CS 304 Lecture 7

More on lists and the `Serializable` interface

Xiwei Wang, Ph.D.

Assistant Professor

Department of Computer Science

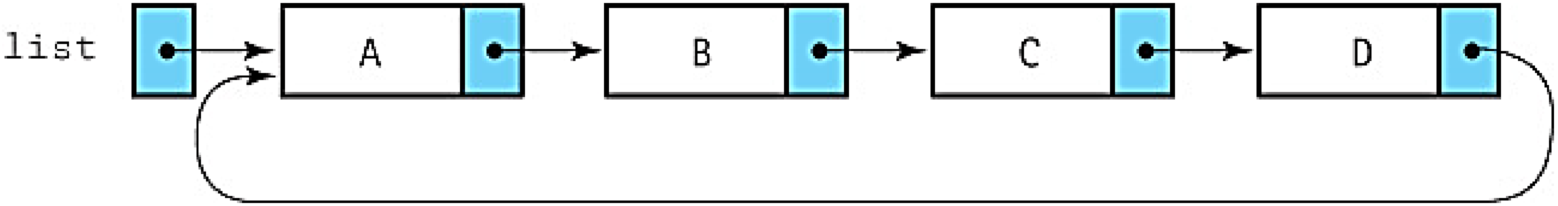
Northeastern Illinois University

Chicago, Illinois, 60625

25 October 2016

Circular linked lists

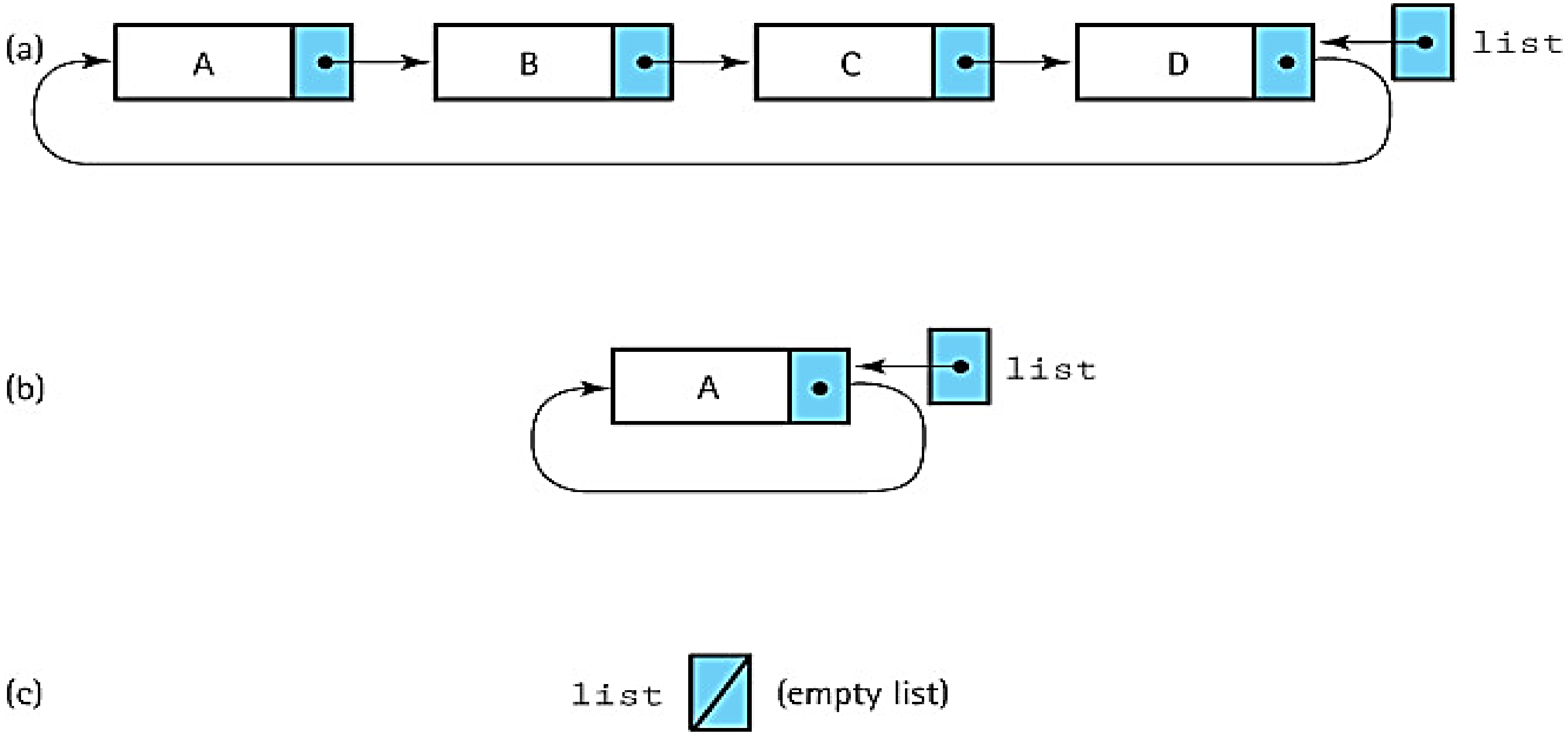
- **Circular linked list** - A list in which every node has a successor; the "last" element is succeeded by the "first" element.



