Iterators

- An iterator permits you to examine the elements of a data structure one at a time.
- C++ iterators
  - Input iterator
  - Output iterator
  - Forward iterator
  - Bidirectional iterator
  - Reverse iterator
Bidirectional Iterator

Allows both forward and backward movement through the elements of a data structure.
Bidirectional Iterator Methods

- **iterator(T* thePosition)**
  Constructs an iterator positioned at specified element
- dereferencing operators * and ->
- Post and pre increment and decrement operators ++ and –
- Equality testing operators == and !=
Iterator Class

- Assume that a bidirectional iterator class `iterator` is defined within the class `arrayList`.

- Assume that methods `begin()` and `end()` are defined for `arrayList`.
  - `begin()` returns an iterator positioned at element 0 of list.
  - `end()` returns an iterator positioned one past last element of list.
Using An Iterator

arrayList<int>::iterator xHere = x.begin();
arrayList<int>::iterator xEnd = x.end();
for (; xHere != xEnd; xHere++)
    examine( *xHere);

vs

for (int i = 0; i < x.size(); i++)
    examine(x.get(i));
Merits Of An Iterator

- it is often possible to implement the `++` and `--` operators so that their complexity is less than that of `get`
- many data structures do not have a `get by index` method
- iterators provide a uniform way to sequence through the elements of a data structure
Linked Representation

• list elements are stored, in memory, in an arbitrary order

• explicit information (called a link) is used to go from one element to the next
Memory Layout

Layout of $L = (a,b,c,d,e)$ using an array representation.

A linked representation uses an arbitrary layout.
Linked Representation

pointer (or link) in e is NULL

use a variable firstNode to get to the first element a
Normal Way To Draw A Linked List

![Diagram of a linked list with nodes labeled a, b, c, d, e, and NULL. The first node is labeled as `firstNode`. The diagram shows the link or pointer field of each node in turquoise and the data field in orange.](image-url)
A chain is a linked list in which each node represents one element.

- There is a link or pointer from one element to the next.
- The last node has a NULL pointer.
Node Representation

template <class T>
struct chainNode
{
  // data members
  T element;
  chainNode<T> *next;

  // constructors come here
};
Constructors Of chainNode

chainNode() {}

chainNode(const T& element)
{this->element = element;}

chainNode(const T& element, chainNode<T>* next)
{this->element = element;
 this->next = next;}
get(0)

firstNode

checkIndex(0);
desiredNode = firstNode;  // gets you to first node
return desiredNode->element;
checkIndex(1);
desiredNode = firstNode->next; // gets you to second node
return desiredNode->element;
checkIndex(2);
desiredNode = firstNode->next->next; // gets you to third node
return desiredNode->element;
get(5)

firstNode

checkIndex(5); // throws exception
desiredNode = firstNode->next->next->next->next->next;
    // desiredNode = NULL
return desiredNode->element; // NULL.element
Erase An Element

erase(0)

deleteNode = firstNode;
firstNode = firstNode->next;
delete deleteNode;
erase(2)

first get to node just before node to be removed

beforeNode = firstNode->next;
erase(2)

firstNode

beforeNode

save pointer to node that will be deleted

deleteNode = beforeNode->next;
erase(2)

firstNode

beforeNode

now change pointer in beforeNode

beforeNode->next = beforeNode->next->next;
delete deleteNode;
insert(0, ’f’)

**Step 1:** get a node, set its data and link fields

```cpp
newNode = new chainNode<char>(theElement, firstNode);
```
insert(0, ’f’)

Step 2: update firstNode

firstNode = newNode;
One-Step insert(0,’f’)

```cpp
firstNode = new chainNode<char>('f', firstNode);
```
**insert(3,’f’)**

- first find node whose index is 2
- next create a node and set its data and link fields

```cpp
chainNode<char>* newNode = new chainNode<char>(‘f’, beforeNode->next);
```

- finally link `beforeNode` to `newNode`

```cpp
beforeNode->next = newNode;
```
```cpp
beforeNode = firstNode->next->next;
beforeNode->next = new chainNode<char>('f', beforeNode->next);
```
The Class chain

firstNode

listSize = number of elements

Use chainNode

next (datatype chainNode<T>*)

element (datatype T)
class chain : public linearList<T> 
{
    public:
        // constructors and destructor defined here

