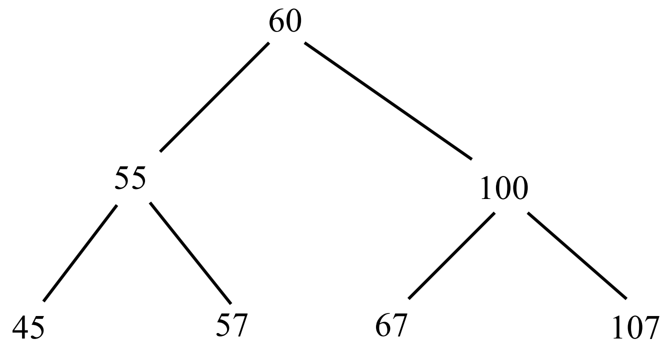
Chapter 19 Binary Search Trees

Objectives

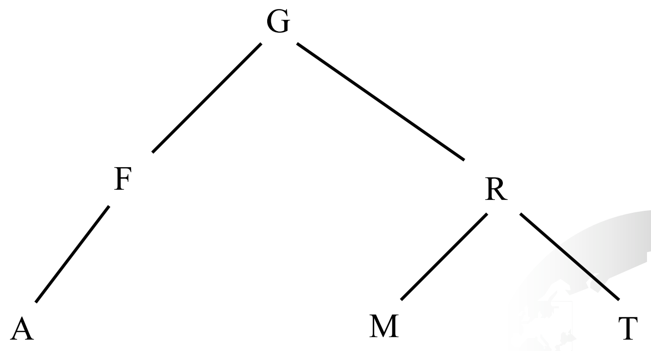
- To design and implement a binary search tree (§19.2).
- To represent binary trees using linked data structures (§19.2.1).
- To search an element in binary search tree (§19.2.2).
- To insert an element into a binary search tree (§19.2.3).
- To traverse elements in a binary tree (§19.2.4).
- To delete elements from a binary search tree (§19.3).
- To display binary tree graphically (§19.4).
- To create iterators for traversing a binary tree (§19.5).
- To implement Huffman coding for compressing data using a binary tree (§19.6).

Binary Trees

A list, stack, or queue is a linear structure that consists of a sequence of elements. A binary tree is a hierarchical structure. It is either empty or consists of an element, called the *root*, and two distinct binary trees, called the *left subtree* and *right subtree*.



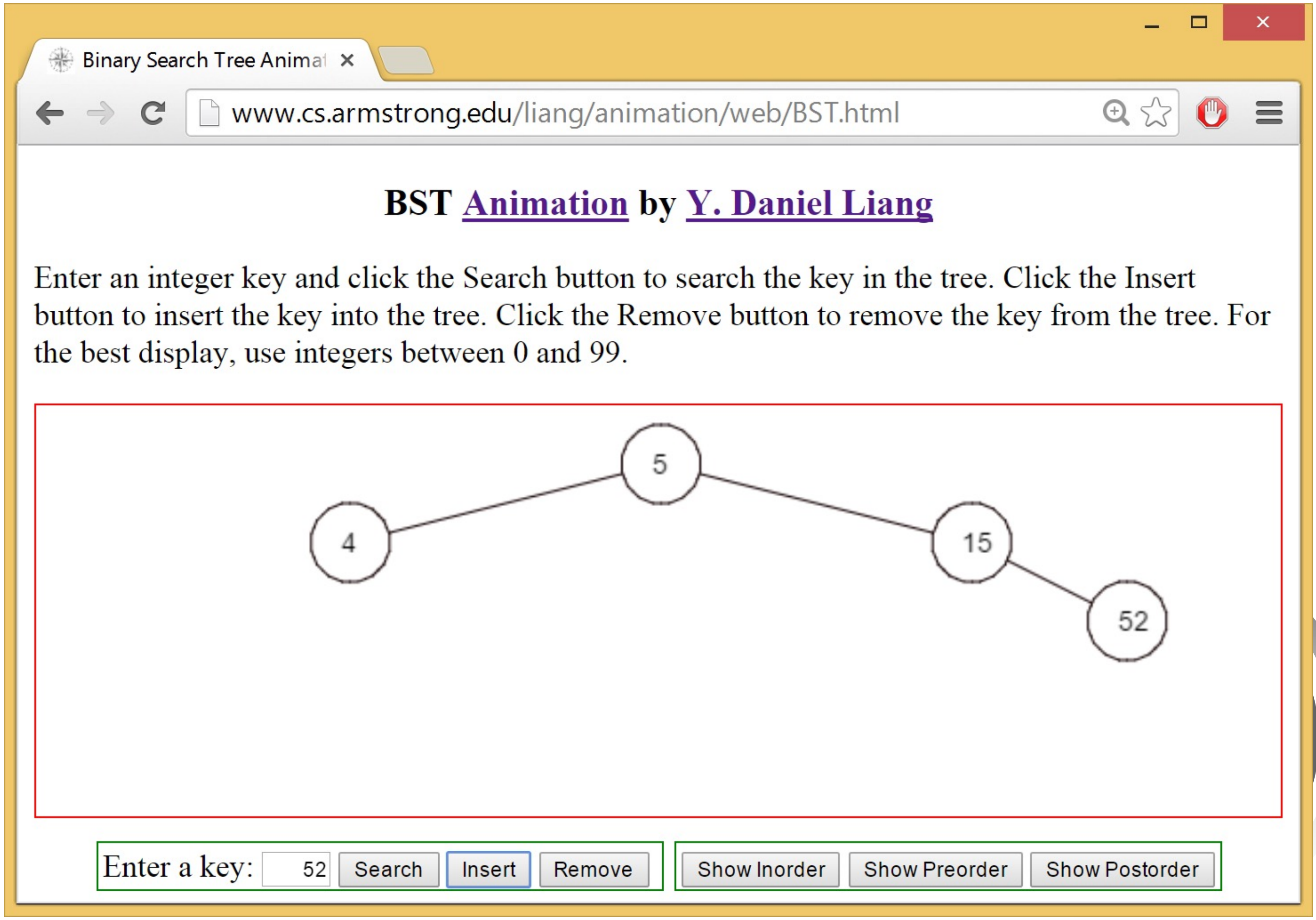
(A)



(B)

