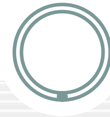
Trees

LECTURE 11

Objectives



- At the end of this lecture the learner is able to:
  - Explain the purpose the structure of tree
  - Understand the main operations on binary tree
  - Build and implement binary tree operations

Outlines

- Introduction
- Binary Trees
- A binary search tree
- Implementation of binary search tree

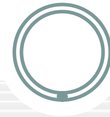
Introduction

- Arrays, Linked lists, stacks and queues are **linear data structures**.
- A tree is a *nonlinear, two-dimensional data structure with special properties.*
- binary trees are trees whose nodes all contain *two links (none,* one, or both of which may be NULL).

Binary Trees

- binary trees are trees whose nodes all contain *two links (none,* one, or both of which may be NULL).
- The root node is the *first node in a tree. Each link in* the root node refers to a child. The left child is the *first node in the left subtree, and the* right child is the *first node in the right subtree. The children of a node are called siblings.*
- A node with *no children is called a leaf node. Computer scientists normally draw trees*from the root node down—exactly the *opposite of trees in nature.*

Binary Trees



root node pointer

B

left subtree
of node
containing B

A

D

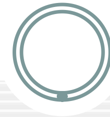
right subtree
of node
containing B

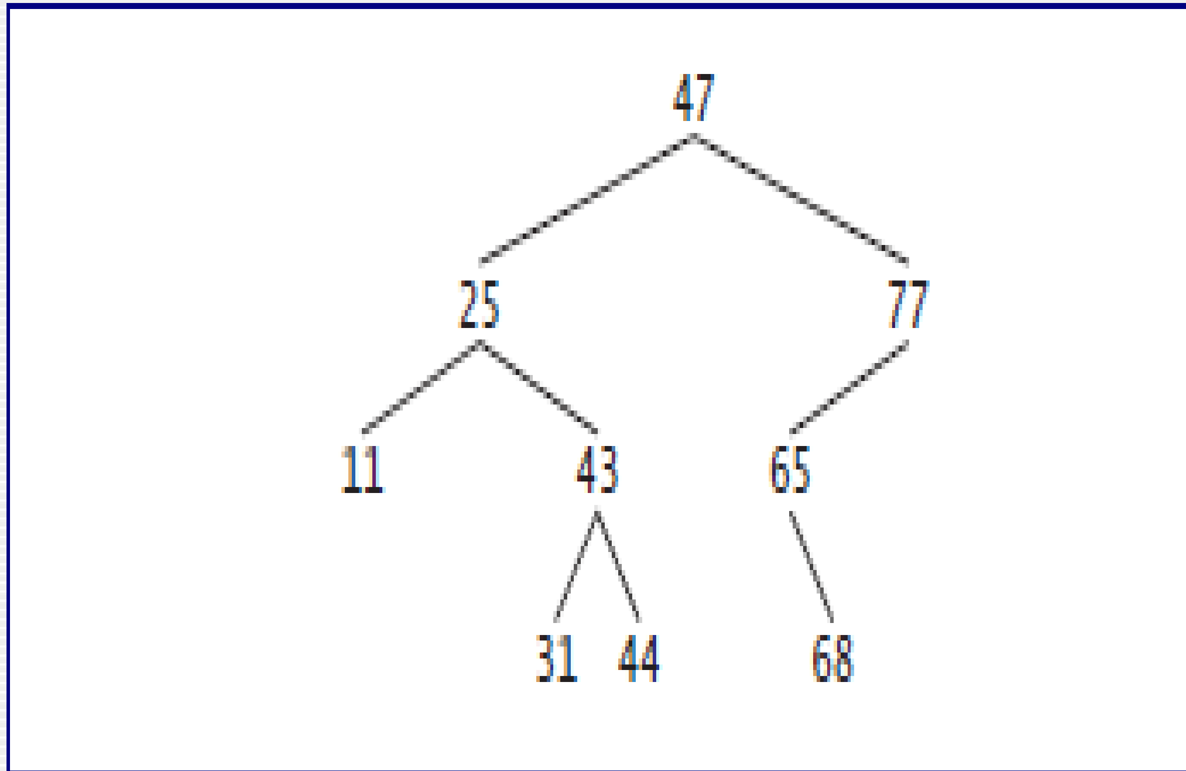
C

A binary search tree

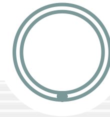
- A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in its parent node, and the values in any right subtree are greater than the value in its **parent node.**
- The shape of the binary search tree that corresponds to a set of data can *vary*, depending on the *order in which the values are inserted into the tree*

