1

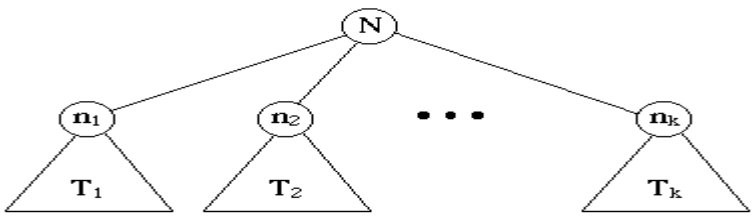
UNIT – 5 : TREES – 1

5.1 Introduction:

A *tree* is a finite set of one or more nodes such that: (i) there is a specially designated node called the *root*; (ii) the remaining nodes are partitioned into *n* 0 disjoint sets T_1, ...,T_n where each of these sets is a tree. T_1, ...,T_n are called the *subtrees* of the root. A tree structure means that the data is organized so that items of information are related by branches. One very common place where such a structure arises is in the investigation of genealogies.

AbstractDataType tree{
instances
A set of elements:
(1) empty or having a distinguished root element
(2) each non-root element having exactly one parent element operations
root()
degree()
child(k)
}
Some basic terminology for trees:

- Trees are formed from *nodes* and *edges*. Nodes are sometimes called *vertices*. Edges are sometimes called *branches*.
- Nodes may have a number of properties including *value* and *label*.
- Edges are used to relate nodes to each other. In a tree, this relation is called "parenthood."
- An edge {a,b} between nodes a and b establishes a as the *parent* of b. Also, b is called a *child* of a.
- Although edges are usually drawn as simple lines, they are really directed from parent to child. In tree drawings, this is top-to-bottom.
- **Informal Definition**: a *tree* is a collection of nodes, one of which is distinguished as "root," along with a relation ("parenthood") that is shown by edges.
- **Formal Definition**: This definition is "recursive" in that it defines tree in terms of itself. The definition is also "constructive" in that it describes how to construct a tree.
    1. A single node is a tree. It is "root."
    2. Suppose N is a node and T_1, T_2, ..., T_k are trees with roots n_1, n_2, ...,n_k, respectively. We can construct a new tree T by making N the parent of the nodes n_1, n_2, ..., n_k. Then, N is the root of T and T_1, T_2, ..., T_k are subtrees.



The tree T, constructed using k subtrees

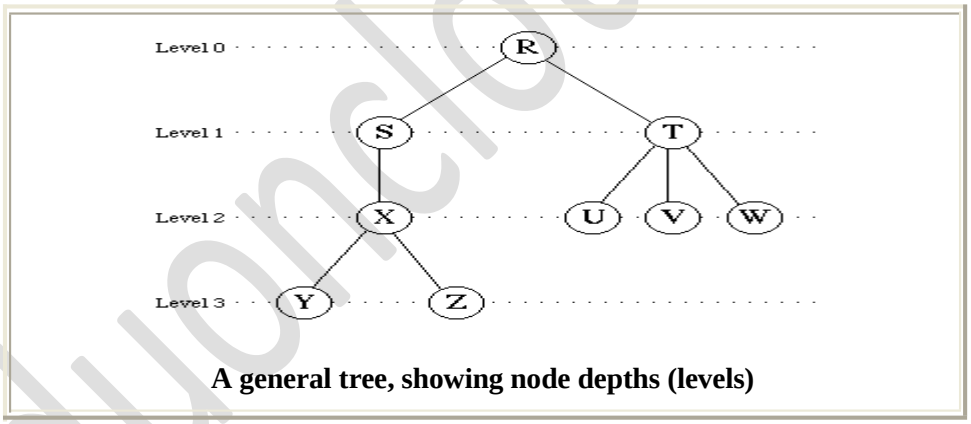
1

More terminology

- A node is either *internal* or it is a *leaf*.
- A *leaf* is a node that has no children.
- Every node in a tree (except root) has exactly one parent.
- The *degree of a node* is the number of children it has.
- The *degree of a tree* is the maximum degree of all of its nodes.

Paths and Levels

- **Definition**: A *path* is a sequence of nodes n_1, n_2, ..., n_k such that node n_i is the parent of node n_{i+1} for all 1 <= i <= k.
- **Definition**: The *length* of a path is the number of edges on the path (one less than the number of nodes).
- **Definition**: The *descendents* of a node are all the nodes that are on some path from the node to any leaf.
- **Definition**: The *ancestors* of a node are all the nodes that are on the path from the node to the root.
- **Definition**: The *depth* of a node is the length of the path from root to the node. The depth of a node is sometimes called its *level*.
- **Definition**: The *height of a node* is the length of the longest path from the node to a leaf.
- **Definition**: the *height of a tree* is the height of its root.



