Chapter 7
More Lists

Chapter 7: More Lists

7.1 – Circular Linked Lists

7.2 – Doubly Linked Lists

7.3 – Linked Lists with Headers and Trailers

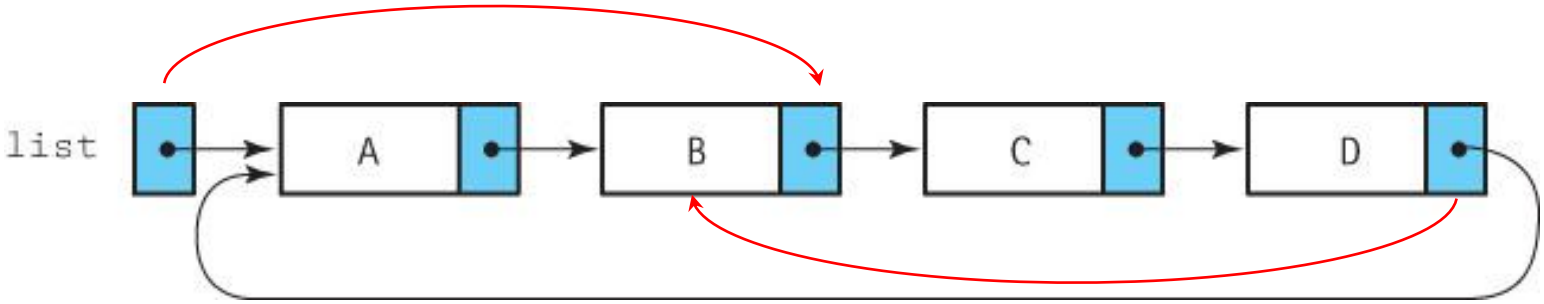
7.4 – A Linked List as an Array of Nodes

7.5 – A Specialized List ADT

7.6 – Case Study: Large Integers

7.1 Circular Linked Lists

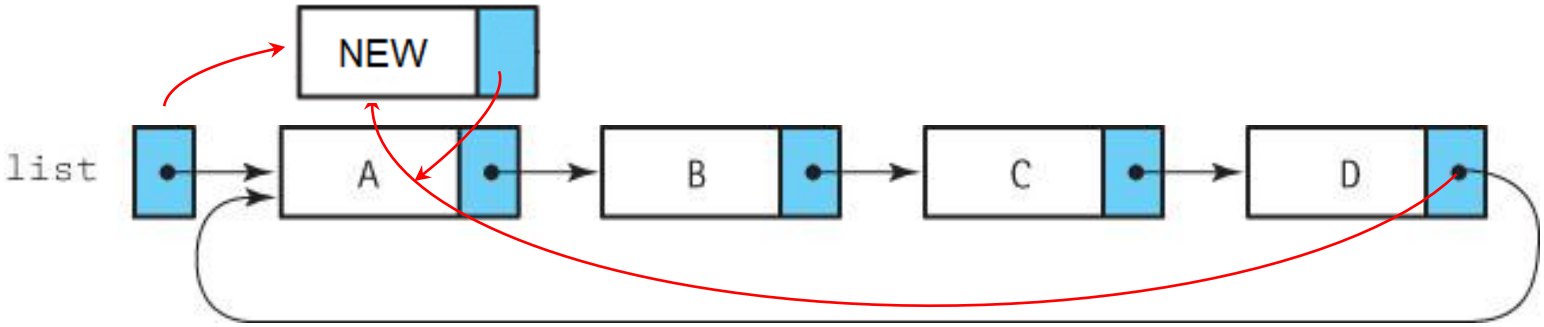
- **Circular linked list**
  - A list in which every node has a successor;
  - The “last” element is succeeded by the “first” element



What does removing an element require?

7.1 Circular Linked Lists

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



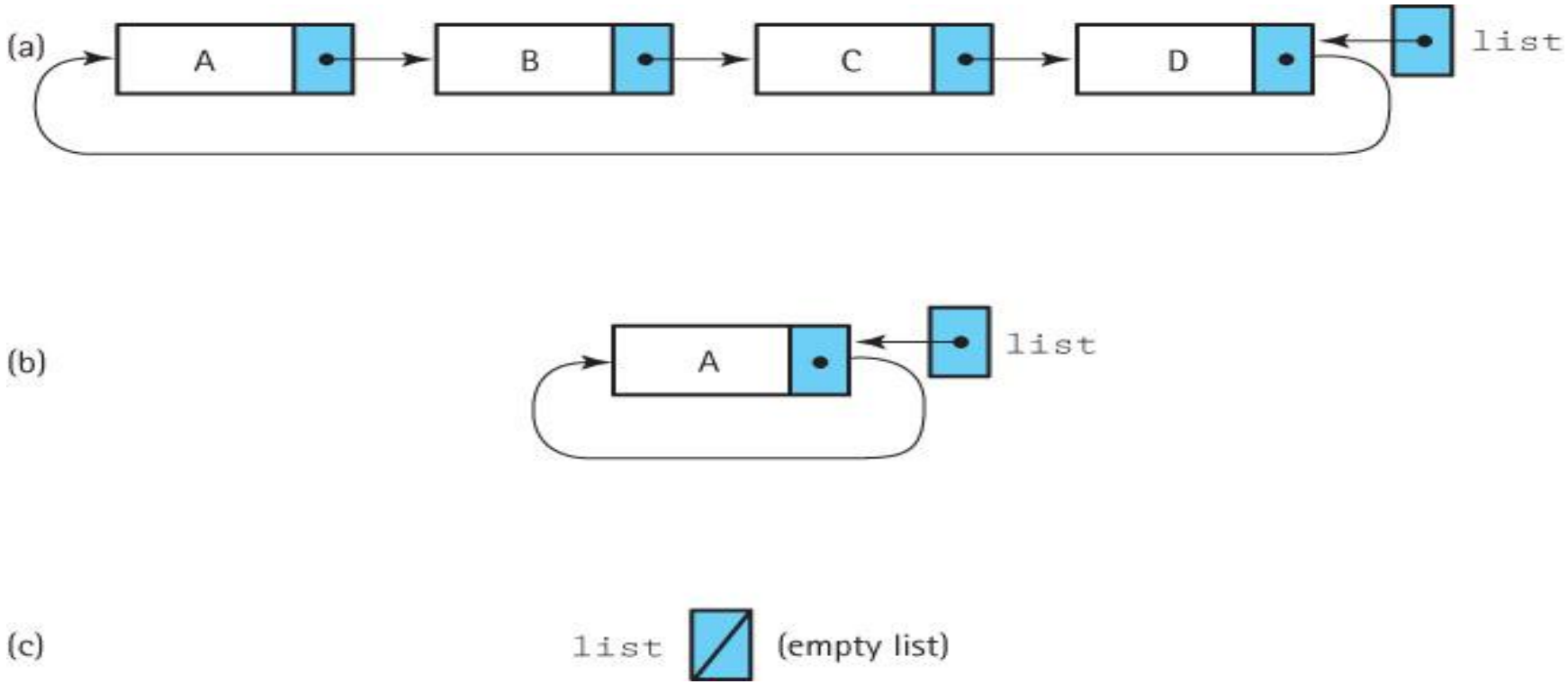
What does adding a new element require?

