Course Code : MCS-208
Course Title : Data Structures and Algorithms

Last Date of Submission : 31st October, 2022 (for July session)

30th April, 2023 (for January session)

There are four questions in this assignment, which carry 80 marks. Each question carries 20 marks. Rest 20 marks are for viva voce. All algorithms should be written nearer to C programming language. You may use illustrations and diagrams to enhance the explanations, if necessary. Please go through the guidelines regarding assignments given in the Programme Guide for the format of presentation.

Question 1: (20 Marks) What are Sparse Matrices? Explain with example(s)

Ans.

What is a sparse matrix?

Sparse matrices are those matrices that have the majority of their elements equal to zero. In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

Why is a sparse matrix required if we can use the simple matrix to store elements?

There are the following benefits of using the sparse matrix -

Storage - We know that a sparse matrix contains lesser non-zero elements than zero, so less memory can be used to store elements. It evaluates only the non-zero elements.

Computing time: In the case of searching in sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

Representation of sparse matrix

Now, let's see the representation of the sparse matrix. The non-zero elements in the sparse matrix can be stored using triplets that are rows, columns, and values. There are two ways to represent the sparse matrix that are listed as follows -

- **Array representation**
- **Linked list representation**

Array representation of the sparse matrix

Representing a sparse matrix by a 2D array leads to the wastage of lots of memory. This is because zeroes in the matrix are of no use, so storing zeroes with non-zero elements is wastage of memory. To avoid such wastage, we can store only non-zero elements. If we store only non-zero elements, it reduces the traversal time and the storage space.

In 2D array representation of sparse matrix, there are three fields used that are named as -

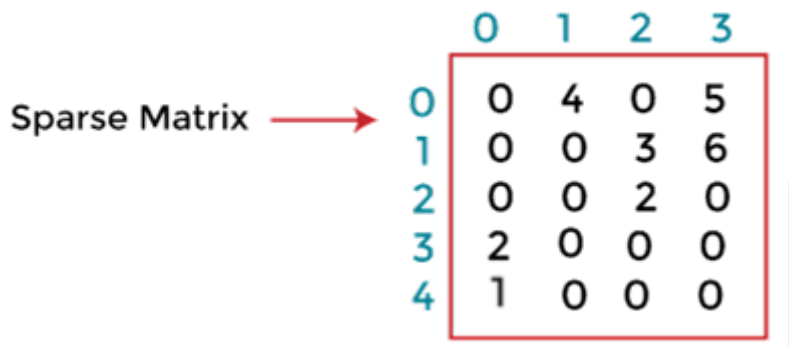
- **Row - It is the index of a row where a non-zero element is located in the matrix.**

- **Column - It is the index of the column where a non-zero element is located in the matrix.**
- **Value - It is the value of the non-zero element that is located at the index (row, column).**

Example -

Let's understand the array representation of sparse matrix with the help of the example given below -

Consider the sparse matrix -



In the above figure, we can observe a 5x4 sparse matrix containing 7 non-zero elements and 13 zero elements. The above matrix occupies 5x4 = 20 memory space. Increasing the size of matrix will increase the wastage space.

The tabular representation of the above matrix is given below -

Table Structure

Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
4	0	1
5	4	7

In the above structure, first column represents the rows, the second column represents the columns, and the third column represents the non-zero value. The first row of the table represents the triplets. The first triplet represents that the value 4 is stored at 0th row and 1st column. Similarly, the second triplet represents that the value 5 is stored at the 0th row and 3rd column. In a similar manner, all triplets represent the stored location of the non-zero elements in the matrix.

The size of the table depends upon the total number of non-zero elements in the given sparse matrix. Above table occupies 8x3 = 24 memory space which is more than the space occupied by the sparse matrix. So, what's the benefit of using the sparse matrix? Consider the case if the matrix is 8*8 and there are only 8 non-zero elements in the matrix, then the space occupied by the sparse matrix would be 8*8 = 64, whereas the space occupied by the table represented using triplets would be 8*3 = 24.

Question 2: (20 Marks) How many different traversals of a Binary Tree are possible? Explain them with example(s).

