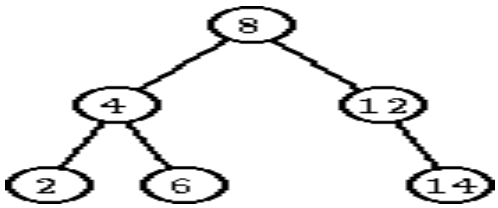


Deletion

The way the deletion is made depends on the type of node holding the key.

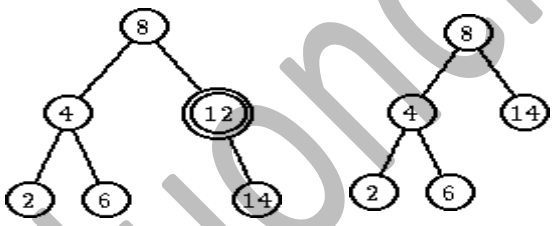
Node of degree 0

Delete the node



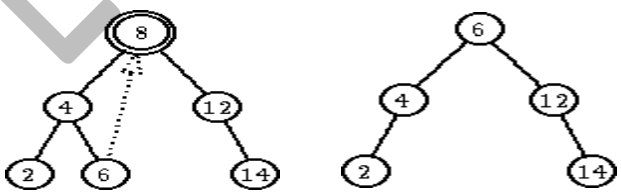
Node of degree 1

Delete the node, while connecting its predecessor to the successor.



Node of degree 2

Replace the node containing the deleted key with the node having the largest key in the left subtree, or with the node having the smallest key in the right subtree.

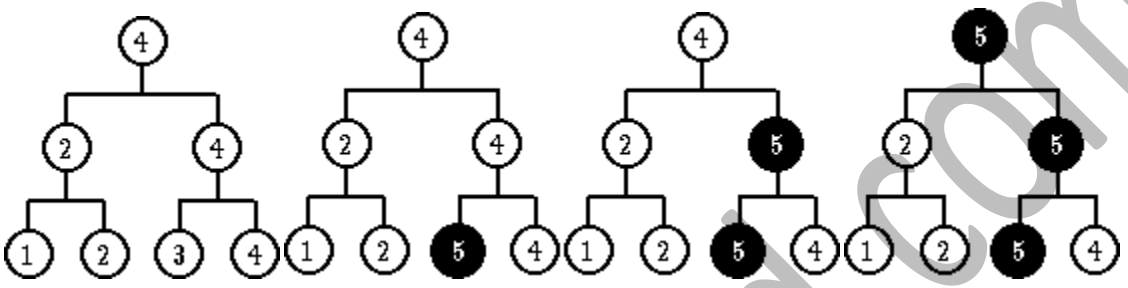


6.2. Selection Trees

A **selection tree** is a complete binary tree in which the leaf nodes hold a set of keys, and each internal node holds the ―winner‖ key among its children.

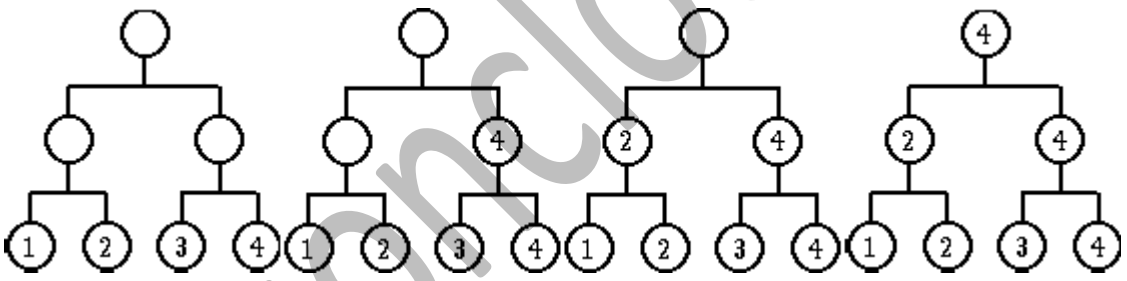
Modifying a Key

It takes O(log n) time to modify a selection tree in response to a change of a key in a leaf.