Adding and removing elements at the front of a list is not efficient!
Why?

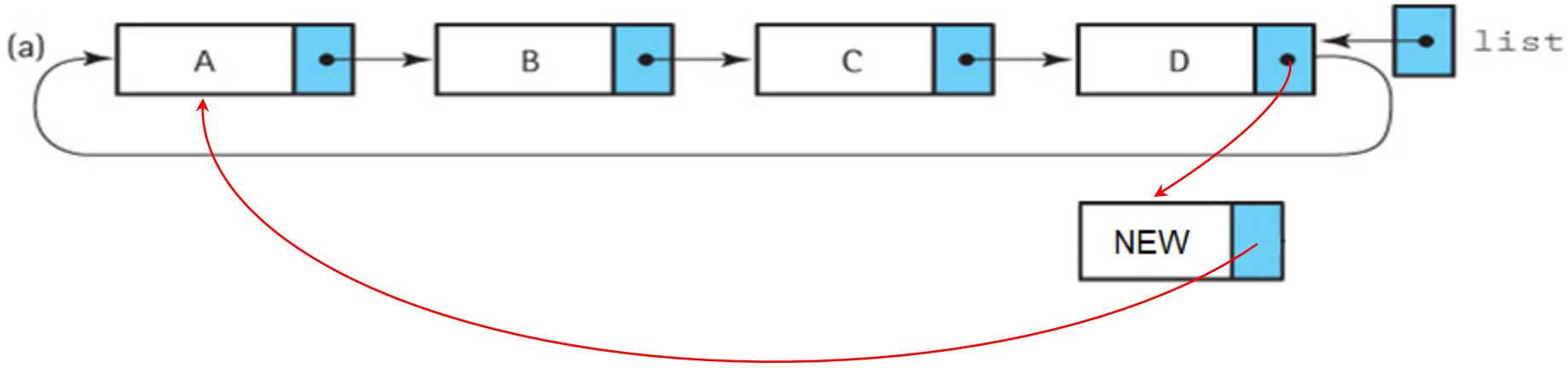
We are required to traverse the list to make the last element point to the first…

A more efficient approach

- Setting the list reference to the last node solves the traversal problem
  - now there is easy access to both the first and the last elements in the list

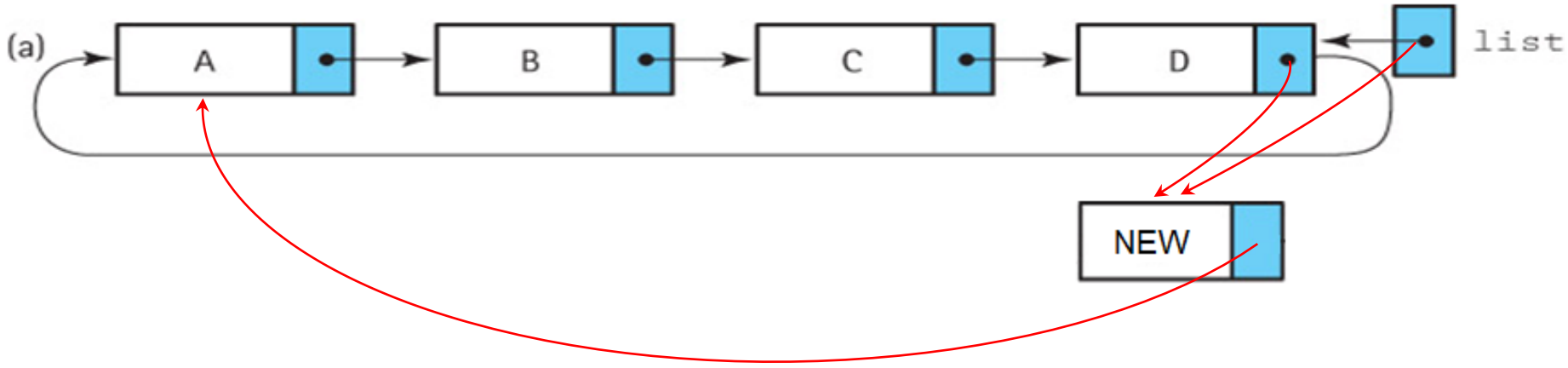


Addition at the front



No traversal at all!

