UCL DEPARTMENT OF ELECTRONIC & ELECTRICAL ENGINEERING

ELEC0021 - PROGRAMMING II
OBJECT-ORIENTED PROGRAMMING

DATA STRUCTURES

Prof. George Pavlou
Communication and Information Systems Group
Dept. of Electronic and Electrical Engineering
http://www.ee.ucl.ac.uk/~gpavlou/
g.pavlou@ee.ucl.ac.uk

Data Structures

- A data structure is a way of organising and storing data so that they can be accessed efficiently
- The simplest form of data structure is the array, which is a fixed-size data structure
  - The size is specified at creation time
- More advanced data structures are dynamic; these do not have a predefined size but can grow and shrink at execution time
  - They use dynamic memory allocation

Dynamic Data Structures

- Linked List: a collection of data “linked up in a chain”; insertions and removals are possible anywhere in a linked list
  - Used in simulation software and many other applications
- Stack: similar to a linked list but insertions and removals are only possible at the top of the stack
  - Used in programming language runtime systems etc.
- Queue: similar to a linked list but insertions are only possible at the back and removals at the front
  - Models a waiting line
- Binary Tree: a hierarchical structure in which a (parent) node has at most two children nodes
  - Used in file systems, compilers, etc.

Dynamic Memory Allocation

- Dynamic data structures require dynamic memory allocation: the ability to obtain memory at execution time to hold new nodes of that data structure
  - E.g. ListNode node = new ListNode(new Integer(10));
- Allocated memory does not need to be released as Java provides automatic garbage collection, unlike C and C++
  - The memory of objects no longer referenced is eventually claimed by the garbage collector, e.g.

```
ListNode node = new ListNode(new Integer(10));
node = null;
// …
```

  - The ListNode/Integer memory will be eventually claimed

Type-Wrapper Classes

- Each one of Java's eight primitive types has a corresponding type-wrapper class so that it can be treated as an object
  - Byte, Short, Integer, Long, Float, Double, Boolean, Character
- An example of using the Integer type-wrapper class:
  - Integer i = new Integer(10); int ival = i.intValue();
- We have also used before its static parseInt method:
  - int i = Integer.parseInt("10");
- This is useful as most data structures store objects (i.e. of class Object) as common form of data
  - This way we can store primitive type data in object form

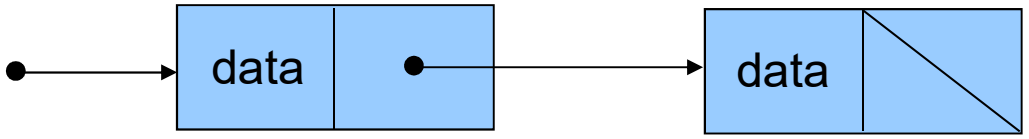
Arrays

- An array is a fixed size data structure which is an object in Java, so the *new* operator is used to allocate space for it
  - A one dimensional array is declared as <type>[] <var> and is created with space is allocated as new <type>[<size>], e.g.
    int[] intArray = new int[10]; // an array of 10 integers, in the heap
  - Note that in C/C++ arrays are declared slightly differently and their space is in the stack, like primitive types e.g.
    int intArray[10]; // an array of 10 integers, in the stack
- The array public instance variable length provides its size
- The first array element is <var>[0] while the last one is <var>[length-1]; array elements are accessed in a for loop:

```
Random randGen = new Random();
for (i = 0; i < intArray.length; i++)
    intArray[i] = randGen.nextInt(100)+1;  // initialise with 1..100
```

Self-Referential Classes

- A self-referential class contains a reference instance variable that refers to another object of the same class
- Self-referential objects can be linked together to form dynamic data structures such as lists, queues, stacks and trees
- An example pictorial representation of two linked self-referential objects is shown below
  - The backslash represents a null reference



A ListNode Self-Referential Class

```
public class ListNode {

    // public instance variables as there is nothing to "protect"
    public Object data;       // the contained data object
    public ListNode next;   // the self-reference

    // constructors
    public ListNode (Object newData, ListNode newNext) {
        data = newData;
        next = newNext;
    }

    public ListNode () {
        data = null;
        next = null;
    }

} // end class ListNode
```