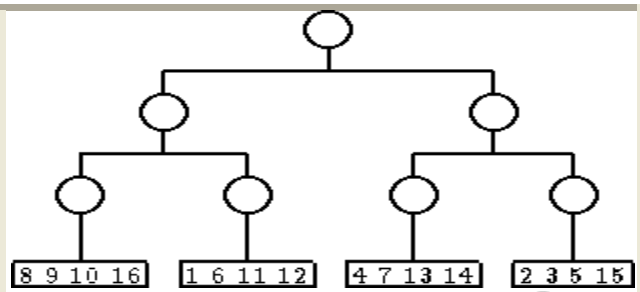
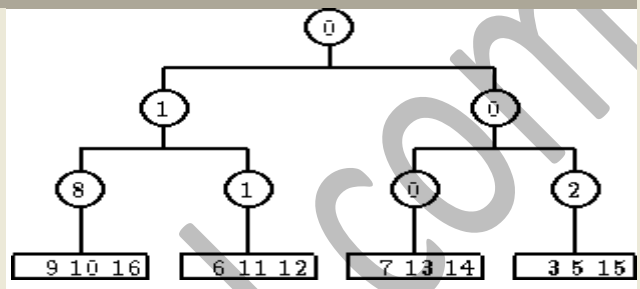
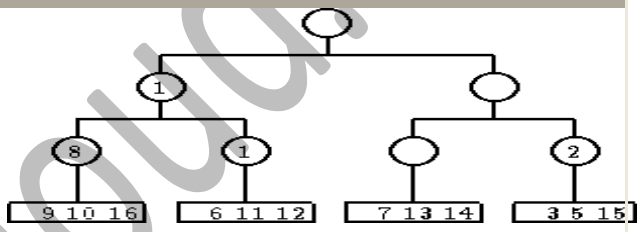
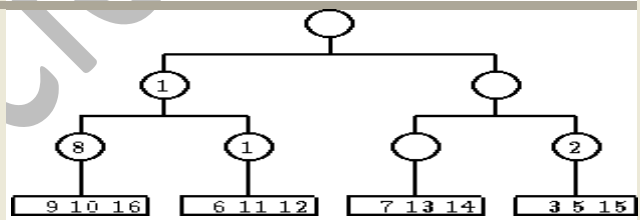
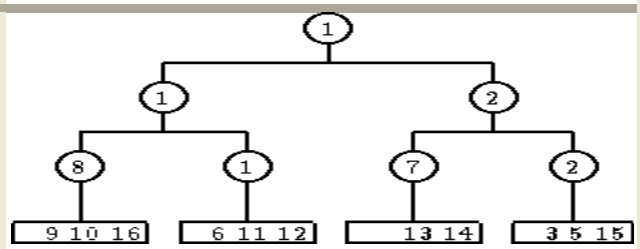
Initialization

The construction of a selection tree from scratch takes O(n) time by traversing it level-wise from bottom up.



Application: External Sort

Given a set of n values	16 9 10 8 6 11 12 1 4 7 14 13 2 15 5 3				n = 16
divide it into M chuncks,	16 9 10 8	6 11 12 1	4 7 14 13	2 15 5 3	M = 4
internally sort each chunk,	8 9 10 16	1 6 11 12	4 7 13 14	2 3 5 15	

construct complete binary tree of M leaves with the chunks attached to the leaves.	
Convert the tree into a selection tree with the keys being fed to the leaves from the chunks	
Remove the winner from the tree	
Feed to the empty leaf the next value from its corresponding chunk	
Adjust the selection tree to the change in the leaf	
Repeat the deletion sub process until all the values are consumed.	

- The algorithm takes $O(M \frac{n}{M} \log \frac{n}{M})$ time to internally sort the elements of the chunks, O(M) to initialize the selection tree, and O(n log M) to perform the selection sort. For M « n the total time complexity is O(n log n).
- To reduce I/O operations, inputs from the chunks to the selection tree should go through buffers.

6.3 Forests

The default interdomain trust relationships are created by the system during domain controller creation. The number of trust relationships that are required to connect *n* domains is *n* –1, whether the domains are linked in a single, contiguous parent-child hierarchy or they constitute two or more separate contiguous parent-child hierarchies.

When it is necessary for domains in the same organization to have different namespaces, create a separate tree for each namespace. In Windows 2000, the roots of trees are linked automatically by two-way, transitive trust relationships. Trees linked by trust relationships form a forest A single tree that is related to no other trees constitutes a forest of one tree.

