Advanced Data Structures

Lecture 5

(Based on Paul Kube course materials)
Lecture 5

• C++ iterators
• The binary search tree successor function

Reading: Weiss Ch 4, sections 1-4
C++ STL iterators

- In the *iterator pattern* of OO design, a container has a way to supply to a client an iterator object which is to be used by the client to access the data in the container sequentially, without exposing the container’s underlying representation.

- Containers in the STL implement the iterator pattern.

- For example here’s a typical way client code can iterate over (and print out) all of the data in an STL container, in this case a `set`:

```cpp
set<string> c;
...
// get an iterator pointing to container’s first element
set<string>::iterator itr = c.begin();
// get an iterator pointing past container’s last element
set<string>::iterator end = c.end();
// loop while itr is not past the last element
while(itr != end) {
    cout << *itr << endl; // dereference the itr to get data
    ++itr;                // increment itr to point to next element
}
```
Implementing the iterator pattern in C++

- To implement the iterator pattern for a container in STL fashion, we need to consider:
  - How to define the `begin()` member function of the container
    - this function must return an iterator object “pointing to” the first element of the container
  - How to define the `end()` member function of the container
    - this function must return an iterator object “pointing just past” the last element of the container
  - How to define these operators for iterators:
    - `!=` (not equal test). This operator must return true (1) or false (0) according to whether the two iterators are pointing to the same element or not
    - `*` (dereference). This operator must return a reference to (or possibly just a copy of) the data item contained in the element the iterator is currently pointing to
    - `++`(pre-increment). This operator must cause the iterator to point to the next element of the container
    - (Note: some iterators also allow decrement `--`, and some allow arbitrary pointer arithmetic)
    - We’ll consider the iterator operators first
An iterator class template for a BST

• Suppose a BST’s nodes are instances of a class template Node as shown before
• At each step in an iteration, an iterator for the BST only needs to keep a pointer to the current Node in the BST
• So, define a BSTIterator class template with one member variable that is a pointer to the current node; then a constructor and overloaded operators for the class are easy to define:

```cpp
template <typename T>
class BSTIterator {
private:
    Node<T>* curr;
public:
    /** Constructor */
    BSTIterator(Node<T>* n) : curr(n) {}
};
```
An iterator class template for a BST, cont’d

```cpp
public:
    /** Inequality test operator */
    bool operator!=(BSTIterator<T> const & other) const {
        return this->curr != other.curr;
    }

    /** Dereference operator */
    T operator*() const {
        return curr->data;
    }

    /** Pre-increment operator */
    BSTIterator<T>& operator++() {
        curr = curr->successor(); // point to next node
        return *this;
    }
};
```

• How to define the successor() member function of Node?...
A container’s end() function

• How to define the end() member function of the container
  • this function must return an iterator object “pointing just past” the last element of the container
  • the end iterator must be an iterator object that can be distinguished from any iterator that is “pointing to” an actual element
  • idea: create an iterator initialized with nullptr

```cpp
template <typename T>
class BST {
  private:
    Node<T>* root;
  public:
    typedef BSTIterator<T> iterator;
    iterator end() const {
      return BSTIterator<T>(nullptr);
    }
};
```
A container’s begin() function

- How to define the `begin()` member function of the container?
  - this function must return an iterator object “pointing to” the first element of the container
  - with our partial definition of the `BSTIterator` template, this function can just return an iterator initialized with the `Node` holding the first (i.e. smallest) data item in the BST
    ```cpp
    iterator begin() const {
        return BSTIterator<T>( first(root) );
    }
    ```
- Now... how to define the `first()` function?
  - (It is just a helper function, not part of the interface to the BST class, so it should be private...)
  - (And if we define it to take a `Node` pointer as argument, it can be a static function; the function will not need to refer to “this” BST...)
Defining a first() function

private:
    /** Return a pointer to the node containing
        the
        * smallest data item in the BST subtree
        rooted at n,
        * or nullptr if n is nullptr
        */
    static Node<T>* first(Node<T>* n) { }

The successor relation in binary search trees

• Inorder traversal of a binary tree can be defined recursively:

```c++
template <typename T> void inorder(Node<T>* n) {
    if(nullptr == n) return; inorder(n->left); visit(n); inorder(n->right);
}
```

• An inorder traversal of a binary search tree starting at the root will visit its nodes in sorted order, according to the ordering relation on the data contained in the nodes

• An iterator for a binary search tree backed container should iterate over the container’s data in sorted order
  • That is, the iterator should essentially simulate an inorder traversal, one step at a time

• If the iterator is currently pointing to a node X in a BST, when it is incremented it should then point to X’s successor, that is, the node that would come immediately after X during an inorder traversal of the BST (if any)
Finding the successor of a BSTNode

- Suppose X is a node in a BST that has just been visited during an inorder traversal. What is the next node that will be visited; that is, what is the successor of X?

- Consider cases:
  - X has a right child
  - X has no right child, and is the left child of its parent
  - X has no right child, and is the right child of its parent
  - X has no right child, and has no parent
Next time

- Binary search tree average cost analysis
- The importance of being balanced

Reading: Weiss Ch 4, sections 1-4