A general tree, showing node depths (levels)

In the example above:

- The nodes Y, Z, U, V, and W are leaf nodes.
- The nodes R, S, T, and X are internal nodes.
- The degree of node T is 3. The degree of node S is 1.
- The depth of node X is 2. The depth of node Z is 3.
- The height of node Z is zero. The height of node S is 2. The height of node R is 3.
- The height of the tree is the same as the height of its root R. Therefore the height of the tree is 3.
- The sequence of nodes R,S,X is a path.
- The sequence of nodes R,X,Y is not a path because the sequence does not satisfy the "parenthood" property (R is not the parent of X).

Representation of trees

1. List representation

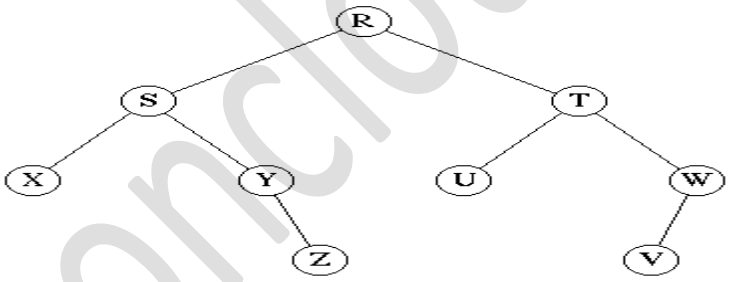
2. left child right sibling representation

3. Representation as a degree two tree

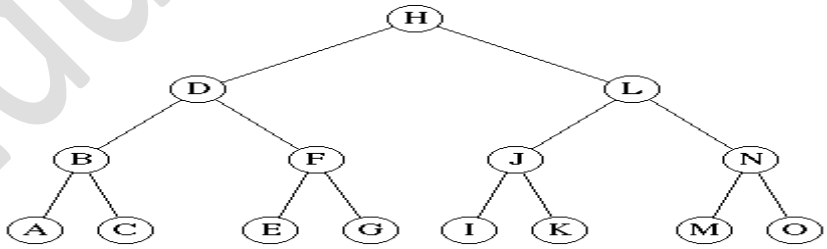
5.2 Binary Trees:

- **Definition**: A binary tree is a tree in which each node has degree of exactly 2 and the children of each node are distinguished as "left" and "right." Some of the children of a node may be empty.
- **Formal Definition**: A binary tree is:
  1. either empty, or
  2. it is a node that has a left and a right subtree, each of which is a binary tree.
- **Definition**: A *full binary tree* (FBT) is a binary tree in which each node has exactly 2 non-empty children or exactly two empty children, and all the leaves are on the same level. (Note that this definition differs from the text definition).
- **Definition**: A *complete binary tree* (CBT) is a FBT except, perhaps, that the deepest level may not be completely filled. If not completely filled, it is filled from left-to-right.
- A FBT is a CBT, but not vice-versa.

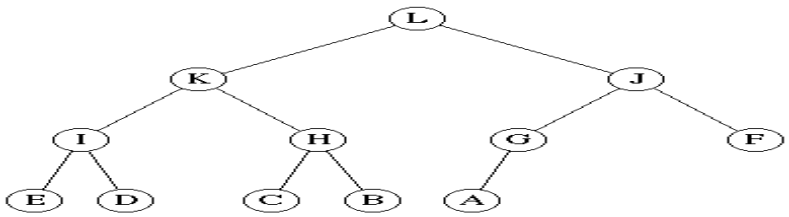
Examples of Binary Trees



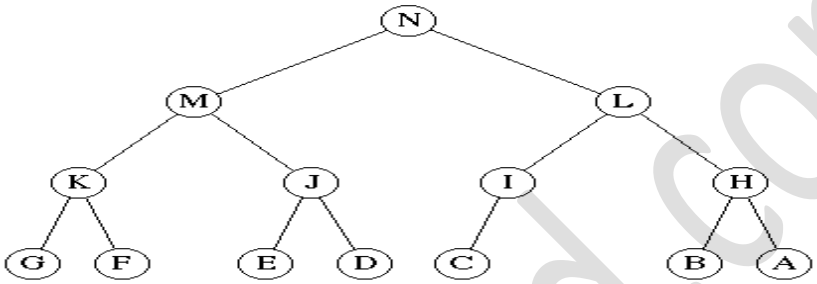
A Binary Tree. As usual, the empty children are not explicitly shown.