The tree structures for the entire Windows 2000 forest are stored in Active Directory in the form of parent-child and tree-root relationships. These relationships are stored as trust account objects (class *trustedDomain*) in the System container within a specific domain directory partition. For each domain in a forest, information about its connection to a parent domain (or, in the case of a tree root, to another tree root domain) is added to the configuration data that is replicated to every domain in the forest. Therefore, every domain controller in the forest has knowledge of the tree structure for the entire forest, including knowledge of the links between trees. You can view the tree structure in Active Directory Domain Tree Manager.

6.4 Representation of Disjoint Sets

Set

In computer science, a **set** is an abstract data structure that can store certain values, without any particular order, and no repeated values. It is a computer implementation of the mathematical concept of a finite set. Some set data structures are designed for **static sets** that do not change with time, and allow only query operations — such as checking whether a given value is in the set, or enumerating the values in some arbitrary order. Other variants, called **dynamic** or **mutable sets**, allow also the insertion and/or deletion of elements from the set.

A set can be implemented in many ways. For example, one can use a list, ignoring the order of the elements and taking care to avoid repeated values. Sets are often implemented using various flavors of trees, tries, hash tables, and more.

A set can be seen, and implemented, as a (partial) associative array, in which the value of each key-value pair has the unit type. In type theory, sets are generally identified with their indicator function: accordingly, a set of values of type may be denoted by or . (Subtypes and subsets may be modeled by refinement types, and quotient sets may be replaced by setoids.)

Operations

Typical operations that may be provided by a static set structure *S* are

• element_of(*x*,*S*): checks whether the value *x* is in the set *S*.
• empty(*S*): checks whether the set *S* is empty.
• size(*S*): returns the number of elements in *S*.
• enumerate(*S*): yields the elements of *S* in some arbitrary order.
• pick(*S*): returns an arbitrary element of *S*.
• build(*x*1,*x*2,…,*xn*,): creates a set structure with values *x*1,*x*2,…,*xn*.

The enumerate operation may return a list of all the elements, or an iterator, a procedure object that returns one more value of *S* at each call.

Dynamic set structures typically add:

• create(*n*): creates a new set structure, initially empty but capable of holding up to *n* elements.

• add(*S*,*x*): adds the element *x* to *S*, if it is not there already.
• delete(*S*,*x*): removes the element *x* from *S*, if it is there.
• capacity(*S*): returns the maximum number of values that *S* can hold.
Some set structures may allow only some of these operations. The cost of each operation will depend on the implementation, and possibly also on the particular values stored in the set, and the order in which they are inserted. There are many other operations that can (in principle) be defined in terms of the above, such as:
• pop(*S*): returns an arbitrary element of *S*, deleting it from *S*.
• find(*S*, *P*): returns an element of *S* that satisfies a given predicate *P*.
• clear(*S*): delete all elements of *S*.
In particular, one may define the Boolean operations of set theory:
• union(*S*,*T*): returns the union of sets *S* and *T*.
• intersection(*S*,*T*): returns the intersection of sets *S* and *T*.
• difference(*S*,*T*): returns the difference of sets *S* and *T*.
• subset(*S*,*T*): a predicate that tests whether the set *S* is a subset of set *T*.
Other operations can be defined for sets with elements of a special type:
• sum(*S*): returns the sum of all elements of *S* (for some definition of "sum").
• nearest(*S*,*x*): returns the element of *S* that is closest in value to *x* (by some criterion).
In theory, many other abstract data structures can be viewed as set structures with additional operations and/or additional axioms imposed on the standard operations. For example, an abstract heap can be viewed as a set structure with a min(*S*) operation that returns the element of smallest value.

Implementations

Sets can be implemented using various data structures, which provide different time and space trade-offs for various operations. Some implementations are designed to improve the efficiency of very specialized operations, such as nearest or union. Implementations described as "general use" typically strive to optimize the element_of, add, and delete operation.

Sets are commonly implemented in the same way as associative arrays, namely, a self-balancing binary search tree for sorted sets (which has O(log n) for most operations), or a hash table for unsorted sets (which has O(1) average-case, but O(n) worst-case, for most operations). A sorted linear hash table[1] may be used to provide deterministically ordered sets.

Other popular methods include arrays. In particular a subset of the integers 1..*n* can be implemented efficiently as an *n*-bit bit array, which also support very efficient union and intersection operations. A Bloom map implements a set probabilistically, using a very compact representation but risking a small chance of false positives on queries. The Boolean set operations can be implemented in terms of more elementary operations (pop, clear, and add), but specialized algorithms may yield lower asymptotic time bounds. If sets are implemented as sorted lists, for example, the naive algorithm for union(*S*,*T*) will take code proportional to the length *m* of *S* times the length *n* of *T*; whereas a variant of the list merging algorithm will do the job in time proportional to *m*+*n*. Moreover, there are specialized set data structures (such as the union-find data structure) that are optimized for one or more of these operations, at the expense of others.

