Binary Search Tree

A **Binary Search Tree** is a Binary Tree in which, at every node \(v\), the values stored in the left subtree of \(v\) are less than the value at \(v\) and the values stored in the right subtree are greater.

The elements in the BST must be comparable. Duplicates are not allowed in our discussion.

Note that each subtree of a BST is also a BST.
A BST of integers

Describe the values which might appear in the subtrees labeled A, B, C, and D
BST Implementation

The SearchTree ADT

– A search tree is a binary search tree which stores homogeneous elements with no duplicates.
– It is dynamic.
– The elements are ordered in the following ways
  • inorder -- as dictated by operator<
  • preorder, postorder, levelorder -- as dictated by the structure of the tree
BST Implementation

template <typename Comparable>
class BinarySearchTree
{
  public:
    BinarySearchTree( );
    BinarySearchTree( const BinarySearchTree & rhs );
    ~BinarySearchTree( );

    const Comparable & findMin( ) const;
    const Comparable & findMax( ) const;
    bool contains( const Comparable & x ) const;
    bool isEmpty( ) const;
    void printTree( ) const;

    void makeEmpty( );
    void insert( const Comparable & x );
    void remove( const Comparable & x );
BST Implementation (2)

```cpp
const BinarySearchTree &
    operator=( const BinarySearchTree & rhs );

private:
    struct BinaryNode
    {
        Comparable element;
        BinaryNode *left;
        BinaryNode *right;

        BinaryNode( const Comparable & theElement,
                   BinaryNode *lt, BinaryNode *rt )
            : element( theElement ), left( lt ), right( rt )
            { }
    };
```
BST Implementation (3)

// private data
  BinaryNode *root;

// private recursive functions
void insert( const Comparable & x, BinaryNode * & t ) const;
void remove( const Comparable & x, BinaryNode * & t ) const;
BinaryNode * findMin( BinaryNode * t ) const;
BinaryNode * findMax( BinaryNode * t ) const;
bool contains( const Comparable & x, BinaryNode * t ) const;
void makeEmpty( BinaryNode * & t );
void printTree( BinaryNode * t ) const;
BinaryNode * clone( BinaryNode * t ) const;
};
BST “contains” method

// Returns true if x is found (contained) in the tree.
bool contains( const Comparable & x ) const
{
    return contains( x, root );
}

// Internal (private) method to test if an item is in a subtree.
// x is item to search for.
// t is the node that roots the subtree.
bool contains( const Comparable & x, BinaryNode *t ) const
{
    if( t == NULL )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true;   // Match
Performance of “contains”

Searching in randomly built BST is $O(\lg n)$ on average
– but generally, a BST is not randomly built

Asymptotic performance is $O(\text{height})$ in all cases
Predecessor in BST

Predecessor of a node v in a BST is the node that holds the data value that immediately precedes the data at v in order.

Finding predecessor

- v has a left subtree
  - then predecessor must be the largest value in the left subtree (the rightmost node in the left subtree)
- v does not have a left subtree
  - predecessor is the first node on path back to root that does not have v in its left subtree
Successor in BST

Successor of a node $v$ in a BST is the node that holds the data value that immediately follows the data at $v$ in order.

Finding Successor

- $v$ has right subtree
  - successor is smallest value in right subtree (the leftmost node in the right subtree)
- $v$ does not have right subtree
  - successor is first node on path back to root that does not have $v$ in its right subtree
The remove Operation

// Internal (private) method to remove from a subtree.
// x is the item to remove.
// t is the node that roots the subtree.
// Set the new root of the subtree.
void remove( const Comparable & x, BinaryNode * & t )
{
    if ( t == NULL )
        return;          // x not found; do nothing
    if ( x < t->element )
        remove( x, t->left );
    else if ( t->element < x )
        remove( x, t->right );
    else if ( t->left != NULL && t->right != NULL ) // two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else // zero or one child
    {
        BinaryNode *oldNode = t;
        t = ( t->left != NULL ) ? t->left : t->right;
        delete oldNode;
    }
}
The insert Operation

// Internal method to insert into a subtree.
// x is the item to insert.
// t is the node that roots the subtree.
// Set the new root of the subtree.

