Cpt S 122 – Data Structures

Data Structures
Trees

Nirmalya Roy

School of Electrical Engineering and Computer Science
Washington State University

Motivation

- Trees are one of the most important and extensively used data structures in computer science
- File systems of several popular operating systems are implemented as trees
  - For example

```
My Documents/
        My Pictures/
                0001.jpg
                0002.jpg
        My Videos/
                Nice.mpg
        My Music/
        School Files/
                CptS223/
                FunStuffs/
```

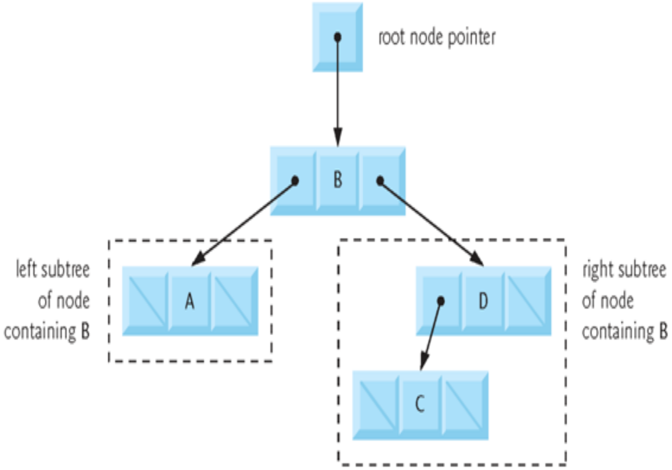
Topics

- Binary Trees
- Binary Search Tree
  - `insertNode, deleteNode`
  - `inOrder, preOrder, postOrder`
- Applications
  - Duplicate elimination, Searching etc

Trees

- Linked lists, stacks and queues are linear data structures.
- A tree is a *nonlinear*, *two-dimensional data structure* with special properties.
- Tree nodes contain *two or more* links.

Trees (Cont.)



- Binary trees
  - 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.

Trees (Cont.)

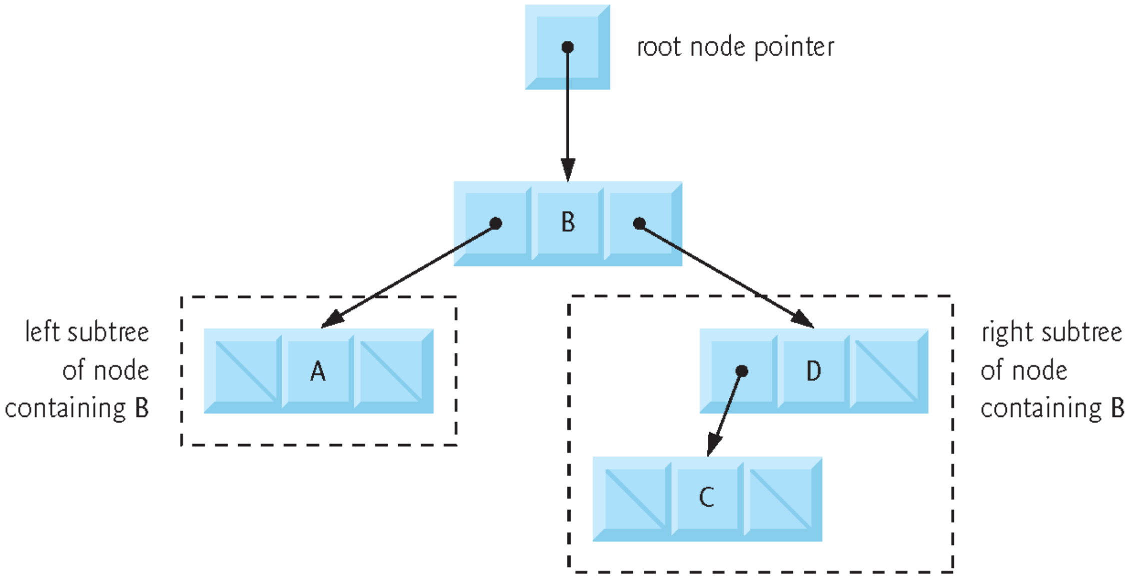
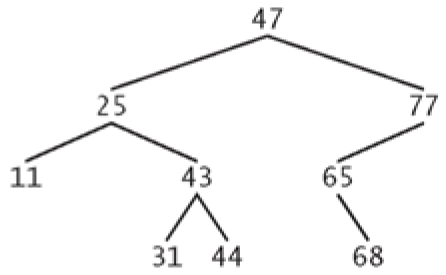