A Full Binary Tree. In a FBT, the number of nodes at level **i** is 2^i.



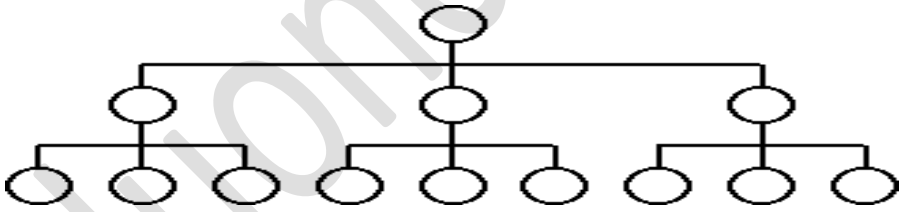
A Complete Binary Tree.



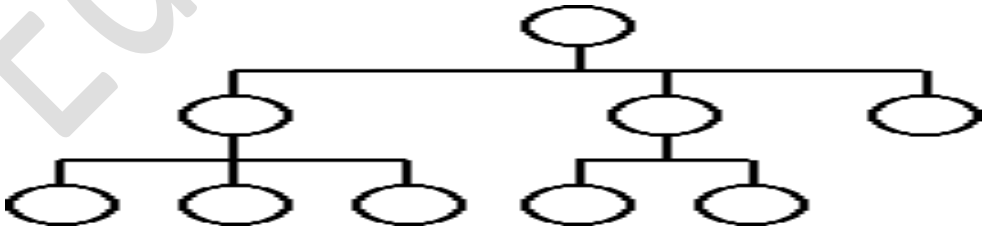
Not a Complete Binary Tree. The tree above is **not** a CBT because the deepest level is not filled from left-to-right.

Properties

Full tree **A tree with all the leaves at the same level, and all the non-leaves having the same degree.**



- **Complete Tree** A full tree in which the _last' elements are deleted.



- Level h of a full tree has d^{h-1} nodes.

  The first h levels of a full tree have nodes.

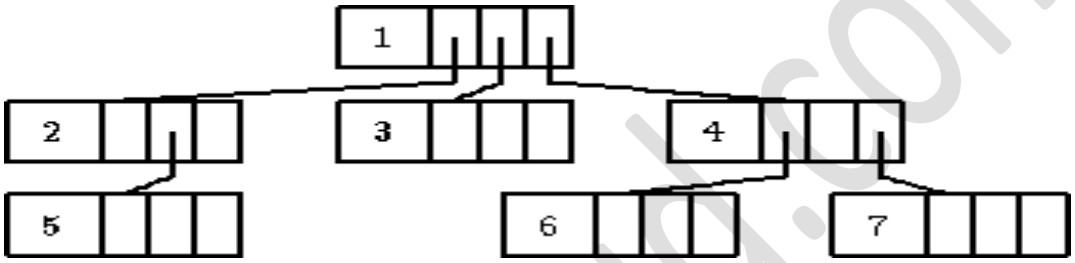
$$1 + d + d^2 + \cdots + d^{h-1} = \frac{d^h - 1}{d - 1}$$

- A tree of height h and degree d has at most $d^h - 1$ elements

$$N(h) = \begin{cases} dN(h-1) & \text{if } h > 1 \\ 1 & \text{if } h = 1 \end{cases}$$

Representations

1. Nodes consisting of a data field and k pointers

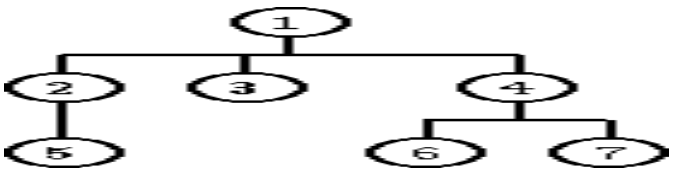


2. Nodes consisting of w data field and two pointers: a pointer to the first child, and a pointer to the next sibling.
3. A tree of degree k assumes an array for holding a complete tree of degree k, with empty cells assigned for missing elements.