Linked List

- A linked list or simply list is a linear collection of self-referential objects connected by reference links, hence the term “linked list”
  - A program accesses a linked list via a reference to its first node
  - The reference of the last node is set to null
- A linked list may contain any data, i.e. (a reference to) an Object can serve as generic form of data
- New nodes can be added dynamically, as needs arise
- Key List class methods and their signatures:
  - public void insertAtFront (Object newData)
  - public void insertAtBack (Object newData)
  - public Object removeFromFront ()
  - public Object removeFromBack ()
  - public ListNode getFirst ()

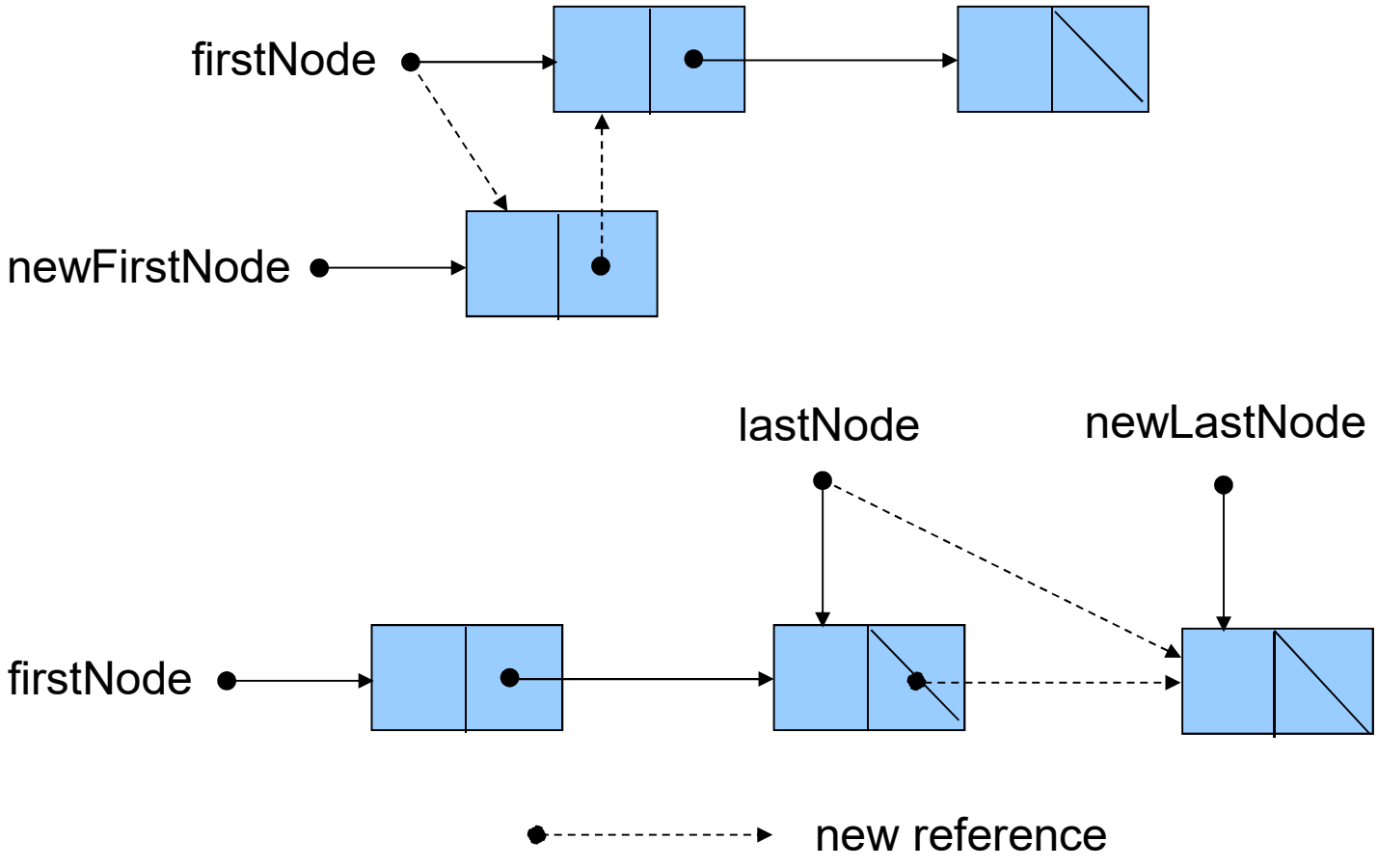
Linked Lists and Arrays

- A linked list can be thought as similar to an array as a list node can contain data, in a similar fashion to an array cell
- But a linked list does not have a limited size, it can rather grow arbitrarily in size
  - Until the machine's virtual memory is exhausted that is
- Also adding an element to a sorted array requires moving a number of elements one position up in order to create a free slot; this can be costly
  - Adding a new element to a sorted linked list simply requires to *link* a node in the right position, which is much more efficient
- On the other hand, accessing an array element is done directly while in a linked list it means traversing the list up to that element
  - More processing is required in this case