Fig. 12.17 | Binary tree graphical representation.

Trees (Cont.)

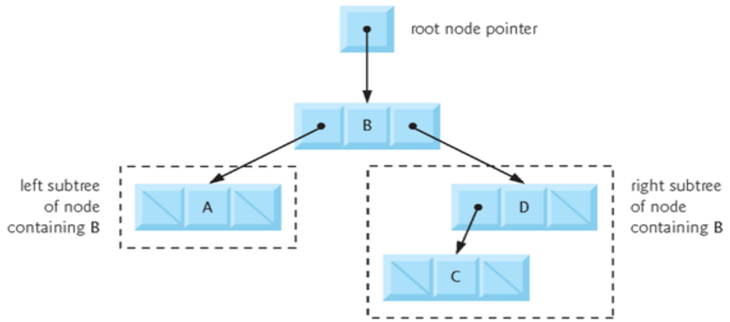
- A special binary tree is called a binary search tree.
- A binary search tree (with no duplicate node values) has following properties.
  - the values in any left subtree are less than the value in its parent node.
  - the values in any right subtree are greater than the value in its parent node.
- 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.



Trees (Cont.)

- Operations or functions
  - Inserting a node into the tree
  - Deleting a node from the tree
  - Searching a node in the tree
  - Traversals
    - Inorder
    - Preorder
    - Postorder

Trees self-referential structure




```
// 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 treeNode
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 );

```

Fig. 12.19 | Creating and traversing a binary tree. (Part 1 of 5.)

Trees Example

```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 );
```

Fig. 12.19 | Creating and traversing a binary tree. (Part 2 of 5.)

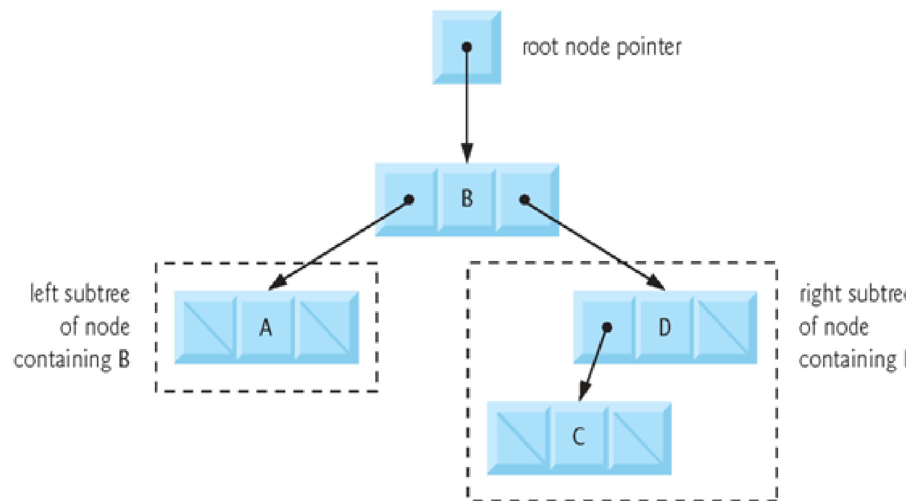
Function insertNode

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

// 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
```

Fig. 12.19 | Creating and traversing a binary tree. (Part 3 of 5.)

Function `insertNode`