Ans.

Binary Tree

A binary tree is a finite collection of elements or it can be said it is made up of nodes. Where each node contains the left pointer, right pointer, and a data element. The root pointer points to the topmost node in the tree. When the binary tree is not empty, so it will have a root element and the remaining elements are partitioned into two binary trees which are called the left pointer and right pointer of a tree.

Traversing in the Binary Tree

Tree traversal is the process of visiting each node in the tree exactly once. Visiting each node in a graph should be done in a systematic manner. If search result in a visit to all the vertices, it is called a traversal. There are basically three traversal techniques for a binary tree that are,

1. Preorder traversal
2. Inorder traversal
3. Postorder traversal

1) Preorder traversal

To **traverse a binary tree in preorder**, following operations are carried out:

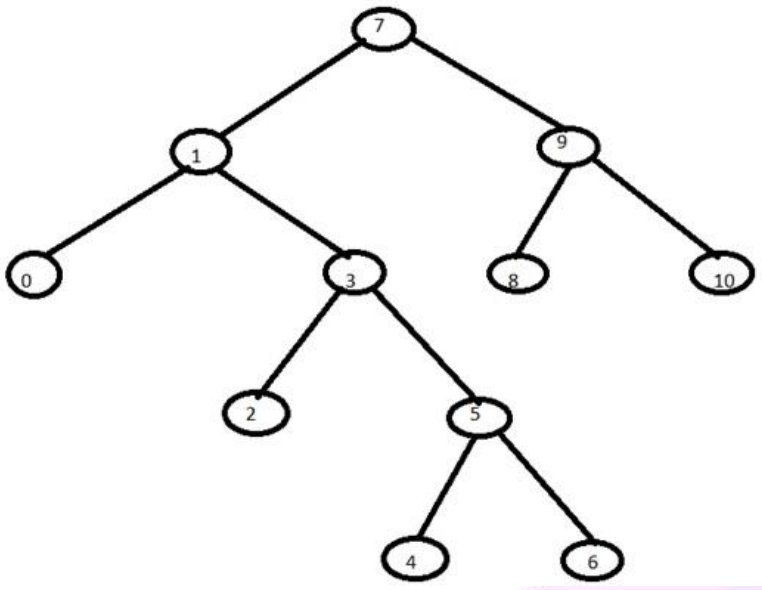
1. Visit the root.
2. Traverse the left sub tree of root.
3. Traverse the right sub tree of root.

Note: Preorder traversal is also known as NLR traversal.

Algorithm:

```
Algorithm preorder(t)
/*t is a binary tree. Each node of t has three fields:
lchild, data, and rchild.*/
{
        If t! =0 then
        {
                Visit(t);
                Preorder(t->lchild);
                Preorder(t->rchild);
        }
}
```

Example: Let us consider the given binary tree,



Therefore, the preorder traversal of the above tree will be: **7,1,0,3,2,5,4,6,9,8,10**

2) Inorder traversal

To traverse a binary tree in inorder traversal, following operations are carried out:

