LISTS AND ITERATORS

1

Lists

- List is a data structure which provides the facility to maintain the ordered collection.
- It contains index-based methods to insert, update, delete and search the elements.
- It can have the duplicate elements and null elements in the list.
- The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface.
- Through the ListIterator, user can iterate the list in forward and backward directions.
- The implementation classes of List interface are **ArrayList**, **LinkedList**, **Stack** and Vector.
- The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

2

The java.util.List ADT

- The java.util.List interface includes the following methods:

size():	Returns the number of elements in the list.
isEmpty():	Returns a boolean indicating whether the list is empty.
get(i):	Returns the element of the list having index i; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.
set(i, e):	Replaces the element at index i with e, and returns the old element that was replaced; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.
add(i, e):	Inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list; an error condition occurs if i is not in range $[0, \text{size}()]$.
remove(i):	Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.

3

Example

- A sequence of List operations:

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	“error”	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	“error”	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	“error”	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

4

List Interface

```
/** A simplified version of the java.util.List interface. */
public interface List<E> {
  /** Returns the number of elements in this list. */
  int size();

  /** Returns whether the list is empty. */
  boolean isEmpty();

  /** Returns (but does not remove) the element at index i. */
  E get(int i) throws IndexOutOfBoundsException;

  /** Replaces the element at index i with e, and returns the replaced element. */
  E set(int i, E e) throws IndexOutOfBoundsException;

  /** Inserts element e to be at index i, shifting all subsequent elements later. */
  void add(int i, E e) throws IndexOutOfBoundsException;

  /** Removes/returns the element at index i, shifting subsequent elements earlier. */
  E remove(int i) throws IndexOutOfBoundsException;
}
```

5

Java List Example

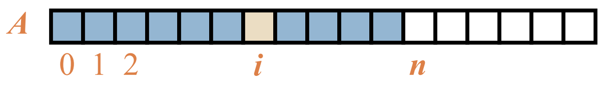
```
import java.util.*;
public class ListExample1{
public static void main(String args[]){
        //Creating a List
        List<String> list=new ArrayList<String>();
        //Adding elements in the List
        list.add("Mango");
        list.add("Apple");
        list.add("Banana");
        list.add("Grapes");
        //Iterating the List element using for-each loop
        for(String fruit:list)
                System.out.println(fruit);

}
}
```

6

Array Lists

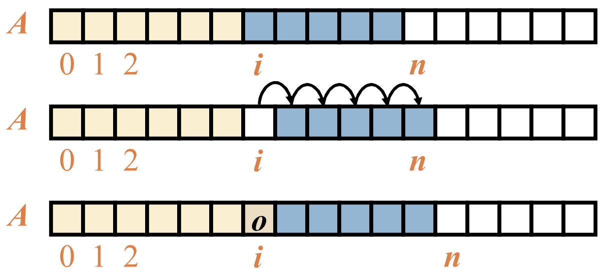
- An obvious choice for implementing the list ADT is to use an array, **A,** where **A[i]** stores (a reference to) the element with index **i.**
- With a representation based on an array **A,** the get(**i**) and set(**i, e**) methods are easy to implement by accessing **A[i]** (assuming **i** is a legitimate index).



7

Insertion

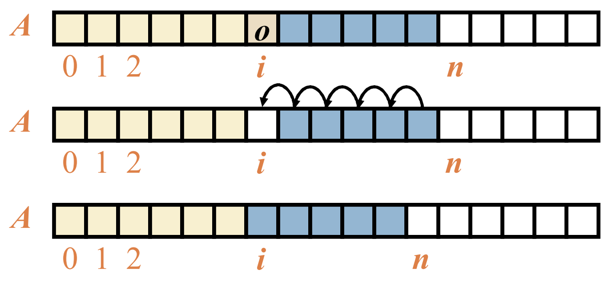
- In an operation $add(i, o)$, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



8

Element Removal

- In an operation *remove*(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \dots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Performance

Method	Running Time
size()	$O(1)$
isEmpty()	$O(1)$
get(i)	$O(1)$
set(i, e)	$O(1)$
add(i, e)	$O(n)$
remove(i)	$O(n)$