See How a Binary Search Tree Works

https://liveexample.pearsoncmg.com/dsanimation/BSTeBook.html


Binary Search Tree Animat
www.cs.armstrong.edu/liang/animation/web/BST.html

BST Animation by Y. Daniel Liang

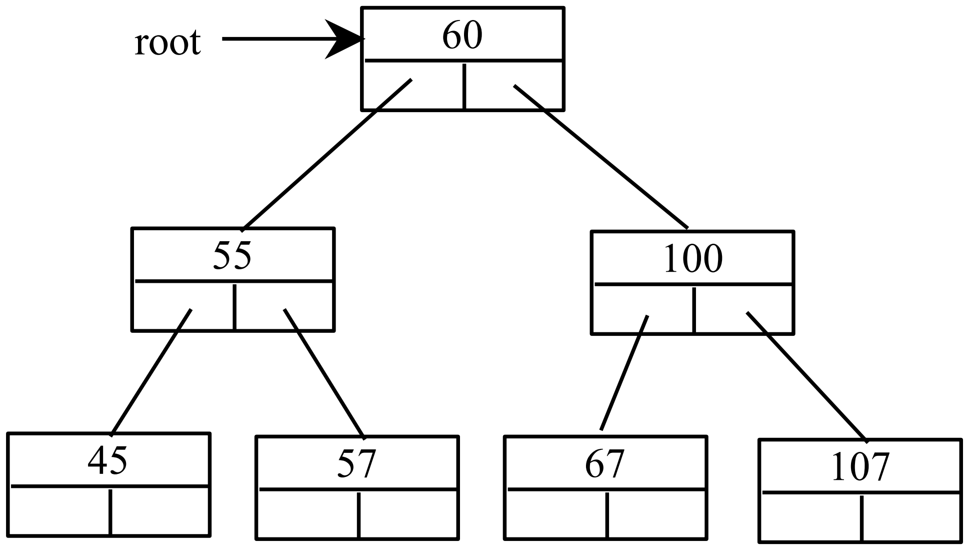
Enter an integer key and click the Search button to search the key in the tree. Click the Insert button to insert the key into the tree. Click the Remove button to remove the key from the tree. For the best display, use integers between 0 and 99.

Binary Tree Terms

The root of left (right) subtree of a node is called a *left (right) child* of the node. A node without children is called a *leaf*. A special type of binary tree called a *binary search tree* is often useful. A binary search tree (with no duplicate elements) has the property that for every node in the tree the value of any node in its left subtree is less than the value of the node and the value of any node in its right subtree is greater than the value of the node. The binary trees in Figure 19.1 are all binary search trees. This section is concerned with binary search trees.

Representing Binary Trees

A binary tree can be represented using a set of linked nodes. Each node contains a value and two links named *left* and *right* that reference the left child and right child, respectively, as shown in Figure 19.2.



```
class TreeNode:
    def _init_(self, e):
        self.element = e
        self.left = None
        self.right = None
```

Searching an Element in a Binary Tree

```
def search(element):
    current = root # Start from the root
    while current != None:
        if element < current.element:
            current = current.left # Go left
        elif element > current.element:
            current = current.right # Go right
        else: # Element matches current.element
            return True # Element is found
    return False # Element is not in the tree
```

Inserting an Element to a Binary Tree

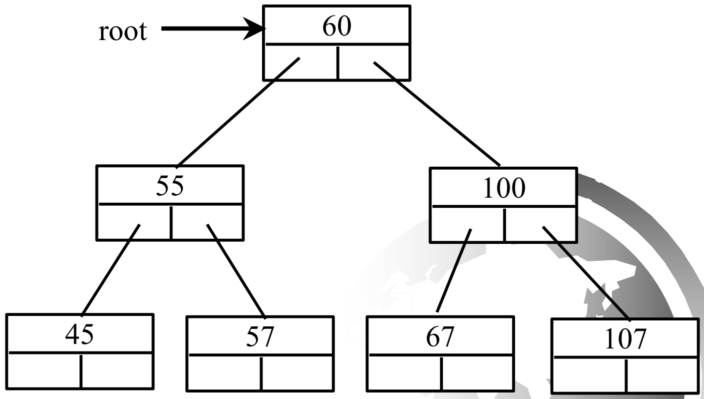
If a binary tree is empty, create a root node with the new element. Otherwise, locate the parent node for the new element node. If the new element is less than the parent element, the node for the new element becomes the left child of the parent. If the new element is greater than the parent element, the node for the new element becomes the right child of the parent. Here is the algorithm:

Inserting an Element to a Binary Tree

```
def insert(e):
  if tree is empty:
    # Create the node for e as the root
  else:
    # Locate the parent node
    parent = current = root
    while current != None:
      if e < the value in current.element:
        parent = current  # Keep the parent
        current = current.left # Go left
      elif e > the value in current.element:
        parent = current # Keep the parent
        current = current.right # Go right
      else:
        return False # Duplicate node not inserted

    # Create a new node for e and attach it to parent
    return True # Element inserted
```

Insert 101 into the following tree.



Tree Traversal

Tree traversal is the process of visiting each node in the tree exactly once. There are several ways to traverse a tree. This section presents *inorder*, *preorder*, *postorder, depth-first, and breadth-first* traversals.

The inorder traversal is to visit the left subtree of the current node first recursively, then the current node itself, and finally the right subtree of the current node recursively.

The postorder traversal is to visit the left subtree of the current node first, then the right subtree of the current node, and finally the current node itself.

Tree Traversal, cont.

The preorder traversal is to visit the current node first, then the left subtree of the current node, and finally the right subtree of the current node.

Tree Traversal, cont.

The breadth-first traversal is to visit the nodes level by level. First visit the root, then all children of the root from left to right, then grandchildren of the root from left to right, and so on.

For example, in the tree in Figure 19.2, the inorder is 45 55 57 59 60 67 100 101 107. The postorder is 45 59 57 55 67 101 107 100 60. The preorder is 60 55 45 57 59 100 67 107 101. The breadth-first traversal is 60 55 100 45 57 67 107 59 101.

