Binary Search Trees

October 21, 2020

Binary Search Trees

Binary Search Trees are a special kind of binary tree.

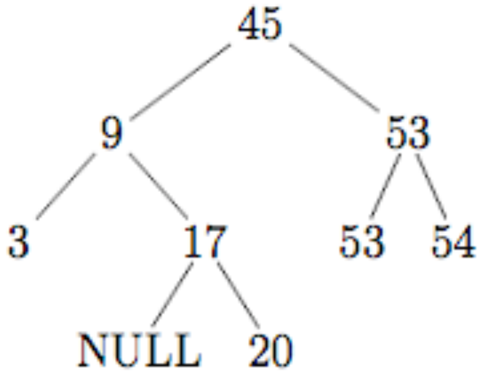
Q: Why are they special?
A: Organized (structure)

Q: How are they organized?
A: Using a couple of rules

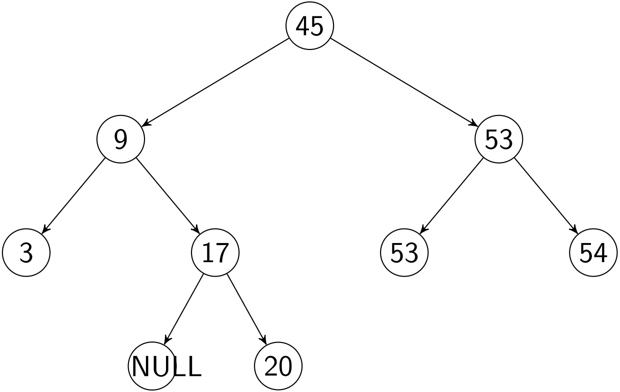
Binary Search Tree Storage Rules

1. Every entry in n's left subtree is less than or equal to the entry in node n.
2. Every entry in n's right subtree is greater than (or equal to) the entry in node n.

Example Binary Search Tree



Example Binary Search Tree



Utility of Binary Search Trees

Q: What is the benefit of this organization?

A: Contents are *sorted* when processed using an *inorder* traversal.

Sample Binary Search Tree


```
static DATA_TYPE A[] = { 51, 31, 67, 23, 43, 57, 83, 17, 29, 79 };
static int nA = sizeof(A)/sizeof(DATA_TYPE);

for( int i = 0 ; i < nA ; i++ )
   t1.AddNode( A[i] );

t1:
        83
            79
    67
        57
51
        43
    31
            29
        23
            17
```

Binary Search Trees – Interface


```
/*  BSTree2.h
 *
 *  Binary Search Tree class Interface WITH deletion.
 */

#ifndef _BSTREE_H_
#define _BSTREE_H_

typedef int  DATA_TYPE;  // Type of node's data

class BinarySearchTree
{
private:
    typedef struct  BSTreeNode
    {
       DATA_TYPE     data;
       BSTreeNode   *leftPtr;
       BSTreeNode   *rightPtr;
    } *TreePtr;

    TreePtr rootPtr;   //  root of the BST
```

Binary Search Tree – Private Methods


```
void      InitBSTree()
              { rootPtr = NULL; }

void      DeleteBST( TreePtr& treePtr );

void      DeleteNode(
                  TreePtr& treePtr, DATA_TYPE theItem );
void      DeleteNodeItem( TreePtr& treePtr );
void      ProcessLeftMost(
                  TreePtr& treePtr, DATA_TYPE& theItem );

bool      IsLeaf( TreePtr treePtr );

TreePtr   SearchNodeInBST( TreePtr treePtr,
                           DATA_TYPE searchKey );
```

Binary Search Trees – Private Print Methods


```
void     PrintBST_InOrder(   TreePtr treePtr );
void     PrintBST_PreOrder(  TreePtr treePtr );
void     PrintBST_PostOrder( TreePtr treePtr );

void     PrintBST_BackwardInOrder(
                             TreePtr treePtr, int depth );
```

Binary Search Trees – Public Methods


```
public:
    BinarySearchTree()    { InitBSTree(); }
    ~BinarySearchTree();

    bool      IsEmpty()
                  { return (rootPtr == NULL); }

    void      AddNode( DATA_TYPE newData );
    void      SearchNode( DATA_TYPE searchKey );
    void      DeleteNode( DATA_TYPE val );

    void      PrintTree();

    void      PrintInOrder();
    void      PrintPreOrder();
    void      PrintPostOrder();

    void      PrintBackwardInOrder();
};
#endif
```