```c
    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

       // 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
```



Function `insertNode`

- The function used to create a binary search tree is recursive.
- Function `insertNode` receives the *address of the tree* and an *integer* to be stored in the tree as arguments.
- *A node can be inserted only as a leaf node in a binary search tree.*

Function `insertNode` (Cont.)

- The steps for inserting a node in a binary search tree are as follows:
  - If `*treePtr` is `NULL`, create a new node.
  - Call `malloc`, assign the allocated memory to `*treePtr`,
  - Assign to `(*treePtr)->data` the integer to be stored,
  - Assign to `(*treePtr)->leftPtr` and `(*treePtr)->rightPtr` the value `NULL`
  - Return control to the caller (either `main` or a previous call to `insertNode`).

Function `insertNode` (Cont.)

- If the value of `*treePtr` is not `NULL` and the value to be inserted is less than `(*treePtr)->data`,
  - function `insertNode` is called with the address of `(*treePtr)->leftPtr` to insert the node in the left subtree of the node pointed to by `treePtr`.
- If the value to be inserted is *greater than* `(*treePtr)->data`,
  - function `insertNode` is called with the address of `(*treePtr)->rightPtr` to insert the node in the right subtree of the node pointed to by `treePtr`.
- Otherwise, the *recursive steps* continue until a `NULL` pointer is found, then Step *1* is executed to *insert the new node*.

Tree Traversal Function

- The functions used to traverse the tree are recursive.

- Traversal functions are `inOrder`, `preOrder` and `postOrder`.

- Each receive a *tree* (i.e., the *pointer to the root node of the tree*) and *traverse* the tree.

Traversals: Function `inOrder`

- The steps for an `inOrder` traversal are: left, root, right
  - Traverse the left subtree `inOrder`.
  - Process the value in the node.
  - Traverse the right subtree `inOrder`.
- The value in a node is not processed until the values in its left subtree are processed.
- The `inOrder` traversal of the tree is:
  - `6 13 17 27 33 42 48`

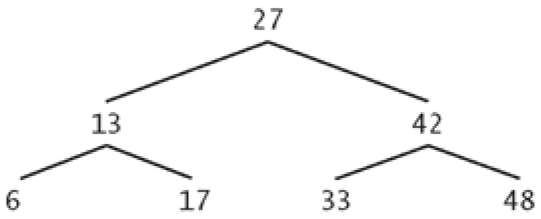
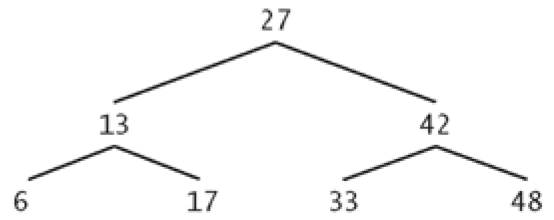


Fig. 12.21 | Binary search tree with seven nodes.

inOrder Function

- `inOrder` traversal is: left, root, right

```c
// 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
```

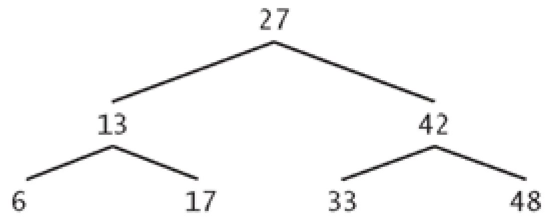


Traversals: Function `inOrder`

- The `inOrder` traversal of a binary search tree prints the node values in *ascending* order.
- The process of creating a binary search tree actually sorts the data
  - this process is called the binary tree sort.

Traversals: Function `preOrder`

- The steps for a `preOrder` traversal are: root, left, right
  - Process the value in the node.
  - Traverse the left subtree `preOrder`.
  - Traverse the right subtree `preOrder`.
- The value in each node is processed as the node is visited.
  - After the value in a given node is processed, the values in the left subtree are processed, then those in the *right* subtree are processed.
- The `preOrder` traversal of the tree is:
  - `27 13 6 17 42 33 48`



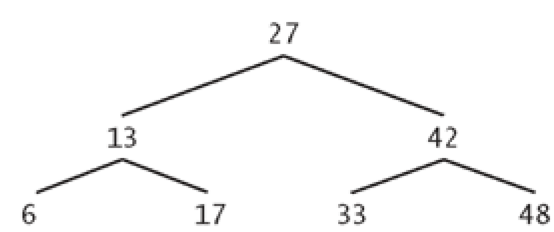
`preOrder` Function

- `preOrder` traversal is: root, left, right