Addition at the rear



One step!

Review CH6 RefUnsortedList.java

```
int size();                    // Returns the number of elements on this list.
void add(T element);           // Adds element to this list.
boolean remove (T element); // Removes an element e from this list and returns true;
                               // if no such element exists, returns false.
boolean contains (T element); // Returns true if this list contains an element e,
                               // such that e.equals(element); otherwise, returns false.
T get(T element); // Returns an element e from this list such that e.equals(element);
                  // if no such element exists, returns null.
String toString(); // Returns a formatted string that represents this list.
void reset();      // Initializes current position for an iteration through this list,
                   // to the first element on this list.
T getNext();       // Preconditions:   The list is not empty
                   //                  The list has been reset
                   //                  The list has not been modified since the most recent reset
                   // Returns the element at the current position on this list.
                   // If the current position is the last element, then advance the value
                   // of the current position to the first element; otherwise, advance
                   // the value of the current position to the next element.
```

See CH6 Code.

The `CRefUnsortedList` Class

- `CRefUnsortedList` vs. `RefUnsortedList`.

circular vs. linear

- Reuse the `size` method
- Reuse the `contains` method*
- Reuse the `get` method*
- Modify the `find` helper method*
  - should function as the `find` in `RefUnsortedList`
  - Reuse both the `contains` and `get` methods.
- Modify the `toString` method
- Modify the `reset` method
- Reuse the `getNext` method

The `find` method

```
protected void find(T target)
{
  location = list;
  found = false;

  if (list != null)
    do
    {
      // move search to the next node
      previous = location;
      location = location.getLink();

      // check for a match
      if (location.getInfo().equals(target))
         found = true;
    }
    while ((location != list) && !found);
}
```

The Iterator methods

- the `reset` method becomes more complicated and the `getNext` method becomes simpler

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

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

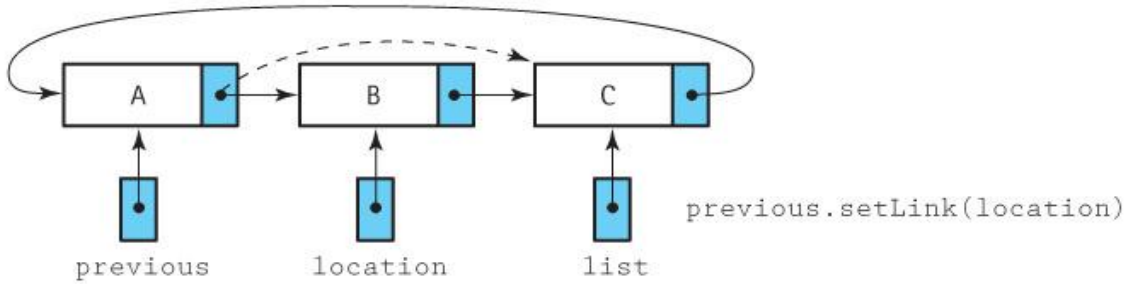
```
// Previous linked list from CH6
public void reset()
{
   currentPos = list
}

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

The `toString` method

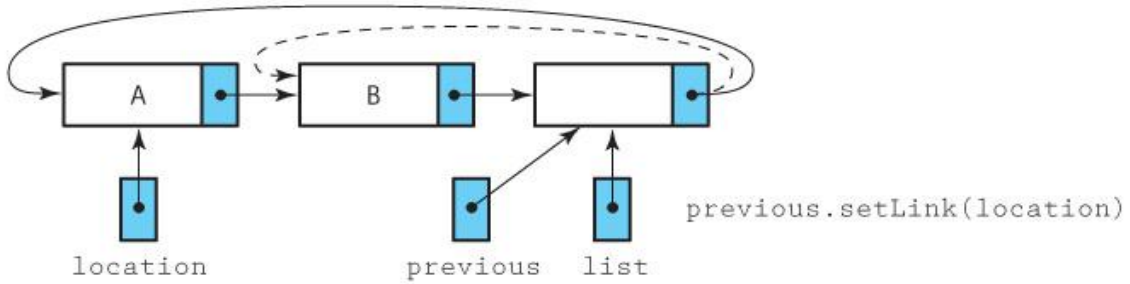
```
// Circular linked list
public String toString()
{

  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

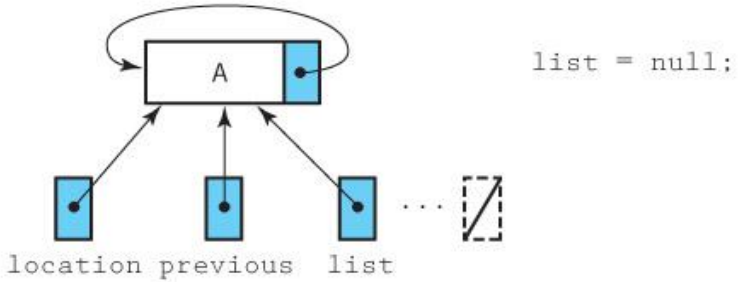
(a) The general case (remove B)