- Adding and removing elements at the front of a list might be a common operation for some applications. Then what is the problem with the above list structure?
  - You have to traverse the list from its beginning to the end and then insert the new node.
  - This would take a long time when the list contains a large number of elements.

Circular linked lists

- We can fix this problem by letting our list reference point to the last element in the list rather than the first; now we have easy access to both the first and the last elements in the list.



The **CRefUnsortedList** Class

- Recall the **reset** and **getNext** methods in the **RefUnsortedList** class:

```
public void reset()
{
  currentPos = list;
}

public T getNext()
{
  T next = currentPos.getInfo();
  if (currentPos.getLink() == null)
    currentPos = list;
  else
    currentPos = currentPos.getLink();
  return next;
}
```

The **CRefUnsortedList** Class

- The **reset** and **getNext** methods in the **CRefUnsortedList** class need to be modified:

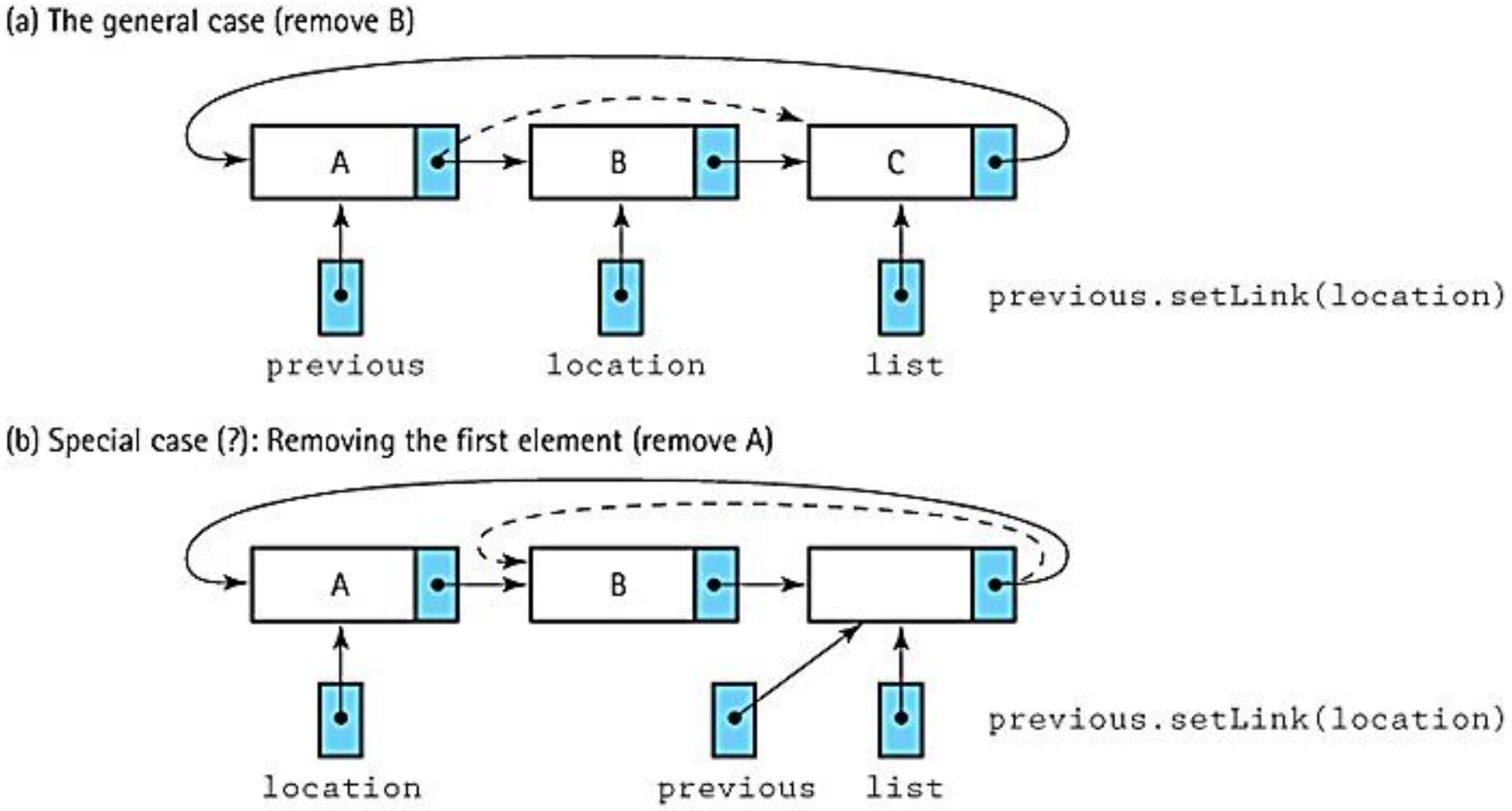
```
public void reset()
{
  if (list != null)
    currentPos = list.getLink();
}

public T getNext()
{
  T next = currentPos.getInfo();
  currentPos = currentPos.getLink();
  return next;
}
```