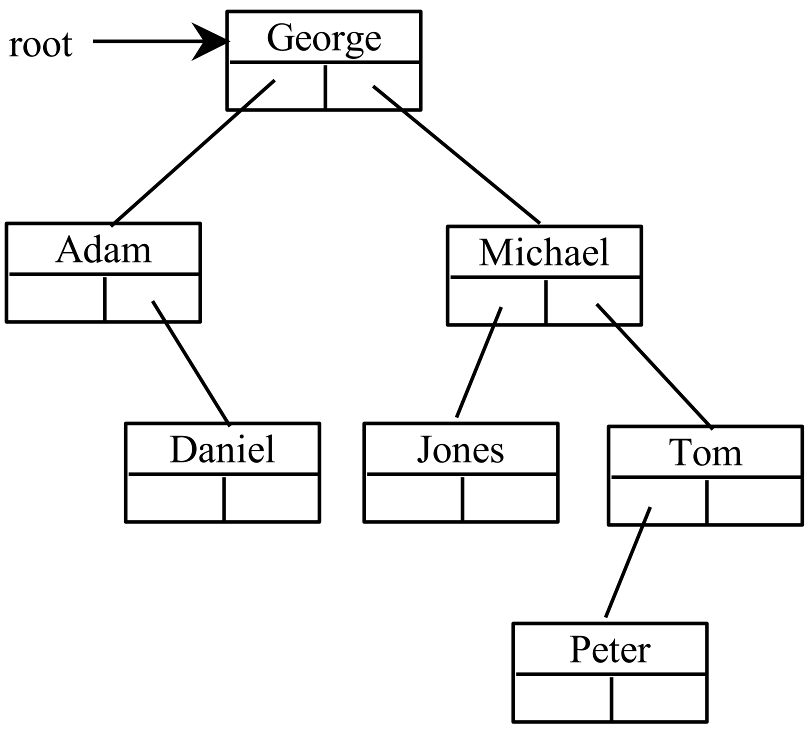
The BinaryTree Class

BST	
root: TreeNode	The root of the tree.
size: int	The size of the tree.
BinaryTree()	Constructs an empty tree.
search(e: object): bool	Returns true if the specified element is in the tree.
insert(e: object): bool	Returns true if the element is added successfully.
delete(e: object): bool	Returns true if the element is removed from the tree successfully.
inorder(): None	Prints the nodes in inorder traversal.
preorder(): None	Prints the nodes in preorder traversal.
postorder(): None	Prints the nodes in postorder traversal.
getSize(): int	Returns the number of elements in the tree.
isEmpty(): bool	Returns true if the tree is empty.
clear(): None	Removes all elements from the tree.
path(e: object): list	Returns the path of nodes from the root leading to the node for the specified element. The element may not be in the tree.

BST

TestBST

Tree After Insertions



Inorder: Adam, Daniel George, Jones, Michael, Peter, Tom

Postorder: Daniel Adam, Jones, Peter, Tom, Michael, George

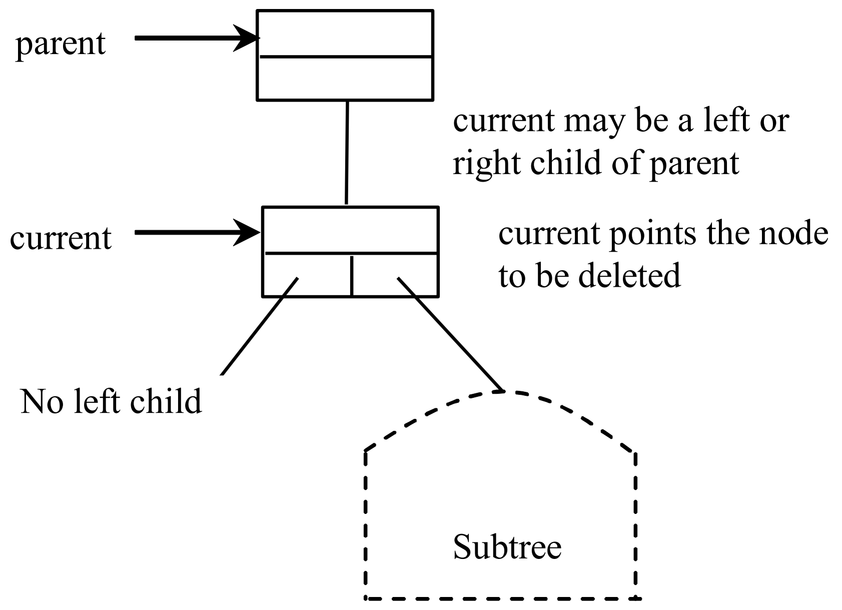
Preorder: George, Adam, Daniel, Michael, Jones, Tom, Peter

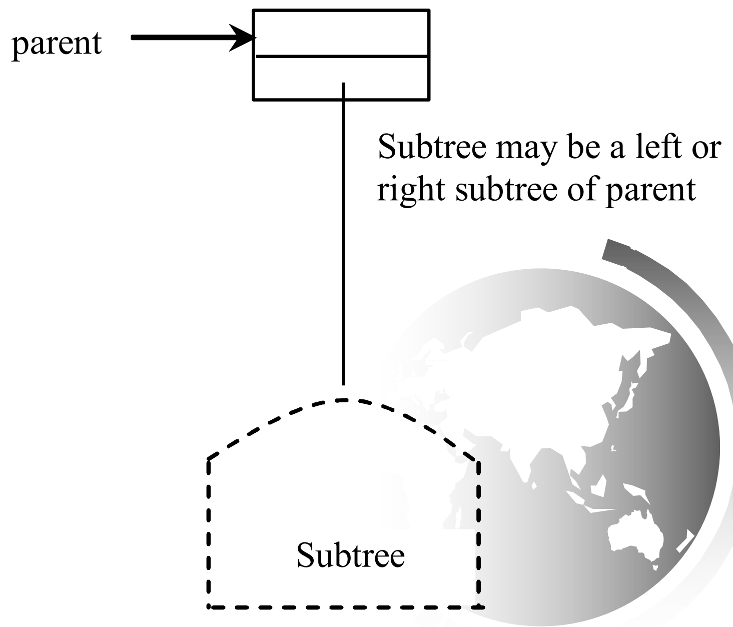
Deleting Elements in a Binary Search Tree

To delete an element from a binary tree, you need to first locate the node that contains the element and also its parent node. Let current point to the node that contains the element in the binary tree and parent point to the parent of the current node. The current node may be a left child or a right child of the parent node. There are two cases to consider:

Deleting Elements in a Binary Search Tree

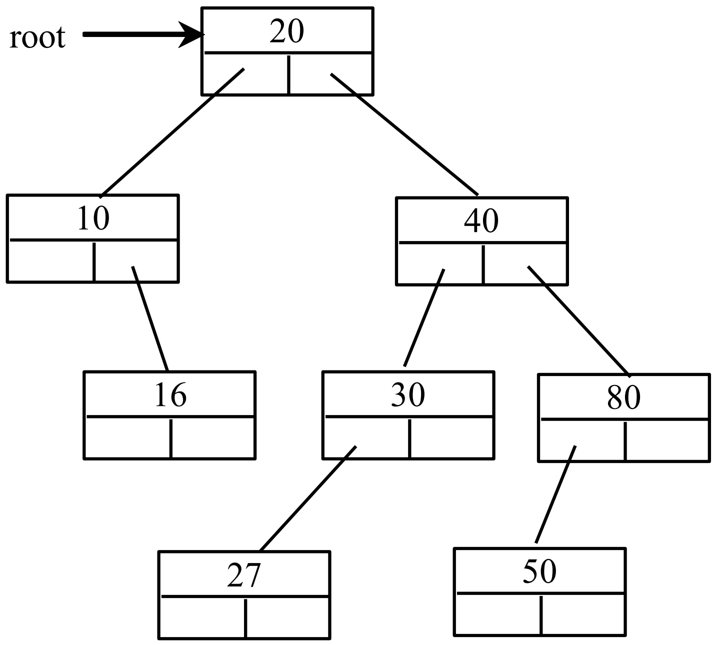
Case 1: The current node does not have a left child, as shown in this figure (a). Simply connect the parent with the right child of the current node, as shown in this figure (b).