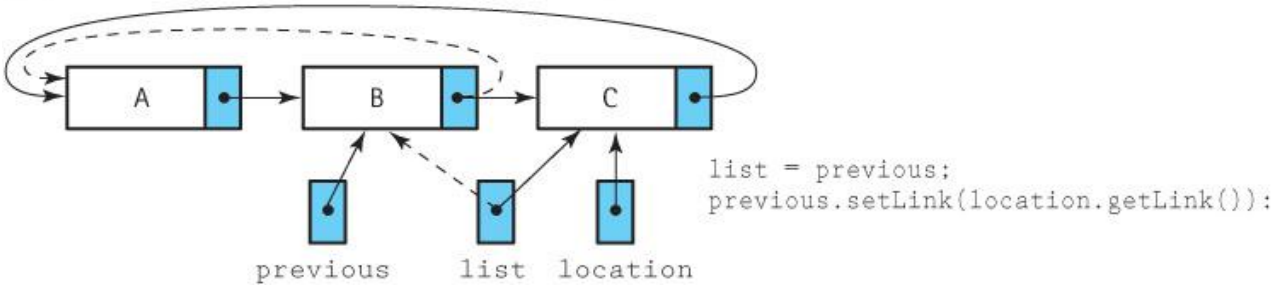
(b) Special case (?): Removing the first element (remove A)



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



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

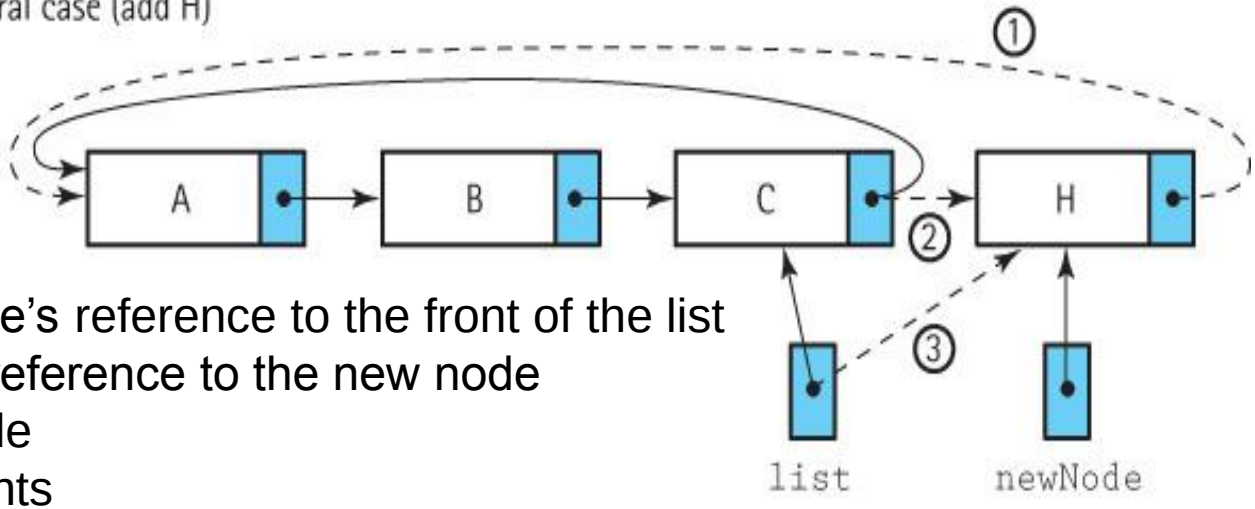


The `remove` method

```
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;
}
```

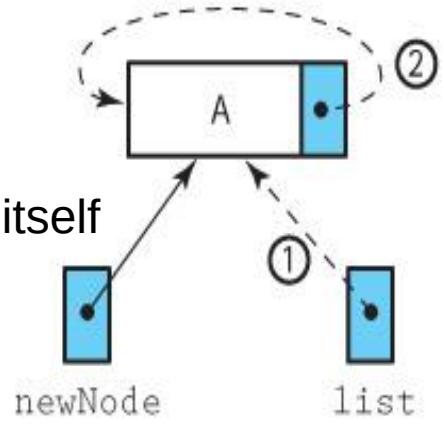
Adding a node

(a) The general case (add H)



1. Create newNode
   and point the newNode's reference to the front of the list
2. Point the rear node's reference to the new node
3. Set list to the new node
4. Increment numElements

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



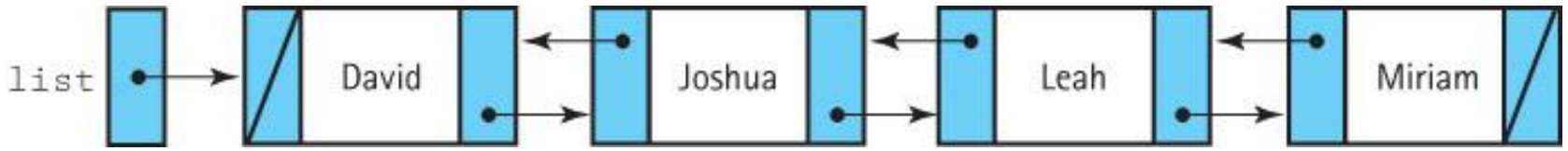
1. Create newNode
   and set list to the new node
2. Point the newNode's reference to itself
3. Increment numElements