The `toString` method

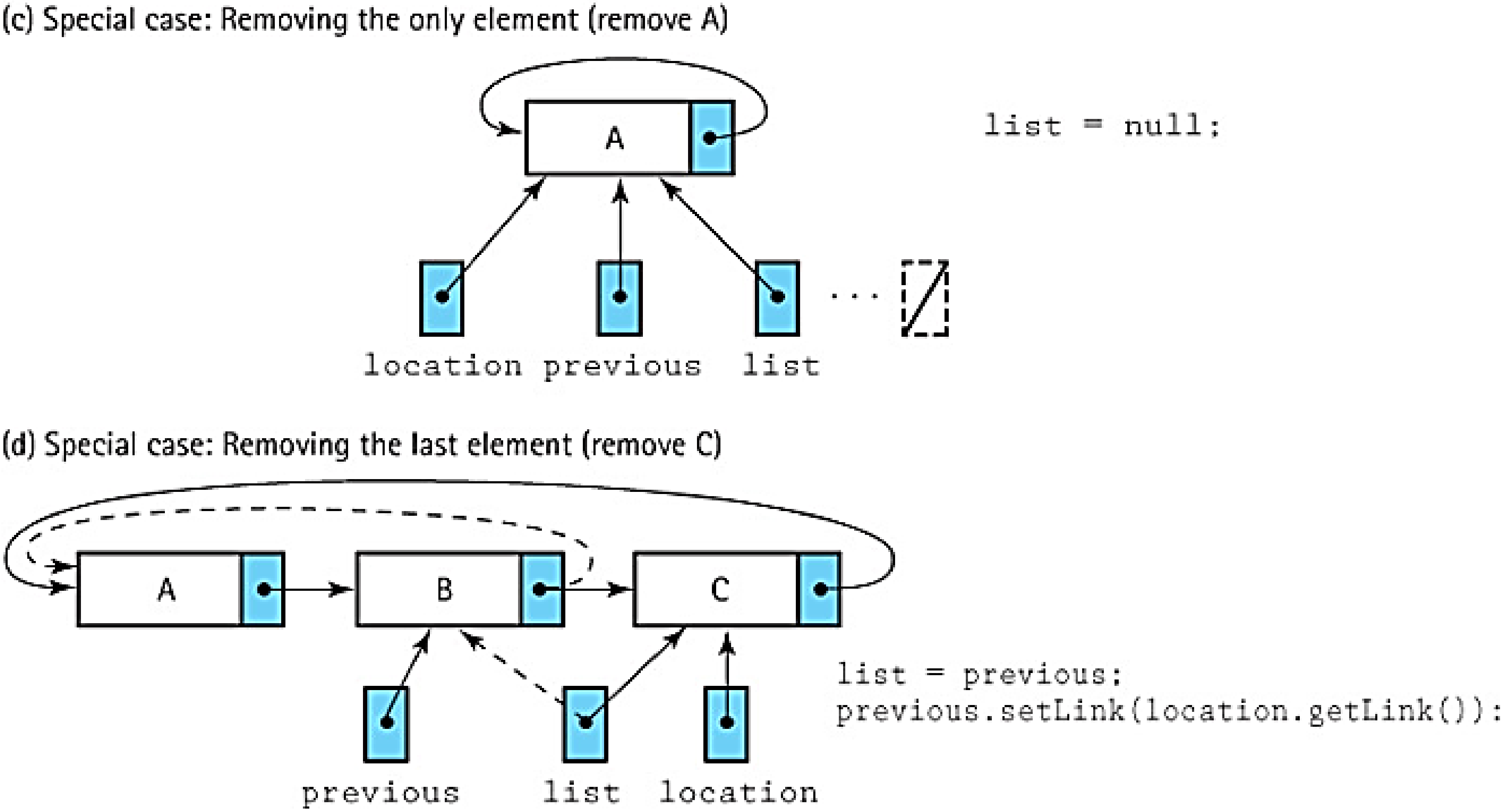
```
public String toString()
// Returns a nicely formatted String that represents this
// list.
{
  String listString = "List:\n";
  if (list != null)
  {
    LLNode<T> prevNode = list;
    do
    {
      listString = listString + "  " +
                   prevNode.getLink().getInfo() + "\n";
      prevNode = prevNode.getLink();
    }
    while (prevNode != list);
  }
  return listString;
}
```