Binary Search Tree Implementation


```
/*  BSTree2.cpp
 *
 *  Binary Search Tree Implementation with Deletion.
 */

#include <iostream.h>

#include "BSTree2.h"

//#define DEBUG_DELETE  /*  Uncomment for extra debugging  */


//  ~BinarySearchTree() --- Delete BST object

BinarySearchTree::~BinarySearchTree()
{
   DeleteBST( rootPtr );
}
```

IsLeaf()


```
// IsLeaf()  --- Test if a node is a leaf

bool  BinarySearchTree::IsLeaf( TreePtr treePtr)
{
   return ((treePtr->leftPtr  == NULL) &&
           (treePtr->rightPtr == NULL)    );
}
```

AddNode()

```
// AddNode()
//   Add (insert) new item into the BST, whose
//   root node is pointed to by "rootPtr".  If
//   the data already exists, it is ignored.

void BinarySearchTree::AddNode( DATA_TYPE newData )
{
   TreePtr newPtr;

   newPtr = new BSTreeNode;
   // Add new data in the new node's data field
   newPtr->data     = newData;
   newPtr->leftPtr  = NULL;
   newPtr->rightPtr = NULL;

   //  If the BST is empty, insert the new data in root
   if( rootPtr == NULL )
   {
      rootPtr = newPtr;
   }
```

AddNode()


```
else   //  Look for the insertion location
{
   TreePtr   treePtr = rootPtr;
   TreePtr   targetNodePtr;

   while( treePtr != NULL )
   {
     targetNodePtr = treePtr;
     if( newData == treePtr->data )
        // Found same data; ignore it.
        return;
     else if( newData < treePtr->data )
        // Search left subtree for insertion location
        treePtr = treePtr->leftPtr;
     else   //  newData > treePtr->data
        // Search right subtree for insertion location
        treePtr = treePtr->rightPtr;
   }
```

AddNode()


```
        //  "targetNodePtr" is the pointer to the
        //  parent of the new node.  Decide where
        //  it will be inserted.
        if( newData < targetNodePtr->data )
           targetNodePtr->leftPtr = newPtr;
        else  // insert it as its right child
           targetNodePtr->rightPtr = newPtr;
    }
}
```

DeleteBST()


```
// DeleteBST()
//   Delete an entire BST.  All memory is released
//   using a "PostOrder" traversal method.

void BinarySearchTree::DeleteBST( TreePtr& treePtr )
{
   if( treePtr != NULL )
   {
      DeleteBST( treePtr->leftPtr );
      DeleteBST( treePtr->rightPtr );

      delete  treePtr;
      treePtr = NULL;
   }
}
```

DeleteNode() – Later

The process is somewhat complicated. We will discuss it later.

SearchNode() – public


```
void BinarySearchTree::SearchNode( DATA_TYPE searchKey )
{
   TreePtr srchPtr = NULL;

   srchPtr = SearchNodeInBST( rootPtr, searchKey );
   if( srchPtr != NULL )
   {
      cout << "\n Node: " << srchPtr->data << " found in the BST" << end
   }
   else
   {
      cout << "\n Node: " << searchKey << " not found" << endl;
   }
}
```

SearchNodeInBST() – private


```
// SearchNodeInBST()
//   Find a given node by "key" in BST.  If successful, it
//   returns the pointer that points to the node with "key";
//   otherwise, it returns NULL.  It uses preorder traversal.

BinarySearchTree::TreePtr
BinarySearchTree::SearchNodeInBST(
              TreePtr treePtr, DATA_TYPE key )
{
   if( treePtr != NULL ) {
      if( key == treePtr->data )
         return  treePtr;
      else if( key < treePtr->data )
         // Search for "key" in left subtree
         SearchNodeInBST( treePtr->leftPtr, key );
      else   // (key > tree_ptr->data)
         // Search for "key" in right subtree
         SearchNodeInBST( treePtr->rightPtr, key );
   }
   else {
     return NULL;
   }
```

PrintTree() – public

```
// PrintTree()
//   Print a BST tree uses InOrder traversal by default.

void BinarySearchTree::PrintTree()
{
    PrintBST_InOrder( rootPtr );
}
```

PrintInOrder()


```
// PrintInOrder()
//   Print BST using InOrder traversal

void BinarySearchTree::PrintInOrder()
{
   PrintBST_InOrder( rootPtr );
}

void BinarySearchTree::PrintBST_InOrder(
                TreePtr treePtr )
{
   if( treePtr != NULL)
   {
     // Print left BST subtree
     PrintBST_InOrder( treePtr->leftPtr );
     // Print Root node data
     cout << treePtr->data << endl;
     // Print right BST subtree
     PrintBST_InOrder( treePtr->rightPtr );
   }
}
```

