UNIVERSITY OF WESTMINSTER

5COSC001W – Object Oriented Programming Week 7

Dr. Barbara Villarini

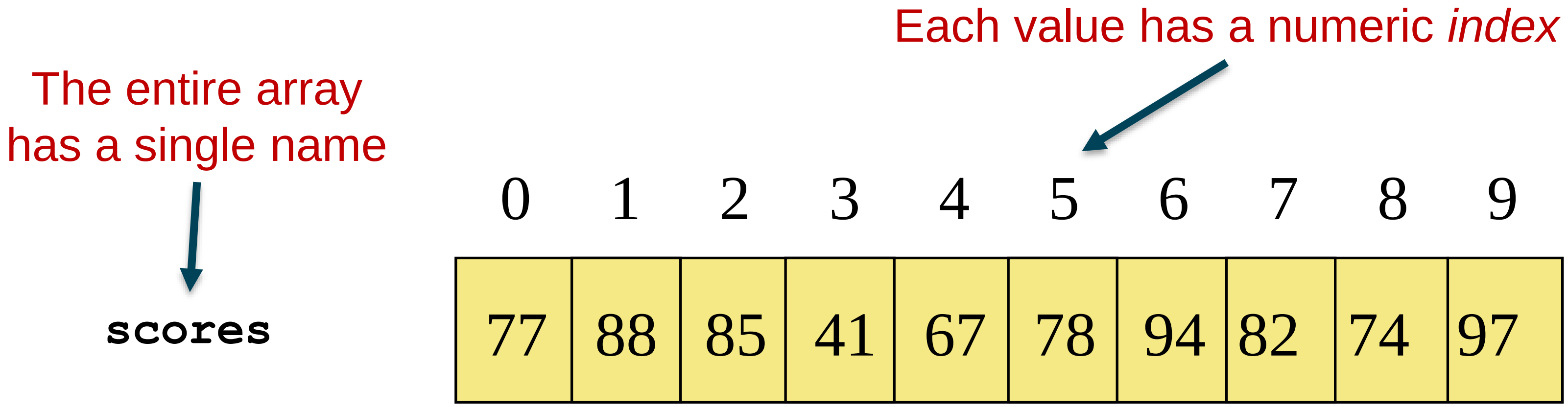
b.villarini@westminster.ac.uk

Summary

- Collections and Data Structure
- Arrays
- List, Queue, Map
- Searching and Sorting
- Linked List

Recap on Arrays

- Arrays are the fundamental mechanism for collecting multiple values.
- An array is an ordered list o values



Each value has a numeric *index*

The entire array has a single name

`scores`

0	1	2	3	4	5	6	7	8	9
77	88	85	41	67	78	94	82	74	97

- An array of size N is indexed from zero to N-1
- This array holds 10 values that are indexed from 0 to 9

Arrays

	0	1	2	3	4	5	6	7	8	9
`scores`	77	88	85	41	67	78	94	82	74	97

- A particular value in an array is referenced using the array name followed by the index in brackets

`scores[2]` → 85 (the 3rd value in the array)

Refers to the value

- That expression represents a place to store a single integer and can be used wherever an integer variable can be used

Arrays

- The values held in an array are called *array elements*
- An array stores multiple values of the same type – the *element type*
- The element type can be a primitive type or an object
- We can create an array of integers, an array of characters, an array of `String` objects, an array of `Coin` objects, etc.
- In Java, the array itself is an object that must be instantiated

Declaring arrays

- To declare an array you will use this general form

type`[]` *array-name*`= new` *type*`[`*size*`];`

```
int[] scores = new int[10];
```

- The type of the variable `scores` is `int[]` (an array of integers)
- The reference variable `scores` is set to a new array object that can hold 10 integers
- When you declare an array, you can specify the initial values and in this case you don't use the new operator:

```
int[] scores = {77, 88, 85, 50, 67, 78, 94, 82, 74, 97}
```

Bound Checking

	0	1	2	3	4	5	6	7	8	9
`scores`	77	88	85	41	67	78	94	82	74	97

- If the array `scores` can hold 10 values, it can be indexed using only the numbers 0 to 9

- If the value of `count` is 10, then the following reference will cause an exception to be thrown:

```
System.out.println (scores[count]);
```

- It's common to introduce *off-by-one errors* when using arrays

Problem

```
for (int index=0; index <= 10; index++)
   scorse[index] = index*5 + epsilon;
```

- The Java interpreter throws an `ArrayIndexOutOfBoundsException` if an array index is out of bounds

Length

- Each array object has a public constant called `length` that stores the size of the array

- It is referenced using the array name:

```
scores.length
```

- Note that `length` holds the number of elements, not the largest index

Arrays as Parameters

- An entire array can be passed as a parameter to a method
- Like any other object, the reference to the array is passed, making the formal and actual parameters aliases of each other
- Therefore, changing an array element within the method changes the original
- An individual array element can be passed to a method as well, in which case the type of the formal parameter is the same as the element type

Example – array as Parameters

```
public static void doubleValues(int[] x) {
       for (int i = 0; i < x.length; i++) {
           x[i] *= 2;
       }
   }
```

Arrays of Objects

- The elements of an array can be object references
- The following declaration reserves space to store 3 references to `String` objects

```
String[] words = new String[3];
```