The `remove` method



(a) The general case (remove B)
A
B
C
previous
location
list
previous.setLink(location)
(b) Special case (?): Removing the first element (remove A)
A
B
location
previous
list
previous.setLink(location)

The **`remove`** method



(c) Special case: Removing the only element (remove A)

```
list = null;
```

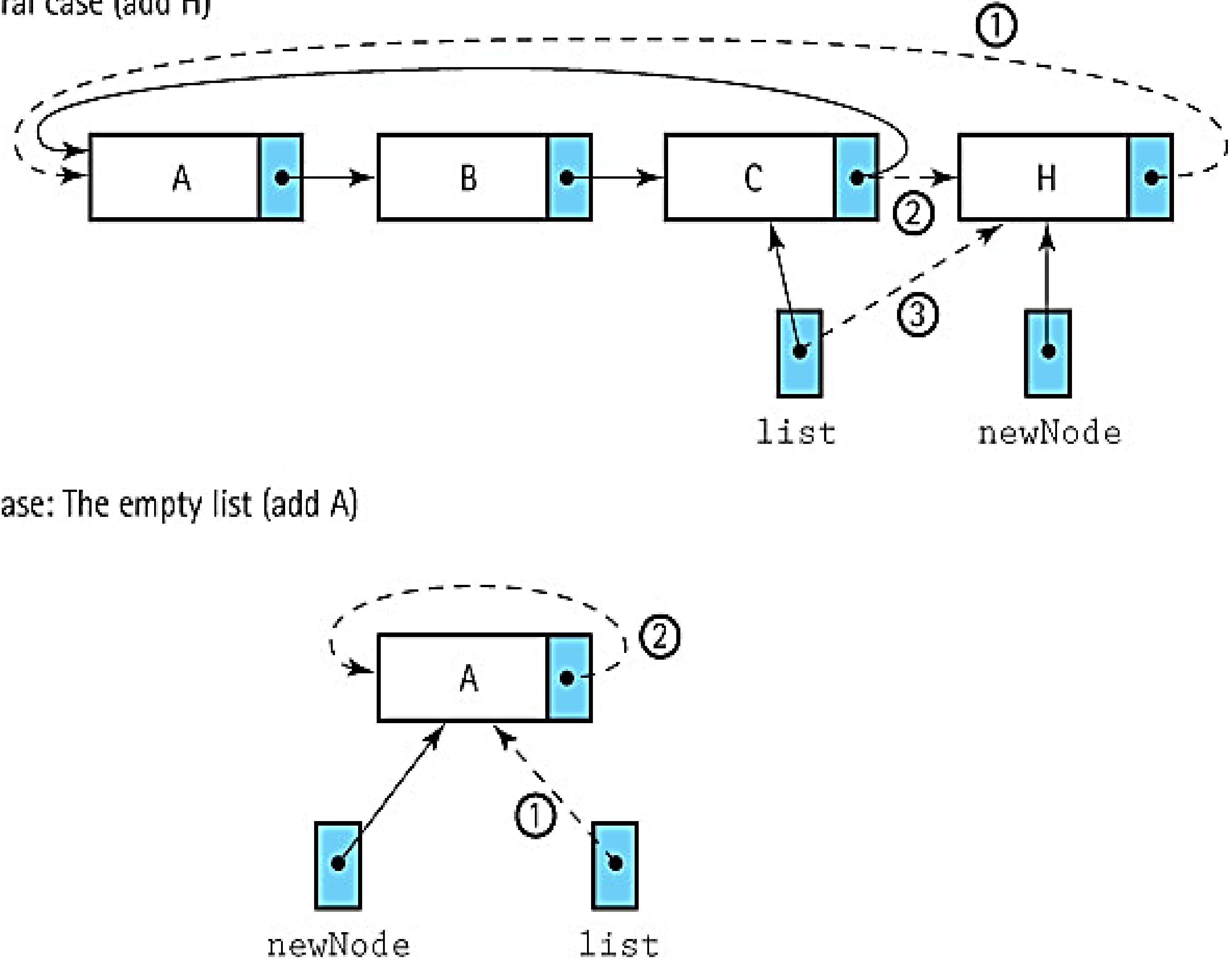
(d) Special case: Removing the last element (remove C)

```
list = previous;
previous.setLink(location.getLink());
```

The **remove** method

```
public boolean remove (T element)
// Removes an element e from this list such that
// e.equals(element) and returns true;
// if no such element exists returns false.
{
  find(element);
  if (found)
  {
    if (list == list.getLink())    // if single element list
      list = null;
    else
      if (previous.getLink() == list) // if removing last node
        list = previous;
      previous.setLink(location.getLink()); // remove node
    numElements--;
  }
  return found;
}
```