List Class (1/4)

```
public class List {
   protected ListNode firstNode;      // the first node
   protected ListNode lastNode;       // the last node
   protected String      name;        // a string name

   public void insertAtFront (Object newData) {   // insert Object at front
      if (firstNode == null)  // empty list
         firstNode = lastNode = new ListNode(newData, null);
      else {
         ListNode newFirstNode = new ListNode(newData, firstNode);
         firstNode = newFirstNode;
      }
   }
   public void insertAtBack (Object newData) {    // insert Object at back
      if (firstNode == null)   // empty list
         firstNode = lastNode = new ListNode(newData, null);
      else {
         ListNode newLastNode = new ListNode(newData, null);
         lastNode.next = newLastNode;
         lastNode = newLastNode;
      }
   }
// continues on the next page
```

Graphical Representation of insertAtFront and insertAtBack



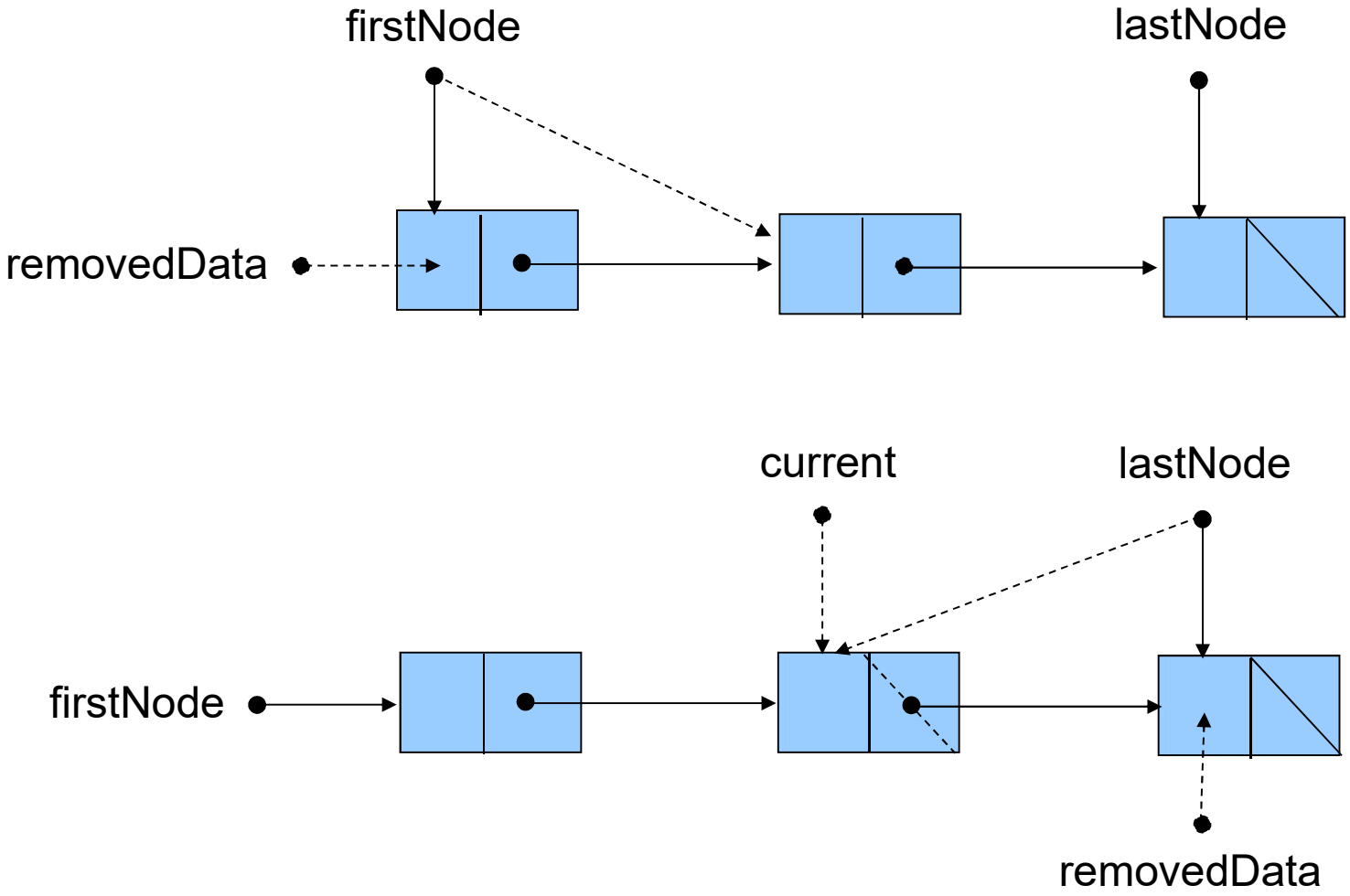
List Class (2/4)

```
// continues from the previous page

    public Object removeFromFront () {   // remove Object from front
        if (firstNode == null)  // empty list
            return null;

        Object removedData = firstNode.data;

        if (firstNode == lastNode)  // only one list node
            firstNode = lastNode = null;
        else
            firstNode = firstNode.next;

        return removedData;
    }

// continues on the next page
```

