



香港中文大學 (深圳)
The Chinese University of Hong Kong

(Materials of this lecture will NOT be tested in the final exam)

CSC3100 Data Structures

Lecture 24: Java data structures

Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen



Outline

- ▶ Java collection framework
 - Iterator/iterable interface
 - List interface
 - Queue/Deque interface
 - Set/SortedSet interface
 - Map interface



Java collection framework

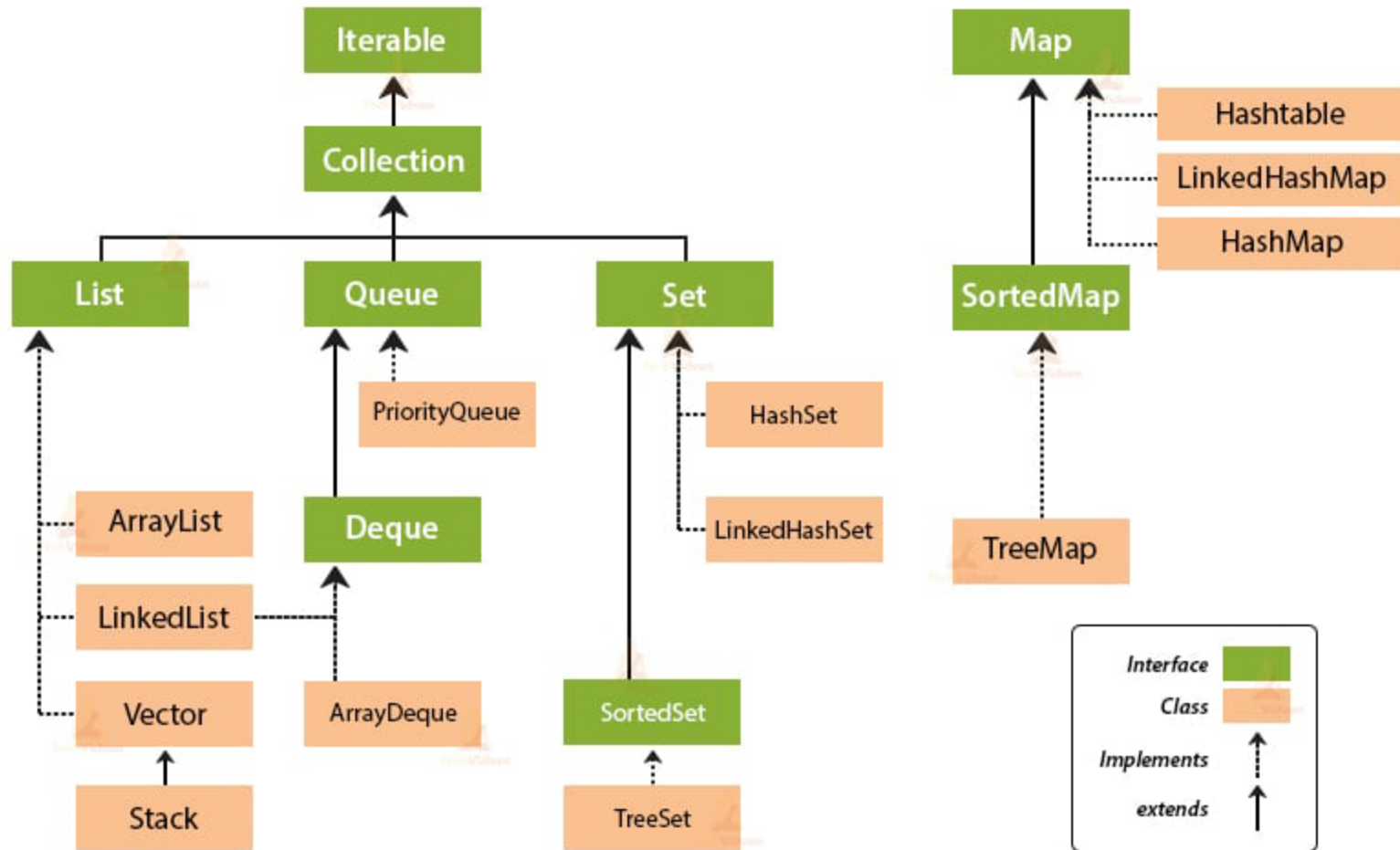
- ▶ What is a **framework** in Java?
 - It provides readymade architecture
 - It represents a set of classes and interfaces
 - It is optional
- ▶ What is a **collection framework**?
 - Represents a unified architecture for storing and manipulating a group of objects
- ▶ Java **collection framework** has:
 - Interfaces and its implementations, i.e., classes
 - Some algorithms

An Interface is defined as an abstract type used to specify what a class must do and not how, i.e., a blueprint of a class



Hierarchy of collection framework

Collection Framework Hierarchy in Java



No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.
10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.
15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.
19	public int hashCode()	It returns the hash code number of the collection.



