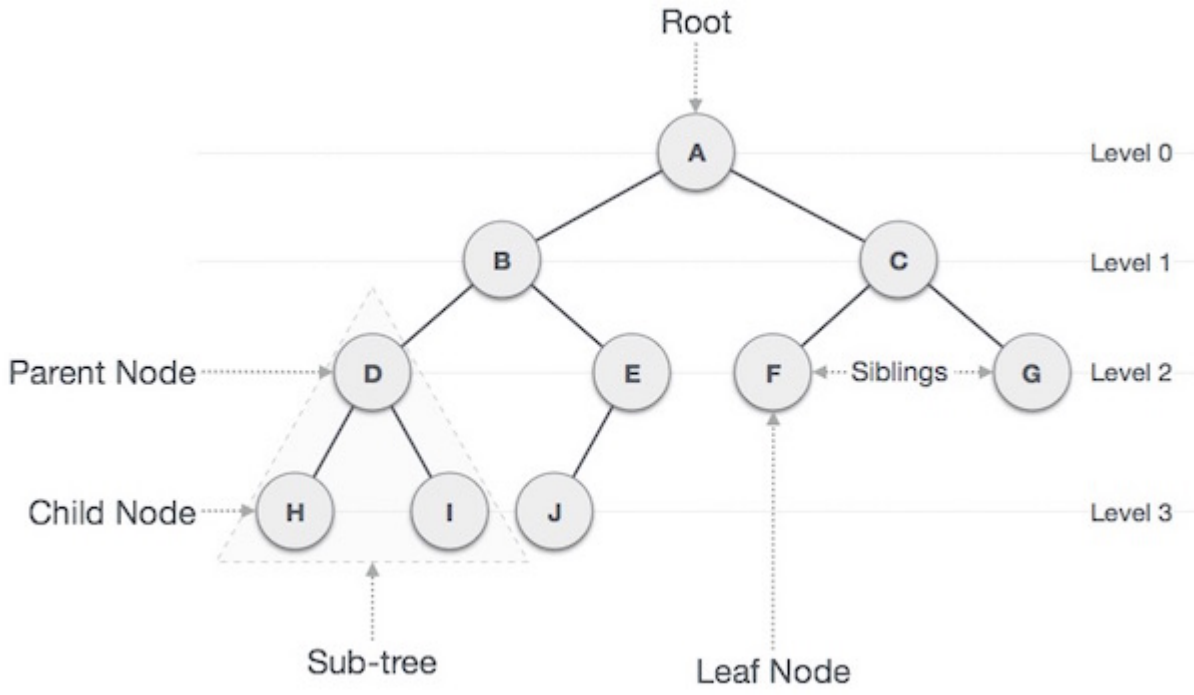
DATA STRUCTURE - TREE

Tree represents nodes connected by edges. We'll going to discuss binary tree or binary search tree specifically.

Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have two children at maximum. A binary tree have benefits of both an ordered array and a linked list as search is as quick as in sorted array and insertion or deletion operation are as fast as in linked list.



Terms

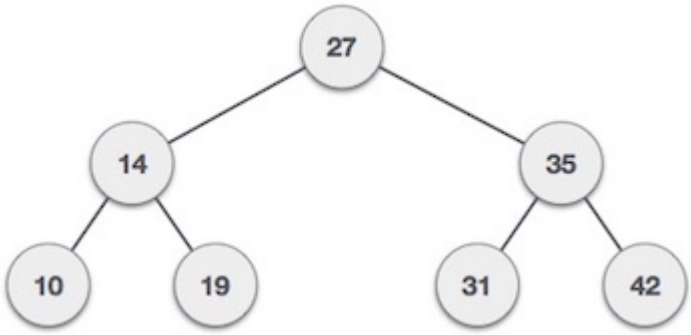
Following are important terms with respect to tree.

- **Path** − Path refers to sequence of nodes along the edges of a tree.
- **Root** − Node at the top of the tree is called root. There is only one root per tree and one path from root node to any node.
- **Parent** − Any node except root node has one edge upward to a node called parent.
- **Child** − Node below a given node connected by its edge downward is called its child node.
- **Leaf** − Node which does not have any child node is called leaf node.
- **Subtree** − Subtree represents descendents of a node.
- **Visiting** − Visiting refers to checking value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If root node is at level 0, then its next child node is at level 1, its grandchild is at level 2 and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Search Tree Representation

Binary Search tree exhibits a special behaviour. A node's left child must have value less than its

parent's value and node's right child must have value greater than it's parent value.



We're going to implement tree using node object and connecting them through references.

Node

A tree node should look like the below structure. It has data part and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

In a tree, all nodes share common construct.

BST Basic Operations

The basic operations that can be performed on binary search tree data structure, are following −

- **Insert** − insert an element in a tree / create a tree.
- **Search** − search an element in a tree.
- **Preorder Traversal** − traverse a tree in a preorder manner.
- **Inorder Traversal** − traverse a tree in an inorder manner.
- **Postorder Traversal** − traverse a tree in a postorder manner.

We shall learn creating *insertinginto* tree structure and searching a data-item in a tree in this chapter. We shall learn about tree traversing methods in the coming one.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted. First locate its proper location. Start search from root node then if data is less than key value, search empty location in left subtree and insert the data. Otherwise search empty location in right subtree and insert the data.

Algorithm

```
If root is NULL
   then create root node
return

If root exists then
   compare the data with node.data

   while until insertion position is located

      If data is greater than node.data
         goto right subtree
      else
```

```
         goto left subtree

   endwhile

   insert data

end If
```

Implementation

The implementation of insert function should look like this −

```
void insert(int data) {
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;

   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty, create root node
   if(root == NULL) {
      root = tempNode;
   }else {
      current = root;
      parent  = NULL;

      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }

         //go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}
```

Search Operation

Whenever an element is to be search. Start search from root node then if data is less than key value, search element in left subtree otherwise search element in right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
   return root
else
```

```
   while data not found

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

      If data found
         return node

   endwhile

   return data not found

end if
```

The implementation of this algorithm should look like this.

```
struct node* search(int data) {
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data) {
      if(current != NULL)
      printf("%d ",current->data);

      //go to left tree

      if(current->data > data) {
         current = current->leftChild;
      }
      //else go to right tree
      else {
         current = current->rightChild;
      }

      //not found
      if(current == NULL) {
         return NULL;
      }

      return current;
   }
}
```

To see the implementation of binary search tree data structure, please click here.