05-0: **Abstract Data Types**

- Recall that an Abstract Data Type is a definition of a type based on the operations that can be performed on it.
- An ADT is an *interface*
- Data in an ADT cannot be manipulated directly – only through operations defined in the interface

05-1: **List ADT**

- A List is an ordered collection of elements
- Each element in the list has a position
  - Element 0, Element 1, Element 2, ...
- We can access elements in the list through an *iterator*

05-2: **List ADT Operations**

- Create an empty list
- Add (append) an element to the end of the list
- Add (insert) an element at a specified index
- Get the size (length) of the list
- Remove an element at a specific index
- Remove the first occurrence of an element
- Get an element at a specific index
- Get an iterator to traverse the list

05-3: **Iterators**

- Think of an iterator as a “smart bookmark” that is associated with a specific data structure
- Often used to examine every element in a data structure

05-4: **Iterators**

Some operation on iterators:

- Retrieve the current element
- Move the iterator forward, to the next element in the data structure
  - C++ has two different operations: “Get current” and “Move forward”
  - Java has a single operation: “Get current and move forward”
- Move the iterator backwards, to the previous element in the data structure
  - Not all iterators can go backwards
  - Java also combines going backwards as “Get previous element and move iterator backwards”

05-5: **Iterators**

Some operation on iterators:
• Delete element at current location (not always allowed)
• Insert an element at the current location (not always allowed)
• Operations specific to the particular data structure

05-6: **List Iterator (first pass)**
• Get the next element (moving the iterator one forward)
• Check if there is a next element
• Remove the object at the current position (current position == last element that was returned from a “next”)
• Insert an element at the current position (right before the “next” element)

05-7: **Java Interfaces**
• A Java interface is a set of methods.
• Any class that implements an interface must implement all of these methods

05-8: **Java List Interface**

```java
public interface List
{
    public void clear();
    public void add(Object o);
    public void add(int index, Object o);
    public void remove(int index);
    public void remove(Object o);
    public int size();
    public Object get(int index);
    public ListIterator listIterator();
    public ListIterator listIterator(int index);
}
```

05-9: **Java List Iterator Interface**

```java
public interface ListIterator
{
    public void add(Object o);
    public boolean hasNext();
    public Object next();
    public void remove();
    public void set(Object o);
}
```

05-10: **Using Iterators**
• Print out a list $L$: 
List L;
...
ListIterator it = L.listIterator();

for (it.hasNext())
{
    System.out.println(it.next());
}

05-11: **Array Implementation**

- Data is stored in an array
- Iterator stores index of next location
- To add an element to the current position:
  - Shift all elements with index $i = \text{current}$ one to right
- To remove and element from the middle of the array:
  - Shift all elements with index $i = \text{current}$ to the right
- List has a maximum size (unless we use growable arrays)

05-12: **Array Implementation** $\Theta()$ Running Time for each operation:

<table>
<thead>
<tr>
<th>List Operations</th>
<th>Iterator Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(append)</td>
<td>next</td>
</tr>
<tr>
<td>add(insert)</td>
<td>hasNext</td>
</tr>
<tr>
<td>remove</td>
<td>add</td>
</tr>
<tr>
<td>listIterator()</td>
<td>remove</td>
</tr>
<tr>
<td>listIterator(n)</td>
<td>set</td>
</tr>
<tr>
<td>size</td>
<td>get</td>
</tr>
</tbody>
</table>

05-13: **Array Implementation** $\Theta()$ Running Time for each operation:

<table>
<thead>
<tr>
<th>List Operations</th>
<th>Iterator Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(append)</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>add(insert)</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>remove</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>listIterator()</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>listIterator(n)</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>size</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>get</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

05-14: **Linked-List Implementation**

- Data is stored in a linked list
- Maintain a pointer to first element in list
- Iterator maintains a pointer to the next element
- To find the $i$th element:
  - Start at the front of the list
  - Skip past $i$ elements
How do we insert an element before the next element? How do we remove the “current” element?

05-15: **Linked-List Implementation**

- Data is stored in a linked list
- Maintain a pointer to first element in list
- Iterator maintains a pointer to the element before the next element (“current” element) and a pointer to the element before the current element.
- To find the \( i \)th element:
  - Start at the front of the list
  - Skip past \( i \) elements

What should “current” pointer be when the “next” element is the first element in the list?

05-16: **Linked-List Implementation**

- Data is stored in a linked list – with a dummy first element
- Maintain a pointer to first (dummy) element in list
- Iterator maintains a pointer to the element before the next element (“current” element) and the “previous” element (what should “previous” be when the first element of the list is the next element in the list?)
- To find the \( i \)th element:
  - Start at the front of the list
  - Skip past \((i+1)\) elements

05-17: **Linked-List Implementation** \( \Theta() \) Running Time for each operation:

<table>
<thead>
<tr>
<th>List Operations</th>
<th>Iterator Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(append)</td>
<td>next</td>
</tr>
<tr>
<td>add(insert)</td>
<td>hasNext</td>
</tr>
<tr>
<td>remove</td>
<td>add</td>
</tr>
<tr>
<td>listIterator()</td>
<td>remove</td>
</tr>
<tr>
<td>listIterator(n)</td>
<td>set</td>
</tr>
<tr>
<td>size</td>
<td></td>
</tr>
<tr>
<td>get</td>
<td></td>
</tr>
</tbody>
</table>

05-18: **Linked-List Implementation** \( \Theta() \) Running Time for each operation:

<table>
<thead>
<tr>
<th>List Operations</th>
<th>Iterator Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(append)</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>add(insert)</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>remove</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>listIterator()</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>listIterator(n)</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>size</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>get</td>
<td>( \Theta(n) )</td>
</tr>
</tbody>
</table>

05-19: **Adding Previous**

- Add a new operation to the iterator: previous
  - Move the iterator back one element, return the previous element
  - next() followed by previous(), both return same element
• How would we implement previous for an array implementation

05-20: Adding Previous
• Add a new operation to the iterator: previous
  • Move the iterator back one element, return the previous element
    next() followed by previous(), both return same element
• How would we implement previous for an array implementation
  • Subtract one from the index of the current location

05-21: Adding Previous
• Add a new operation to the iterator: previous
  • Move the iterator back one element
• How would we implement previous for a linked list implementation

05-22: Adding Previous
• Add a new operation to the iterator: previous
  • Move the iterator back one element
• How would we implement previous for a linked list implementation
  • Start a temp pointer at the front of the list, advance it until temp.next = current pointer
  • How can we improve the running time of previous for the linked list version?

05-23: Doubly-Linked Lists
• Each element in the list has two pointers – next and previous
  • Can locate the previous element of any element in the list in time $O(1)$, instead of time $O(n)$
  • More space is required (two pointers for each element, instead of one)
  • Do we still need a “dummy” element?

05-24: Multiple Iterators
• We can have more than one iterator going in the same list
  • Handy for comparing every element in the list to every other element in the list
• Can have a problem when one iterator modifies the list while another iterator is active
  • Examples

05-25: Multiple Iterators
• We can have more than one iterator going in the same list
• Can have a problem when one iterator modifies the list while another iterator is active
  • Solutions:
    • Throw exception (how java libraries do it)
    • Inform the other iterators
      • List maintains a pointer to each active iterators
      • When a change is made, each active iterator needs to be updated, too