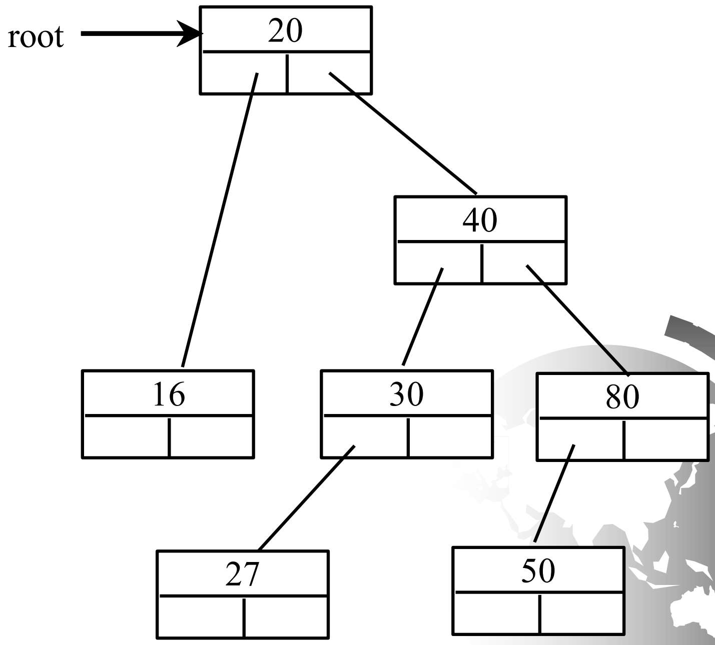

parent
current may be a left or right child of parent
current
current points the node to be deleted
No left child
Subtree



Deleting Elements in a Binary Search Tree

For example, to delete node 10 in Figure 19.9a. Connect the parent of node 10 with the right child of node 10, as shown in Figure 19.9b.



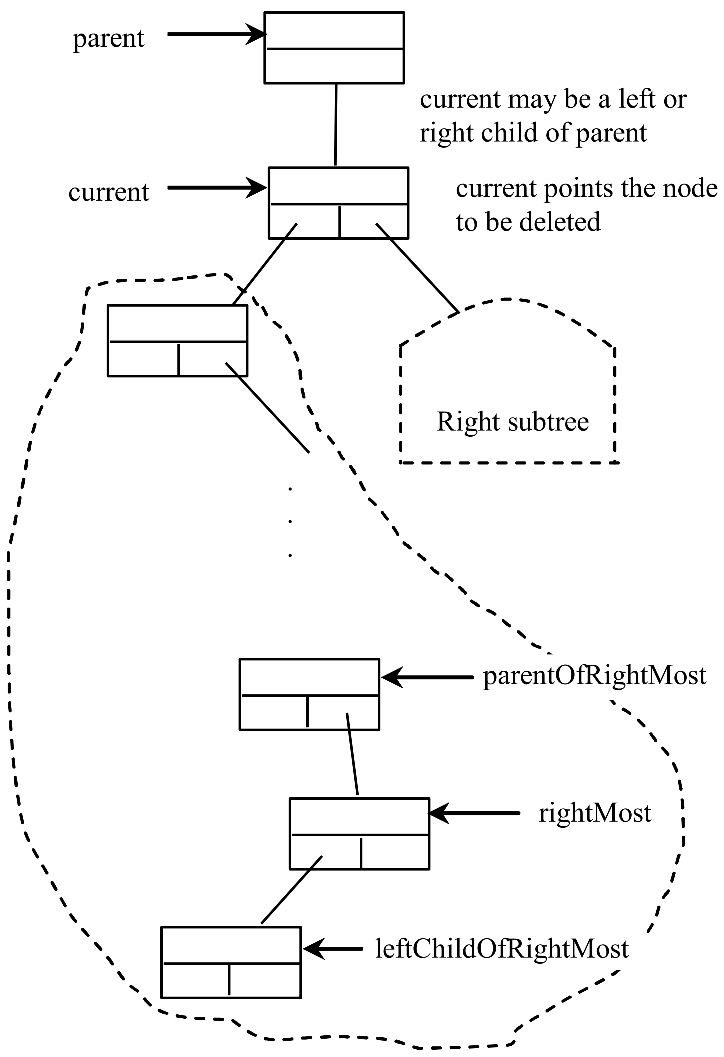


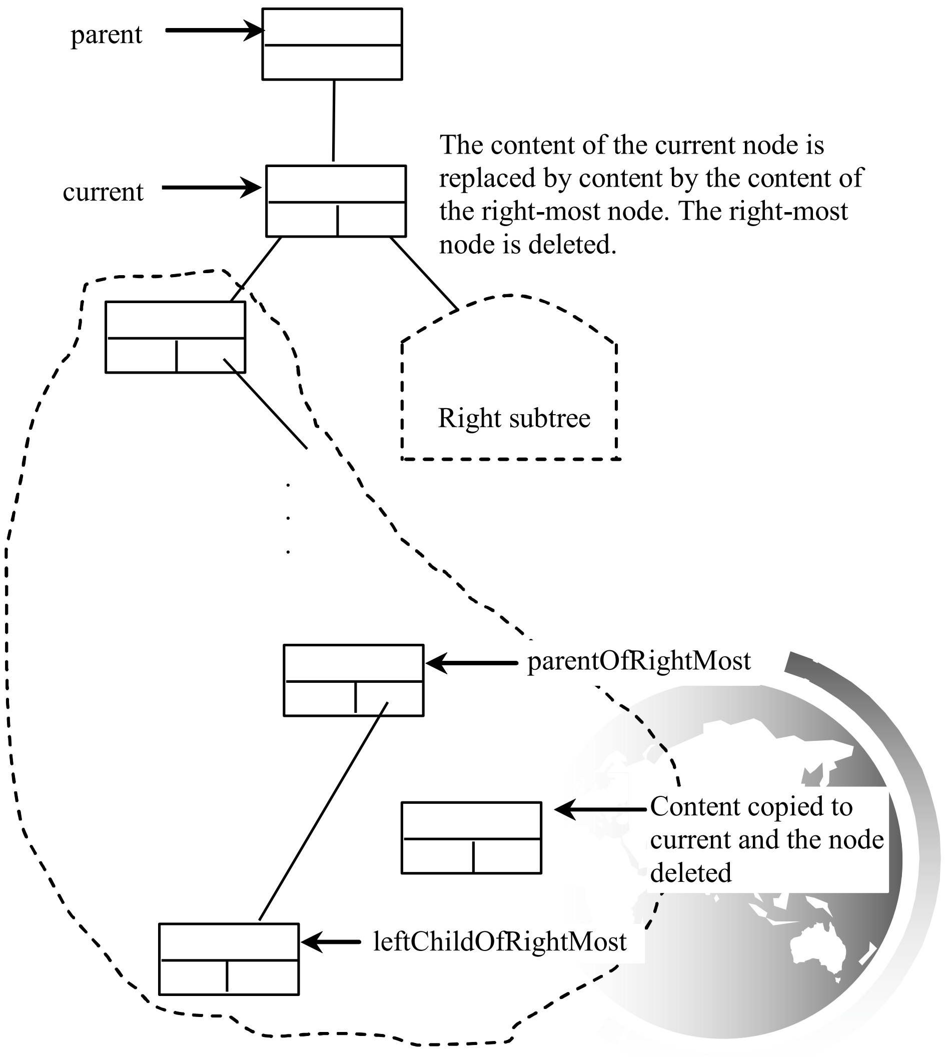
Deleting Elements in a Binary Search Tree

