CMSC 341

Binary Search Trees

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.

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
- Each BST maintains a simple object, known as ITEM_NOT_FOUND, that is guaranteed to not be an element of the tree. ITEM_NOT_FOUND is provided to the constructor. (author’s code)

BinarySearchTree class

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

   const Comparable& findMin() const;
   const Comparable& findMax() const;
   const Comparable& find(const Comparable& x) const;
   bool isEmpty() const;
   void printTree() const;
   void makeEmpty();
   void insert (const Comparable& x);
   void remove (const Comparable& x);
   const BinarySearchTree &operator=(const
                       BinarySearchTree& rhs);
```

BinarySearchTree class (cont)

```
private:
   BinaryNode<Comparable> *root;
   const Comparable ITEM_NOT_FOUND;
   const Comparable&
        elementAt(BinaryNode<Comparable> *t) const;
   void insert (const Comparable& x,
        BinaryNode<Comparable> *& t) const;
   void remove (const Comparable& x,
        BinaryNode<Comparable> *& t) const;
   BinaryNode<Comparable>*
        findMin(BinaryNode<Comparable>* t const;
   BinaryNode<Comparable>*
        findMax(BinaryNode<Comparable>* t)const;
   BinaryNode<Comparable>*
        find(const Comparable& x, BinaryNode<Comparable>* t) const;
   void makeEmpty(BinaryNode<Comparable> *& t) const;
   void printTree(BinaryNode<Comparable* t) const;
   BinaryNode<Comparable>* clone(BinaryNode<Comparable>* t)const;
};
```

BinarySearchTree Implementation

```
template <class Comparable>
const Comparable &BinarySearchTree<Comparable> ::
find(const Comparable& x) const {
        return elementAt(find (x, root));
}

template <class Comparable>
const Comparable& BinarySearchTree<Comparable> ::
elementAt(BinaryNode<Comparable>* t) const {
   return t == NULL ? ITEM_NOT_FOUND : t->element;
}

template <class Comparable>
BinaryNode<Comparable>* BinarySearchTree<Comparable> ::
find(const Comparable& x, BinaryNode<Comparable>* t) const
{
   if (t == NULL) return NULL;
   else if (x < t->element) return find(x, t->left);
   else if (t->element < x) return find(x, t->right);
   else return t; // Match
}
```

Performance of find

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

```
template <class Comparable>
void BinarySearchTree<Comparable>::
remove(const Comparable& x, BinaryNode<Comparable> *& t) const
{
   if (t == NULL)
        return;         // item 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)) {
        t->element = (findMin (t->right))->element;
        remove (t->element, t->right);
   } else {
        BinaryNode<Comparable> *oldNode = t;
        t = (t->left != NULL) ? t->left : t->right;
        delete oldNode;
   }
}
```

The insert Operation

```
template <class Comparable>
void BinarySearchTree<Comparable>::
insert(const Comparable& x)   // public insert( )
{
   insert (x, root);          // calls private insert( )
}

template <class Comparable>
void BinarySearchTree<Comparable>::
insert(const Comparable& x, BinaryNode<Comparable> *&t ) const
{
   if (t == NULL)
        t = new BinaryNode<Comparable>(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 <class Comparable>
void BinarySearchTree<Comparable>::
makeEmpty()                        // public makeEmpty ()
{
   makeEmpty(root);                // calls private makeEmpty ( )
}

template <class 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;
}
```

Tree Iterators

Could provide separate iterators for each desired order

- `Iterator<T> *GetInorderIterator();`
- `Iterator<T> *GetPreorderIterator();`
- `Iterator<T> *GetPostorderIterator ();`
- `Iterator<T> *GetLevelorderIterator ();`

Tree Iterator Implementation

Approach 1: Store traversal in list (private data member). Return iterator for list.

```
Iterator<T> BinaryTree::GetInorderIterator() {
    m_theList = new ArrayList<T>;
    FullListInorder(m_theList, getRoot());
    return m_theList->GetIterator();
}

void FillListInorder(ArrayList<T> *lst, Bnode<T> *node)
{
   if (node == NULL) return;
   FillListInorder(lst, node->left);
   lst->Append(node->data);
   FillListInorder(lst, node->right);
}
```

Tree Iterators (cont)

Approach 2: store traversal in stack to mimic recursive traversal

```
template <class T>
class InOrderIterator
{
  private:
      Stack<* BNode<T> > m_stack;

  public:
      InOrderIterator(BinaryTree<T> *t);
      bool hasNext()    // aka isPastEnd
            {return !m_stack.isEmpty(); }
      T Next();         // aka advance()
};
```

Tree Iterators (cont'd)

```
template <class T>
InOrderIterator<T>::InOrderIterator(BinaryTree<T> *t)
{
   BNode<T> *v = t->getRoot();
   while (v != NULL) {
        m_stack.Push(v);     // push root
        v = v->left;         // and all left descendants
   }
}
```

Tree Iterators (cont'd)

```
template <class T>
T InOrderIterator<T>::Next()
{
   Bnode<T> *top = m_stack.Top();
   m_stack.Pop();
   BNode<T> *v = top->right;
   while (v != NULL) {
        m_stack.Push(v);     // push right child
        v = v->left;         // and all left descendants
   }
    return top->element;
}
```