6.5 Counting Binary Trees:

Definition: A binary tree has a special vertex called its root. From this vertex at the top, the rest of the tree is drawn downward.Each vertex may have a left child and/or a right child.
Example. The number of binary trees with 1, 2, 3 vertices is:
Example. The number of binary trees with 4 vertices is:

Conjecture: The number of binary trees on n vertices is .
Proof: Every binary tree either:
! Has no vertices (x0) –or–
! Breaks down as one root vertex (x)
along with two binary trees beneath (B(x)2).
Therefore, the generating function for binary trees satisfies B(x) = 1 + xB(x)2. We conclude bn = 1 n+1#2n n $.
Another way: Find a recurrence for bn. Note:
b4 = b0b3 + b1b2 + b2b1 + b3b0.
In general, bn =%n−1 i=0 bibn−1−i .
Therefore, B(x) equals 1 + & n"1 '&n−1 i=0 bibn−1−I (xn = 1 + x & n"1 '&n−1 i=0 bibn−1−I (xn−1 = 1+x & k"0 '&k i=0 bibk−I (xk = 1+x '& k"0 bkxk ('& k"0 bkxk (= 1+xB(x)2.

6.6 The Graph Abstract Data Type:

In computer science, a **graph** is an abstract data type that is meant to implement the graph and hypergraph concepts from mathematics.A graph data structure consists of a finite (and possibly mutable) set of ordered pairs, called **edges** or **arcs**, of certain entities called **nodes** or **vertices**. As in mathematics, an edge (*x*,*y*) is said to **point** or **go from** *x* **to** *y*. The nodes may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some **edge value**, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

Algorithms

Graph algorithms are a significant field of interest within computer science. Typical higher-level operations associated with graphs are: finding a path between two nodes, like depth-first search and breadth-first search and finding the shortest path from one node to another, like Dijkstra's algorithm. A solution to finding the shortest path from each node to every other node also exists in the form of the Floyd–Warshall algorithm.A directed graph can be seen as a flow network, where each edge has a capacity and each edge receives a flow. The Ford–Fulkerson algorithm is used to find out the maximum flow from a source to a sink in a graph

Operations

The basic operations provided by a graph data structure *G* usually include:

- adjacent(*G*, *x*, *y*): tests whether there is an edge from node *x* to node *y*.
- neighbors(*G*, *x*): lists all nodes *y* such that there is an edge from *x* to *y*.
- add(*G*, *x*, *y*): adds to *G* the edge from *x* to *y*, if it is not there.
- delete(*G*, *x*, *y*): removes the edge from *x* to *y*, if it is there.
- get_node_value(*G*, *x*): returns the value associated with the node *x*.
- set_node_value(*G*, *x*, *a*): sets the value associated with the node *x* to *a*.

Structures that associate values to the edges usually also provide:

- get_edge_value(*G*, *x*, *y*): returns the value associated to the edge (*x*,*y*).
- set_edge_value(*G*, *x*, *y*, *v*): sets the value associated to the edge (*x*,*y*) to *v*.

Representations

Different data structures for the representation of graphs are used in practice:

- **Adjacency list** – Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices. This data structure allows the storage of additional data on the vertices.
- **Incidence list** – Vertices and edges are stored as records or objects. Each vertex stores its incident edges, and each edge stores its incident vertices. This data structure allows the storage of additional data on vertices and edges.
- **Adjacency matrix** – A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices.
- **Incidence matrix** – A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges. The entries indicate whether the vertex at a row is incident to the edge at a column.

6.7. RECOMMENDED QUESTIONS

1. Define forest. Explain the forest traversals.

2. Define Binary search tree. Explain with example?

3. Define a path in a tree, terminal nodes in a tree

4. Why it is said that searching a node in a binary search tree is efficient than that of a simple binary tree?

5. List the applications of set ADT.

6. What do you mean by disjoint set ADT

7. List the abstract operations in the set.

8. Define Graph. What is a directed graph & undirected graph?

9. What is a weighted graph? Define path in a graph?

10. Define outdegree of a graph & indegree of a graph?