The `add` method



(a) The general case (add H)

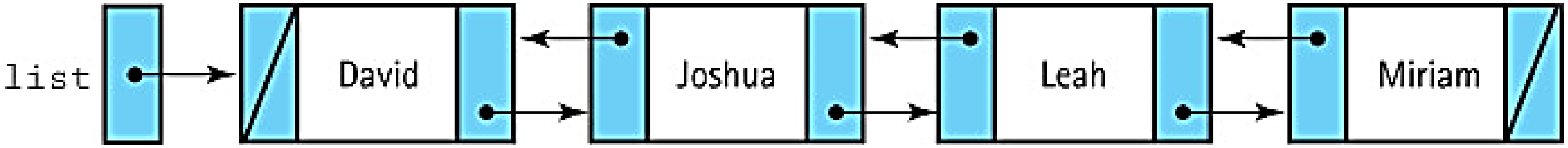
(b) Special case: The empty list (add A)

The add method

```
public void add(T element)
// Adds element to this list.
{
  LLNode<T> newNode = new LLNode<T>(element);
  if (list == null)
  {
    // add element to an empty list
    list = newNode;
    newNode.setLink(list);
  }
  else
  {
    // add element to a non-empty list
    newNode.setLink(list.getLink());
    list.setLink(newNode);
    list = newNode;
  }
  numElements++;
}
```

Doubly linked lists

- **Doubly linked list** - A linked list in which each node is linked to both its successor and its predecessor.



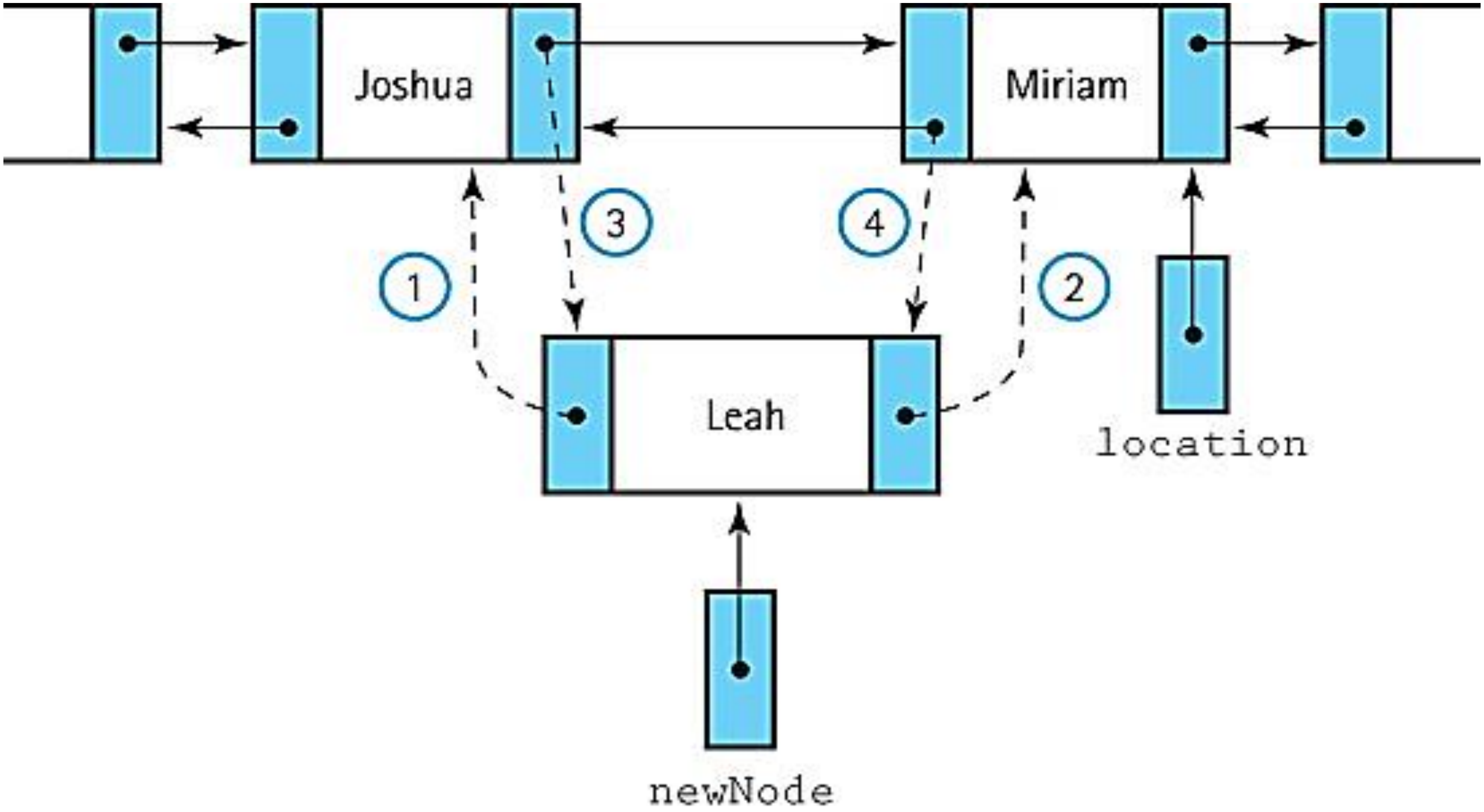
```
public class DLLNode<T> extends
LLNode<T>
{
  private DLLNode<T> back;

  public DLLNode(T info)
  {
    super(info);
    back = null;
  }
```

