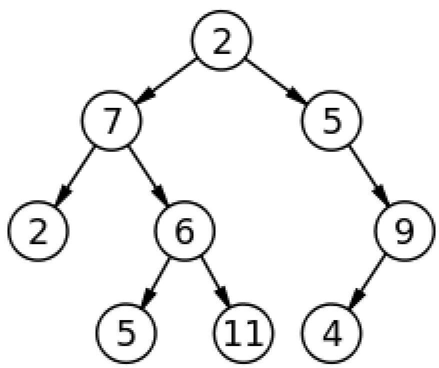
Tree

In computer science, a **tree** is a widely used abstract data type (ADT) or data structure implementing this ADT that simulates a hierarchical tree structure, with a root value and subtrees of children, represented as a set of linked nodes.

A tree data structure can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root. A tree can be defined abstractly as a whole (globally) as an ordered tree, with a value assigned to each node. Both these perspectives are useful: while a tree can be analyzed mathematically as a whole, when actually represented as a data structure it is usually represented and worked with separately by node (rather than as a list of nodes and an adjacency list of edges between nodes, as one may represent a digraph, for instance). For example, looking at a tree as a whole, one can talk about "the parent node" of a given node, but in general as a data structure a given node only contains the list of its children, but does not contain a reference to its parent (if any)



A simple unordered tree; in this diagram, the node labeled 7 has two children, labeled 2 and 6, and one parent, labeled 2. The root node, at the top, has no parent.

CODES (C)

```
typedef struct tree
{
	int no;
	struct tree *l,*r;
}node;
node * insert(node *t,int num)
{
	node *p,*q,*x;
	q=(node *)malloc(sizeof(node));
	if(q==NULL)
		printf("\n Overflow");
	else
	{
		q->l=q->r=NULL;
		q->no=num;
		if(!t)
			t=q;
		else
		{
			for(p=t;p;x=p,p=num>p->no?p->r:p->r);
			num>x->no?x->r=q:x->l=q;
		}
	}
	return t;
}
```

```
int pop(st *s)
{
	int a=0;
	if(isempty(s)==1)
		printf("Stack is empty");
	else
	{
		a=s->a[s->tos];
		s->tos--;
	}
	return a;
}
char peek(st *s)
{
	char p;
	if(isempty(s)==1)
		printf("Stack is empty");
	else
	{
		p=s->a[s->tos];
	}
	return p;
}
```

Deletions(C)

```
void delete ( struct btreenode **root, int num )
{
    int found ;
    struct btreenode *parent, *x, *xsucc ;
    /* if tree is empty */
    if ( *root == NULL )
   {
     printf ( "\nTree is empty" ) ;
     return ;
   }
   parent = x = NULL ;
   /* call to search function to find the node to be deleted */
   search ( root, num, &parent, &x, &found ) ;
   /* if the node to deleted is not found */
   if ( found == FALSE )
   {
      printf ( "\nData to be deleted, not found" ) ;
      return ;
   }
    /* if the node to be deleted has two children */
   if ( x -> leftchild != NULL && x -> rightchild != NULL )
   {
      parent = x ;
      xsucc = x -> rightchild ;
      while ( xsucc -> leftchild != NULL )
      {
        parent = xsucc ;
        xsucc = xsucc -> leftchild ;
      }
      x -> data = xsucc -> data ;
      x = xsucc ;
    }
    /* if the node to be deleted has no child */
    if ( x -> leftchild == NULL && x -> rightchild == NULL )
    {
      if ( parent -> rightchild == x )
        parent -> rightchild = NULL ;
     else
        parent -> leftchild = NULL ;
        free ( x ) ;
     return ;
     }
    /* if the node to be deleted has only rightchild */
   if ( x -> leftchild == NULL && x -> rightchild != NULL )
   {
      if ( parent -> leftchild == x )
         parent -> leftchild = x -> rightchild ;
     else
        parent -> rightchild = x -> rightchild ;
     free ( x ) ;
```

```
    return ;
  }
  /* if the node to be deleted has only left child */
  if ( x -> leftchild != NULL && x -> rightchild == NULL )
  {
     if ( parent -> leftchild == x )
        parent -> leftchild = x -> leftchild ;
     else
        parent -> rightchild = x -> leftchild ;
    free ( x ) ;
    return ;
  }
 }
 /*returns the address of the node to be deleted, address of its parent and whether the node is
found or not */
 void search ( struct btreenode **root, int num, struct btreenode **par, struct btreenode **x,
int *found )
 {
    struct btreenode *q ;
    q = *root ;
    *found = FALSE ;
    *par = NULL ;
    while ( q != NULL )
    {
      /* if the node to be deleted is found */
      if ( q -> data == num )
      {
        *found = TRUE ;
        *x = q ;
         return ;
      }
      *par = q ;
      if ( q -> data > num )
         q = q -> leftchild ;
      else
         q = q -> rightchild ;
    }
}
```

CODES (JAVA)

```
public void insert(Node node, int value)
 {
        if (value < node.value)
        {
            if (node.left != null)
            {
                insert(node.left, value);
            }
            else
            {
                System.out.println("  Inserted " + value + " to left of Node " + node.value);
                node.left = new Node(value);
            }
        }
       else if (value > node.value)
        {
            if (node.right != null) {
                insert(node.right, value);
            } else {
                System.out.println("  Inserted " + value + " to right of Node " + node.value);
                node.right = new Node(value);
            }
        }
```

Deletion (JAVA)

```
public class BinarySearchTree {
    public boolean remove(int value) {
        if (root == null)
            return false;
        else {
            if (root.getValue() == value) {
                BSTNode auxRoot = new BSTNode(0);
                auxRoot.setLeftChild(root);
                boolean result = root.remove(value, auxRoot);
                root = auxRoot.getLeft();
                return result;
            } else {
                return root.remove(value, null);
            }
        }
    }
}
public class BSTNode {
    …
    public boolean remove(int value, BSTNode parent) {
        if (value < this.value) {
            if (left != null)
                return left.remove(value, this);
            else
                return false;
        } else if (value > this.value) {
            if (right != null)
                return right.remove(value, this);
            else
                return false;
        } else {
            if (left != null && right != null) {
                this.value = right.minValue();
                right.remove(this.value, this);
            } else if (parent.left == this) {
                parent.left = (left != null) ? left : right;
            } else if (parent.right == this) {
                parent.right = (left != null) ? left : right;
            }
            return true;
        }
    }

    public int minValue() {
        if (left == null)
            return value;
        else
            return left.minValue();
    }
}
```