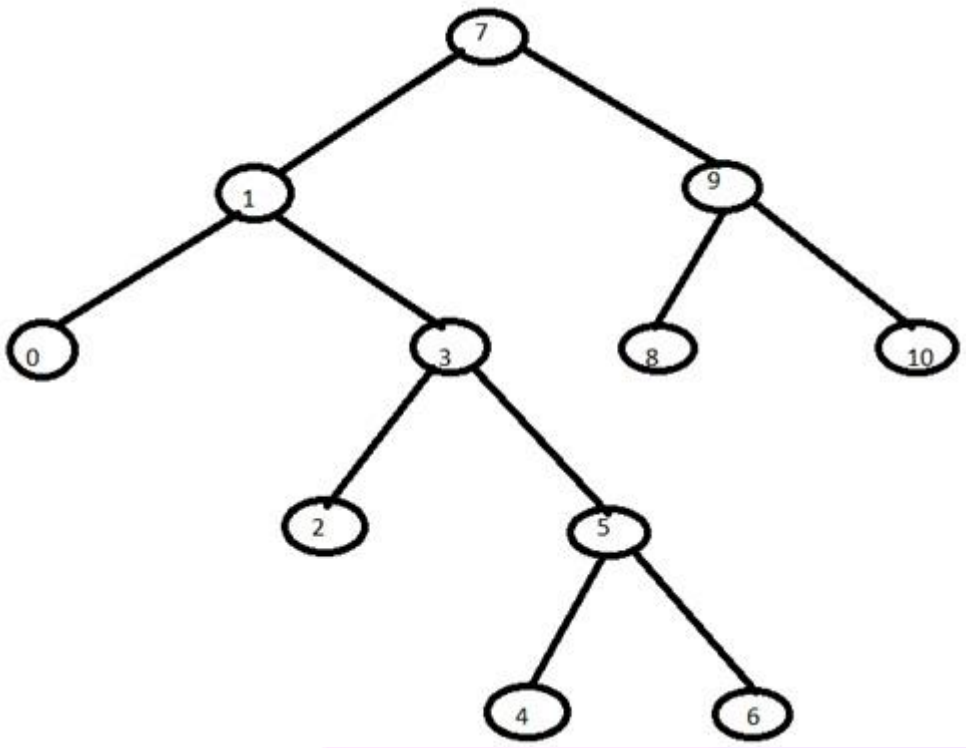
1. Traverse the left most sub tree.
2. Visit the root.
3. Traverse the right most sub tree.

Note: Inorder traversal is also known as LNR traversal.

Algorithm:

```
Algorithm inorder(t)


/*t is a binary tree. Each node of t has three fields:

lchild, data, and rchild.*/

{

        If t! =0 then

        {

                Inorder(t->lchild);

                Visit(t);

                Inorder(t->rchild);

        }
```

Example: Let us consider a given binary tree.



Therefore the inorder traversal of above tree will be: **0,1,2,3,4,5,6,7,8,9,10**

3) Postorder traversal

To traverse a binary tree in postorder traversal, following operations are carried out:

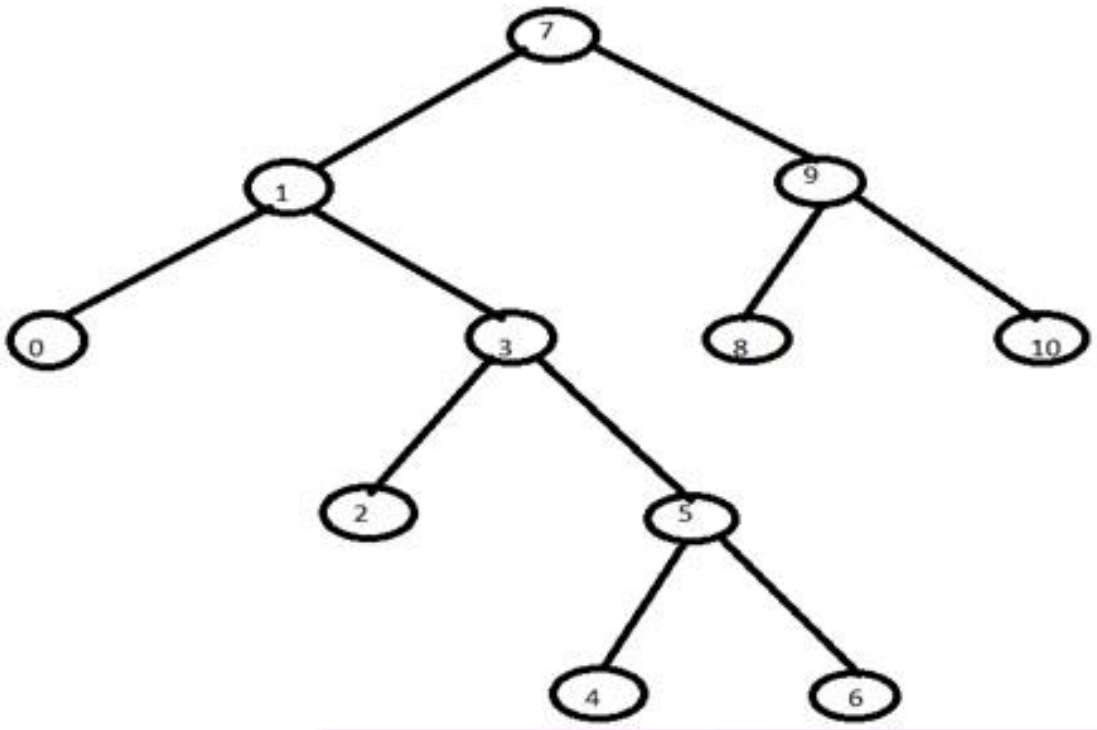
1. Traverse the left sub tree of root.
2. Traverse the right sub tree of root.
3. Visit the root.