PrintPreOrder()


```
// PrintPreOrder()
//   Print BST using PreOrder traversal

void BinarySearchTree::PrintPreOrder()
{
   PrintBST_PreOrder( rootPtr );
}

void BinarySearchTree::PrintBST_PreOrder(
                TreePtr treePtr )
{
   if( treePtr != NULL )
   {
     // Print node data
     cout << treePtr->data << endl;
     // Print left subtree
     PrintBST_PreOrder( treePtr->leftPtr );
     // Print right subtree
     PrintBST_PreOrder( treePtr->rightPtr );
   }
}
```

PrintPostOrder()


```
// PrintPostOrder()
//   Print BST using PostOrder traversal

void BinarySearchTree::PrintPostOrder()
{
   PrintBST_PostOrder( rootPtr );
}

void BinarySearchTree::PrintBST_PostOrder(
                TreePtr treePtr )
{
   if( treePtr != NULL )
   {
     // Print left BST subtree
     PrintBST_PostOrder( treePtr->leftPtr );
     // Print right BST subtree
     PrintBST_PostOrder( treePtr->rightPtr );
     // Print node data
     cout << treePtr->data << endl;
   }
}
```

PrintBackwardInOrder()


```
//   Print BST using InOrder traversal

void BinarySearchTree::PrintBackwardInOrder()
{
   PrintBST_BackwardInOrder( rootPtr, 0 );
}

void BinarySearchTree::PrintBST_BackwardInOrder(
                TreePtr treePtr, int depth )
{
   const int INDENT = 4;

   if( treePtr != NULL ) {
        // Print right BST subtree
     PrintBST_BackwardInOrder( treePtr->rightPtr, depth+1 );
        // Print data in root node
     //cout << setw(INDENT*depth) << " ";
     for( int i = 0 ; i < INDENT*depth ; i++ ) cout << " ";
     cout << treePtr->data << endl;
        // Print left BST subtree
     PrintBST_BackwardInOrder( treePtr->leftPtr, depth+1 );
   }
```

Binary Search Tree Testing

```
/* TestBSTree.cpp
 *   Test Binary Search Tree (BST)
 */

#include <iostream.h>

#include "BSTree2.h"


int main()
{
   static DATA_TYPE A[8] = { 15, 53, 13, 61,
                              57, 47, 21, 51 };
   static int nA = sizeof(A)/sizeof(DATA_TYPE);

   static DATA_TYPE B[] = { 48, 54, 14, 52,
                              8, 16, 63, 10, 1 };
   static int nB = sizeof(B)/sizeof(DATA_TYPE);
```

Build Trees


```
BinarySearchTree  t1;
BinarySearchTree  t2;

for (int i = 0; i < nA; i++)
    t1.AddNode( A[i] );

for (int i = 0; i < nB; i++)
    t2.AddNode( B[i] );
```

Test Trees


```
    //  Test display methods
cout << "\n The Binary Search Tree using ";
cout << "a Backward InOrder traversal:" << endl;
t1.PrintBackwardInOrder();

cout << "\n The Binary Search Tree using ";
cout << "an InOrder traversal:" << endl;
t1.PrintInOrder();

cout << "\n The Binary Search Tree using ";
cout << "a PreOrder traversal:" << endl;
t1.PrintPreOrder();

cout << "\n The Binary Search Tree using ";
cout << "a PostOrder traversal:" << endl;
t1.PrintPostOrder();
```

Deletion/Search

```
        //  Test deletion
    cout << "\n\n Deleting node(s)" << endl;
    cout << "    deleting  61 and 47" << endl;
    t1.DeleteNode( 61 );
    t1.DeleteNode( 47 );
    t1.PrintBackwardInOrder();

    cout << "\n\n Deleting entire tree" << endl;
    cout << "      (Tree should be empty)" << endl;
    t1.~BinarySearchTree();
    t1.PrintInOrder();

        //  Display second tree
    cout << "\n The Binary Search Tree using ";
    cout << "a Backward InOrder traversal:" << endl;
    t2.PrintBackwardInOrder();

    t2.SearchNode(45);
    t2.SearchNode(48);

    return EXIT_SUCCESS;
}
```

Output

```
 The Binary Search Tree using a Backward InOrder traversal:
        61
            57
    53
            51
        47
            21
15
    13
```

Output: InOrder Traversal


```
The Binary Search Tree using an InOrder traversal:

13
15
21
47
51
53
57
61
```

Output: PreOrder Traversal


```
The Binary Search Tree using a PreOrder traversal:

15
13
53
47
21
51
61
57
```

Output: PostOrder Traversal

```
 The Binary Search Tree using a PostOrder traversal:

13
21
51
47
57
61
53
15
```