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++;
}
```

7.2 Doubly Linked Lists

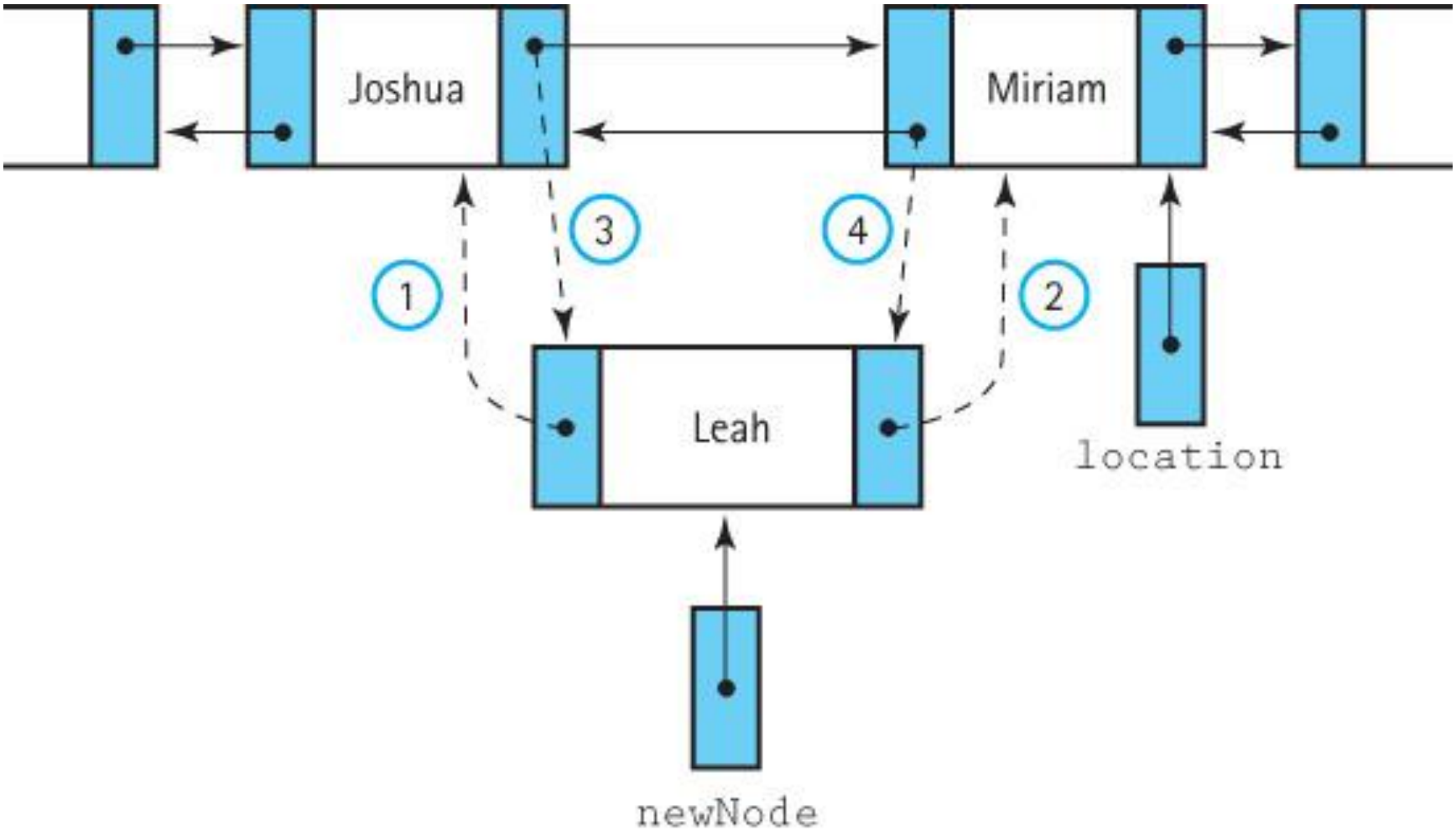
- A linked list in which each node is linked to both its successor and its predecessor



DLLNode Class

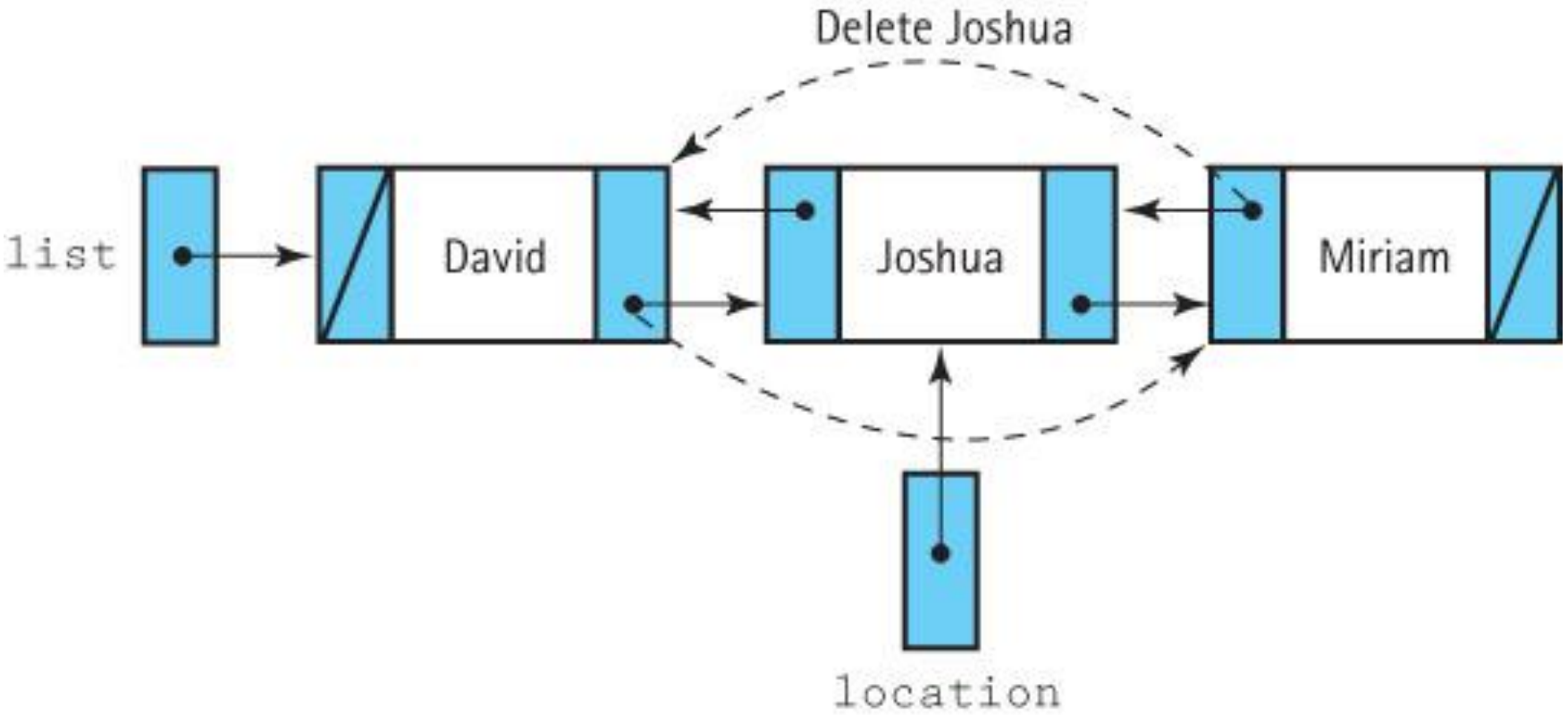
```
package support;