List Class (3/4)

```
// continues from the previous page

    public Object removeFromBack () {   // remove object from back
        if (firstNode == null)  // empty list
            return null;

        Object removedData = lastNode.data;

        if (firstNode == lastNode)  // only one node in the list
            firstNode = lastNode = null;
        else {
            ListNode current = firstNode;
            while (current.next != lastNode)
                current = current.next;
            lastNode = current;
            current.next = null;
        }

        return removedData;
    }

// continues on page 19
```

Graphical Representation of removeFromFront and removeFromBack



Insert and Remove Methods Logic

- The logic of the insert methods is as follows:
  - We first check to see if the list is empty, in which case insertion is trivial and the same as initialisation
  - If not, we create the new node and link it appropriately
- The logic of the remove methods is as follows:
  - We first check to see if the list is empty, in which case we simply return null
  - If not and there is only one node, removing is trivial, we simply set the first and last references to null
  - Otherwise, in removeFromFront we simply advance the first reference
  - In removeFromBack, which is more complicated, we need to advance a current reference until the penultimate node and set the last reference and next accordingly

Insert at Back Pseudo Code

```
method insertAtBack
    input: object to insert
    output: nothing

    if the list is empty
        make new node with object to insert
        set first and last node references to it
    else
        make new last node with object to insert
        link current last node with new last node
        set last node reference to the new last node
end insertAtBack
```

Remove From Back Pseudo Code

```
method removeFromBack
    input: none
    output: the last object stored in the list

    if the list is empty
        return nothing

    if there is only one node in the list
        nullify references to first and last node
    else
        advance a temporary reference until the penultimate node
        make this the last node

    return the last object in the list
end removeFromBack
```

List Class (4/4)

```
// getFirst allows one to get a “handle” to navigate through the list in a loop
public ListNode getFirst () {
    return firstNode;
}

@override
public String toString () {      // print list content to string
    String output = new String();
    ListNode current = firstNode;

    output = name + “:”;
    while (current != null) {
        // we are implicitly calling the data object toString method
        output += “ ” + current.data;
        current = current.next;
    }
    return output;
}

// constructors
public List (String listName) { firstNode = lastNode = null; name = listName; }
public List () { this(“List”); }

}  // end class List
```

An Example ListTest Program

```
public class ListTest {
    public static void main (String[] args) {
        List list = new List();

        list.insertAtFront(new Integer(2));
        list.insertAtBack(new Integer(-1));
        list.insertAtBack(new Integer(9));
        list.insertAtBack(new Integer(5));
        System.out.println(list);

        Object removedData = list.removeFromFront();
        System.out.println("removed data is: " + removedData);

        removedData = list.removeFromBack();
        // we know removedData is of class Integer so
        // we "downcast" a superclass (Object) to a subclass (Integer)
        Integer i = (Integer) removedData;
        System.out.println("removed data is: " + i);

        System.out.println(list);
    }
}  // end class ListTest
```