Case 2: The current node has a left child. Let rightMost point to the node that contains the largest element in the left subtree of the current node and parentOfRightMost point to the parent node of the rightMost node, as shown in Figure 19.10). Note that the rightMost node cannot have a right child, but may have a left child. Replace the element value in the current node with the one in the rightMost node, connect the parentOfRightMost node with the left child of the rightMost node, and delete the rightMost node, as shown in Figure 19.10b.

Deleting Elements in a Binary Search Tree

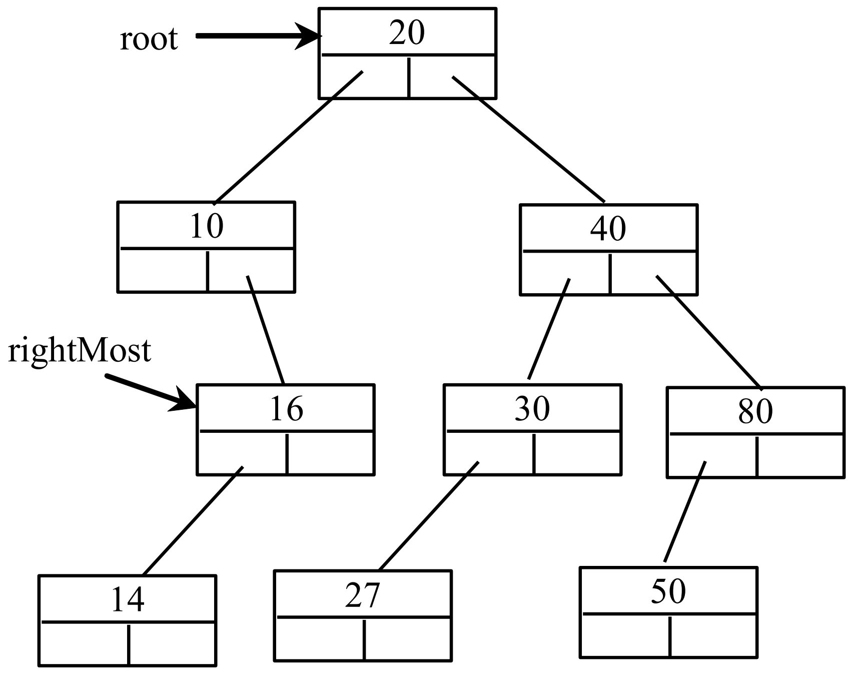
Case 2 diagram

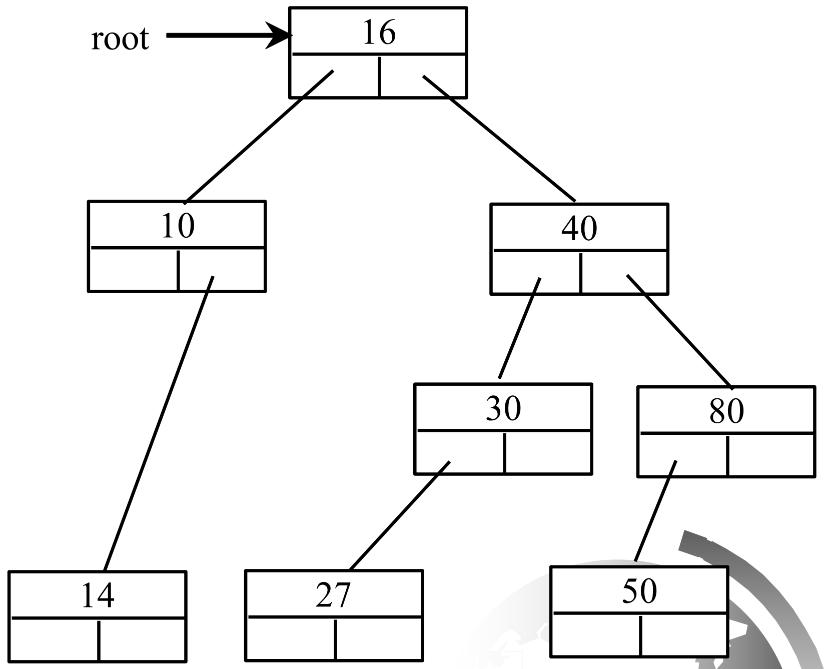

parent
current may be a left or right child of parent
current
current points the node to be deleted
Right subtree
parentOfRightMost
rightMost
leftChildOfRightMost


parent
current
The content of the current node is replaced by content by the content of the right-most node. The right-most node is deleted.
Right subtree
parentOfRightMost
Content copied to current and the node deleted
leftChildOfRightMost