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

  public DLLNode(T info)
  {
    super(info); // calls constructor of the superclass
    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



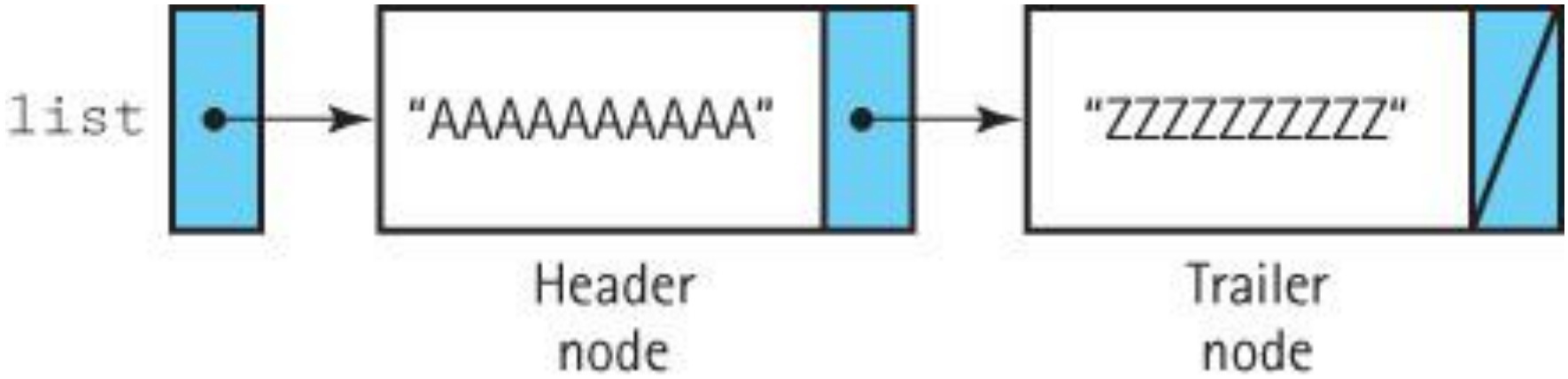
What happens if the steps are taken out of order?
What is the rule of thumb for all LLs?

The remove operation



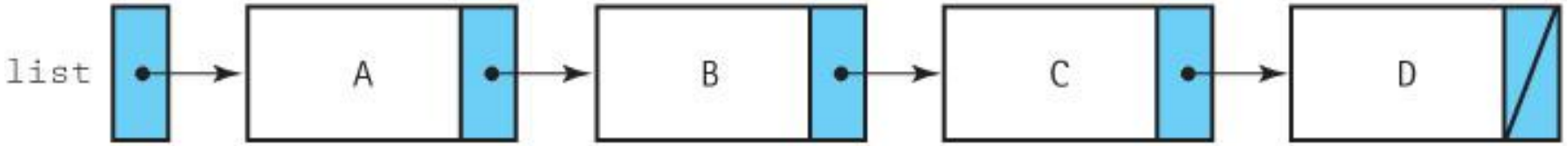
7.3 Linked Lists with Headers and Trailers