5.3 Binary tree Traversals:

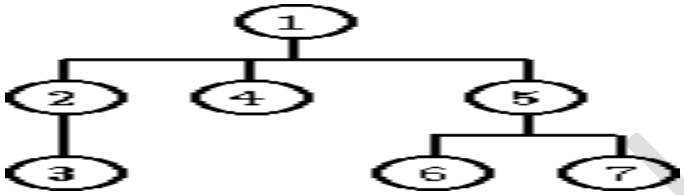
There are many operations that we often want to perform on trees. One notion that arises frequently is the idea of traversing a tree or visiting each node in the tree exactly once. A full traversal produces a linear order for the information in a tree. This linear order may be familiar and useful. When traversing a binary tree we want to treat each node and its subtrees in the same fashion. If we let *L, D, R* stand for moving left, printing the data, and moving right when at a node then there are six possible combinations of traversal: *LDR, LRD, DLR, DRL, RDL,* and *RLD*. If we adopt the convention that we traverse left before right then only three traversals remain: *LDR, LRD* and *DLR*. To these we assign the names inorder, postorder and preorder because there is a natural correspondence between these traversals and producing the infix, postfix and prefix forms of an expression.

Level order



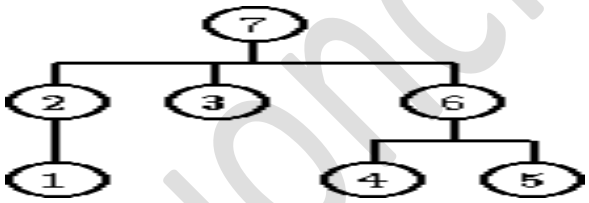
```
x := root()
if( x ) queue (x)
while( queue not empty ){
  x := dequeue()
  visit()
  i=1; while( i <= degree() ){
     queue( child(i) )
  }
}
```

Preorder




```
procedure preorder(x){
  visit(x)
  i=1; while( i <= degree() ){
     preorder( child(i) )
  }
}
```

Postorder




```
procedure postorder(x){
  i=1; while( i <= degree() ){
     postorder( child(i) )
  }
  visit(x)
}
```

Inorder

Meaningful just for binary trees.

```
procedure inorder(x){
  if( left_child_for(x) ) { inorder( left_child(x)  ) }
  visit(x)
  if( right_child_for(x) ) { inorder( right_child(x)  ) }
}
```

Usages for _visit‘: determine the height, count the number of elements .

5.4. Threaded Binary trees:

If we look carefully at the linked representation of any binary tree, we notice that there are more null links than actual pointers. As we saw before, there are $n + 1$ null links and $2n$ total links. A clever way to make use of these null links has been devised by A. J. Perlis and C. Thornton. Their idea is to replace the null links by pointers, called threads, to other nodes in the tree. If the RCHILD(*P*) is normally equal to zero, we will replace it *by a pointer to the node which would be printed after P when traversing the tree in inorder*. A null LCHILD link at node *P* is replaced *by a pointer to the node which immediately precedes node P in inorder*.

The tree *T* has 9 nodes and 10 null links which have been replaced by threads. If we traverse *T* in inorder the nodes will be visited in the order *H D I B E A F C G*. For example node *E* has a predecessor thread which points to B and a successor thread which points to *A*.

In the memory representation we must be able to distinguish between threads and normal pointers. This is done by adding two extra one bit fields LBIT and RBIT.

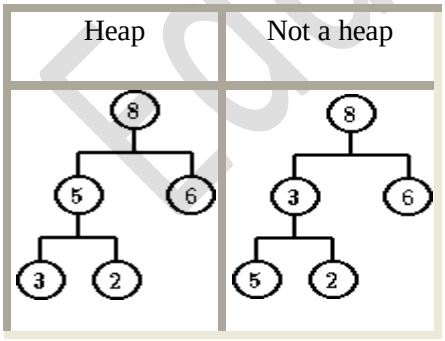
LBIT(*P*) =1 if LCHILD(*P*) is a normal pointer
LBIT(*P*) = 0 if LCHILD(*P*) is a thread
RBIT(*P*) = 1 if RCHILD(*P*) is a normal pointer
RBIT(*P*) = 0 if RCHILD(*P*) is a thread

5.5. Heaps

A **heap** is a complete tree with an ordering-relation R holding between each node and its descendant.

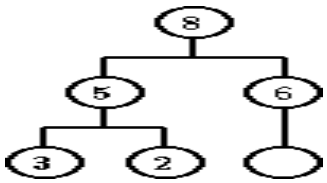
Examples for R: smaller-than, bigger-than

Assumption: In what follows, R is the relation _bigger-than‘, and the trees have degree 2.