void insert( const Comparable &x, BinaryNode * &t )
{
  if( t == NULL )
    t = new BinaryNode( x, NULL, NULL );
  else if( x < t->element )
    insert( x, t->left );
  else if( t->element < x )
    insert( x, t->right );
  else
    ; // Duplicate; do nothing
}
Implementation of makeEmpty

template <typename Comparable>
void BinarySearchTree<Comparable>::
makeEmpty()  // public makeEmpty ()
{
    makeEmpty( root );  // calls private makeEmpty ()
}

template <typename Comparable>
void BinarySearchTree<Comparable>::
makeEmpty( BinaryNode<Comparable> * & t ) const
{
    if ( t != NULL ) {
        // post order traversal
        makeEmpty ( t->left );
        makeEmpty ( t->right );
        delete t;
    }
    t = NULL;
}
Implementation of Assignment Operator

// operator= makes a deep copy via cloning
const BinarySearchTree & operator=( const BinarySearchTree & rhs )
{
    if( this != &rhs )
    {
        makeEmpty(); // free LHS nodes first
        root = clone( rhs.root ); // make a copy of rhs
    }
    return *this;
}

//Internal method to clone subtree -- note the recursion
BinaryNode * clone( BinaryNode *t ) const
{
    if( t == NULL )
        return NULL;
    return new BinaryNode(t->element, clone(t->left), clone(t->right));
}
Performance of BST methods

What is the asymptotic performance of each of the BST methods?

<table>
<thead>
<tr>
<th>Method</th>
<th>Best Case</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>findMin/Max</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>makeEmpty</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>assignment</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2/21/2006
Building a BST

Given an array/vector of elements, what is the performance (best/worst/average) of building a BST from scratch?
Tree Iterators

As we know there are several ways to traverse through a BST. For the user to do so, we must supply different kind of iterators. The iterator type defines how the elements are traversed.

- `InOrderIterator<T> *InOrderBegin();`
- `PerOrderIterator<T> *PreOrderBegin();`
- `PostOrderIterator<T> *PostOrderBegin();`
- `LevelOrderIterator<T> *LevelOrderBegin();`
Using Tree Iterator

main ( )
{
    Tree<int> tree;

    // store some ints into the tree

    InOrderIterator<int> itr = tree.InOrderBegin( );
    while (itr.HasNext( ))
    {
        int x = itr.Next( );

        // do something with x
    }
}
InOrder Tree Iterator Implementation
Approach 1: Store traversal in list (private data member). Return iterator for list.

template <typename T>
InOrderIterator<T> BinaryTree::InorderBegin()
{
    m_theList = new List<T>;
    FillListInorder(m_theList,getRoot());
    return m_theList->GetIterator();
}

template <typename T>
void FillListInorder(List<T> *lst, BinaryNode<T> *node)
{
    if (node == NULL) return;
    FillListInorder(lst, node->left);
    lst->Append(node->data);
    FillListInorder(lst, node->right);
}
InOrder Tree Iterator Implementation (2)

Approach 2: store traversal in stack to mimic recursive traversal

```cpp
template <typename T>
class InOrderIterator
{
    private:
        Stack<BinaryNode<T>> m_stack;

    public:
        InOrderIterator(BinaryNode<T> *t);

        bool HasNext() // aka end()
        { return !m_stack.isEmpty(); }

        T Next(); // aka op++
};
```

2/21/2006
InOrder Tree Iterator Implementation (3)

template <class T>
InOrderIterator<T>::InOrderIterator( BinaryNode<T> *t )
{
    BinaryNode<T> *v = t->GetRoot();
    while (v != NULL) {
        m_stack.Push(v);  // push root
        v = v->left;      // and all left descendants
    }
}
InOrder Tree Iterator Implementation (4)

template <typename T>
T InOrderIterator<T>::Next()
{
    BinaryNode<T> *top = m_stack.Top();
    m_stack.Pop();
    BinaryNode<T> *v = top->right;
    while (v != NULL) {
        m_stack.Push(v);  // push right child
        v = v->left;      // and all left descendants
    }
    return top->element;
}
More Recursive Binary (Search) Tree Functions

- `bool isBST ( BinaryNode<T> *t )` returns true if the Binary tree is a BST

- `const T& findMin( BinaryNode<T> *t )` returns the minimum value in a BST

- `int CountFullNodes ( BinaryNode<T> *t )` returns the number of full nodes (those with 2 children) in a binary tree

- `int CountLeaves( BinaryNode<T> *t )` counts the number of leaves in a Binary Tree