- **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

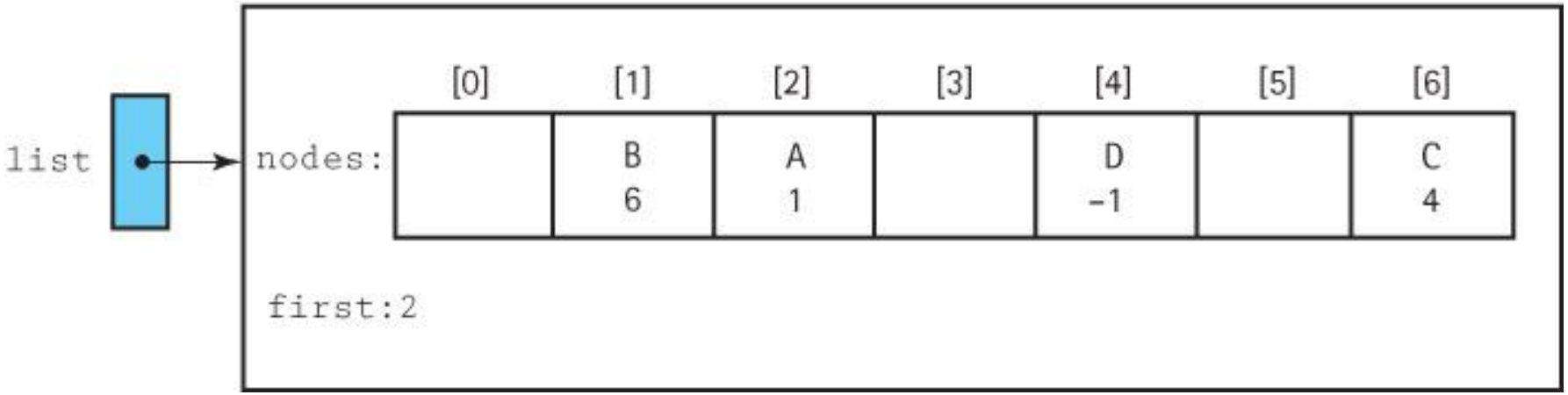


7.4 A Linked List as an Array of Nodes

(a) A linked list in dynamic storage



(b) A linked list in static storage



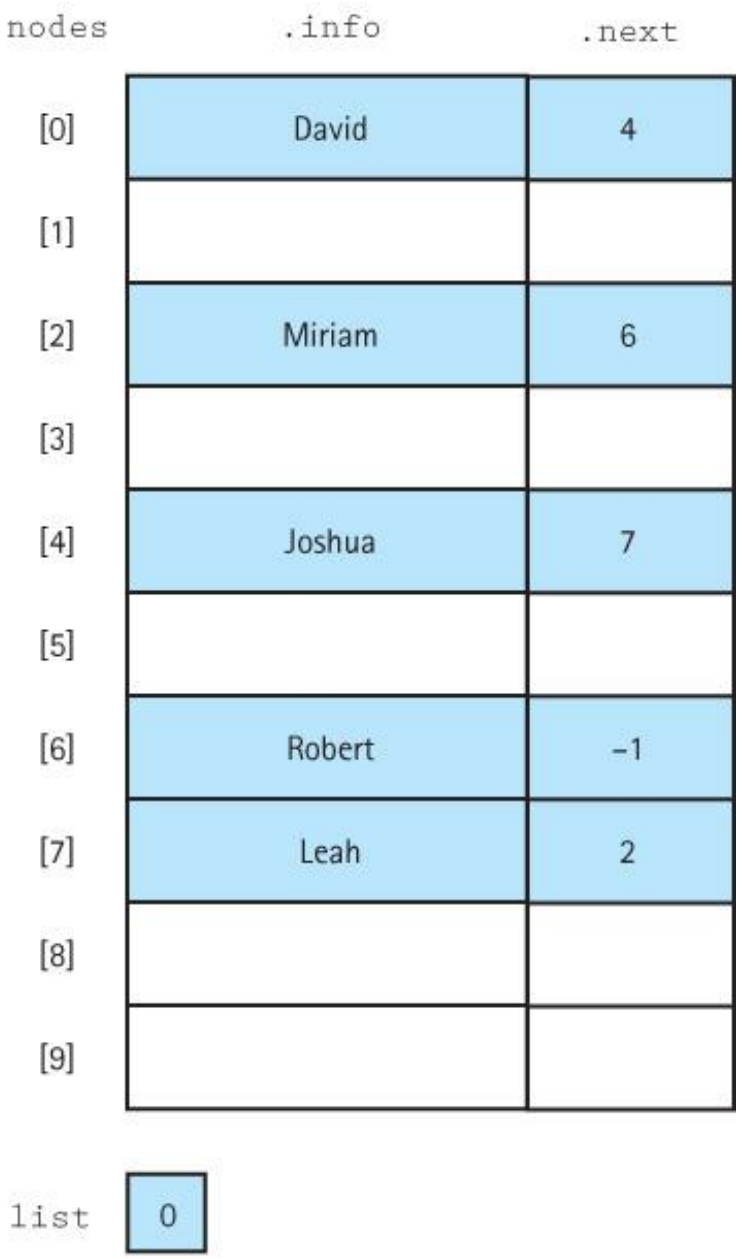
Why Use an Array?

- Flexibility: we can manage the freespace
- Some languages do not support dynamic allocation or reference types
  - C, Fortran77 (most non-OO languages)
  - Surprise! Mac OS X Java Applets (??blog)
- Dynamic allocation of each node, one at a time, can be too costly in terms of time

Boundedness

- static allocation
  - primary motivation for the array-based linked approach
- Our lists will not grow as needed in this section
- Applications should not add elements to a full list
- What method will then be required in addition to all the other standard list operations?
  - The `isFull` operation

