List and Binary Tree Iterator Implementation

Lecture 37
Section 9.4

Robb T. Koether

Hampden-Sydney College

Wed, Apr 22, 2009

Outline

List Traversals

Definition (Traverse)

To traverse a list is to move systematically through its nodes, "visiting" each node along the way. Forward traversals go from head to tail. Reverse traversals go from tail to head.

- The meaning of "visiting" a node is left unspecified at this point.
- The meaning will be specified at the time of the traversal.

The Traversal Function

The traverse() Function

```
void traverse(void (*visit)(Iterator&));
```

- Introduce a new List member function `traverse()`.
- The parameter `visit` is a pointer to a function.
- The `visit()` function has prototype

```
void visit(Iterator& it);
```

Traversals and Iterators

Traversal Implementation

```
void traverse(void (*visit)(Iterator&))
{
    for (Iterator it = begin(); it != end(); ++it)
        visit(it);
    return;
}
```

- The `traverse()` function is implemented as a **`for`** loop.

Example - Print the List

The `print()` Function

```
void print(Iterator& it)
{
    cout << *it << endl;
    return;
}
        ⋮
list.traverse(print);
```

- For example, we could write a `print()` function and then use `traverse()` to print all the values in the list.

Reverse Iterators

- A reverse iterator is an iterator that advances in the opposite direction, from tail to head.
- It is initialized to the last element in the list.
- It "advances" until it has gone *beyond* the head of the list.
- Because a reverse iterator is an iterator, we will derive the `ReverseIterator` class from the `Iterator` class.

`ReverseIterator` Member Functions

Additional `ReverseIterator` Member Functions

- `ReverseIterator(const LinkedListwIter<T>* lst, LinkedListNode<T>* p);`

  Construct a `ReverseIterator`.

- `ReverseIterator& operator++();`

  Advance the `ReverseIterator` to the next node.

`LinkedListwIter` Member Functions

Additional `LinkedListwIter` Member Functions

- `ReverseIterator rbegin() const;`

  Create a `ReverseIterator` set to the beginning of the list.

- `ReverseIterator rend() const;`

  Create a `ReverseIterator` set to the end of the list.

`LinkedListwIter` Member Functions

- The other `LinkedListwIter` member functions that use iterators, such as the iterator version of `getElement()`, can accept reverse iterators as well because...
- That is because

  `A ReverseIterator` IS-A `Iterator.`

Implemention of Reverse Iterators

- To construct a reverse iterator for a linked list,
  - Introduce a stack data member.
  - Push `NULL` onto the stack.
  - Then push the addresses of the nodes onto the stack as the list is traversed from head to tail.
  - Stop with all but the final `NULL` pointer on the stack.
  - Now the reverse iterator is initialize.
- To increment the reverse list iterator
  - Pop an address off the stack.
  - Assign it to the node pointer.

Binary Tree Iterators

- In a list, we could traverse in only two ways:
  - Head to tail (forward)
  - Tail to head (reverse)
- In a binary tree, there is a variety of ways in which we can traverse the structure.
  - Pre-order
  - In-order
  - Post-order
  - Level-order, etc.

Binary Tree Iterators

- Accordingly, we create the following binary tree iterator classes.
  - `PreorderIterator` class
  - `InorderIterator` class
  - `PostorderIterator` class
  - `LevelorderIterator` class

Binary Tree Iterators

- Furthermore, these are all subclasses of a base class `Iterator`.
- By using inheritance, all we have to implement for each subclass is
  - The constructor.
  - The `++` operator.

Binary Tree Preorder Iterators

- We will build on what we learned about reverse iterators for lists.
- To reach the “next” node from a root node, a preorder iterator must move to the root of the left subtree.
- It should also push the pointer to the right subtree, if there is one.

Binary Tree Preorder Iterators

Preorder Iterator Constructor

```
PreorderIterator(const BinaryTree<T>* tr,
    BinaryTreeNode<T>* rt) : Iterator(tr, rt)
        {stack.push(NULL);}
```

Binary Tree Preorder Iterators

Preorder Iterator `operator++()`

```
PreorderIterator& operator++()
{
    if (node != NULL)
    {
    //  Store pointer to right subtree
        if (node->rightNode() != NULL)
            stack.push(node->rightNode());
    //  Go to root of left subtree
        if (node->leftNode() != NULL)
            node = node->leftNode();
    //  Or, use stack to get next node
        else
            node = stack.pop();
    }
    return *this;
}
```

Binary Tree Inorder Iterators

- To reach the “next” node from a root node, an inorder iterator must travel to the right subtree, and then as far left as possible, pushing node pointers along the way.

Binary Tree Inorder Iterators

Inorder Iterator Constructor

```
InorderIterator(const BinaryTree<T>* tr,
    BinaryTreeNode<T>* rt) : Iterator(tr, rt)
{
    stack.push(NULL);

//  Find the leftmost node

    if (node != NULL)
        while (node->leftNode() != NULL)
        {
            stack.push(node);
            node = node->leftNode();
        }
    return;
}
```

Binary Tree Inorder Iterators

Inorder Iterator `operator++()`

```
InorderIterator& operator++()
{
    if (node != NULL)
    {
        if (node->rightNode() != NULL)
        {
            node = node->rightNode();
            while (node->leftNode() != NULL)
            {
                stack.push(node);
                node = node->leftNode();
            }
        }
        else
            node = stack.pop();
    }
    return *this;
}
```

Binary Tree Postorder Iterators

- To the “next” node from a root node, a postorder iterator must do the following.
- If the current node is a left child, then the iterator must following the leftmost branch of the sibling right subtree all the way to a leaf node, pushing nodes along the way.

Binary Tree Postorder Iterators

Postorder Iterator Constructor

```
PostorderIterator(const BinaryTree<T>* tr,
    BinaryTreeNode<T>* rt) : Iterator(tr, rt)
{
    stack.push(NULL);
//  Find the leftmost leaf
    if (node != NULL)
        while (node->leftNode() != NULL
        || node->rightNode() != NULL)
        {
            stack.push(node);
            if (node->leftNode() != NULL)
                node = node->leftNode();
            else
                node = node->rightNode();
        }
    return;
}
```

Binary Tree Postorder Iterators

Postorder Iterator `operator++()`

```
PostorderIterator& operator++()
{
    if (node != NULL)
    {
        BinaryTreeNode<T>* parent = stack.top();
        if (parent != NULL && node == parent->leftNode()
        && parent->rightNode() != NULL)
        {
            node = parent->rightNode();
            while (node->leftNode() != NULL
            || node->rightNode() != NULL)
            {
                stack.push(node);
                if (node->leftNode() != NULL)
                    node = node->leftNode();
                else
                    node = node->rightNode();
            }
        }
        else
        {
            stack.pop();
            node = parent;
        }
    }
    return *this;
}
```

Assignment

Homework

- Create a `LevelorderIterator` class for binary trees.