- In an array-based implementation of a dynamic list:
  - The space used by the data structure is **O**(***n***)
  - Indexing the element at i takes **O**(1) time
  - ***add*** and ***remove*** run in **O**(***n***) time
- In an ***add*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one …

Java Implementation

```
// public methods
/** Returns the number of elements in the array list. */
public int size( ) { return size; }
/** Returns whether the array list is empty. */
public boolean isEmpty( ) { return size == 0; }
/** Returns (but does not remove) the element at index i. */
public E get(int i) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  return data[i];
}
/** Replaces the element at index i with e, and returns the replaced element. */
public E set(int i, E e) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  E temp = data[i];
  data[i] = e;
  return temp;
}
```

11

Java Implementation, 2

```
/** Inserts element e to be at index i, shifting all subsequent elements later. */
public void add(int i, E e) throws IndexOutOfBoundsException,
                                        IllegalStateException {
  checkIndex(i, size + 1);
  if (size == data.length)                   // not enough capacity
    throw new IllegalStateException("Array is full");
  for (int k=size−1; k >= i; k−−)            // start by shifting rightmost
    data[k+1] = data[k];
  data[i] = e;                               // ready to place the new element
  size++;
}
/** Removes/returns the element at index i, shifting subsequent elements earlier. */
public E remove(int i) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  E temp = data[i];
  for (int k=i; k < size−1; k++)             // shift elements to fill hole
    data[k] = data[k+1];
  data[size−1] = null;                       // help garbage collection
  size−−;
  return temp;
}
// utility method
/** Checks whether the given index is in the range [0, n−1]. */
protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
  if (i < 0 || i >= n)
    throw new IndexOutOfBoundsException("Illegal index: " + i);
}
}
```

12

Dynamic Arrays

- The ArrayList implementations so far as well as those for a stack, queue, and deque from previous chapters has a serious limitation; it requires that a fixed maximum capacity be declared, throwing an exception if attempting to add an element once full.
- This is a major weakness, and there is risk that either too large of an array will be requested, causing an inefficient waste of memory, or that too small of an array will be requested, causing a fatal error when exhausting that capacity.
- Java's ArrayList class provides a more robust abstraction, allowing a user to add elements to the list, with no apparent limit on the overall capacity.
- To provide this abstraction, Java relies on an algorithmic sleight of hand that is known as a dynamic array.
- In reality, elements of an ArrayList are stored in a traditional array, and the precise size of that traditional array must be internally declared in order for the system to properly allocate a consecutive piece of memory for its storage.

13

Dynamic Arrays

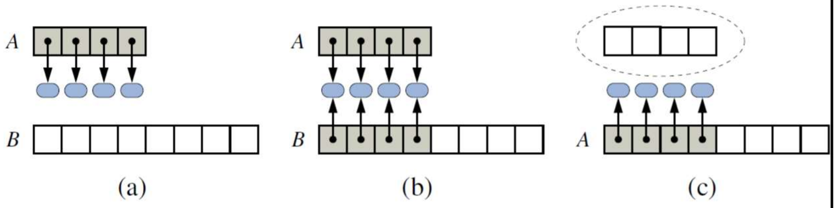
- Because the system may allocate neighboring memory locations to store other data, the capacity of an array cannot be increased by expanding into subsequent cells. The first key to providing the semantics of an unbounded array is that an array list instance maintains an internal array that often has greater capacity than the current length of the list. For example, while a user may have created a list with five elements, the system may have reserved an underlying array capable of storing eight object references (rather than only five). This extra capacity makes it easy to add a new element to the end of the list by using the next available cell of the array.
- If a user continues to add elements to a list, all reserved capacity in the underlying array will eventually be exhausted. In that case, the class requests a new, larger array from the system, and copies all references from the smaller array into the beginning of the new array.

