

BiIterator.h

```

template <class T> class BiIterator {
private:
    List<T>::Node* node;

    // only the List<T> class can call the constructor that initializes the iterator
    // with a List<T>::Node*
    friend class List<T>;
    BiIterator (List<T>::Node* node);

public:

    BiIterator ();

    bool operator == (const BiIterator<T> &other) const;
    bool operator != (const BiIterator<T> &other) const;

    // ++i advances the iterator so that it refers to the next element in the container
    // and returns the new value of the iterator
    BiIterator<T>& operator ++ ();

    // i++ advances the iterator so that it refers to the next element in the container
    // container, and returns the old value of the iterator.
    BiIterator<T> operator ++ (int);

    // --i modifies the iterator so that it refers to the previous element in
    // the container, and returns a reference to the new value of the iterator.
    BiIterator<T>& operator -- ();

    // i-- modifies the iterator so that it refers to the previous element in
    // the container, and returns the old value of the iterator.
    BiIterator<T> operator -- (int);

    // *i returns a reference to the element that the iterator refers to. The
    // value of the element within the container may be updated by assignment
    // to the returned reference.

    T &operator * () const;
};

```

8/5/00

1 of 4

BiIterator Construction and Equality Operators

The public default constructor is used to allow a user of an Iterator for a list to instantiate an iterator, that can then be assigned to by `List<T>::begin()`.

```

template<class T> inline BiIterator<T>::BiIterator () : node(NULL) {}

List<int> list;
List<int>::iterator i;
for (i = list.begin(); i != list.end(); i++)
    ...

```

The constructor taking a single `Node` argument is private, and since `List<T>` is a friend class, only a method of the `List<T>` class can instantiate an iterator object and initialize it's node pointer to point at a node in the list.

```

template<class T> inline BiIterator<T>::BiIterator (List<T>::Node* n) : node(n) {}

```

For example, the `begin()` method on the `List` class returns an iterator to the first node in the list:

```

template<class T> iterator List<T>::begin() { BiIterator<T> i = head->front; return i; }

```

The equality operators are trivial, they just compare two pointers.

```

template<class T> inline bool BiIterator<T>::operator != (const BiIterator<T> &other) const
{
    return (node != other.node);
}

template<class T> inline bool BiIterator<T>::operator == (const BiIterator<T> &other) const
{
    return (node == other.node);
}

```

8/5/00

2 of 4

Traversing the List

The way the iterator traverses a linked list depends on how the `List` node structure is defined. You could simply follow the `next` (or `prev`) pointers, but that assumes you have implemented the `Node` as a doubly linked list. To insulate the iterator from the implementation of the `Node` structure, you can define a couple of methods on the `Node` structure to help out.

```

struct Node {
    T* elem;
    Node* next, prev;
    ..
    Node* forw() { return next; }
    Node* back() { return prev; }
};

template<class T> inline BiIterator<T>& BiIterator<T>::operator ++ () {
    node = node->forw();
    return *this;
}

template<class T> inline BiIterator<T> BiIterator<T>::operator ++ (int) {
    BiIterator<T> tmp = *this;
    node = node->forw();
    return tmp;
}

template<class T> inline BiIterator<T>& BiIterator<T>::operator -- () {
    node = node->back();
    return *this;
}

template<class T> inline BiIterator<T> BiIterator<T>::operator -- (int) {
    BiIterator<T> tmp = *this;
    node = node->back();
    return tmp;
}

```

8/5/00

3 of 4

Deferecing an Iterator

The `BiIterator<T>::operator*` method dereferences an iterator, which should result in a reference to an object of type `T`. How this method is implemented depends on how you implement the `List` node.

If `Node` has a `T*` `elem`, then use:

```

template<class T> inline T& BiIterator<T>::operator* () { return *(node->elem); }

```

Else, if `Node` has a `T` `elem`, then use:

```

template<class T> inline T& BiIterator<T>::operator*() { return node->elem; }

```

The better solution is to add another method to the `Node` class so that it doesn't matter to the iterator how the `Node` stores the `T` object.

```

struct Node {
    T* elem; // could be T elem
    Node* prev, next;
    ..
    Node* forw() { return next; }
    Node* back() { return prev; }

    T& elem () { return *elem; } // or just elem if defined as a T instead of a T*
};

```

You then implement the `BiIterator<T>::operator*` method to call the `Node::elem()` method, which does the "right thing" to return a `T&` to the caller.

```

template<class T> inline T& BiIterator<T>::operator*() { return node->elem(); }

```

8/5/00

4 of 4