Iterator interface

- ▶ Iterator interface provides the facility of iterating the elements in a **forward** direction only

It contains only one abstract method. i.e.,

```
Iterator<T> iterator()
```

It returns the iterator over the elements of type T.

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.



Iterable and Collection interfaces

- ▶ The **Iterable interface** is the root interface for all the collection classes
- ▶ The **Collection interface** extends the Iterable interface
 - All the subclasses of Collection interface also implement the Iterable interface
 - The Collection interface declares the methods that every collection will have, e.g.,
 - `boolean add (Object obj)`
 - `boolean addAll (Collection c)`
 - `void clear()`, etc.



List interface

- ▶ List interface is the child interface of Collection interface
 - It inhibits a list type data structure which can store the ordered collection of objects
 - It can have duplicate values
 - List interface is implemented by the classes `ArrayList`, `LinkedList`, `Vector`, and `Stack`

To instantiate the List interface, we must use :

```
List <data-type> list1 = new ArrayList();  
List <data-type> list2 = new LinkedList();  
List <data-type> list3 = new Vector();  
List <data-type> list4 = new Stack();
```

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.


```

1  import java.util.*;
2  class TestJavaCollection1
3  {
4      public static void main(String args[])
5      {
6          ArrayList<String> list = new ArrayList<String>(); //Creating arraylist
7          list.add("Ravi");//Adding object in arraylist
8          list.add("Vijay");
9          list.add("Ravi");
10         list.add("Ajay");
11         //Traversing list through Iterator
12         Iterator itr = list.iterator();
13         while(itr.hasNext())
14         {
15             System.out.println(itr.next());
16         }
17     }
18 }

```

```

Ravi
Vijay
Ravi
Ajay

```

The **ArrayList** class implements the List interface

It uses a dynamic array to store the duplicate element of different data types

The ArrayList class maintains the insertion order and is non-synchronized

The elements stored in the ArrayList class can be randomly accessed

```

1  import java.util.*;
2  public class TestJavaCollection2
3  {
4      public static void main(String args[])
5      {
6          LinkedList<String> al = new LinkedList<String>();
7          al.add("Ravi");
8          al.add("Vijay");
9          al.add("Ravi");
10         al.add("Ajay");
11         Iterator<String> itr = al.iterator();
12         while(itr.hasNext())
13         {
14             System.out.println(itr.next());
15         }
16     }
17 }

```

```

Ravi
Vijay
Ravi
Ajay

```

LinkedList implements the Collection interface

It uses a doubly linked list internally to store the elements

It can store the duplicate elements

It maintains the insertion order and is non-synchronized

The manipulation is fast because no shifting is required

```

1  import java.util.*;
2  public class TestJavaCollection3
3  {
4      public static void main(String args[])
5      {
6          Vector<String> v = new Vector<String>();
7          v.add("Ayush");
8          v.add("Amit");
9          v.add("Ashish");
10         v.add("Garima");
11         Iterator<String> itr = v.iterator();
12         while(itr.hasNext())
13         {
14             System.out.println(itr.next());
15         }
16     }
17 }

```

```

Ayush
Amit
Ashish
Garima

```

Vector uses a dynamic array to store the data elements, which is similar to `ArrayList`. However, it is synchronized and contains many methods that are not the part of Collection framework.

```

1  import java.util.*;
2  public class TestJavaCollection4
3  {
4      public static void main(String args[])
5      {
6          Stack<String> stack = new Stack<String>();
7          stack.push("Ayush");
8          stack.push("Garvit");
9          stack.push("Amit");
10         stack.push("Ashish");
11         stack.push("Garima");
12         stack.pop();
13         Iterator<String> itr = stack.iterator();
14         while(itr.hasNext())
15         {
16             System.out.println(itr.next());
17         }
18     }
19 }