```
// 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
```

Traversals: Function `postOrder`

- The steps for a `postOrder` traversal are: left, right, root
  - Traverse the left subtree `postOrder`.
  - Traverse the right subtree `postOrder`.
  - Process the value in the node.
- The value in each node is not printed until the values of its children are printed.
- The `postOrder` traversal of the tree is:
  - `6 17 13 33 48 42 27`



`postOrder` Function

- `postOrder` traversal is: left, right, root

```c
// 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
```

BST Applications: Duplicate Elimination

- The binary search tree facilitates duplicate elimination.
- An attempt to insert a duplicate value will be recognized
  - a duplicate will follow the same “go left” or “go right” decisions on each comparison as the original value did.
- The duplicate will eventually be compared with a node in the tree containing the same value.
- The duplicate value may simply be discarded at this point.

Binary Tree Search

- Searching a binary tree for a value that matches a key value is fast.
- If the tree is tightly packed, each level contains about twice as many elements as the previous level.
- A binary search tree with n elements would have a maximum of $\log_2 n$ levels.
  - a maximum of $\log_2 n$ comparisons would have to be made either to find a match or to determine that no match exists.
- Searching a (tightly packed) 1,000,000 element binary search tree requires no more than 20 comparisons
  - $2^{20} > 1{,}000{,}000$.

Other Binary Tree Operations

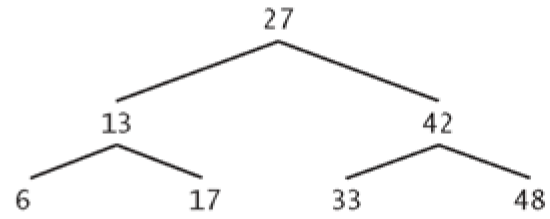
- The *level order traversal* of a binary tree visits the nodes of the tree *row-by-row* starting at the root node level.
  - On each level of the tree, the nodes are visited from left to right.
  - The level order traversal is not a recursive algorithm.

Exercise

- Implement the *level order binary tree traversal* using a common data structure we have discussed in the class.
  - Write the pseudo code of this algorithm.

Level Order Binary Tree Traversal

- Use the Queue data structure to control the output of the level order binary tree traversal.
- Algorithm
  - `Insert/enqueue` the root node in the queue
  - While there are nodes left in the queue,
    - `Get/dequeue` the node in the queue
    - Print the node's value
    - If the pointer to the left child of the node is not null
      - `Insert/enqueue` the left child node in the queue
    - If the pointer to the right child of the node is not null
      - `Insert/enqueue` the right child node in the queue



Other Common Tree Data Strictures

- Binary search trees (BSTs)
  - Support $O(\log_2 N)$ operations
  - Balanced trees
    - AVL trees, Splay trees
- B-trees for accessing secondary storage

Conclusions

- Accessing elements in a linear linked list can be prohibitive especially for large amounts of input.
- Trees are simple data structures for which the running time of most operations is O(log N) on average.
- For example, if N = 1 million:
  - Searching an element in a linear linked list requires at most O(N) comparisons (i.e. 1 million comparisons)
  - Searching an element in a binary search tree (a kind of tree) requires $O(\log_2 N)$ comparisons ($\approx$ 20 comparisons)