- It does NOT create the `String` objects themselves
- Initially an array of objects holds `null` references
- Each object stored in an array must be instantiated separately

Example – Arrays of Objects

```
Person[] p = new Person[20];

System.out.println(p[0]); // output null

p[0] = new Person(”Mark”, “Smith");
p[1] = new Person(”Paul”, “Barne");
 .
 .

System.out.println(p[0].getName()); // output Mark
```

Example

```
public class Grade {
   private String name;
   private int lowerBound;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this Grade object with the specified
   //  grade name and numeric lower bound.
   //-----------------------------------------------------------------
   public Grade (String grade, int cutoff)  {
      name = grade;
      lowerBound = cutoff;
   }

   //-----------------------------------------------------------------
   //  Returns a string representation of this grade.
   //-----------------------------------------------------------------
   public String toString()  {
      return name + "\t" + lowerBound;
   }
}
```

```
public static void main(String[] args){
Grade[] grades =
      {
         new Grade("A", 95), new Grade("A-", 90),
         new Grade("B+", 87), new Grade("B", 85), new Grade("B-", 80),
         new Grade("C+", 77), new Grade("C", 75), new Grade("C-", 70),
         new Grade("D+", 67), new Grade("D", 65), new Grade("D-", 60),
         new Grade("F", 0)
      };

      Grade currentGrade;
      for (int i = 0; i < grades.length; i++){
               currentGrade= grades[i];
               System.out.println (currentGrade);
      }
}
```

COLLECTIONS

Collections

- **Collection** is an object that stores data
  - the objects stored are called **elements**
  - some collections maintain an ordering; some allow duplicates
  - typical operations: *add*, *remove*, *clear*, *contains* (search), *size*

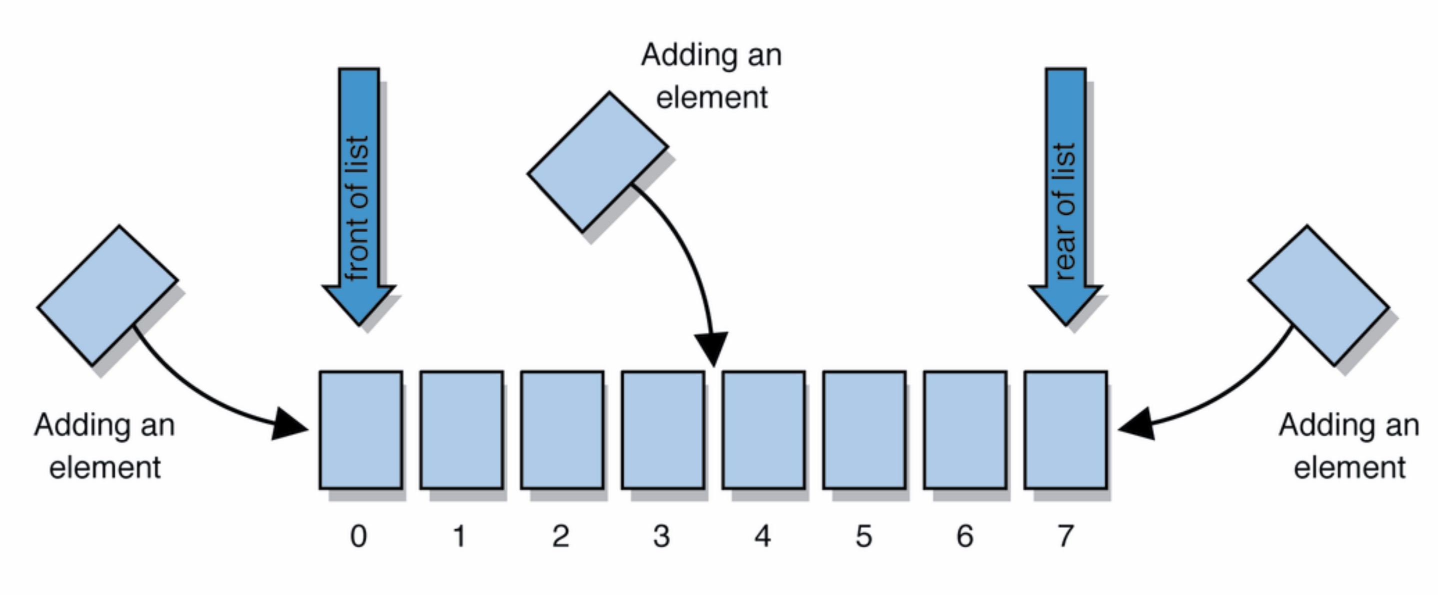
  - examples found in the Java class libraries:
    - **`ArrayList`**, **`LinkedList`**, **`HashMap`**, **`TreeSet`**, **`PriorityQueue`**

  - all collections are in the **`java.util`** package

```
import java.util.*;
```

List

- A collection storing an ordered sequence of elements
  - each element is accessible by a 0-based **index**
  - a list has a **size** (number of elements that have been added)
  - elements can be added to the front, back, or elsewhere
  - in Java, a list can be represented as an **`ArrayList`** object



Array Lists in Java

- You don't always know how many inputs you will have.
- An array list offers two significant advantages:

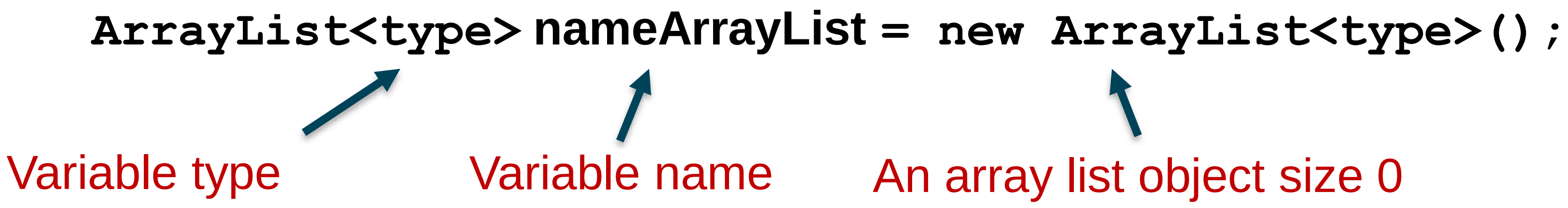
    1. Array lists can grow and shrink as needed. It changes size dynamically as new elements are added – Dynamic Resizing

    2. The ArrayList class supplies methods for common tasks, such as inserting and removing elements.

- In order to use array lists you need to:
    - Use the statement import **`java.util.ArrayList`**

Declaring and using Array Lists

- Any list of String:



```
ArrayList<type> nameArrayList = new ArrayList<type>();
```

Variable type

Variable name

An array list object size 0

- To construct an array list

```
new ArrayList<typeName>()
```

- To access an element

```
nameArrayList .get(index)
nameArrayList .set(index, value)
```

ArrayList methods

add(**value**)	appends value at end of list
add(**index, value**)	inserts given value just before the given index, shifting subsequent values to the right
clear()	removes all elements of the list
indexOf(**value**)	returns first index where given value is found in list (-1 if not found)
get(**index**)	returns the value at given index
remove(**index**)	removes/returns value at given index, shifting subsequent values to the left
set(**index, value**)	replaces value at given index with given value
size()	returns the number of elements in list
toString()	returns a string representation of the list such as "[3, 42, -7, 15]"

ArrayList methods

addAll(**list**)
addAll(**index, list**)
	adds all elements from the given list to this list
	(at the end of the list, or inserts them at the given index)

contains(**value**)	returns true if given value is found somewhere in this list
containsAll(**list**)	returns true if this list contains every element from given list
equals(**list**)	returns true if given other list contains the same elements
lastIndexOf(**value**)	returns last index value is found in list (-1 if not found)
remove(**value**)	finds and removes the given value from this list
removeAll(**list**)	removes any elements found in the given list from this list
retainAll(**list**)	removes any elements *not* found in given list from this list
subList(**from, to**)	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
toArray()	returns the elements in this list as an array

Example

```
ArrayList<String> band = new ArrayList();

band.add ("Paul");      band.add ("Peter");
band.add ("John");      band.add ("George");

System.out.println (band);

int location = band.indexOf ("Peter");
band.remove (location);

System.out.println (band);
System.out.println ("At index 1: " + band.get(1));

band.add (2, "Ringo");

System.out.println (band);
System.out.println ("Size : " + band.size());
```

ArrayList vs Array

- construction

```
String[] list= new String[5];
ArrayList<String> list = new ArrayList<String>();
```

- storing a value

```
list[0] = "Jessica";
list.add("Jessica");
```

- retrieving a value

```
String s = list[0];
String s = list.get(0);
```

- doing something to each value that starts with "B"

```
for (int i = 0; i < list.length; i++) {
    if (list[i].startsWith("B")) { ... }
}
for (int i = 0; i < list.size(); i++) {
    if (list.get(i).startsWith("B")) { ... }
}
```

- seeing whether the value "Benson" is found

```
for (int i = 0; i < list.length; i++) {
    if (list[i].equals("Benson")) { ... }
}
if (list.contains("Benson")) { ... }
```

An other collection: Set Interface

- A Set is a collection, it is an unordered list and has no duplicates.
- It is an interface so you can't say new Set( )
- There are four implementations:
  - HashSet is best for most purposes, backed by a hash table
  - TreeSet guarantees that an iterator will return elements in sorted order
  - LinkedHashSet guarantees that an iterator will return elements in the order they were inserted
  - AbstractSet is a "helper" abstract class for new implementations

```
Set s = new HashSet( );
HashSet s = new HashSet( );
```

- https://docs.oracle.com/javase/8/docs/api/java/util/Set.html

Map interface

- A Map is an object that maps keys to values: `Map<Key, Value>`

- A map cannot contain duplicate keys and each key can map to at most one value

- Map is an interface; you can’t say new Map( )
- Here are two implementations:
  - HashMap is the faster
  - TreeMap guarantees the order of iteration

```
Map map<KeyType, ValueType> = new HashMap <KeyType,ValueType> ( );
```

- https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

Map example

```
import java.util.*;

public class MapExample {

    public static void main(String[] args) {
        Map<String, String> fruit = new HashMap<String,
String>();
        fruit.put("Apple", "red");
        fruit.put("Pear", "yellow");
        fruit.put("Plum", "purple");
        fruit.put("Cherry", "red");
        for (String key : fruit.keySet()) {
            System.out.println(key + ": " +
fruit.get(key));
        }
    }
}
```