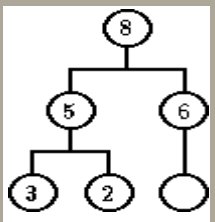
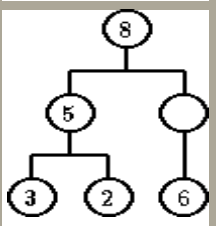
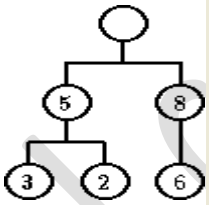


Adding an Element

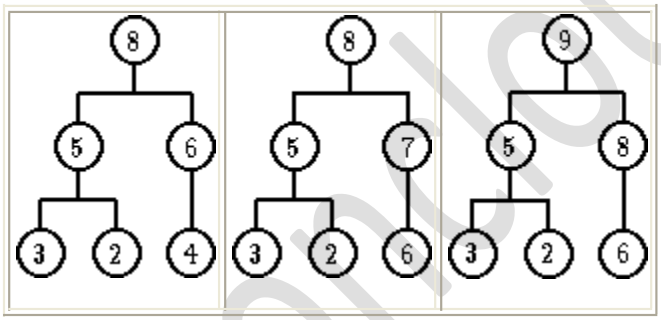
1. Add a node to the tree



2. Move the elements in the path from the root to the new node one position down, if they are smaller than the new element

new element	4	7	9
modified tree			

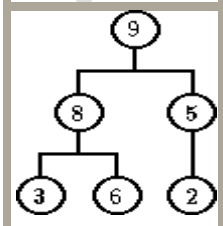
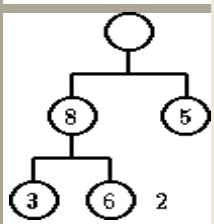
3. Insert the new element to the vacant node



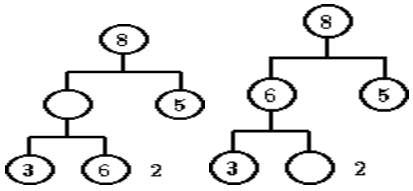
4. A complete tree of n nodes has depth $\lceil \log n \rceil$, hence the time complexity is O(log n)

Deleting an Element

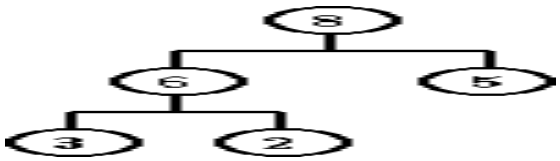
1. Delete the value from the root node, and delete the last node while saving its value.

before	after
	

2. As long as the saved value is smaller than a child of the vacant node, move up into the vacant node the largest value of the children.



3. Insert the saved value into the vacant node



4. The time complexity is O(log n)

Initialization:

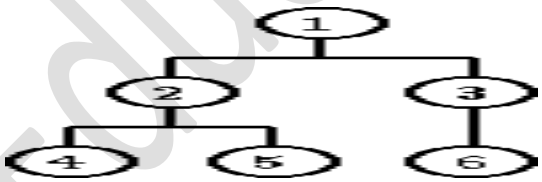
Brute Force

Given a sequence of n values e_1, ..., e_n, repeatedly use the insertion module on the n given values.

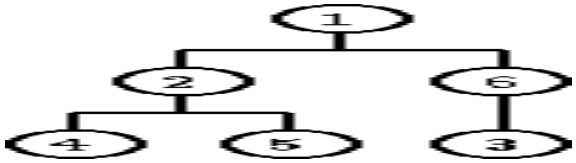
- Level h in a complete tree has at most $2^{h-1} = O(2^n)$ elements
- Levels 1, ..., h - 1 have $2^0 + 2^1 + \ldots + 2^{h-2} = O(2^h)$ elements
- Each element requires O(log n) time. Hence, brute force initialization requires O(n log n) time.