14

Implementing a Dynamic Array

```
Algorithm push(o)
    if t = S.length − 1 then
        A ← new array of
                size …
        for i ← 0 to n−1 do
            A[i] ← S[i]
        S ← A
    n ← n + 1
    S[n−1] ← o
```

- Let push(o) be the operation that adds element o at the end of the list
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size



```
/** Resizes internal array to have given capacity >= size. */
protected void resize(int capacity) {
  E[ ] temp = (E[ ]) new Object[capacity];  // safe cast; compiler may give warning
  for (int k=0; k < size; k++)
    temp[k] = data[k];
  data = temp;                               // start using the new array
}
```

Code Fragment 7.4: An implementation of the ArrayList.resize method.

15

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations
- We assume that we start with an empty list represented by a growable array of size 1
- We call amortized time of a push operation the average time taken by a push operation over the series of operations, i.e., $T(n)/n$

16

Incremental Strategy Analysis

- Over n push operations, we replace the array $k = n/c$ times, where c is a constant
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$
$$n + c(1 + 2 + 3 + \ldots + k) =$$
$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- Thus, the amortized time of a push operation is $O(n)$

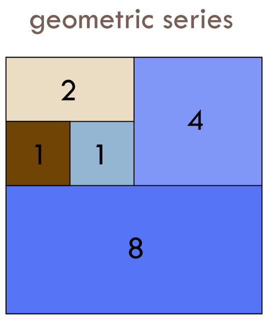
17

Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of a push operation is $O(1)$



18

Positional Lists

- To provide for a general abstraction of a sequence of elements with the ability to identify the location of an element, we define a **positional list** ADT.
- A position acts as a marker or token within the broader positional list.
- A position p is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.
- A position instance is a simple object, supporting only the following method:
  - P.getElement( ): Return the element stored at position p.

19

Positional List ADT

- Accessor methods:

first( ):	Returns the position of the first element of L (or null if empty).
last( ):	Returns the position of the last element of L (or null if empty).
before(p):	Returns the position of L immediately before position p (or null if p is the first position).
after(p):	Returns the position of L immediately after position p (or null if p is the last position).
isEmpty( ):	Returns true if list L does not contain any elements.
size( ):	Returns the number of elements in list L.

20

Positional List ADT, 2

- Update methods:

addFirst(e):	Inserts a new element e at the front of the list, returning the position of the new element.
addLast(e):	Inserts a new element e at the back of the list, returning the position of the new element.
addBefore(p, e):	Inserts a new element e in the list, just before position p, returning the position of the new element.
addAfter(p, e):	Inserts a new element e in the list, just after position p, returning the position of the new element.
set(p, e):	Replaces the element at position p with element e, returning the element formerly at position p.
remove(p):	Removes and returns the element at position p in the list, invalidating the position.