Note: Postorder traversal is also known as LRN traversal.

Algorithm:

```
Algorithm postorder(t)


/*t is a binary tree .Each node of t has three fields:

lchild, data, and rchild.*/

{

      If t! =0 then

      {

            Postorder(t->lchild);

            Postorder(t->rchild);

            Visit(t);

      }

}
```

Example: Let us consider a given binary tree.



Therefore the postorder traversal of the above tree will be: **0,2,4,6,5,3,1,8,10,9,7**

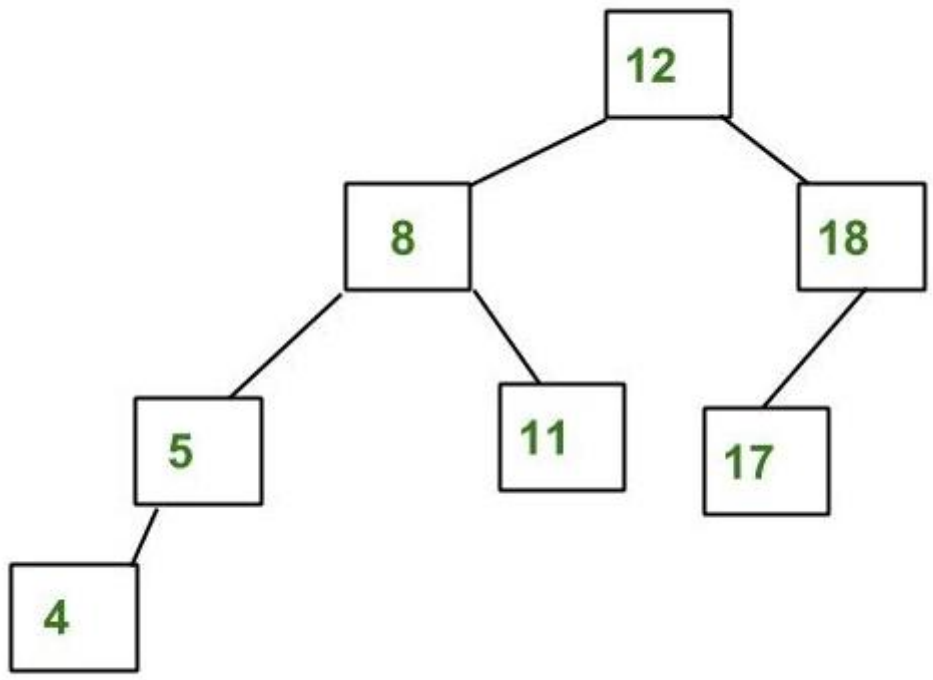
Question 3: (20 Marks) What are AVL trees? How do they differ from Splay trees.

Ans.

