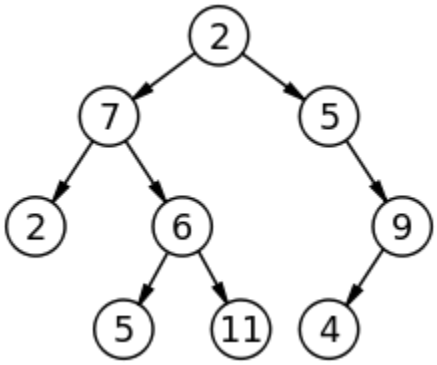
Tree

कंप्यूटर विज्ञान में, एक पेड़ से जुड़े नोड्स के एक सेट के रूप में प्रतिनिधित्व children की एक जड़ मूल्य और subtrees, के साथ एक पदानुक्रमित वृक्ष संरचना, simulates कि इस एडीटी को लागू करने के लिए एक व्यापक रूप से इस्तेमाल सार डेटा प्रकार) एडीटी (या डेटा संरचना है. एक tree data structure नोड्स के लिए संदर्भ की एक सूची") children के साथ ("एक साथ, प्रत्येक नोड एक मूल्य से मिलकर एक डेटा संरचना है जहां नोड्स) एक रूट नोड पर शुरू(, का एक संग्रह के रूप में (स्थानीय) बारी बारी से परिभाषित किया जा सकता है , रूट करने के लिए कोई संदर्भ दोहराया गया है कि बाधाओं, और कोई भी अंक के साथ .एक पेड़ के प्रत्येक नोड को सौंपा एक मूल्य के साथ, एक आदेश के पेड़ के रूप में) विश्व स्तर पर (एक पूरे के रूप में सूक्ष्म रूप में परिभाषित किया जा सकता है .इन दृष्टिकोण दोनों उपयोगी होते हैं :एक पेड़ एक पूरे के रूप में, वास्तव में एक आंकड़ा संरचना के रूप में प्रतिनिधित्व जब यह आम तौर पर प्रतिनिधित्व किया और नोड) बल्कि नोड्स की एक सूची और नोड्स के बीच किनारों की एक संलग्नता सूची के रूप में की तुलना में अलग से साथ काम किया है गणितीय विश्लेषण किया जा सकता है, जबकि , के रूप में एक, उदाहरण के लिए (एक संयुक्ताक्षर प्रतिनिधित्व कर सकते हैं .उदाहरण के लिए, एक पूरे के रूप में एक पेड़ पर देख रहे हैं, एक किसी दिए गए नोड के पैरेंट"नोड "के बारे में बात कर सकते हैं, लेकिन एक data Structure के रूप में सामान्य रूप से किसी दिए गए नोड केवल अपने children की सूची में शामिल है, लेकिन करने के लिए एक संदर्भ शामिल नहीं है अपने parents.



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();
    }
}
```