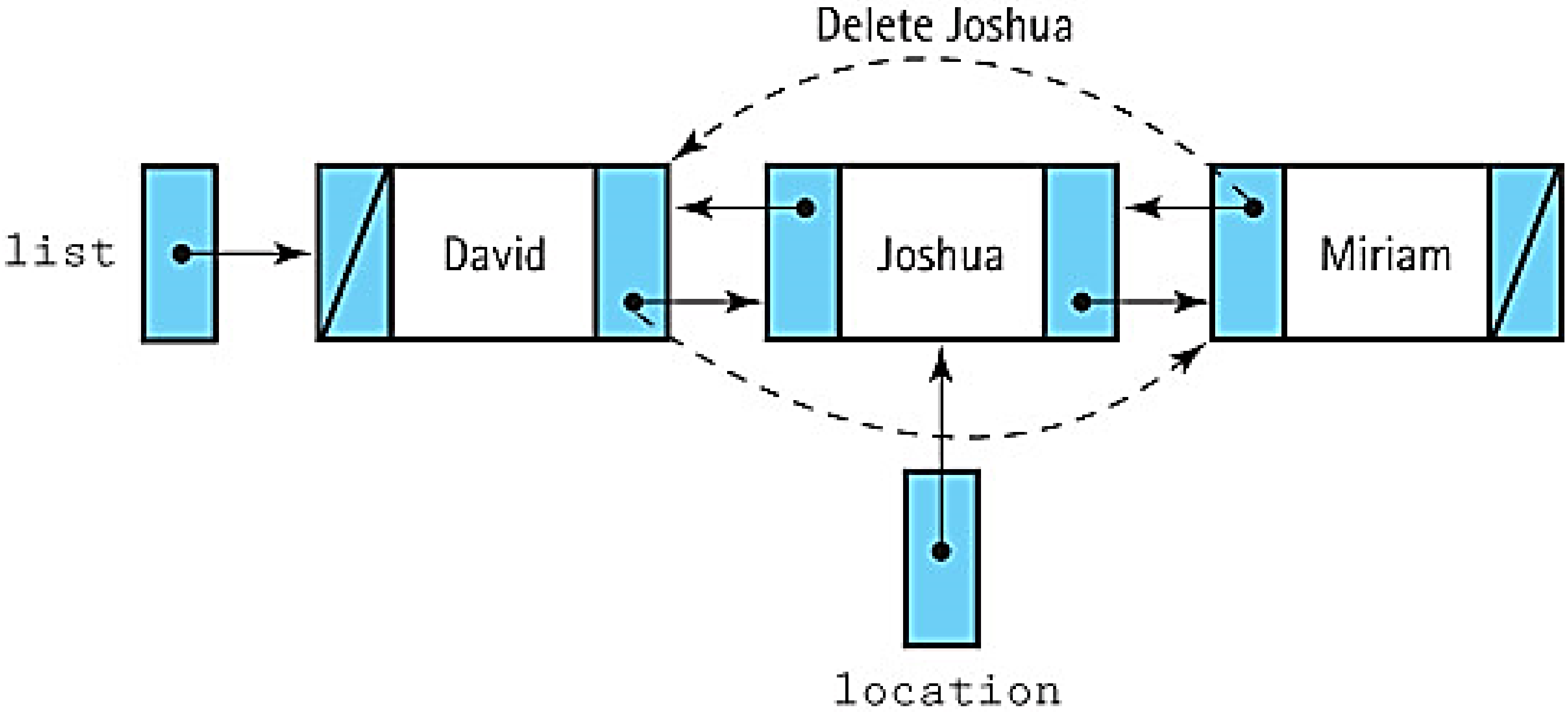
```
  public void setBack(DLLNode<T> back)
  // Sets back link of this
  // DLLNode.
  {
    this.back = back;
  }

  public DLLNode<T> getBack()
  // Returns back link of this
  // DLLNode.
  {
    return back;
  }
}
```

