Chapter 6A More Lists

CS401

Michael Y. Choi, Ph.D.

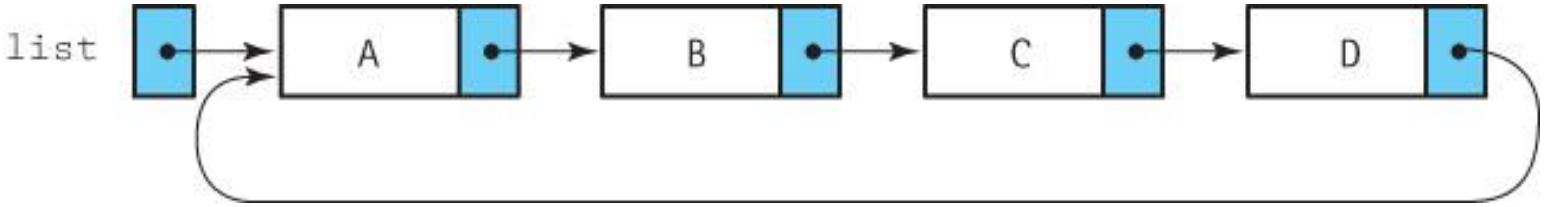
Department of Computer Science

Illinois Institute of Technology

Revised Nell Dale Presentation

Circular Linked Lists

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

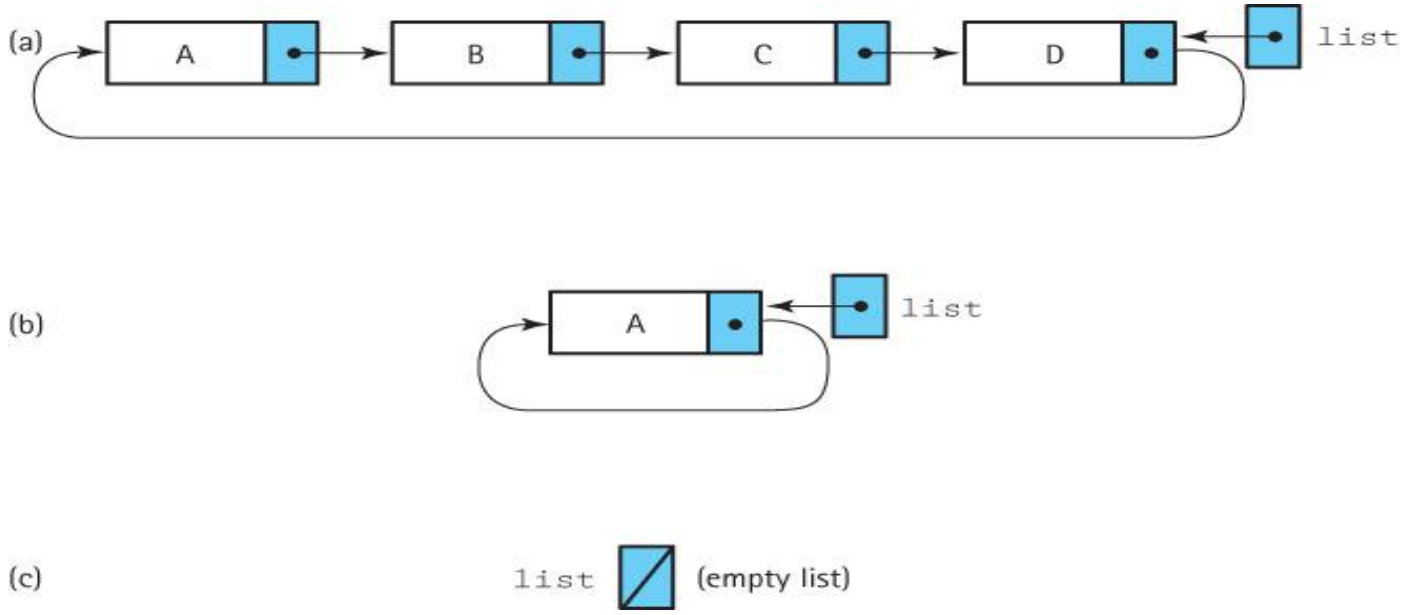


A more efficient approach

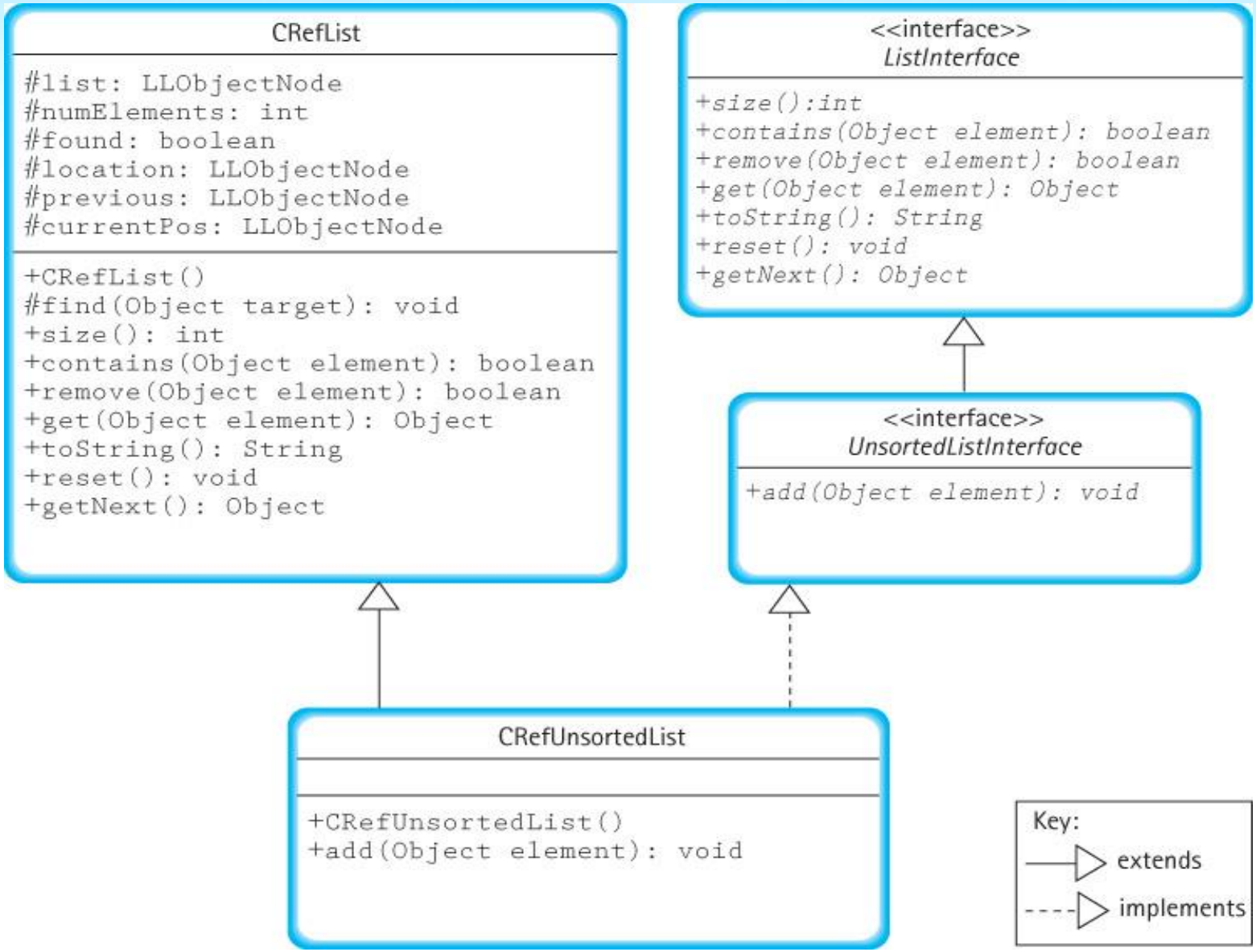
- Adding and removing elements at the front of a list might be a common operation for some applications.
- Our linear linked list approach supports these operations very efficiently

Why?

- The previous slide circular linked list approach does not (we need to access the last element also).
- 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.