AVL Tree:

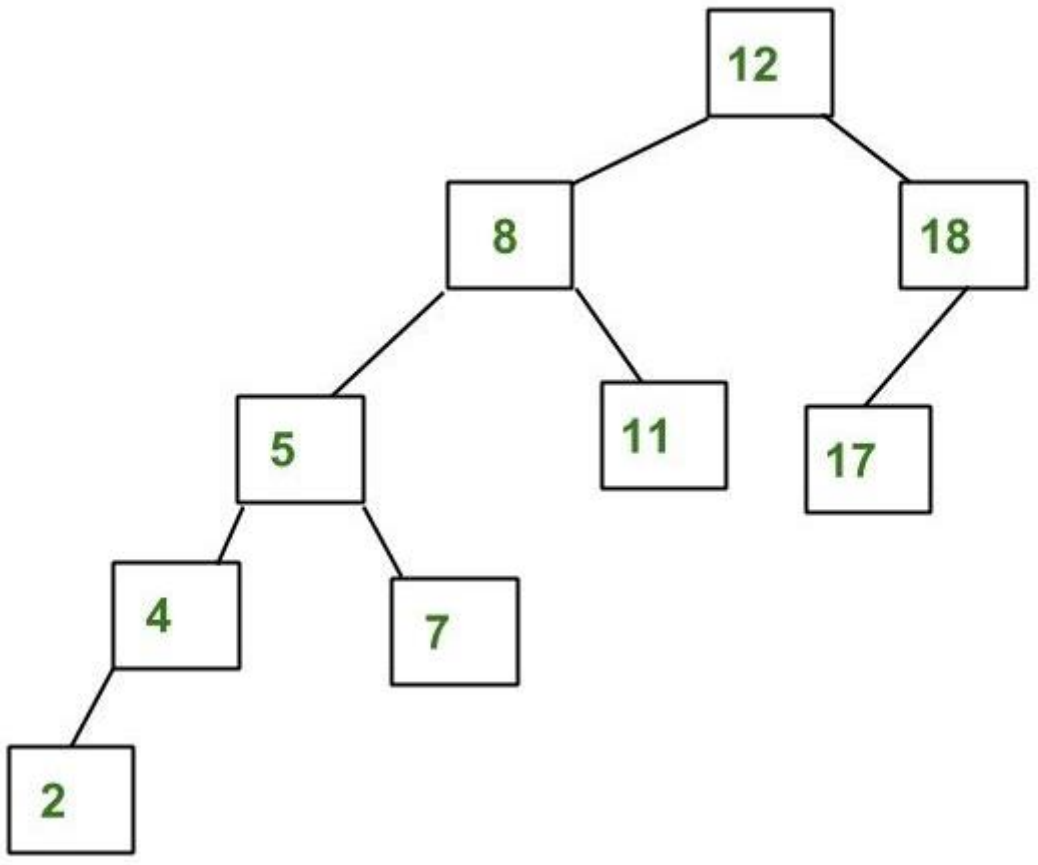
AVL tree is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.

Example of AVL Tree:



The above tree is AVL because the differences between heights of left and right subtrees for every node are less than or equal to 1.

Example of a Tree that is NOT an AVL Tree:



The above tree is not AVL because the differences between the heights of the left and right subtrees for 8 and 12 are greater than 1.

Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log(n)) after every insertion and deletion, then we can guarantee an upper bound of O(log(n)) for all these operations. The height of an AVL tree is always O(log(n)) where n is the number of nodes in the tree.

Both splay trees and AVL trees are binary search trees with excellent performance guarantees, but they differ in how they achieve those guarantee that performance. In an AVL tree, the shape of the tree is constrained at all times such that the tree shape is balanced, meaning that the height of the tree never exceeds O(log n). This shape is maintained on insertions and deletions, and does not change during lookups. Splay trees, on the other hand, maintain efficient by reshaping the tree in response to lookups on it. That way, frequently-accessed elements move up toward the top of the tree and have better lookup times. The shape of splay trees is not constrained, and varies based on what lookups are performed.

Splay trees are more memory-efficient than AVL trees, because they do not need to store balance information in the nodes. However, AVL trees are more useful in multithreaded environments with lots of lookups, because lookups in an AVL tree can be done in parallel while they can't in splay trees. Because splay trees reshape themselves based on lookups, if you only need to access a small subset of the elements of the tree, or if you access some elements much more than others, the splay tree will outperform the AVL tree. Finally, splay trees tend to be easier to implement than AVL trees, since the rotation logic is much easier.