Stack

- A stack is a dynamic data structure in which data can only be inserted and removed from the top (i.e. front)
  - A stack is also referred to as a Last-In-First-Out (LIFO) data structure
- The primary methods for manipulating a stack object are push and pop
- A stack is typically implemented using a linked list or an array
  - This is hidden from its users – an Abstract Data Structure
  - If a list is used, insertAtFront/removeFromFront => push/pop
  - If an array is used, there is a maximum stack size
- Stacks are used in programming language runtime systems to push the arguments of called functions/ methods and pop those of terminating ones
  - Many other uses in compilers, etc.

Stack Implementation

- We can implement a stack by inheriting from a linked list
  - In this case though other public methods of linked list such as insertAtBack/removeFromBack should NOT be used
  - A solution is to redefine those to do nothing (see next)
- A better approach is to implement a stack through "composition" i.e. by having a linked list or array instance variable
- We will show three implementations of a Stack:
  - By inheriting from a linked list
  - By having a linked list instance variable
  - By having an array instance variable
- All these serve the same "class interface", i.e. its public methods - key of OO encapsulation and information hiding

Stack Through List Inheritance

```
public class Stack extends List {   // an “is-a” relationship
    public Stack () {
        super(“Stack”);
    }

    public void push (Object data) {
        insertAtFront(data);
    }

    public Object pop () {
        return removeFromFront();
    }

    @Override  // override and “nullify” insertAtBack
    public void insertAtBack (Object data) {
        System.err.println (“insertAtBack should not be called in Stack!”);
    }

    @Override  // override and “nullify” removeFromBack
    public Object removeFromBack () {
        System.err.println (“removeFromBack should not be called in Stack!”);
        return null;
    }
}  // end class Stack
```

Stack Through List Composition

```
public class Stack {
    private List stackList;  // a contained list – a "has-a" relationship

    public Stack () {
        stackList = new List("Stack");
    }

    public void push (Object data) {
        stackList.insertAtFront(data);
    }

    public Object pop () {
        return stackList.removeFromFront();
    }

    public ListNode getFirst () {
        return stackList.getFirst();
    }

    public String toString () {
        return stackList.toString();
    }
}  // end class Stack
```

Stack Through Array Composition

```
public class StackArray {
    private Object stackArray[];                 // a contained array
    final private int maxStackSize = 1024;   // its max size
    private int current;                             // its current top position

    public StackArray () {
        stackArray = new Object[maxStackSize];
        current = 0;
    }

    public void push (Object data) {
        if (current == maxStackSize)
            return;  // full stack, we should ideally throw an exception here
        stackArray[current++] = data
    }

    public Object pop () {
        if (getFirst() == null)
            return null;  // empty stack
        return stackArray[--current];
    }

// complete with isFull and toString methods
```

Queue

- A queue is a dynamic data structure in which data can only be inserted at the back and removed from the front
  - A queue is also referred to as a First-In-First-Out (FIFO) data structure
- The primary methods for manipulating a queue object are enqueue and dequeue
- A queue is typically implemented using a linked list
  - An array implementation would not be efficient because both ends of the queue are manipulated
- Queues are used in discrete event simulations to store "events to happen", in spooling systems and many other computer applications

Queue Through List Composition

```
public class Queue {
    private List queueList;  // a contained list – a "has-a" relationship

    public Queue () {
        queueList = new List("Queue");
    }

    public void enqueue (Object data) {
        queueList.insertAtBack(data);
    }

    public Object dequeue () {
        return queueList.removeFromFront();
    }

    public ListNode getFirst () {
        return queueList.getFirst();
    }

    public String toString () {
        return queueList.toString();
    }
}  // end class Queue
```

Summary

- Data structures allow programs to organise, store and access data
- Arrays are fixed size data structures while lists, stacks and queues are dynamic self-referential structures
  - It is important to know how to implement the List methods
- We typically store data in dynamic data structures through instances of class Object; the wrapper classes allow to store primitive types through the corresponding type-wrapper classes
  - After retrieving data, we downcast the contained object to the correct class e.g. Integer i = (Integer) removedData;
- You may implement these classes and test them through suitable test programs