The `add` operation

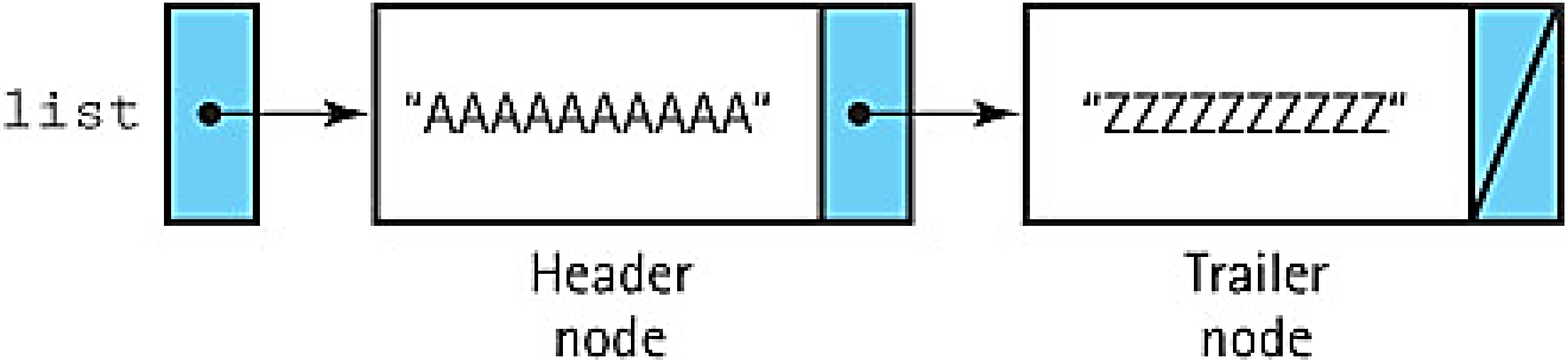


The `remove` operation

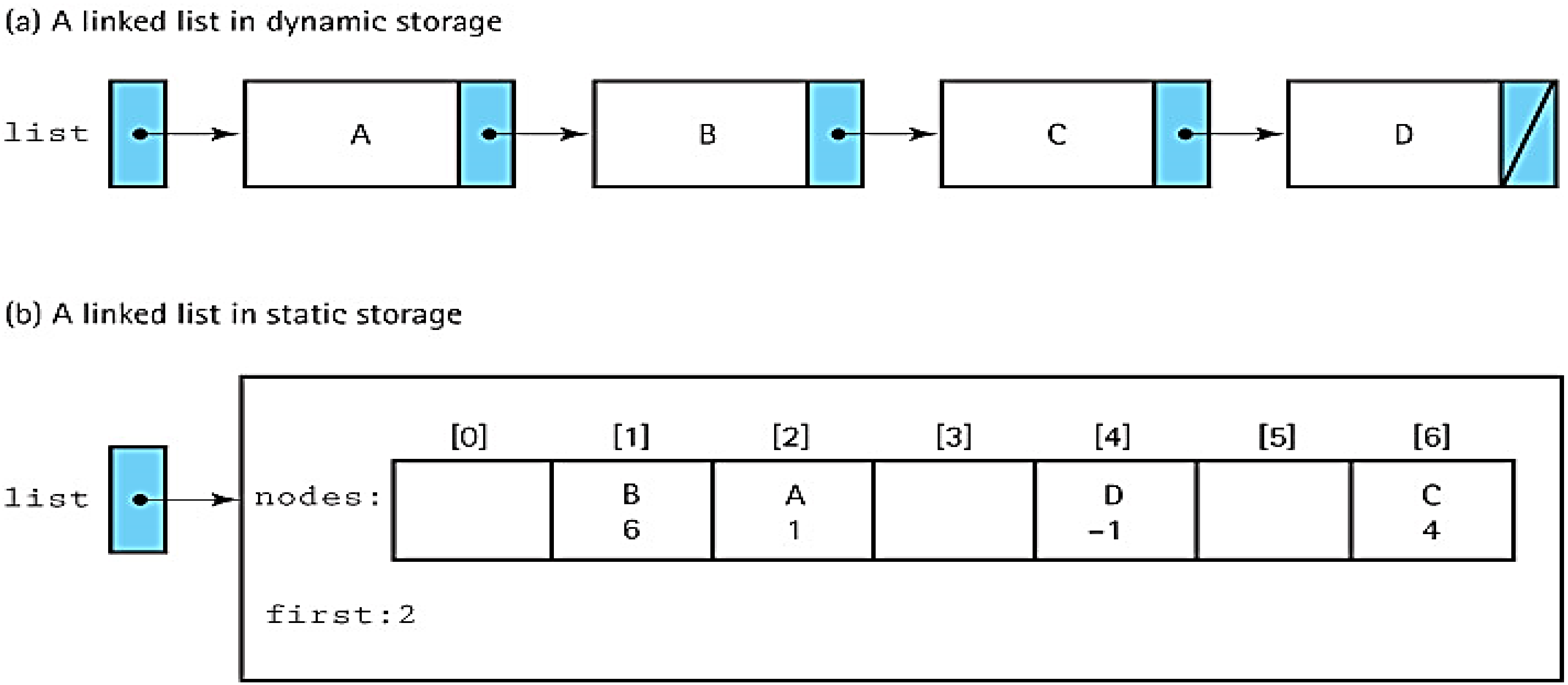


Linked lists with headers and trailers

- **Header node** - A placeholder node at the beginning of a list; used to simplify list processing.
- **Trailer node** - A placeholder node at the end of a list; used to simplify list processing.