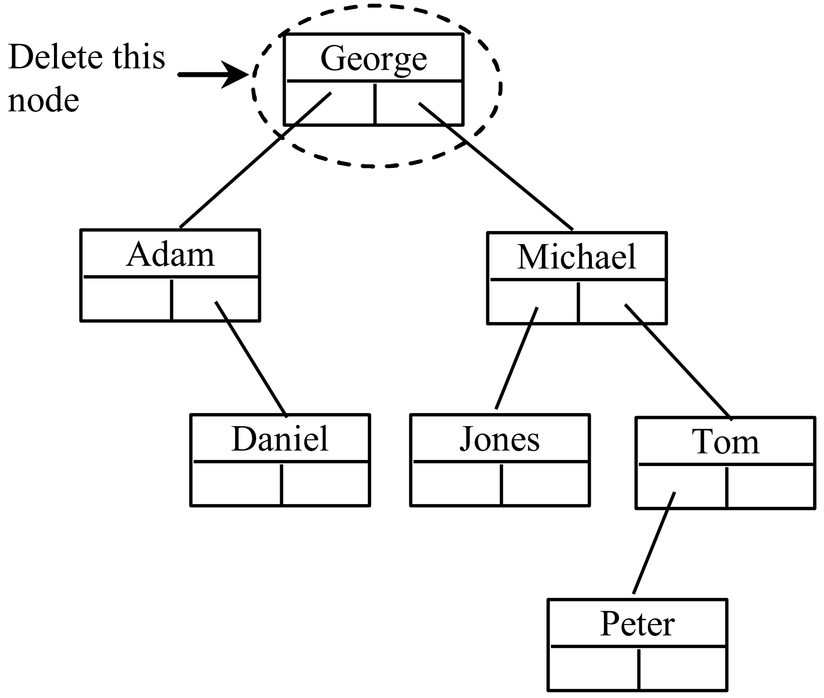
Deleting Elements in a Binary Search Tree