```

```

Ayush
Garvit
Amit
Ashish

```

Stack is the subclass of Vector

It implements the last-in-first-out data structure, i.e., Stack

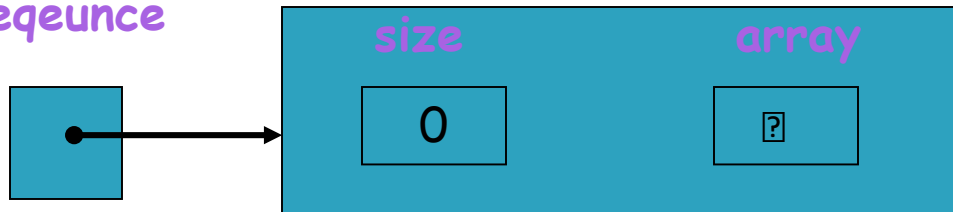
The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which define its properties₁₁



ArrayList class

- ▶ Implements the List using an array
 - By using an **Object** array, it can store any reference type
 - It cannot directly store primitive types, but can indirectly store such values by using instances of their wrapper types
- ▶ Consider the declaration:
`ArrayList addressSequence = new ArrayList();`

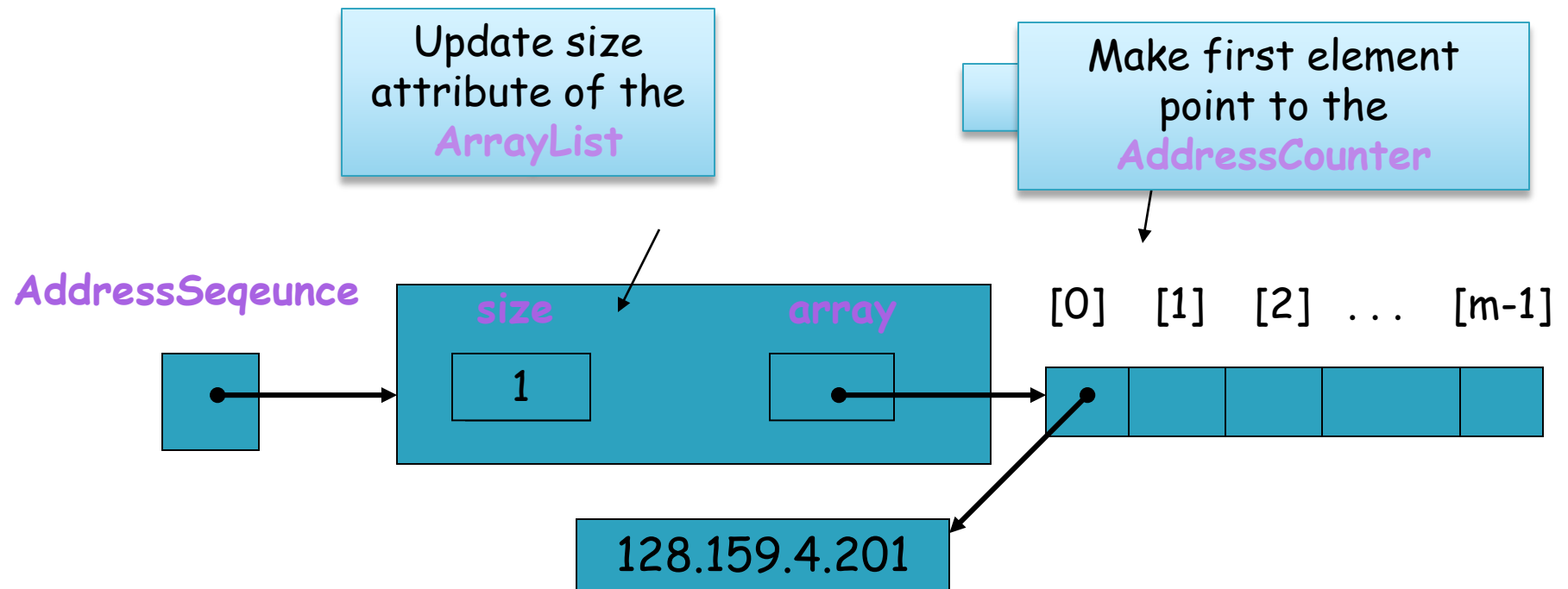
addressSequence





Add to addressSequence

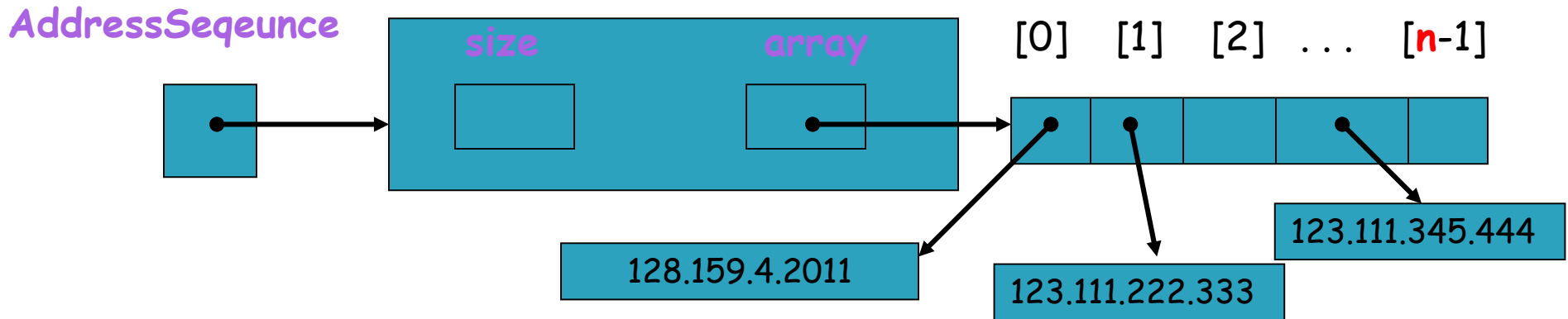
- ▶ The command `addressSequence.add(anAddressCounter);`
 - appends anAddressCounter object to the sequence
- ▶ The system will then ...





Enlarge the AddressSequence array