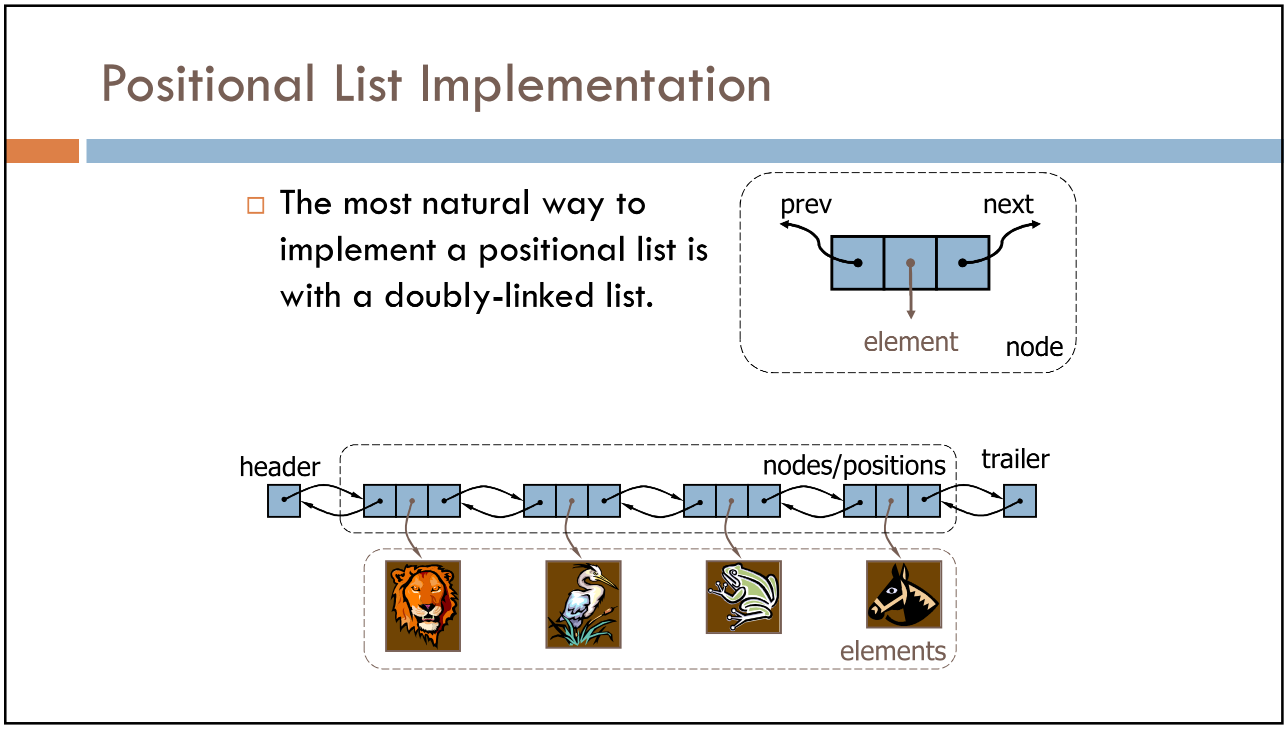
21

Example

- A sequence of Positional List operations:

Method	Return Value	List Contents
addLast(8)	p	(8_p)
first()	p	(8_p)
addAfter(p, 5)	q	$(8_p, 5_q)$
before(q)	p	$(8_p, 5_q)$
addBefore(q, 3)	r	$(8_p, 3_r, 5_q)$
r.getElement()	3	$(8_p, 3_r, 5_q)$
after(p)	r	$(8_p, 3_r, 5_q)$
before(p)	null	$(8_p, 3_r, 5_q)$
addFirst(9)	s	$(9_s, 8_p, 3_r, 5_q)$
remove(last())	5	$(9_s, 8_p, 3_r)$
set(p, 7)	8	$(9_s, 7_p, 3_r)$
remove(q)	"error"	$(9_s, 7_p, 3_r)$