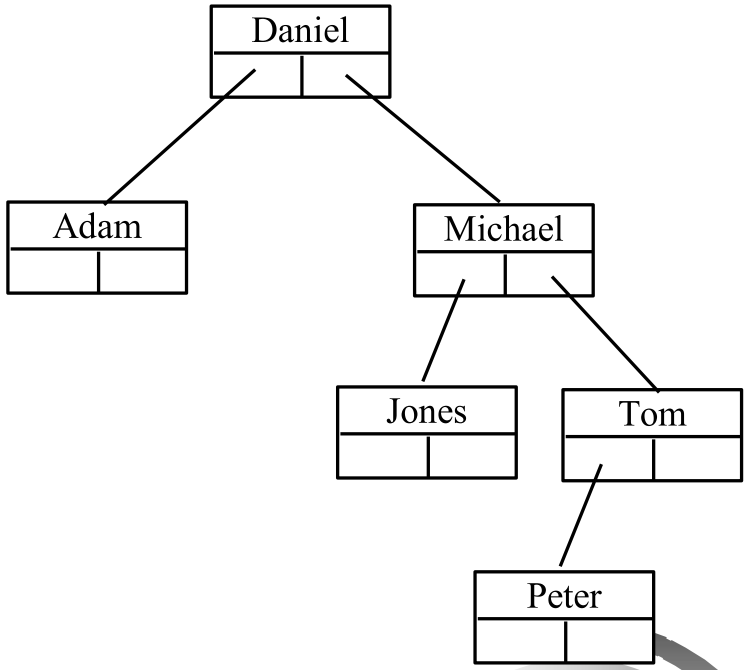
Case 2 example, delete 20



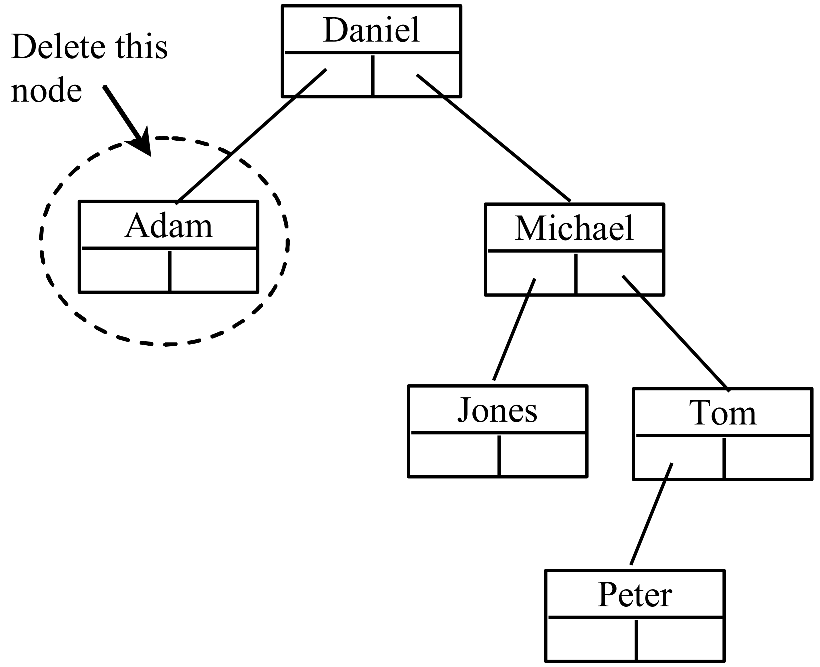


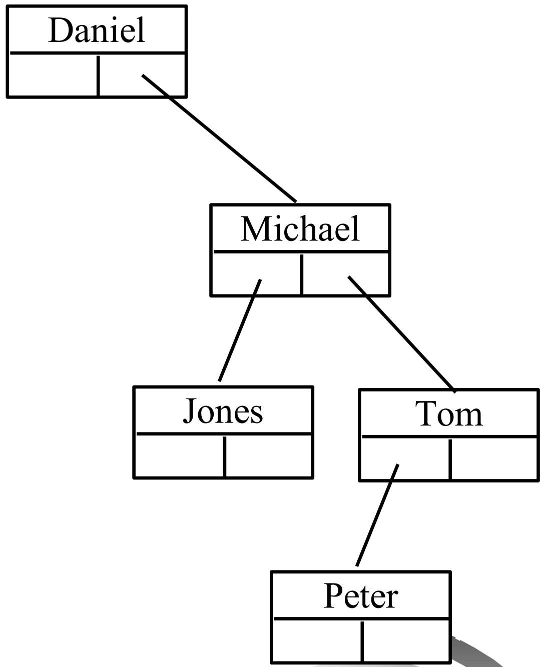
Examples


Delete this node
George
Adam
Michael
Daniel
Jones
Tom
Peter

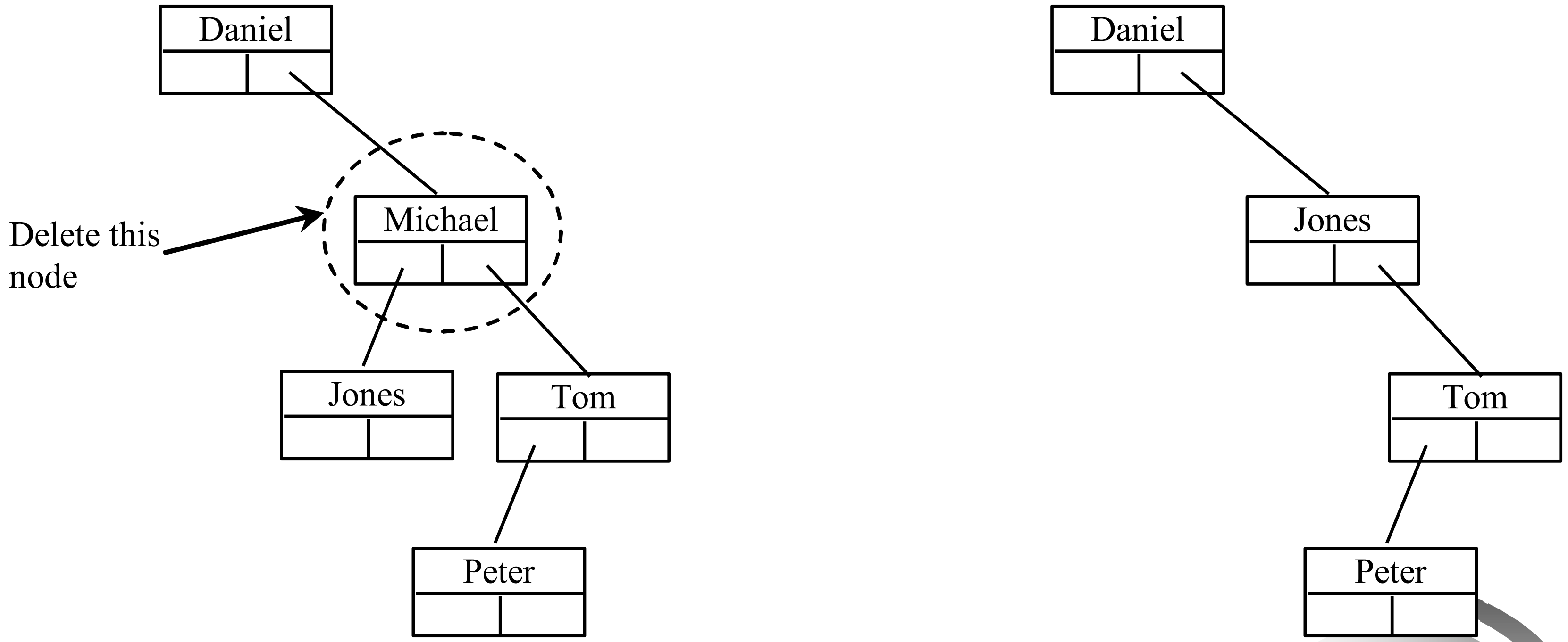

Daniel
Adam
Michael
Jones
Tom
Peter

Examples


Delete this node
Daniel
Adam
Michael
Jones
Tom
Peter


Daniel
Michael
Jones
Tom
Peter

Examples

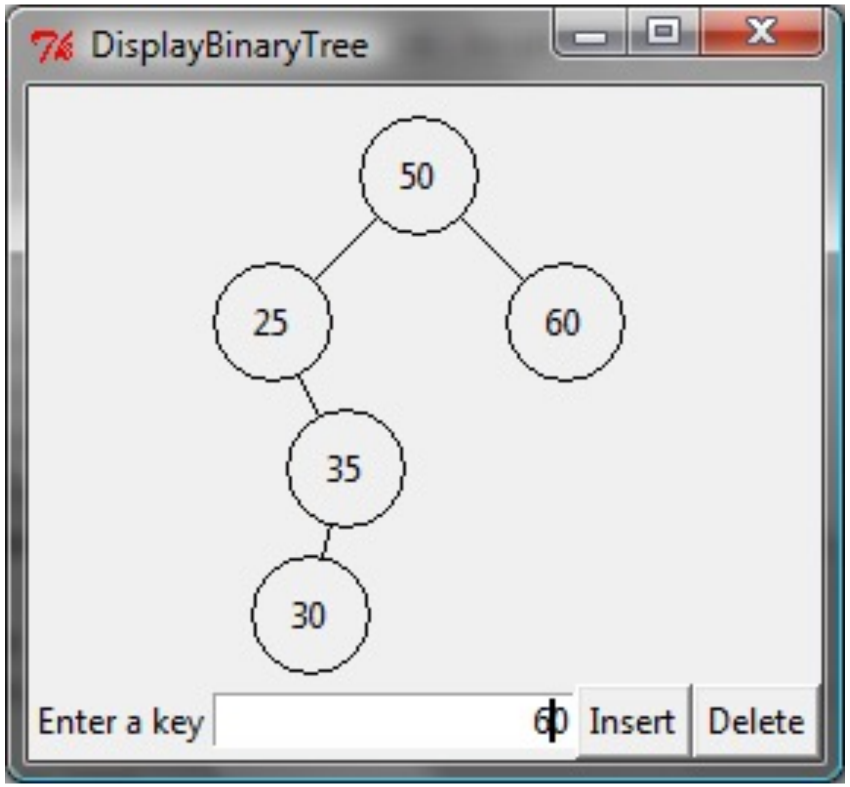


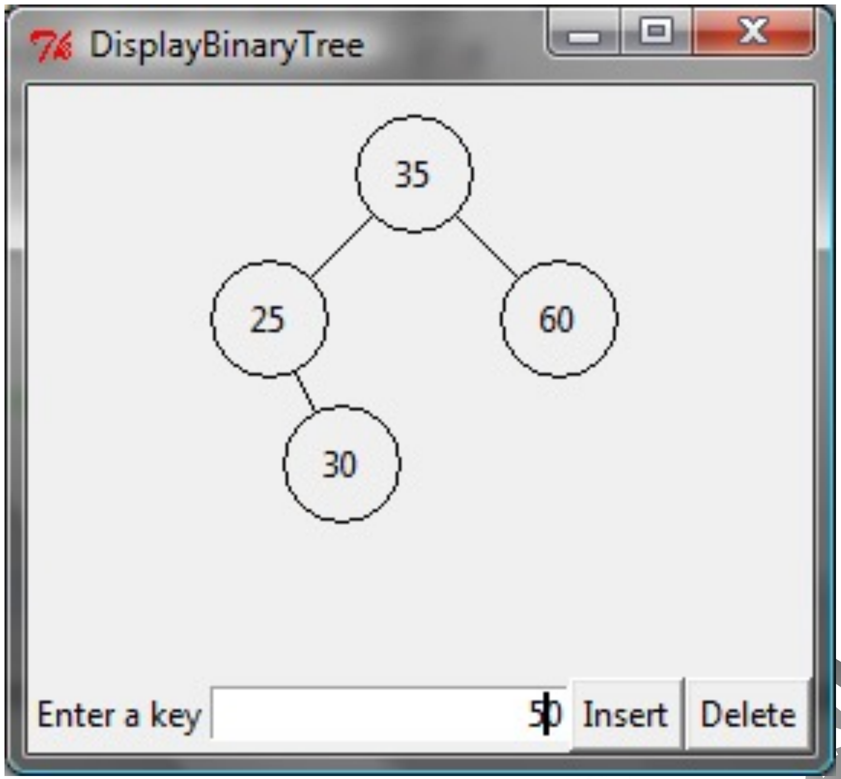
TestBinaryTreeDelete

binary tree time complexity

It is obvious that the time complexity for the inorder, preorder, and postorder is O(n), since each node is traversed only once. The time complexity for search, insertion and deletion is the height of the tree. In the worst case, the height of the tree is O(n).