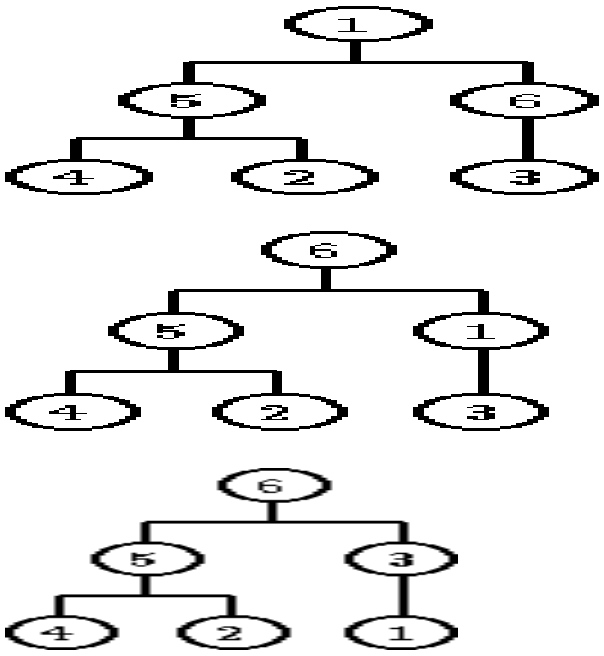
Efficient

- Insert the n elements e_1, ..., e_n into a complete tree



- For each node, starting from the last one and ending at the root, reorganize into a heap the subtree whose root node is given. The reorganization is performed by interchanging the new element with the child of greater value, until the new element is greater than its children.





- The time complexity is O(0 * (n/2) + 1 * (n/4) + 2 * (n/8) + . . .+ (log n) * 1) = O(n(0 .2^{-1} + 1 .2^{-2} + 2 .2^{-3} + . . .+ (log n) .2$^{-\log n}$)) = O(n)

  since the following equalities holds. $\sum_{k=1}^{\infty}(k-1)2^{-k} = 2[\sum_{k=1}^{\infty}(k-1)2^{-k}] - [\sum_{k=1}^{\infty}(k-1)2^{-k}] = [\sum_{k=1}^{\infty}k2^{-k}] - [\sum_{k=1}^{\infty}(k-1)2^{-k}] = \sum_{k=1}^{\infty}[k-(k-1)]2^{-k} = \sum_{k=1}^{\infty}2^{-k} = 1$

Applications:

Priority Queue A dynamic set in which elements are deleted according to a given ordering-relation.
Heap Sort Build a heap from the given set (O(n)) time, then repeatedly remove the elements from the heap (O(n log n)).

5.6. RECOMMENDED QUESTIONS

1. Construct a binary tree for : ((6+(3-2)*5)^2+3)

2. What is threaded binary tree? Explain right in and left in threaded binary trees.

3. Define a tree. Write the procedure to convert general tree to binary tree.

4. Define degree of the node, leaves, root

5. Define internal nodes, parent node, depth and height of a tree

6. State the properties of a binary tree

7. What is meant by binary tree traversal? What are the different binary tree traversal techniques

8. State the merits & demerit of linear representation of binary trees.

9. Define right-in threaded tree & left-in threaded tree

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.

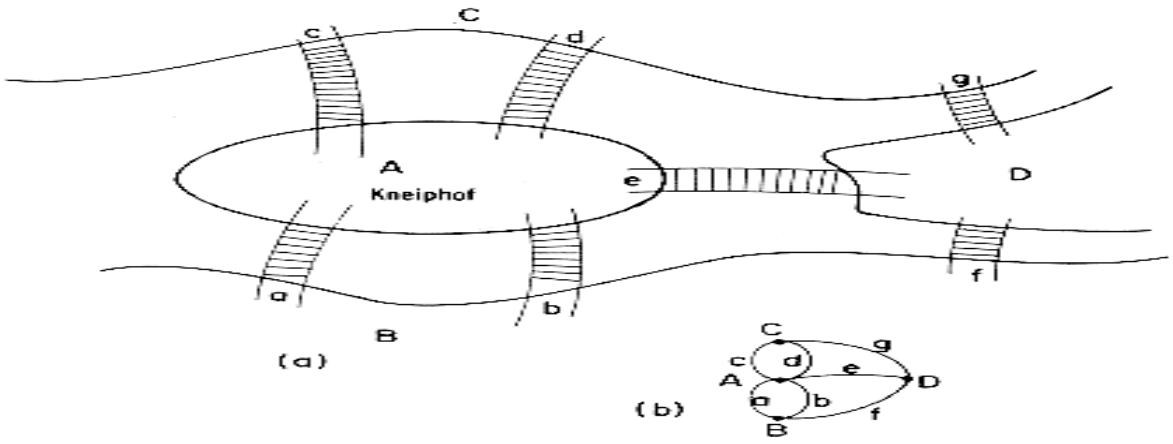


Figure 6.1 Section of the river Pregal in Koenigsberg and Euler's graph.

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.