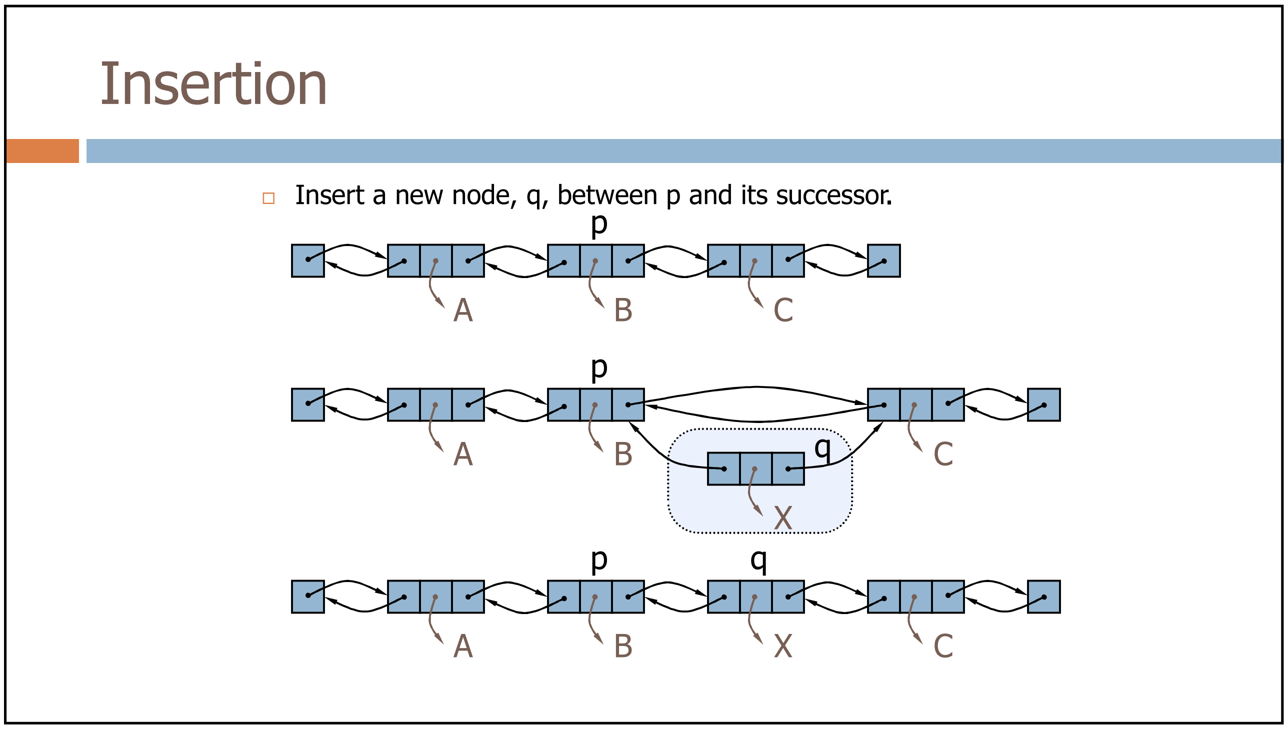
22



Positional List Implementation

- The most natural way to implement a positional list is with a doubly-linked list.

23



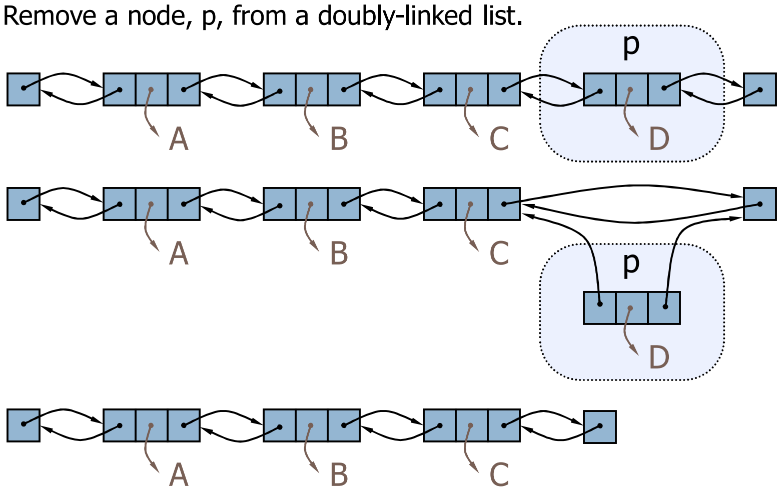
Insertion

- Insert a new node, q, between p and its successor.

24

Deletion

- Remove a node, p, from a doubly-linked list.



25

Iterators

- An iterator is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a time.

hasNext(): Returns true if there is at least one additional element in the sequence, and false otherwise.

next(): Returns the next element in the sequence.

26

The Iterable Interface

- Java defines a parameterized interface, named Iterable, that includes the following single method:
  - iterator( ): Returns an iterator of the elements in the collection.
- An instance of a typical collection class in Java, such as an ArrayList, is iterable (but not itself an iterator); it produces an iterator for its collection as the return value of the iterator( ) method.
- Each call to iterator( ) returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection.

27

The for-each Loop

- Java's Iterable class also plays a fundamental role in support of the "for-each" loop syntax:

```
for (ElementType variable : collection) {
  loopBody                                  // may refer to "variable"
}
```

is equivalent to:

```
Iterator<ElementType> iter = collection.iterator( );
while (iter.hasNext( )) {
  ElementType variable = iter.next( );
  loopBody                                  // may refer to "variable"
}
```

28