BST Visualization





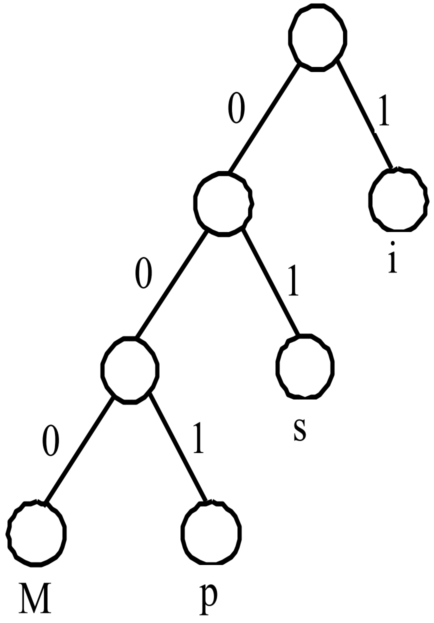
DisplayBST

Iterators

See Exercise 19.11.

Data Compression: Huffman Coding

In ASCII, every character is encoded in 8 bits. Huffman coding compresses data by using fewer bits to encode more frequently occurring characters. The codes for characters are constructed based on the occurrence of characters in the text using a binary tree, called the *Huffman coding tree*.



Character	Code	Frequency
M	000	1
p	001	2
s	01	4
i	1	4

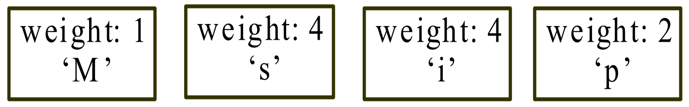
Constructing Huffman Tree

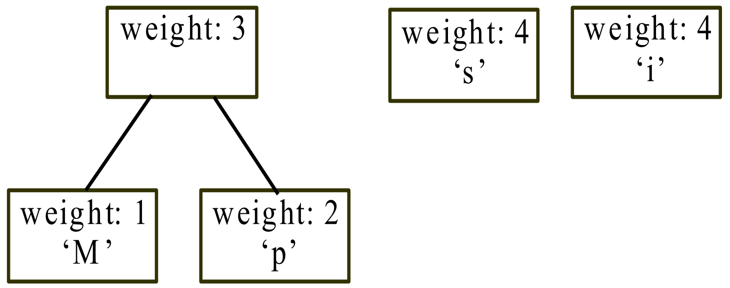
To construct a *Huffman coding tree*, use a greedy algorithm as follows:

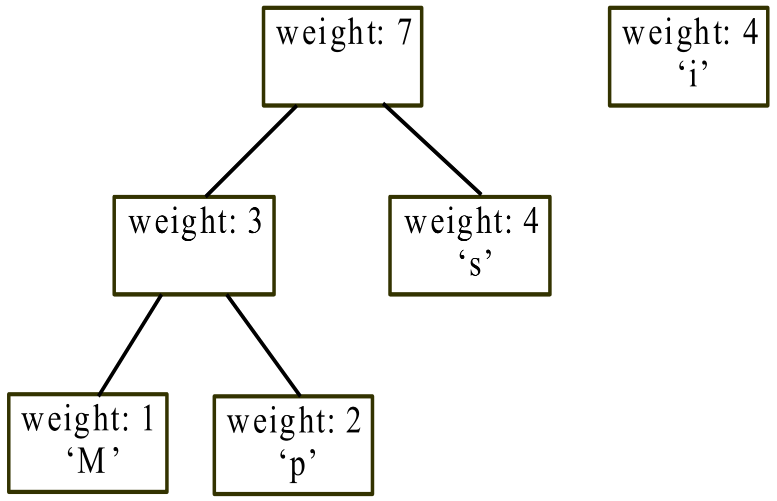
- Begin with a forest of trees. Each tree contains a node for a character. The weight of the node is the frequency of the character in the text.
- Repeat this step until there is only one tree:

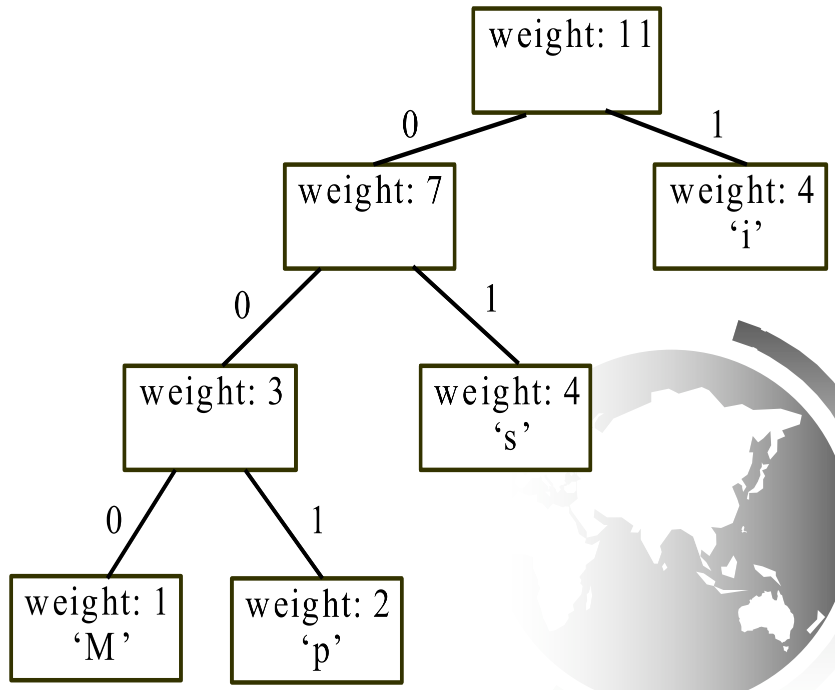
  Choose two trees with the smallest weight and create a new node as their parent. The weight of the new tree is the sum of the weight of the subtrees.

Constructing Huffman Tree









Constructing Huffman Tree

HuffmanCode