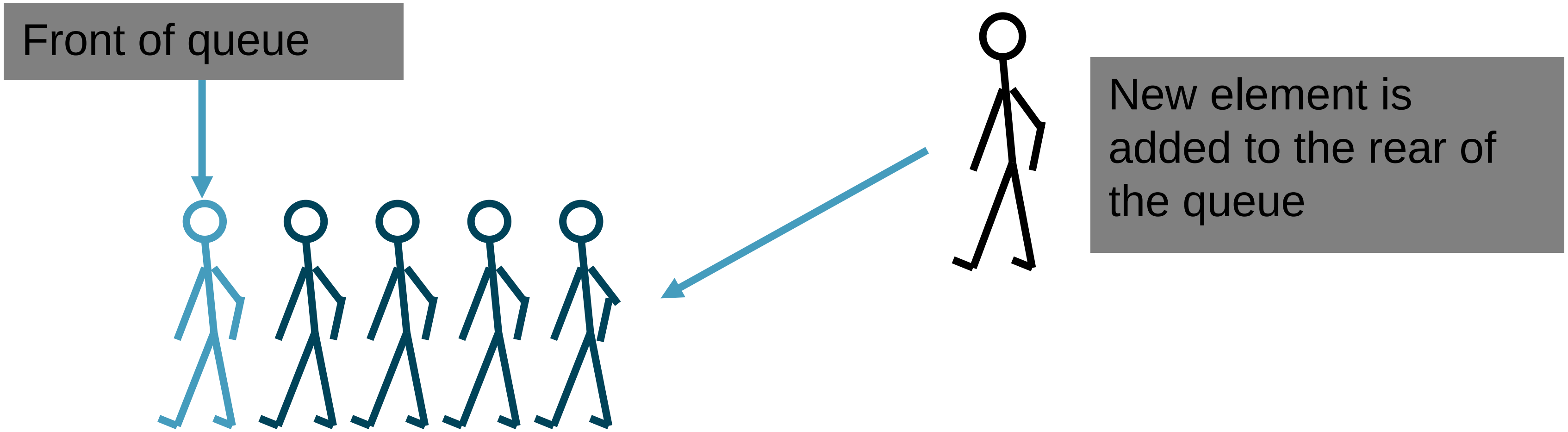
```
Plum: purple
Apple: red
Pear: yellow
Cherry: red
```

Queue

- A collection whose elements are added at one end (the ***rear*** or ***tail*** of the queue) and removed from the other end (the ***front*** or ***head*** of the queue)
- A queue is a ***FIFO*** (first in, first out) data structure
- Any waiting line is a queue:
  - The check-out line at a grocery store
  - The cars at a stop light
  - An assembly line

https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html

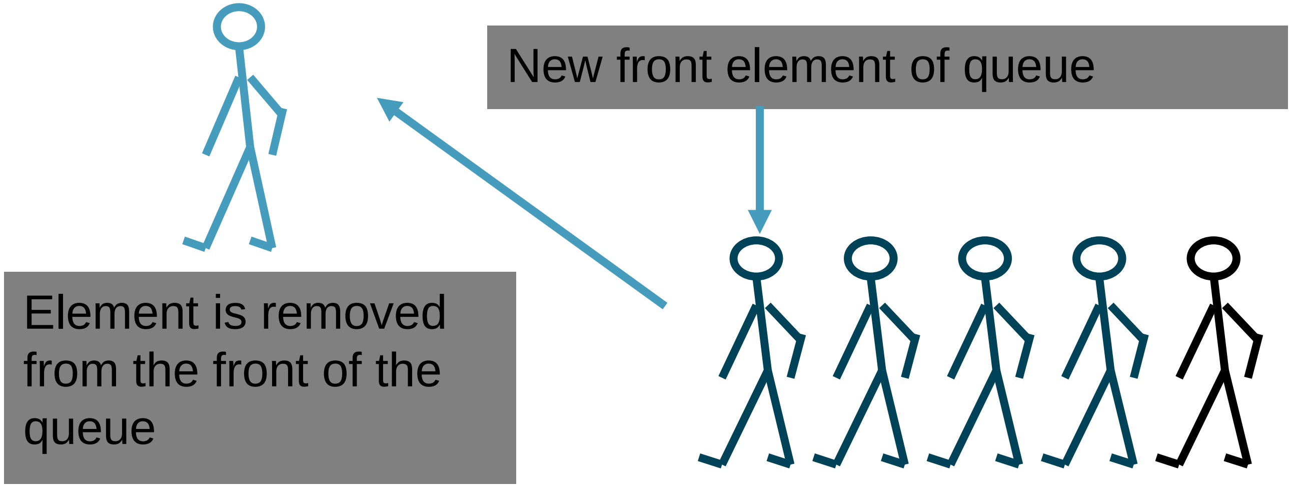
Conceptual view of a Queue



Front of queue

New element is added to the rear of the queue

Removing an element



New front element of queue

Element is removed from the front of the queue

Operation on a Queue

Operation	Description
dequeue	Removes an element from the front of the queue
enqueue	Adds an element to the rear of the queue
first	Examines the element at the front of the queue
isEmpty	Determines whether the queue is empty
size	Determines the number of elements in the queue
toString	Returns a string representation of the queue

SEARCHING AND SORTING

Searching and sorting

- Fundamental problems in computer science and programming
- Sorting done to make searching easier
- Multiple different algorithms to solve the same problem
  - How do we know which algorithm is "better"?