Question 4: (20 Marks) What are Tries? How do they differ from Binary Tries?

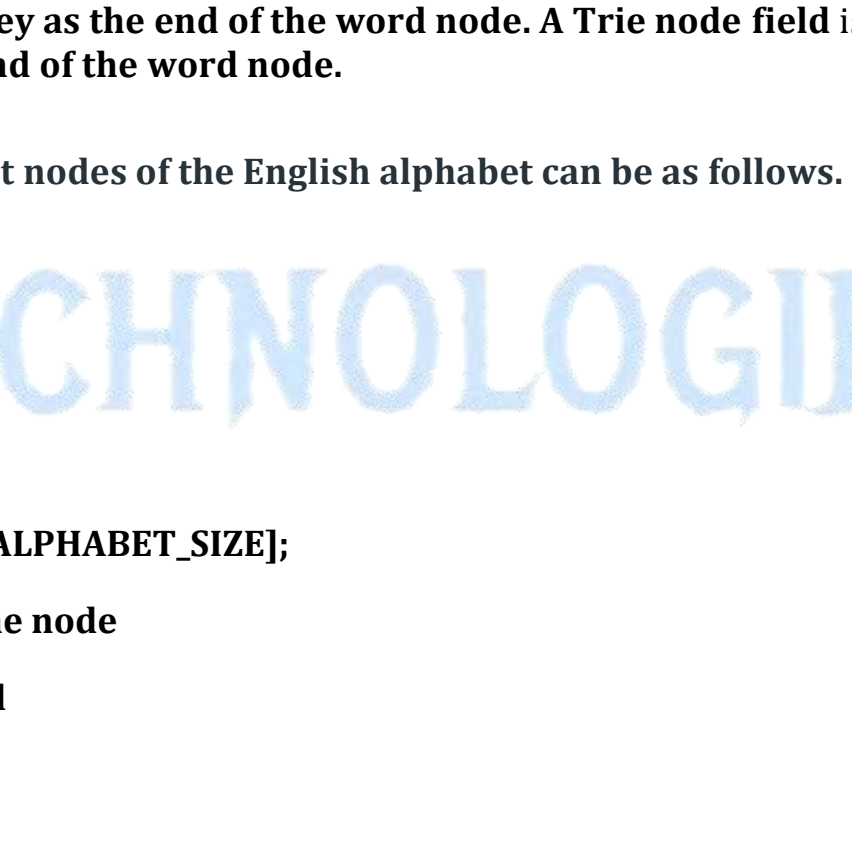
Ans.

What is Trie?

Trie is a type of k-ary search tree used for storing and searching a specific key from a set. Using Trie, search complexities can be brought to optimal limit (key length).

If we store keys in a binary search tree, a well balanced BST will need time proportional to M * log N, where M is the maximum string length and N is the number of keys in the tree. Using Trie, the key can be searched in O(M) time. However, the penalty is on Trie storage requirements

Trie is also known as digital tree or prefix tree.



Trie data structure

Structure of Trie node:

Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. Mark the last node of every key as the end of the word node. A Trie node field isEndOfWord **is used to distinguish the node as the end of the word node.**

Simple structure to represent nodes of the English alphabet can be as follows.

- **C++**

```
// Trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];
    // isEndOfWord is true if the node
    // represents end of a word
    bool isEndOfWord;
};
```

13.1 `BinaryTrie`: A digital search tree

A `BinaryTrie` encodes a set of `w` bit integers in a binary tree. All leaves in the tree have depth `w` and each integer is encoded as a root-to-leaf path. The path for the integer `x` turns left at level `i` if the `i`th most significant bit of `x` is a 0 and turns right if it is a 1. Figure 13.1 shows an example for the case $w = 4$, in which the trie stores the integers 3(0011), 9(1001), 12(1100), and 13(1101).