- ▶ When the allocated array is full, adding another element forces replacing array with larger one
 - A new array of $n > m$ is allocated
 - Values from old array are copied into new array
 - Old array is replaced by new one





ArrayList drawback

- ▶ Problems arise from using an array
 - Values can be added only at back of **ArrayList**
 - To insert a value, “shifting” may be required
 - Similarly, deleting a value requires shifting
- ▶ We need a slightly different structure to allow simple insertions and deletions
 - The **LinkedList** class will accomplish this



The LinkedList class

► Given

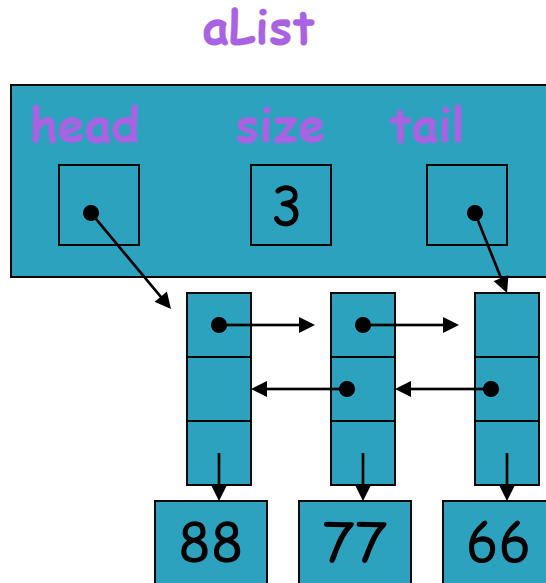
```
LinkedList alist = new LinkedList();
```

```
...
```

```
alist.add(new Integer(88));
```

```
alist.add(new Integer(77));
```

```
alist.add(new Integer(66));
```



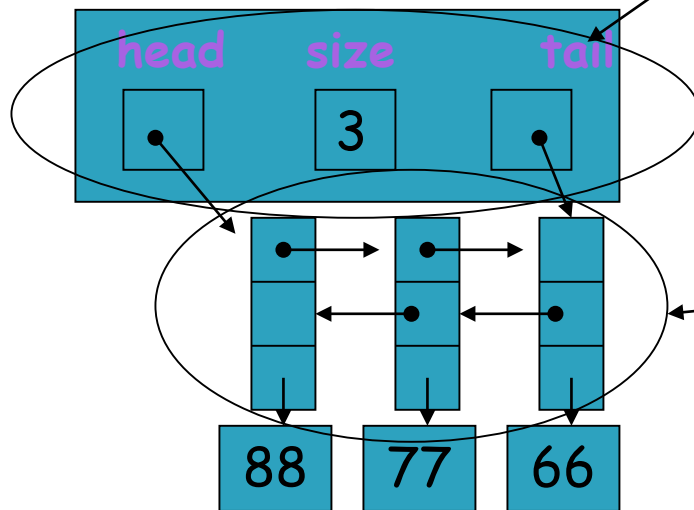


Linkedlist containers

Attributes:

- link to first item in the list
- size of the list
- link to last item in the list

aList



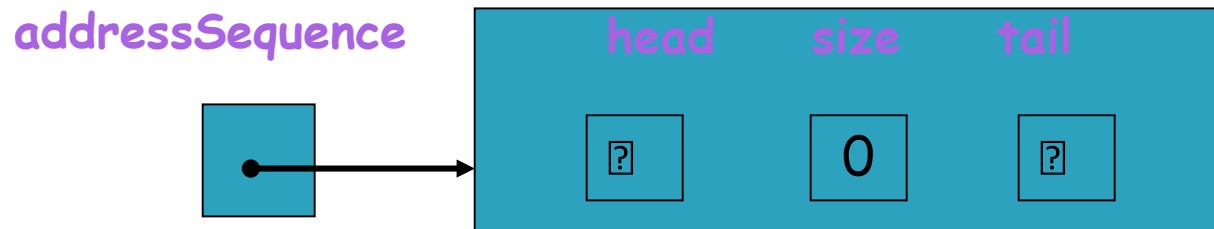
Nodes:

- Contain 3 handles
 - link to next node
 - link to previous node
 - link to stored object
- Links to next and previous make it a doubly linked list



LinkedList class

- ▶ Given the command
`LinkedList addressSequence = new LinkedList();`
- ▶ Uses `LinkedList` constructor to build an empty list

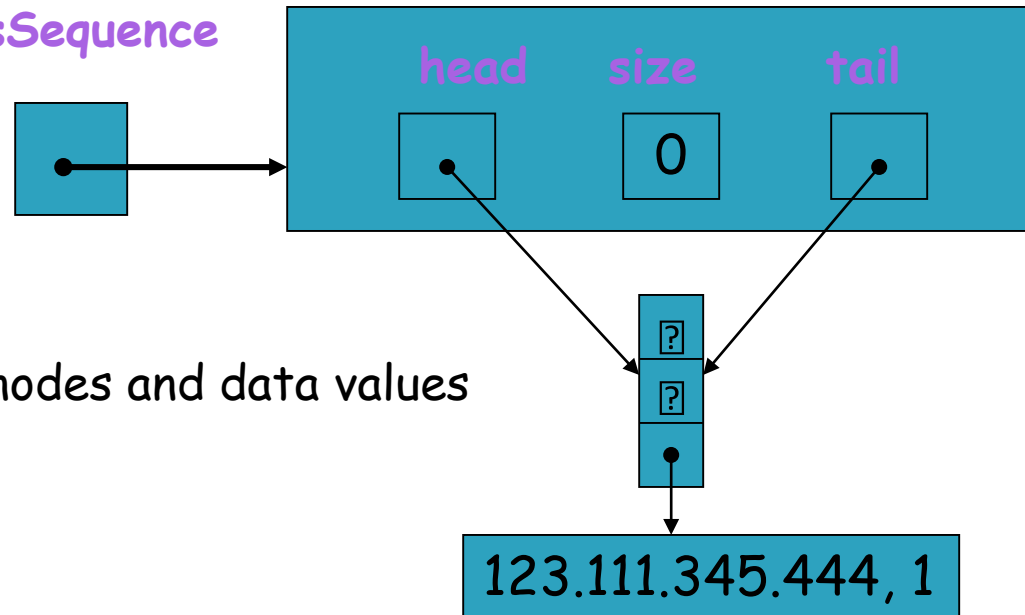




Add to a LinkedList

- ▶ Results of command for first add
`addressSequence.add(anAddressCounter);`

addressSequence



- Successive adds
 - create more nodes and data values
 - adjust links



Accessing values in a `LinkedList`

- ▶ Must use the `get` method

```
((AddressCounter)addressSequence.get(index)).incrementCount();
```

