**Data Structures & Algorithms**

**Lesson Plan 15 (Tree)**

**Duration: 1 hour 30 minutes (90 minutes 2 Lectures)**

**Objectives:**

* Trees
* Binary Trees
* Terminology
* Complete Binary Tree
* Extended Tree
* Tree Traversal
  + Pre-order Tree Traversal
  + In-order Tree Traversal
  + Post-order Tree Traversal
* Array Representation
* Linked List Representation

**Content:**

In this lecture we shall be discussing above mentioned points in detail.

**Methods:**

Using various examples objective of this lecture is met and C++ code is given as task

**Resources:**

Robert Lafore C++ Object Oriented Programming.

**Evaluation:**

Quiz was taken to evaluate students.

**Close:**

Class was concluded with question answer session

**Assignment:**

Assignment regarding topic was

|  |  |  |
| --- | --- | --- |
| **Time**  **Plan**  **(min)** | **Contents of Lecture** | **Lecture**  **Strategy**  **(Remarks)** |
|  | **Trees**  So far in this course we have seen so many types of data and their respective structures. We can easily observe that there is a kind of data we mostly dealt with, is of hierarchical structure. Like,   * Family tree * Organizational diagrams * Table of contents   So here is special data structure to deal with such kind of data type. These are known as trees. As per their name is concerned their structure is like tree, that is a root with so many branches. Though there are so many types and trees but for instant being in which we people are interested is binary tree. And ultimately we will see that is most elegant form of tree.  A tree is a finite set of one or more nodes such that   1.  There is a specially designated node called the root.  2. The remaining nodes are partitioned into disjoint subsets T1, T2… Tn  **Binary Trees**  When there are exactly 0, 1 or 2 nodes in tree it is called binary tree. Any expression can be represented by a binary tree. Like ((A\*B) - (C^D)) + (E/F)    **Terminology**  *Level* of a tree is the depth of three. For a binary tree there could be possible *2r* number of nodes where *r* is number of levels. |  |
| **Time**  **Plan**  **(min)** | **Contents of Lecture** | **Lecture**  **Strategy**  **(Remarks)** |
|  | 0  1  2  3  So the maximum possible number of nodes in a binary tree are *2r*, it means first level contains only 1 element (Root), next contains 2, further 4 and 8 so on.  **Complete Binary Tree**  If all the levels of a binary tree contain exactly *2r* number of nodes except the last level, tree is called complete binary tree. The above given tree is complete binary tree.  **Extended Tree**  If each node in a tree contains exactly 0 or 2 nodes, no node can have further sub node it is said to be an extended tree.  *Root*: the node at level 0 is designated as root.  *Branch Node*: a node with descendants called branch node.  *Terminal or leaf*: the nodes of last level, which are not ancestor or contain no descendant.  *Siblings*: the nodes at same level with same parent.  **Tree Traversal**  One of the most common operations performed on tree structure is that of traversals. This a procedure by which each node in the tree is visited (or processed) exactly once in systematic way. The meaning of “processed” depends on the nature of the application. |  |
| **Time**  **Plan**  **(min)** | **Contents of Lecture** | **Lecture**  **Strategy**  **(Remarks)** |
|  | For example, the tree in the fig. (i) represents an arithmetic expression. In this context, the processing of a node that represents an arithmetic operation would probably mean performing or executing that operation. There are three types of traversal (visiting each element) in binary tree.   1. Pre-order 2. In-order 3. Post-order 4. **Pre-order Tree Traversal**   In this traversal *ROOT* is processed first then *left sub-tree* and at end *right sub-tree*. The general algorithm can be written as:   * 1. If the tree is empty, then exit   2. Process the root node   3. Traverse the left sub-tree in in-order   4. Traverse the right sub-tree in in-order   Result: + - \* A B ^ C D / E F   1. **In-order Tree Traversal**   In this traversal *left sub-tree* is processed first then *ROOT* and at end *right sub-tree*. The general algorithm can be written as:   1. If the tree is empty, then exit 2. Traverse the left sub-tree in in-order 3. Process the root node 4. Traverse the right sub-tree in in-order   Result: (A \* B) – (C ^ D) + (E / F) |  |
| **Time**  **Plan**  **(min)** | **Contents of Lecture** | **Lecture**  **Strategy**  **(Remarks)** |
|  | 1. **Post-order Tree Traversal**   In this traversal *left sub-tree* is processed first then *right sub-tree* and *ROOT* at end. The general algorithm can be written as:   1. If the tree is empty, then exit 2. Traverse the left sub-tree in in-order 3. Traverse the right sub-tree in in-order 4. Process the root node   Result: A B \* C D ^ - E F / +  **Array Representation**  One way to represent the trees in the memory is array approach. An array's length is fixed at compile time, if we use an array to implement a tree we have to set a limit on the number of nodes we will permit in the tree. Our strategy is to fix the maximum height of the tree (H), and make the array big enough to hold any binary tree of this height (or less). We'll need an array of size (2H)-1. Here is the biggest binary tree of depth 3:  Lecture10Fig21  If we picked H=3 as our limit, then every tree we might build will be a subtree of this one - this is the key insight behind our implementation.  What we do now is assign each of nodes to a specific position in the array. This could be done any way you like, but a particular easy and useful way is: |  |
|  |  |  |
| **Time**  **Plan**  **(min)** | **Contents of Lecture** | **Lecture**  **Strategy**  **(Remarks)** |
|  | root of the tree (A): array position 1  root's left child (B): array position 2  root's right child (C): array position 3  ...  left child of node in array position K: array position 2K  right child of node in array position K: array position 2K+1  So D is in position 2\*2 (4), E is in position 2\*2+1 (5), F is in position 2\*3 (6), G is in position 2\*3+1 (7).This figure shows the array position associated with each node:  Lecture10Fig22  This particular arrangement makes it easy to move from a node to its children, just double the node's index (and add 1 to go right). It also makes it easy to go from a node to its parent: the parent of node I has index (I div 2).Using this strategy, a tree with N nodes does not necessarily occupy the first N positions in the array. For example, the tree:  Lecture10Fig23  Somehow we need to keep track of which array elements contain valid information. Two possibilities:   1. as usual, each node stores information saying which of its children exist 2. Each array position stores information saying if it is a valid node. |  |
| **Time**  **Plan**  **(min)** | **Contents of Lecture** | **Lecture**  **Strategy**  **(Remarks)** |
|  | However, if we restrict ourselves to complete trees, these problems go away. Because of the way we assigned nodes to positions, if there are N nodes in a complete tree, they will correspond to the first N positions of the array.  **Linked List Representation**  Since array representation is not very effective in memory management sense, so we go for dynamic memory management by means of linked list. In this way each node is consisted of three portions, one for information, second for left node address (pointer) and third for right node information. Since a binary tree consists of nodes that can have at most two offspring’s, an obvious linked representation of such a tree is expressed by the following definition in C.  struct node {    int info;  node \*leftpointer;  node \*rightpointer;  }; |  |
|  |  |  |