Chapter 24 Implementing Lists, Stacks, Queues, and Priority Queues

Objectives

- To design common features of lists in an interface and provide skeleton implementation in an abstract class (§24.2).
- To design and implement a dynamic list using an array (§24.3).
- To design and implement a dynamic list using a linked structure (§24.4).
- To design and implement a stack class using an array list and a queue class using a linked list (§24.5).

Lists

A list is a popular data structure to store data in sequential order. For example, a list of students, a list of available rooms, a list of cities, and a list of books, etc. can be stored using lists. The common operations on a list are usually the following:

- Retrieve an element from this list.
- Insert a new element to this list.
- Delete an element from this list.
- Find how many elements are in this list.
- Find if an element is in this list.
- Find if this list is empty.

Two Ways to Implement Lists

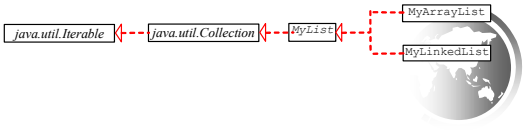
There are two ways to implement a list.

Using arrays to store the elements. If the capacity of the array is exceeded, create a new larger array and copy all the elements from the current array to the new array.

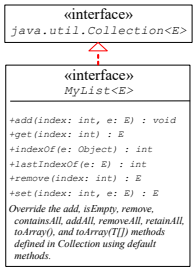
Using linked list for head/tail access in a linked list of nodes. Each node is dynamically created to hold an element. All the nodes are linked together to form a list.

Design of ArrayList and LinkedList

Let's name these two classes: MyArrayList and MyLinkedList. These two classes have common operations, but different data fields. The common operations can be generalized in an interface or an abstract class. A popular design strategy is to define common operations in an interface and provide an abstract class for partially implementing the interface. So, the concrete class can simply extend the abstract class without implementing the full interface.



MyList Interface



Inserts a new element at the specified index in this list.
Returns the element from this list at the specified index.
Returns the index of the first matching element in this list.
Returns the index of the last matching element in this list.
Removes the element at the specified index and returns the removed element.
Sets the element at the specified index and returns the element being replaced.

MyList Run

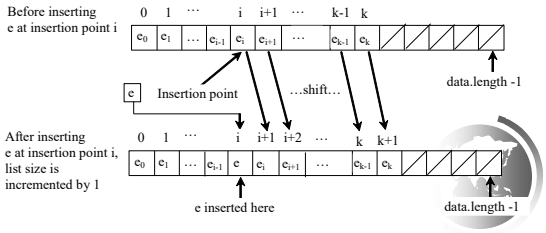
Array for Lists

Array is a fixed-size data structure. Once an array is created, its size cannot be changed. Nevertheless, you can still use arrays to implement dynamic data structures. The trick is to create a new larger array to replace the current array if the current array cannot hold new elements in the list.

Initially, an array, say data of Object[] type, is created with a default size. When inserting a new element into the array, first ensure there is enough room in the array. If not, create a new array with the size twice as the current one. Copy the elements from the current array to the new array. The new array now becomes the current array.

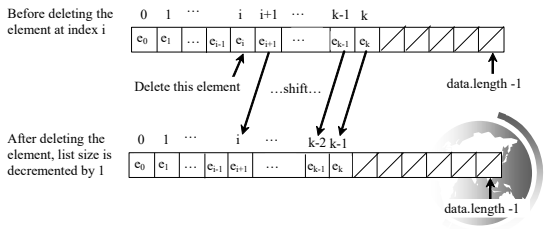
Insertion

Before inserting a new element at a specified index, shift all the elements after the index to the right and increase the list size by 1.



Deletion

To remove an element at a specified index, shift all the elements after the index to the left by one position and decrease the list size by 1.



Linked Lists

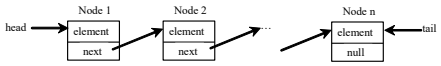
Since MyArrayList is implemented using an array:

- add(int index, Object o) and remove(int index) are inefficient
  - require shifting potentially a large number of elements.

You can use a linked structure to implement a list to improve efficiency for adding and removing an element anywhere in a list.

Nodes in Linked Lists

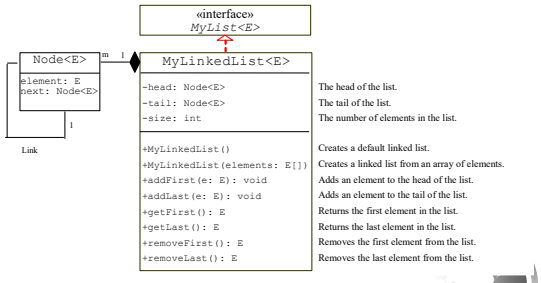
A linked list consists of nodes. Each node contains an element, and each node is linked to its next neighbor. Thus a node can be defined as a class, as follows:




```
class Node<E> {
  E element;
  Node<E> next;

  public Node(E o) {
    element = o;
  }
}
```