In Java

- Class `Arrays` in `java.util` has many useful array methods:

Method name	Description
`binarySearch(`**array**`, `**value**`)`	returns the index of the given value in a *sorted* array (or < 0 if not found)
`binarySearch(`**array**`,` **minIndex**`, `**maxIndex**`, `**value**`)`	returns index of given value in a *sorted* array between indexes *min* / *max* - 1 (< 0 if not found)
`copyOf(`**array**`, `**length**`)`	returns a new resized copy of an array
`equals(`**array1**`, `**array2**`)`	returns `true` if the two arrays contain same elements in the same order
`fill(`**array**`, `**value**`)`	sets every element to the given value
`sort(`**array**`)`	arranges the elements into sorted order
`toString(`**array**`)`	returns a string representing the array, such as `"[10, 30, -25, 17]"`

- Syntax:`Arrays.`**methodName**`(`**parameters**`)`

Comparable in Java

- Very useful for sorting collection.
- We can sort an array/list of primitive types
- We can sort array/list of custom objects.
- Let’s see how it works through the following example

Sorting

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class Lecture5 {

public static void main(String[] args) {

        //sort primitives array like int array
        int[] intArr = {5,29,10,11};
        Arrays.sort(intArr);
        System.out.println(Arrays.toString(intArr));

        //sorting String array
        String[] strArr = {"A", "C", "B", "Z", "F"};
        Arrays.sort(strArr);
        System.out.println(Arrays.toString(strArr));
```

Importing libraries

Sorting array of int

Sorting array of String

The program continues…

```
//sorting list of objects of Wrapper classes
        List<String> strList = new ArrayList<String>();
        strList.add("A");
        strList.add("C");
        strList.add("B");
        strList.add("Z");
        strList.add("F");
        Collections.sort(strList);
        for(String str: strList)
             System.out.print(" "+str);
    }

}
```

We have a List

Sorting an array of custom Objects

- We want to sort objects Employee based on the salary!
- How can we sort objects? Employee class can implement the interface Comparable:

```
public class Employee implements Comparable<Employee> {

    private int id;
    private String name;
    private int age;
    private int salary;

   public Employee(int id, String name, int age, int
                                                   salary) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
```

…continue in the next slide

```
    public int getId() {
          return id;}

     public String getName() {
          return name;}

     public int getAge() {
          return age;}

     public long getSalary() {
          return salary;}

    public String toString(){
          return "Employee name = " + name + ", id = " + id +
         ", age = " + age + ", salary = " + salary + "\n";
     }
```

let's sort the employee based on id in ascending order

```
     @Override
     public int compareTo(Employee emp) {
            return (this.salary - emp.salary);
     }
```

Returns a negative integer, zero, or a positive integer as this employee salary is less than, equal to, or greater than the specified object.

Test Sort Employees

```
import java.util.Arrays;

public class test {

public static void main(String[] args) {

Employee[] empArr = new Employee[4];

empArr[0] = new Employee(10, "Mikey", 25, 10000);
empArr[1] = new Employee(20, "Arun", 29, 20000);
empArr[2] = new Employee(5, "Lisa", 35, 5000);
empArr[3] = new Employee(1, "Pankaj", 32, 50000);