A linked list as an array of nodes



(a) A linked list in dynamic storage

(b) A linked list in static storage

[0]	[1]	[2]	[3]	[4]	[5]	[6]
	B 6	A 1		D -1		C 4

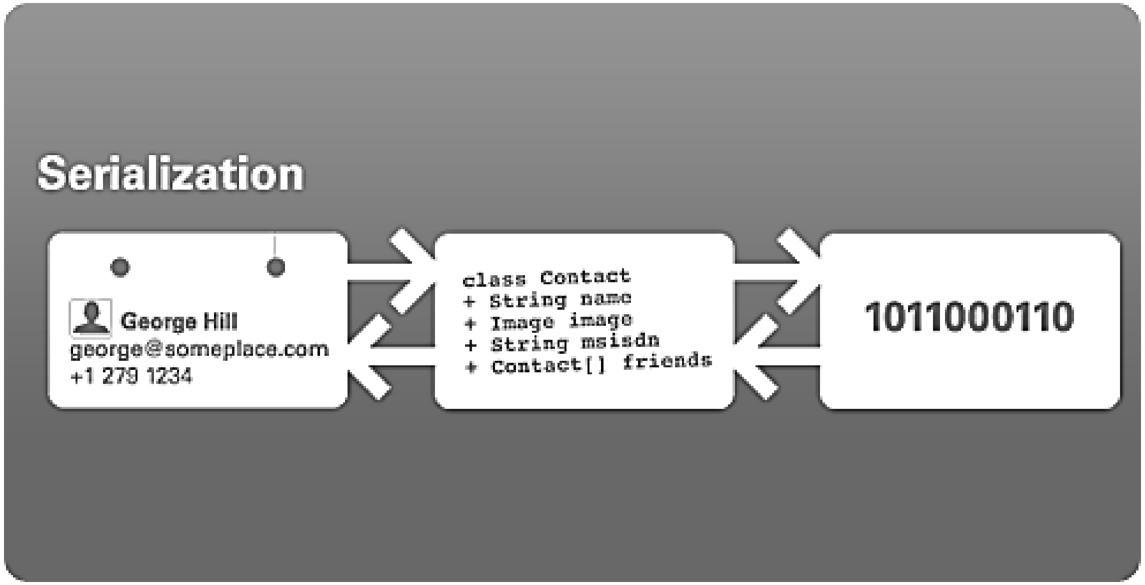
- Why do we use the array-based implementation?
  - There are programming languages that do not support dynamic allocation or reference types.
  - There are times when dynamic allocation of each node, one at a time, is too costly in terms of time.

Storing objects and structures in files

- For bigger programs that involve data access, saving and loading data files are expected. In Java, you can use *serialization* to store objects in files with very little effort.
- Any information we need to save can be represented by its primitive parts.
- As a very simple example, consider a **`Student`** object which has three instance variables:
  - **`private String m_name;`**
  - **`private int m_year;`**
  - **`private double m_gpa;`**
- A **`student`** is not a "primitive" object, but when broken into its constituent parts its information consists of a **`String`**, an **`int`**, and a **`double`**. You can save all of them as **`string`**s in a text file. However, there exists an easier way.

Serialization of objects

- Java provides a mechanism, called *object serialization* where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.



Serialization of objects

- We can write objects using the **`writeObject`** method of the **`ObjectOutputStream`** class.
    - To set up the output of serialized objects to the file **`objects.dat`** using the stream variable **`out`**, we write

```
ObjectOutputStream out = new
ObjectOutputStream(new FileOutputStream("objects.dat"));
```

- We can read objects using the **`readObject`** method of the **`ObjectInputStream`** class.
    - To set up reading from the same file, but this time using the variable **`in`**, we code

```
ObjectInputStream in = new
ObjectInputStream(new FileInputStream("objects.dat"));
```

The `Serializable` interface

- Any class whose objects we plan to serialize must implement the **`Serializable`** interface.
- This interface has no methods!
- It marks a class as potentially being serialized for I/O, so that the Java runtime engine knows to convert references as needed on output or input of class instances.
- To make our objects serializable, we simply state that their class implements the interface.

Action items

- Read book chapter 7.