- ▶ A `LinkedList` has no array with an index to access an element
- ▶ `get` method must ...
 - begin at `head` node
 - iterate through `index` nodes to find match
 - return reference of object in that node
- ▶ Command then does cast and `incrementCount()`



Accessing values in a `LinkedList`

- ▶ To print successive values for the output

```
for (int i = 0; i < addressSequence.size(); i++)  
    System.out.println(addressSequence.get(i));
```

`size` method determines
limit of loop counter

`get(i)` starts at first
node, iterates `i` times to
reach desired node

- Note that each `get(i)` must pass over the same first `i-1` nodes previously accessed; this, of course, is inefficient



Insert nodes anywhere in a `LinkedList`

- ▶ Recall problem with `ArrayList`
 - It can add only at end of the list
 - `LinkedList` has capability to insert nodes anywhere
- ▶ We can use `addressSequence.add(n, new anAddressCounter);`
which will ...
 - build a new node
 - update `head` and `tail` links if required
 - update node handle links to place new node to be n^{th} item in the list
 - allocate memory for the data item



Queue interface

- ▶ Queue interface maintains the property of first-in-first-out
 - It can be defined as **an ordered list** that is used to hold the elements to be processed
 - Classes **PriorityQueue**, **Deque**, and **ArrayDeque** implement the Queue interface

Queue interface can be instantiated as:

```
Queue<String> q1 = new PriorityQueue();  
Queue<String> q2 = new ArrayDeque();
```



```

1 import java.util.*;
2 public class TestJavaCollection5
3 {
4     public static void main(String args[])
5     {
6         PriorityQueue<String> queue = new PriorityQueue<String>();
7         queue.add("Amit Sharma");
8         queue.add("Vijay Raj");
9         queue.add("JaiShankar");
10        queue.add("Raj");
11        System.out.println("head:" + queue.element());
12        System.out.println("head:" + queue.peek());
13        System.out.println("iterating the queue elements:");
14        Iterator itr = queue.iterator();
15        while(itr.hasNext())
16        {
17            System.out.println(itr.next());
18        }
19        queue.remove();
20        queue.poll();
21        System.out.println("after removing two elements:");
22        Iterator<String> itr2 = queue.iterator();
23        while(itr2.hasNext())
24        {
25            System.out.println(itr2.next());
26        }
27    }
28 }

```

```

head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
after removing two elements:
Raj
Vijay Raj

```

PriorityQueue implements the Queue interface
 It holds the elements or objects which are to be processed by their priorities
 PriorityQueue doesn't allow null values to be stored in the queue

```

1 import java.util.*;
2
3 public class HelloWorld {
4     public static void main(String[] args) {
5         PriorityQueue<String> q = new PriorityQueue<String>();
6         q.add("Amit Sharma");
7         q.add("Vijay Raj");
8         q.add("JaiShankar");
9         q.add("Raj");
10
11         Iterator itr = q.iterator();
12         while(itr.hasNext()) System.out.println("iter: " + itr.next());
13         System.out.println();
14
15         while(q.size() > 0){
16             System.out.println("pool:" + q.poll());
17         }
18     }
19 }

```

```

iter: Amit Sharma
iter: Raj
iter: JaiShankar
iter: Vijay Raj

```

```

pool:Amit Sharma
pool:JaiShankar
pool:Raj
pool:Vijay Raj

```

PriorityQueue is implemented by heap data structure
When we remove an element from it, the element with the highest priority will be removed

When we use an iterator to enumerate its elements, the enumeration does not consider the priority values



Deque interface

- ▶ Deque interface extends the Queue interface
- ▶ A double-ended queue which enables us to perform the operations at both the ends

Deque can be instantiated as:

```
Deque d = new ArrayDeque();
```

```

1  import java.util.*;
2  public class TestJavaCollection6
3  {
4      public static void main(String[] args)
5      {
6          //Creating Deque and adding elements
7          Deque<String> deque = new ArrayDeque<String>();
8          deque.add("Gautam");
9          deque.add("Karan");
10         deque.add("Ajay");
11         //Traversing elements
12         for (String str : deque)
13         {
14             System.out.println(str);
15         }
16     }
17 }