//sorting employees array using Comparable interface

Arrays.sort(empArr);
System.out.println("Default Sorting of Employees
list:\n"+Arrays.toString(empArr));
}
}
```

Sorting algorithms

- Sorting algorithms are used to arrange random data into some order
  - can be solved in many ways:
    - there are many sorting algorithms
    - some are faster/slower than others
    - some use more/less memory than others
    - some work better with specific kinds of data
    - some can utilize multiple computers / processors, ...
  - *comparison-based sorting* : determining order by comparing pairs of elements:
    - <, >, `compareTo`, ...

Sorting algorithms

- **bubble sort**: swap adjacent pairs that are out of order
- **selection sort**: look for the smallest element, move to front
- **insertion sort**: build an increasingly large sorted front portion
- **merge sort**: recursively divide the array in half and sort it
- **heap sort**: place the values into a sorted tree structure
- **quick sort**: recursively partition array based on a middle value

other specialized sorting algorithms:

- **bucket sort**: cluster elements into smaller groups, sort them
- **radix sort**: sort integers by last digit, then 2nd to last, then ...
- ...

Linked List

A ***linked*** data structure consists of items, called Nodes, that are linked to other items

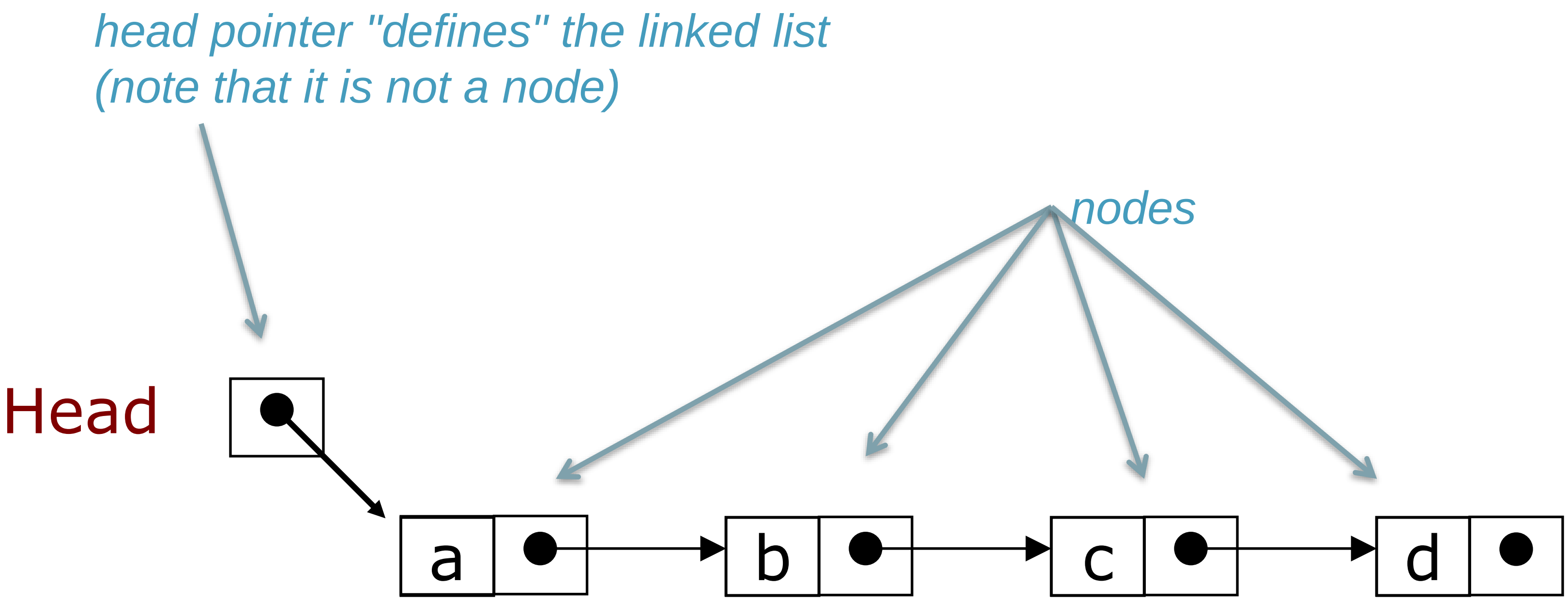
Singly linked list: each item points to the next item

- A linked list is an ordered sequence of items called ***nodes***
  - A node is the basic unit of representation in a linked list
- A ***node*** in a ***singly linked list*** consists of two fields:
  - A ***data*** portion
  - A ***link (pointer)*** to the ***next*** node in the structure
- The first item (node) in the linked list is accessed via a ***front*** or ***head*** pointer