        // ADT methods
        bool empty() const {return listSize == 0;}
        int size() const {return listSize;}
        // other ADT methods defined here

    protected:
        void checkIndex(int theIndex) const;
        chainNode<T>* firstNode;
        int listSize;
};
Constructor

template<class T>
chain<T>::chain(int initialCapacity = 10)
{ // Constructor.
    if (initialCapacity < 1)
    {
        ostringstream s;
        s << "Initial capacity = "
        << initialCapacity << " Must be > 0";
        throw illegalParameterValue(s.str());
    }
    firstNode = NULL;
    listSize = 0;
}
template<class T>
chain<T>::~chain()
{// Chain destructor. Delete all nodes
 // in chain.
while (firstNode != NULL)
{// delete firstNode
  chainNode<T>* nextNode = firstNode->next;
  delete firstNode;
  firstNode = nextNode;
}
```cpp
template<class T>
T& chain<T>::get(int theIndex) const
{
// Return element whose index is theIndex.
    checkIndex(theIndex);
    // move to desired node
    chainNode<T>* currentNode = firstNode;
    for (int i = 0; i < theIndex; i++)
        currentNode = currentNode->next;
    return currentNode->element;
}
```
The Method indexOf

template<class T>
int chain<T>::indexOf(const T& theElement) const
{
    // search the chain for theElement
    chainNode<T>* currentNode = firstNode;
    int index = 0;  // index of currentNode
    while (currentNode != NULL && currentNode->element != theElement)
    {
        // move to next node
        currentNode = currentNode->next;
        index++;
    }
}
The Method indexOf

// make sure we found matching element
if (currentNode == NULL)
    return -1;
else
    return index;
Erase An Element

deleteNode = firstNode;
firstNode = firstNode->next;
delete deleteNode;
template<class T>
void chain<T>::erase(int theIndex)
{
    checkIndex(theIndex);

    chainNode<T>* deleteNode;
    if (theIndex == 0)
    { // remove first node from chain
        deleteNode = firstNode;
        firstNode = firstNode->next;
    }
}
erase(2)

firstNode

beforeNode

Find & change pointer in `beforeNode`

`beforeNode->next = beforeNode->next->next;`

delete `deleteNode;`
Remove An Element

else
{
    // use p to get to beforeNode
    chainNode<T>* p = firstNode;
    for (int i = 0; i < theIndex - 1; i++)
        p = p->next;

    deleteNode = p->next;
    p->next = p->next->next;
}

listSize--; 
delete deleteNode;
}
One-Step insert(0,’f’)

```cpp
firstNode = new chainNode<char>('f', firstNode);
```
template<class T>
void chain<T>::insert(int theIndex,
    const T& theElement)
{
    if (theIndex < 0 || theIndex > listSize)
    {// Throw illegalIndex exception
    }

    if (theIndex == 0)
    {// insert at front
        firstNode = new chainNode<T>
            (theElement, firstNode);
Two-Step insert(3,'f')

beforeNode = firstNode->next->next;
beforeNode->next = new chainNode<char>(’f’, beforeNode->next);
Inserting An Element

else
{
   // find predecessor of new element
   chainNode<T>* p = firstNode;
   for (int i = 0; i < theIndex - 1; i++)
   {
      p = p->next;
   }

   // insert after p
   p->next = new chainNode<T>(theElement, p->next);
}

listSize++;
Performance

50,000 operations of each type
## Performance

50,000 operations of each type

<table>
<thead>
<tr>
<th>Operation</th>
<th>FastArrayLinearList</th>
<th>Chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>1.0ms</td>
<td>13.2sec</td>
</tr>
<tr>
<td>best-case inserts</td>
<td>2.1ms</td>
<td>45.1ms</td>
</tr>
<tr>
<td>average inserts</td>
<td>1.5sec</td>
<td>49.3sec</td>
</tr>
<tr>
<td>worst-case inserts</td>
<td>2.5sec</td>
<td>12.9sec</td>
</tr>
<tr>
<td>best-case removes</td>
<td>2.0ms</td>
<td>2.1ms</td>
</tr>
<tr>
<td>average removes</td>
<td>1.5sec</td>
<td>68.8sec</td>
</tr>
<tr>
<td>worst-case removes</td>
<td>2.5sec</td>
<td>12.9sec</td>
</tr>
</tbody>
</table>
Chain With Header Node

headerNode

a → b → c → d → NULL
Empty Chain With Header Node

headerNode

NULL
Circular List

FirstNode

a → b → c → d → e
Doubly Linked List

firstNode

NULL

a

b

c

d

e

lastNode

NULL
Doubly Linked Circular List

firstNode

a <-> b <-> c <-> d <-> e
Doubly Linked Circular List With Header Node

headerNode

[Diagram of a doubly linked circular list with header node, showing nodes labeled a, b, c, d, and e.]
Empty Doubly Linked Circular List With Header Node
The STL Class list

- Linked implementation of a linear list.
- Doubly linked circular list with header node.
- Has many more methods than chain.
- Similar names and signatures.