```

Gautam

Karan

Ajay

ArrayDeque class implements the Deque interface

Unlike queue, we can add or delete the elements from both ends

ArrayDeque is faster than ArrayList and Stack, and has no capacity restrictions



Set interface

- ▶ Set interface extends the Collection interface
 - It represents the unordered set of elements which doesn't allow duplicate items
 - We can store at most one null value in Set
 - Set is implemented by [HashSet](#), [LinkedHashSet](#), and [TreeSet](#)

Set can be instantiated as:

```
Set<data-type> s1 = new HashSet<data-type>();  
Set<data-type> s2 = new LinkedHashSet<data-type>();  
Set<data-type> s3 = new TreeSet<data-type>();
```

```

1  import java.util.*;
2  public class TestJavaCollection7
3  {
4      public static void main(String args[])
5      {
6          //Creating HashSet and adding elements
7          HashSet<String> set = new HashSet<String>();
8          set.add("Ravi");
9          set.add("Vijay");
10         set.add("Ravi");
11         set.add("Ajay");
12         //Traversing elements
13         Iterator<String> itr = set.iterator();
14         while(itr.hasNext())
15         {
16             System.out.println(itr.next());
17         }
18     }
19 }

```

```

Vijay
Ravi
Ajay

```

HashSet class implements Set interface

It represents the collection that uses a hash table for storage

Hashing is used to store the elements in the HashSet

It contains unique items

```

1  import java.util.*;
2  public class TestJavaCollection8
3  {
4      public static void main(String args[])
5      {
6          HashSet<String> set = new HashSet<String>();
7          set.add("Ravi");
8          set.add("Vijay");
9          set.add("Ravi");
10         set.add("Ajay");
11         Iterator<String> itr = set.iterator();
12         while(itr.hasNext())
13         {
14             System.out.println(itr.next());
15         }
16     }
17 }

```

```

Ravi
Vijay
Ajay

```

HashSet class represents the LinkedList implementation of Set interface
 It extends the HashSet class and implements Set interface
 Like HashSet, it also contains unique elements
 It maintains the insertion order and permits null elements



Sortedset interface

- ▶ SortedSet is the alternate of Set interface that provides a total ordering on its elements
 - The elements of the SortedSet are arranged in the increasing (ascending) order
 - It provides additional methods that inhibit the natural ordering of the elements

The SortedSet can be instantiated as:

```
SortedSet<data-type> set = new TreeSet();
```



```

1  import java.util.*;
2  public class TestJavaCollection9
3  {
4      public static void main(String args[])
5      {
6          //Creating and adding elements
7          TreeSet<String> set = new TreeSet<String>();
8          set.add("Ravi");
9          set.add("Vijay");
10         set.add("Ravi");
11         set.add("Ajay");
12         //traversing elements
13         Iterator<String> itr = set.iterator();
14         while(itr.hasNext())
15         {
16             System.out.println(itr.next());
17         }
18     }
19 }

```

```

Ajay
Ravi
Vijay

```

TreeSet class implements the Set interface that uses a tree for storage
 Like HashSet, TreeSet also contains unique elements
 The access and retrieval time of TreeSet is quite fast
 The elements in TreeSet are stored in ascending order



Map interface

- ▶ The Map interface is not a subtype of the Collection interface, so it behaves a bit differently from the previous collection types
 - A map contains unique keys
- ▶ HashMap class
 - It is an implementation of Map interface based on hash table
 - It stores elements in key & value pair

```
class Demo
{
    public static void main(String args[])
    {
        // Creating HashMap
        HashMap<Integer,String> hashMap = new HashMap<Integer,String>();
        // Adding elements
        hashMap.put(1, "One");
        hashMap.put(2, "Two");
        hashMap.put(3, "Three");
        hashMap.put(4, "Four");
        // Displaying HashMap
        System.out.println(hashMap);
    }
}
```

Output:
{1=One, 2=Two, 3=Three, 4=Four}



Questions for self-study

- ▶ What are the two ways to iterate the elements of a collection?
- ▶ What is the difference between ArrayList and LinkedList classes in collection framework?
- ▶ What is the difference between ArrayList and Vector classes in collection framework?
- ▶ What is the difference between HashSet and ArrayList classes in collection framework?
- ▶ How can we sort the elements of an object?
- ▶ Give an application example that we should use HashSet, instead of other collection classes



Recommended reading

▶ Reading materials

- <https://fileadmin.cs.lth.se/cs/Education/EDA040/common/java21.pdf>
- <https://www.guru99.com/java-tutorial.html>
- Book: Think in Java