A binary search tree



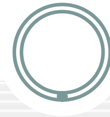


Implementation of binary search tree



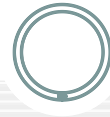
```c
// Fig. 12.19: fig12_19.c
// Creating and traversing a binary tree
// preorder, inorder, and postorder
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// self-referential structure
struct treeNode {
   struct treeNode *leftPtr; // pointer to left subtree
   int data; // node value
   struct treeNode *rightPtr; // pointer to right subtree
}; // end structure treeNode

typedef struct treeNode TreeNode; // synonym for struct tree
typedef TreeNode *TreeNodePtr; // synonym for TreeNode*

// prototypes
void insertNode( TreeNodePtr *treePtr, int value );
void inOrder( TreeNodePtr treePtr );
void preOrder( TreeNodePtr treePtr );
void postOrder( TreeNodePtr treePtr );
```

Implementation of binary search tree



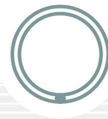
```c
// function main begins program execution
int main( void )
{
   unsigned int i; // counter to loop from 1-10
   int item; // variable to hold random values
   TreeNodePtr rootPtr = NULL; // tree initially empty

   srand( time( NULL ) );
   puts( "The numbers being placed in the tree are:" );

   // insert random values between 0 and 14 in the tree
   for ( i = 1; i <= 10; ++i ) {
      item = rand() % 15;
      printf( "%3d", item );
      insertNode( &rootPtr, item );
   } // end for

   // traverse the tree preOrder
   puts( "\n\nThe preOrder traversal is:" );
   preOrder( rootPtr );

   // traverse the tree inOrder
   puts( "\n\nThe inOrder traversal is:" );
   inOrder( rootPtr );

   // traverse the tree postOrder
   puts( "\n\nThe postOrder traversal is:" );
   postOrder( rootPtr );
} // end main
```

Implementation of binary search tree



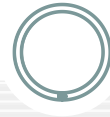
```c
// insert node into tree
void insertNode( TreeNodePtr *treePtr, int value )
{
   // if tree is empty
   if ( *treePtr == NULL ) {
      *treePtr = malloc( sizeof( TreeNode ) );

      // if memory was allocated, then assign data
      if ( *treePtr != NULL ) {
         ( *treePtr )->data = value;
         ( *treePtr )->leftPtr = NULL;
         ( *treePtr )->rightPtr = NULL;
      } // end if
      else {
         printf( "%d not inserted. No memory available.\n", value );
      } // end else
   } // end if
   else { // tree is not empty
      // data to insert is less than data in current node
      if ( value < ( *treePtr )->data ) {
         insertNode( &( ( *treePtr )->leftPtr ), value );
      } // end if
```

Implementation of binary search tree



```
      // data to insert is greater than data in current node
      else if ( value > ( *treePtr )->data ) {
         insertNode( &( ( *treePtr )->rightPtr ), value );
      } // end else if
      else { // duplicate data value ignored
         printf( "%s", "dup" );
      } // end else
   } // end else
} // end function insertNode

// begin inorder traversal of tree
void inOrder( TreeNodePtr treePtr )
{
   // if tree is not empty, then traverse
   if ( treePtr != NULL ) {
      inOrder( treePtr->leftPtr );
      printf( "%3d", treePtr->data );
      inOrder( treePtr->rightPtr );
   } // end if
} // end function inOrder

// begin preorder traversal of tree
void preOrder( TreeNodePtr treePtr )
{
   // if tree is not empty, then traverse
   if ( treePtr != NULL ) {
      printf( "%3d", treePtr->data );
      preOrder( treePtr->leftPtr );
      preOrder( treePtr->rightPtr );
   } // end if
} // end function preOrder
```

Implementation of binary search tree



```
// begin postorder traversal of tree
void postOrder( TreeNodePtr treePtr )
{
   // if tree is not empty, then traverse
   if ( treePtr != NULL ) {
      postOrder( treePtr->leftPtr );
      postOrder( treePtr->rightPtr );
      printf( "%3d", treePtr->data );
   } // end if
} // end function postOrder
```

End