A sorted list



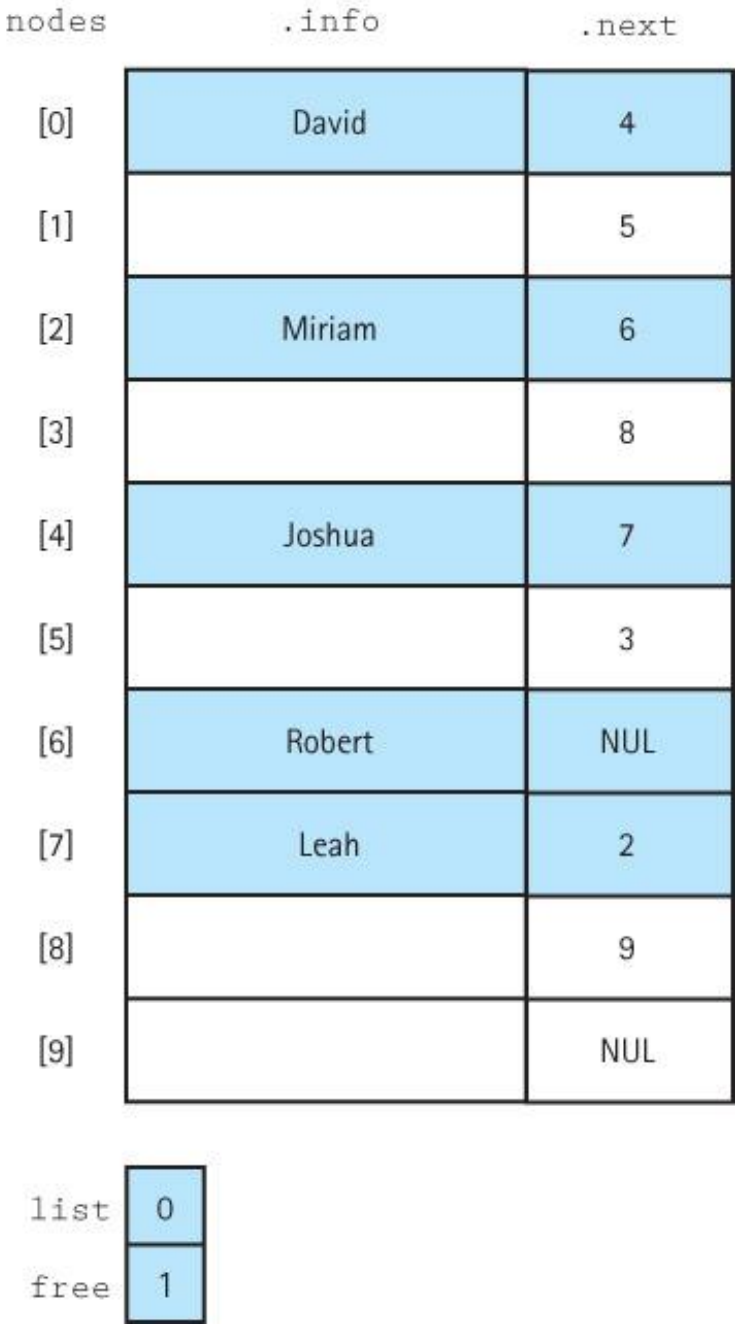
Implementation Issues

- The end of the list is marked with a “null” value
  - This “null” value must be an invalid address for a list element
  - we use the value –1

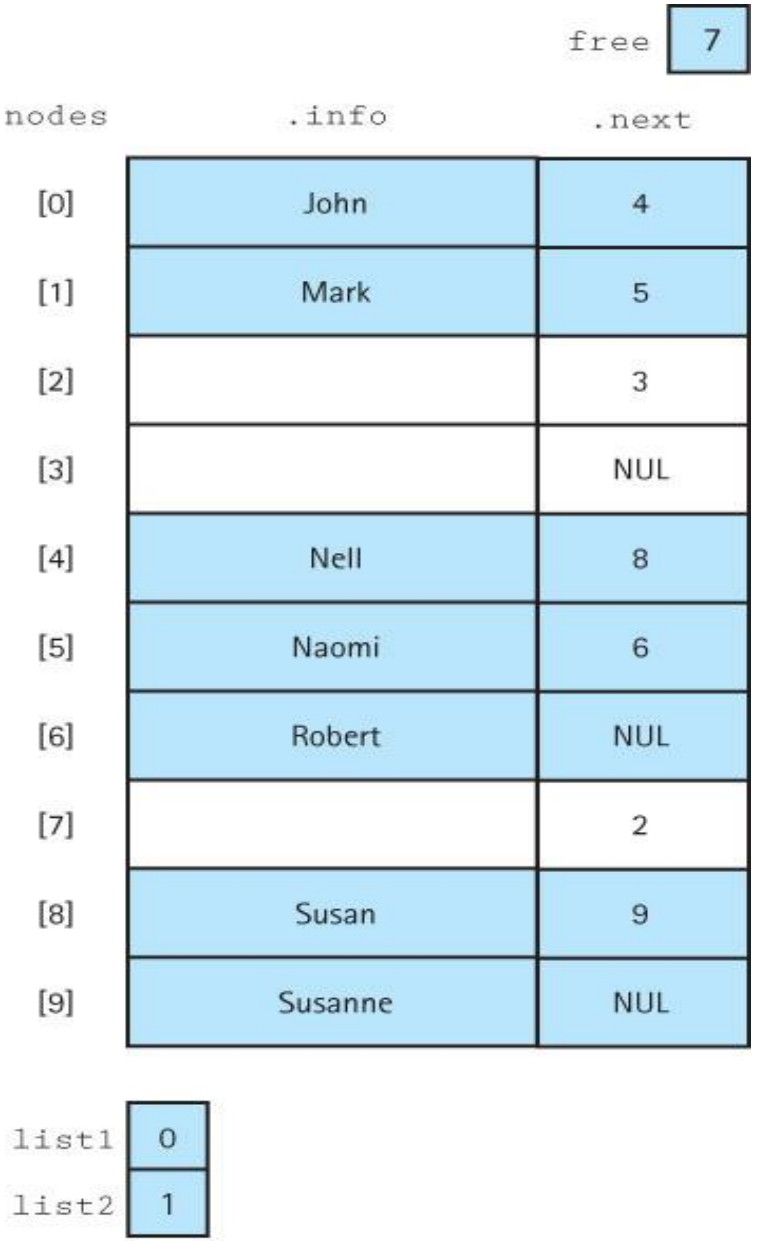
```
private static final int NUL = -1;
```

- Directly manage the free space for new list elements.
  - Link the collection of unused array elements together into a linked list of free nodes.
  - Write a method to allocate nodes from the free space (`getNode`)
    - Use `getNode` when adding new elements onto the list
  - Write a method (`freeNode`) to de-allocate a node
    - Put the node back into the pool of free space

A linked list and free space



More than one list



ADT Comparison: List

Abstract	Array-Based	Reference-Based	Array Index Lists
node at location	N/A	location	nodes[location]
element at location	list[location]	location.getInfo()	nodex[location].info
next location	list[location+1]	location.getLink()	nodes[location].next
allocate a node	N/A	new	getNode()
free a node	N/A	remove links, garbage collector	freeNode(node)