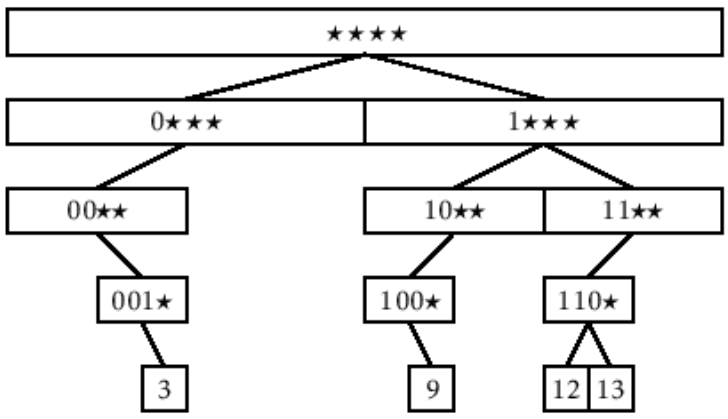


Figure 13.1: The integers stored in a binary trie are encoded as root-to-leaf paths.

Because the search path for a value `x` depends on the bits of `x`, it will be helpful to name the children of a node, `u`, `u.child[0]` (`left`) and `u.child[1]` (`right`). These child pointers will actually serve double-duty. Since the leaves in a binary trie have no children, the pointers are used to string the leaves together into a doubly-linked list. For a leaf in the binary trie `u.child[0]` (`prev`) is the node that comes before `u` in the list and `u.child[1]` (`next`) is the node that follows `u` in the list. A special node, `dummy`, is used both before the first node and after the last node in the list (see Section 3.2).

Each node, `u`, also contains an additional pointer `u.jump`. If `u`'s left child is missing, then `u.jump` points to the smallest leaf in `u`'s subtree. If `u`'s right child is missing, then `u.jump` points to the largest leaf in `u`'s subtree. An example of a `BinaryTrie`, showing `jump` pointers and the doubly-linked list at the leaves, is shown in Figure 13.2.

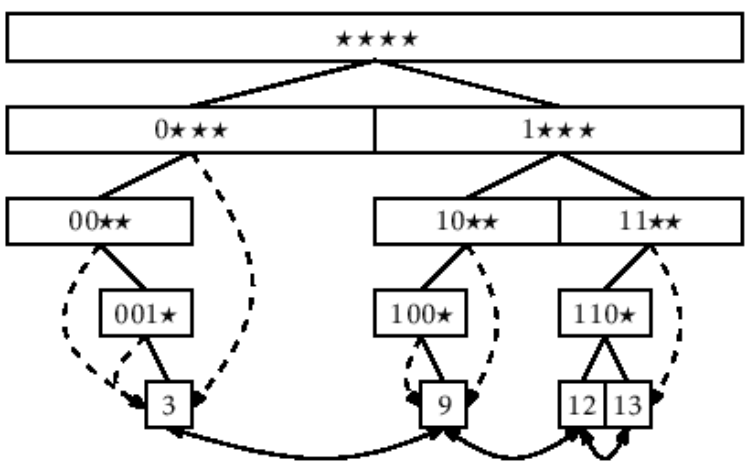


Figure 13.2: A `BinaryTrie` with `jump` pointers shown as curved dashed edges.

The `find(x)` operation in a `BinaryTrie` is fairly straightforward. We try to follow the search path for `x` in the trie. If we reach a leaf, then we have found `x`. If we reach a node `u` where we cannot proceed (because `u` is missing a child), then we follow `u.jump`, which takes us either to the smallest leaf larger than `x` or the largest leaf smaller than `x`. Which of these two cases occurs depends on whether `u` is missing its left or right child, respectively. In the former case (`u` is missing its left child), we have found the node we want. In the latter case (`u` is missing its right child), we can use the linked list to reach the node we want. Each of these cases is illustrated in Figure 13.3.

```
T find(T x) {
    int i, c = 0, ix = it.intValue(x);
    Node u = r;
    for (i = 0; i < w; i++) {
      c = (ix >>> w-i-1) & 1;
      if (u.child[c] == null) break;
      u = u.child[c];
    }
    if (i == w) return u.x;  // found it
    u = (c == 0) ? u.jump : u.jump.child[next];
    return u == dummy ? null : u.x;
}
```