  - The linked list is defined by its head (this is its starting point)

Advantages of Linked List

- The items do ***not*** have to be stored in consecutive memory locations: the successor can be anywhere physically
  - So, can insert and delete items without shifting data
  - Can increase the size of the data structure easily
- Linked lists can grow ***dynamically*** (i.e. at run time) – the amount of memory space allocated can grow and shrink as needed

Singly Linked List


head pointer "defines" the linked list
(note that it is not a node)
nodes
Head
a
b
c
d

Traversing the linked list

How is the first item accessed?
The second?
The last?

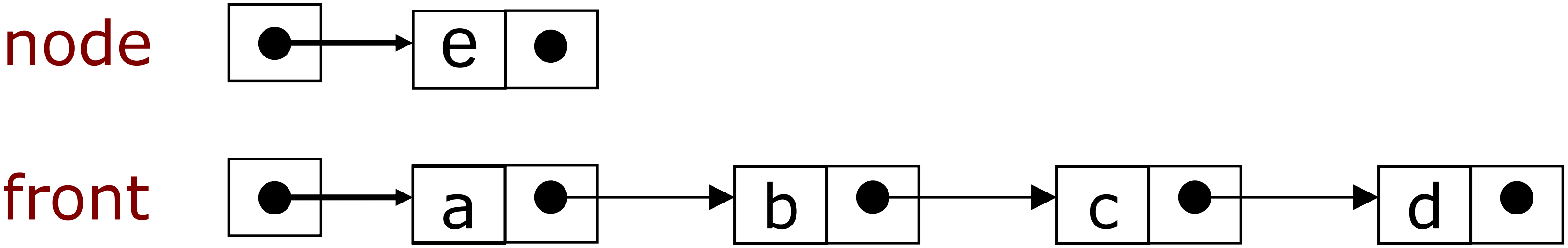
What does the last item point to?
We call this the ***null link***

Some operations

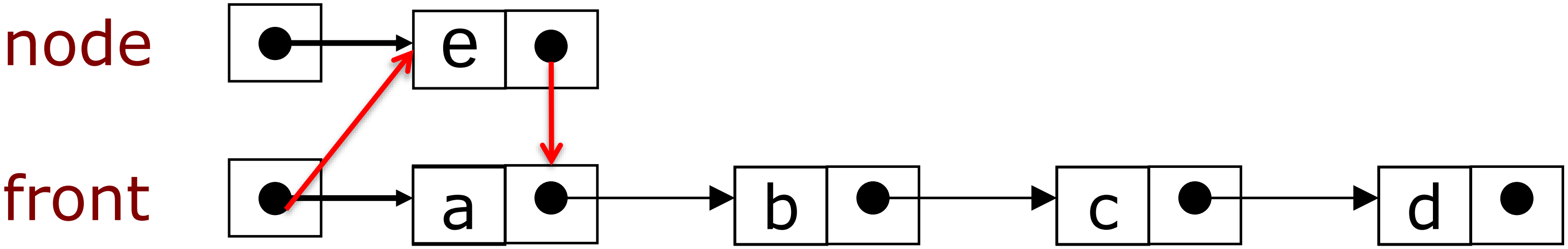
- ***Add*** an item to the linked list
  - We have 3 situations:
    - insert a node at the front
    - insert a node in the middle
    - insert a node at the end
- ***Delete*** an item from the linked list
  - We have 3 situations :
    - delete the node at the front
    - delete an interior node
    - delete the last node

Insert a node at the front

- node points to the new node to be inserted, front points to the first node of the linked list



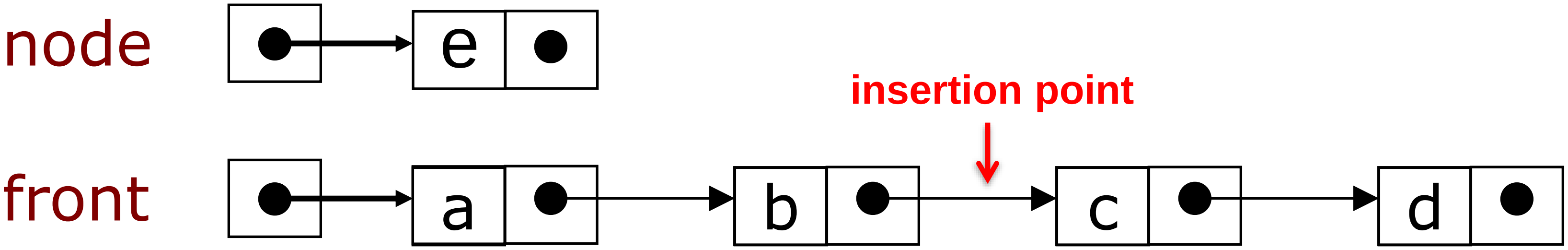
- 1. Make the new node point to the first node (i.e. the node that front points to)



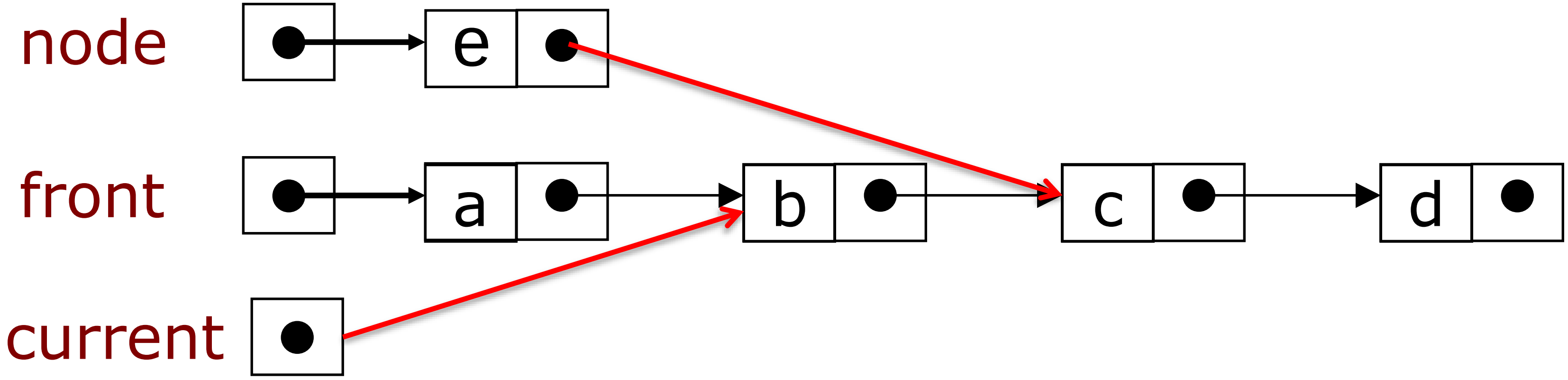
- 2. Make frontpoint to the new node (i.e the node that node points to)

Insert a node in the middle

- Let's insert the new node after the *second* node in the linked list



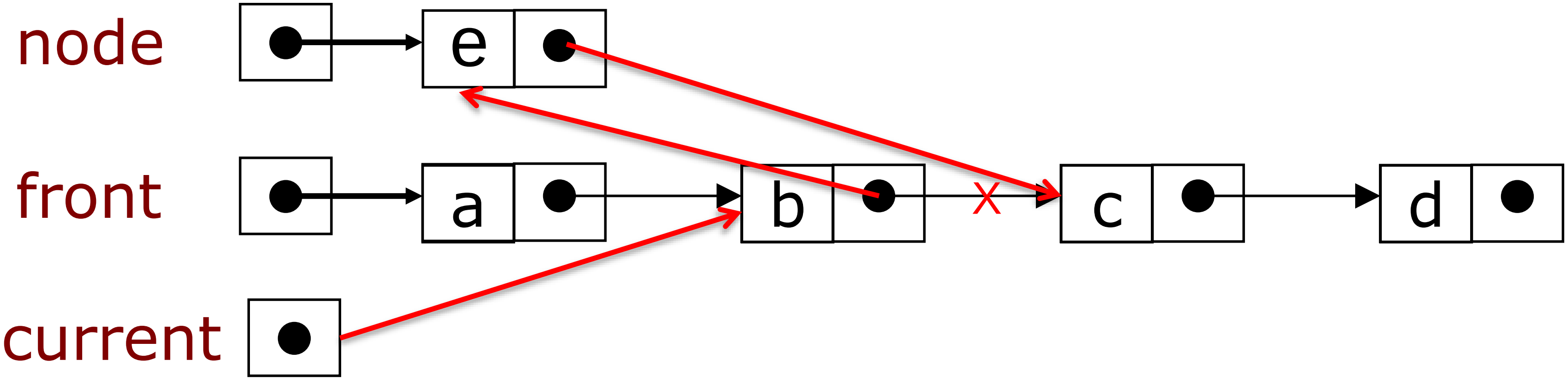
- 1. Locate the node *preceding the insertion point* , since it will have to be modified (make current point to it)