MyLinkedList



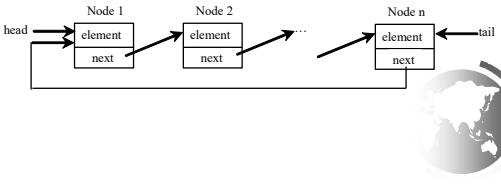
MyLinkedList TestMyLinkedList Run

Time Complexity for ArrayList and LinkedList

Methods	MyArrayList/ArrayList	MyLinkedList/LinkedList
`add(e: E)`	$O(1)$	$O(1)$
`add(index: int, e: E)`	$O(n)$	$O(n)$
`clear()`	$O(1)$	$O(1)$
`contains(e: E)`	$O(n)$	$O(n)$
`get(index: int)`	$O(1)$	$O(n)$
`indexOf(e: E)`	$O(n)$	$O(n)$
`isEmpty()`	$O(1)$	$O(1)$
`lastIndexOf(e: E)`	$O(n)$	$O(n)$
`remove(e: E)`	$O(n)$	$O(n)$
`size()`	$O(1)$	$O(1)$
`remove(index: int)`	$O(n)$	$O(n)$
`set(index: int, e: E)`	$O(n)$	$O(n)$
`addFirst(e: E)`	$O(n)$	$O(1)$
`removeFirst()`	$O(n)$	$O(1)$

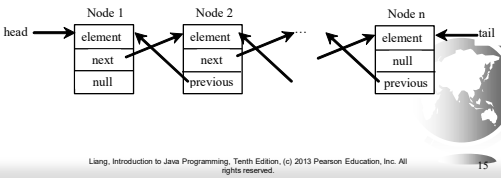
Circular Linked Lists

A *circular, singly linked list* is like a singly linked list, except that the pointer of the last node points back to the first node.



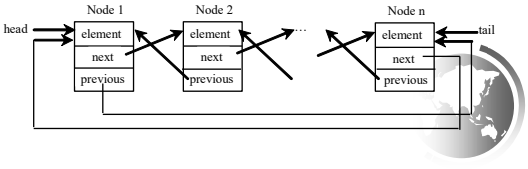
Doubly Linked Lists

A *doubly linked list* contains the nodes with two pointers. One points to the next node and the other points to the previous node. These two pointers are conveniently called *a forward pointer* and *a backward pointer*. So, a doubly linked list can be traversed forward and backward.



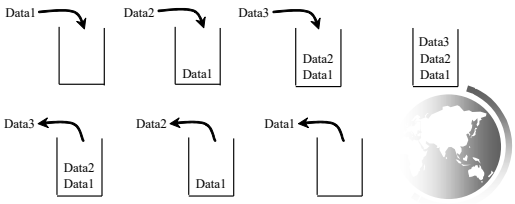
Circular Doubly Linked Lists

A *circular*, *doubly linked list* is doubly linked list, except that the forward pointer of the last node points to the first node and the backward pointer of the first pointer points to the last node.



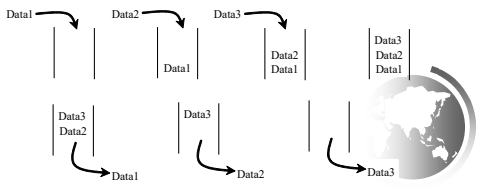
Stacks

A stack can be viewed as a special type of list, where the elements are accessed, inserted, and deleted only from the end, called the top, of the stack.



Queues

A queue represents a waiting list. A queue can be viewed as a special type of list, where the elements are inserted into the end (tail) of the queue, and are accessed and deleted from the beginning (head) of the queue.



Implementing Stacks and Queues

- Stack: Using an ArrayList. Since the insertion and deletion operations on a stack are made only at the end of the stack, using an array list to implement a stack is more efficient than a linked list.
- Queue: Using a linked list. Since deletions are made at the beginning of the list, it is more efficient to implement a queue using a linked list than an array list.

Design of the Stack and Queue Classes

There are two ways to design the stack and queue classes:

– Using inheritance: You can define the stack class by extending the array list class, and the queue class by extending the linked list class.

(a) Using inheritance

– Using composition: You can define an array list as a data field in the stack class, and a linked list as a data field in the queue class.

(b) Using composition

Composition is Better

Both designs are fine, but using composition is better because it enables you to define a complete new stack class and queue class without inheriting the unnecessary and inappropriate methods from the array list and linked list.

MyStack and MyQueue

GenericStack<E>	
-list: java.util.ArrayList<E>	An array list to store elements.
+GenericStack()	Creates an empty stack.
+getSize(): int	Returns the number of elements in this stack.
+peek(): E	Returns the top element in this stack.
+pop(): E	Returns and removes the top element in this stack.
+push(o: E): void	Adds a new element to the top of this stack.
+isEmpty(): boolean	Returns true if the stack is empty.

GenericStack

GenericQueue<E>	
-list: LinkedList<E>	
+enqueue(e: E): void	Adds an element to this queue.
+dequeue(): E	Removes an element from this queue.
+getSize(): int	Returns the number of elements from this queue.

GenericQueue