- 2. Make the new node point to the node after the insertion point (i.e. the node pointed to by the node that current points to)

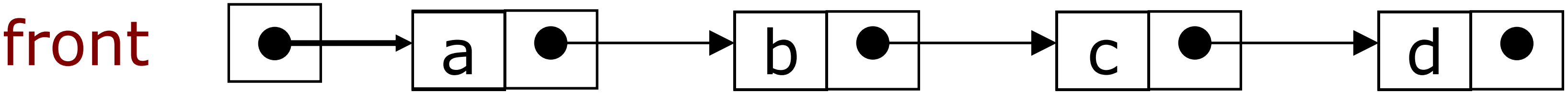
Insert a node in the middle

- 3. Make the node pointed to by current point to the new node

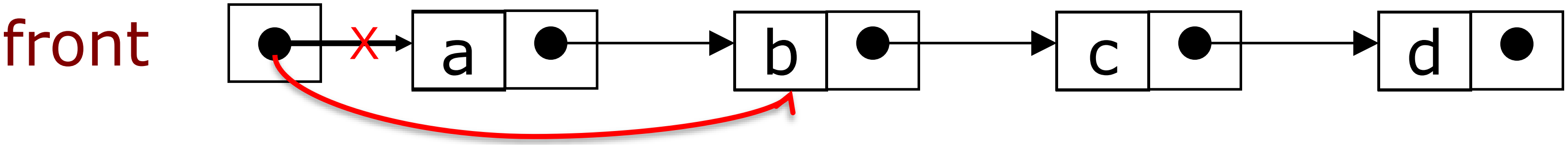


Delete the node at the front

- front points to the first node in the linked list, which points to the second node

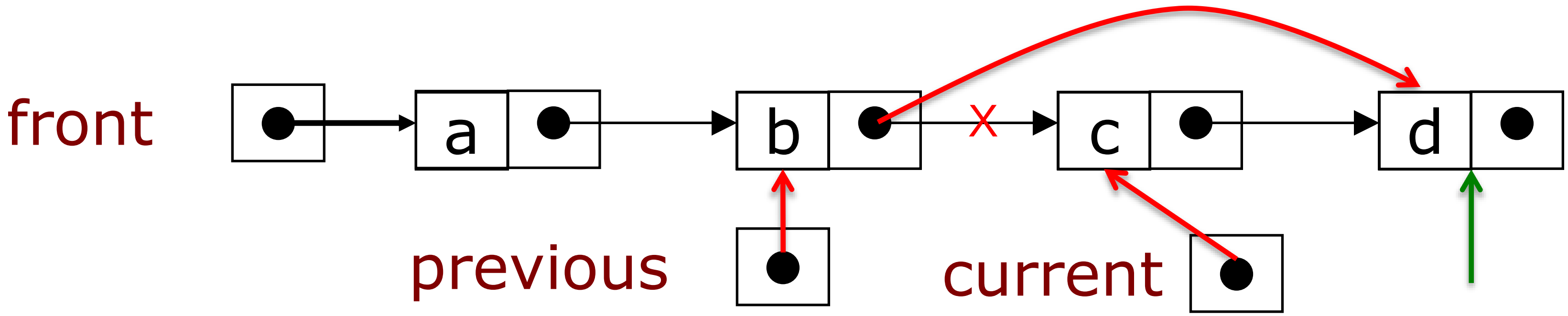


- 1. Make front point to the second node (i.e. the node pointed to by the first node)



Delete an interior node

- 1. Traverse the linked list so that current points to the node to be deleted and previous points to the node prior to the one to be deleted



- 2. We need to get at the node *following the one to be deleted* (i.e. the node pointed to by the node that current points to)

- 3. Make the node that previous points to, point to the node following the one to be deleted

Node implementation

```
class Node<T> {

    private T element;
    private Node<T> next;

    public T getValue() {
        return element;
    }
    public void setValue(T value) {
        this.element = value;
    }
    public Node<T> getNextRef() {
        return next;
    }
    public void setNextRef(Node<T> ref) {
        this.next = ref;
    }
}
```

Linked List implementation

```
public class SinglyLinkedList<T> {

    private Node<T> front;
    private Node<T> end;

    public void add(T newElement){

      // ..code here
    }

    public void addAfter(T newElement, T after){

       // .. code here

    }

    public void deleteFront(){

      // .. code here
    }

    public void deleteAfter(T after){

      // .. code here

    }

    public void traverse(){

      // .. code here
    }}
```