An Unsorted Circular Linked List



The `CRefList` Class

- The `CRefList` class can contain the same set of methods (`size`, `contains`, `remove`, `get`, `toString`, `reset`, and `getNext`) as its linear list counterpart `RefList`.
- The `size` method need not be changed.
- If we provide a revised `find` helper method with functionality that corresponds to the `find` in `RefList`, we can also reuse both the `contains` and `get` methods.

The Iterator methods

- the `reset` method becom ated and the `getNext` method .

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

public Object getNext()
{
  Object next = currentPos.getInfo();
  currentPos = currentPos.getLink();

  return next;
}
```

```
// Linear list

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

```
// Linear list

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

The `toString` method

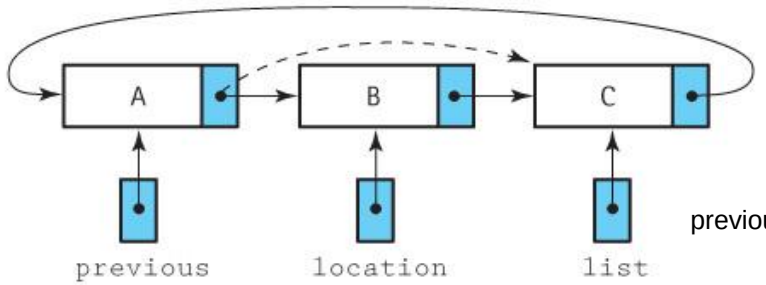
```
public String toString()
// Returns a nicely formatted String that represents this list.
{
  String listString = "List:\n";
  if (list != null)
  {
    LLObjectNode prevNode = list;
    do
    {
      listString = listString + "  " + prevNode.getLink().getInfo() + "\n";
      prevNode = prevNode.getLink();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

The `find` method

```
protected void find(Object target)
{
  boolean moreToSearch;
  location = list;
  found = false;

  moreToSearch = (location != null);
  while (moreToSearch && !found)
  {
    // move search to the next node
    previous = location;
    location = location.getLink();

    // check for a match
    if (location.getInfo().equals(target))
      found = true;

    moreToSearch = (location != list);
  }
}
```

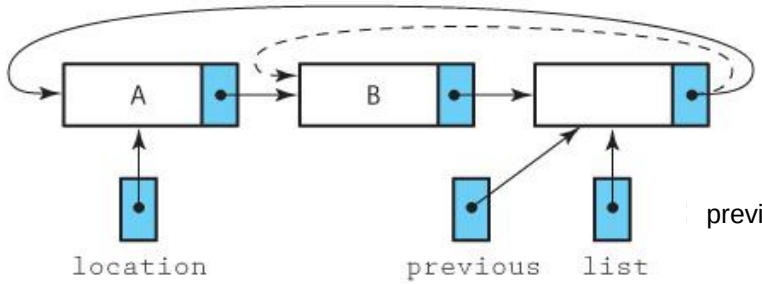
The `remove` method

(a) The general case (remove B)



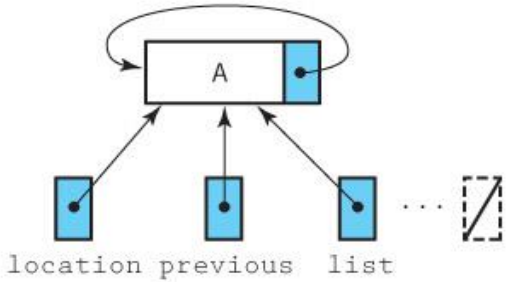

```
previous.setLink(location.getLink());
```

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



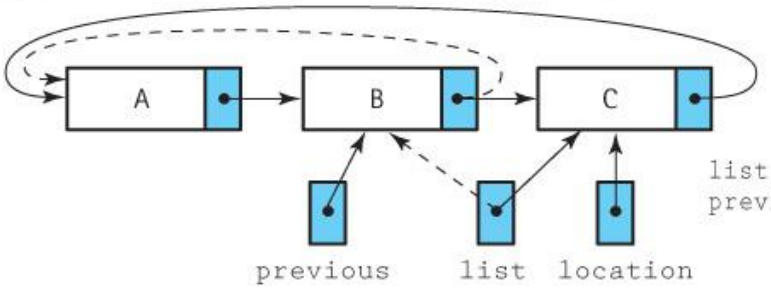

```
previous.setLink(location.getLink());
```

(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());
```

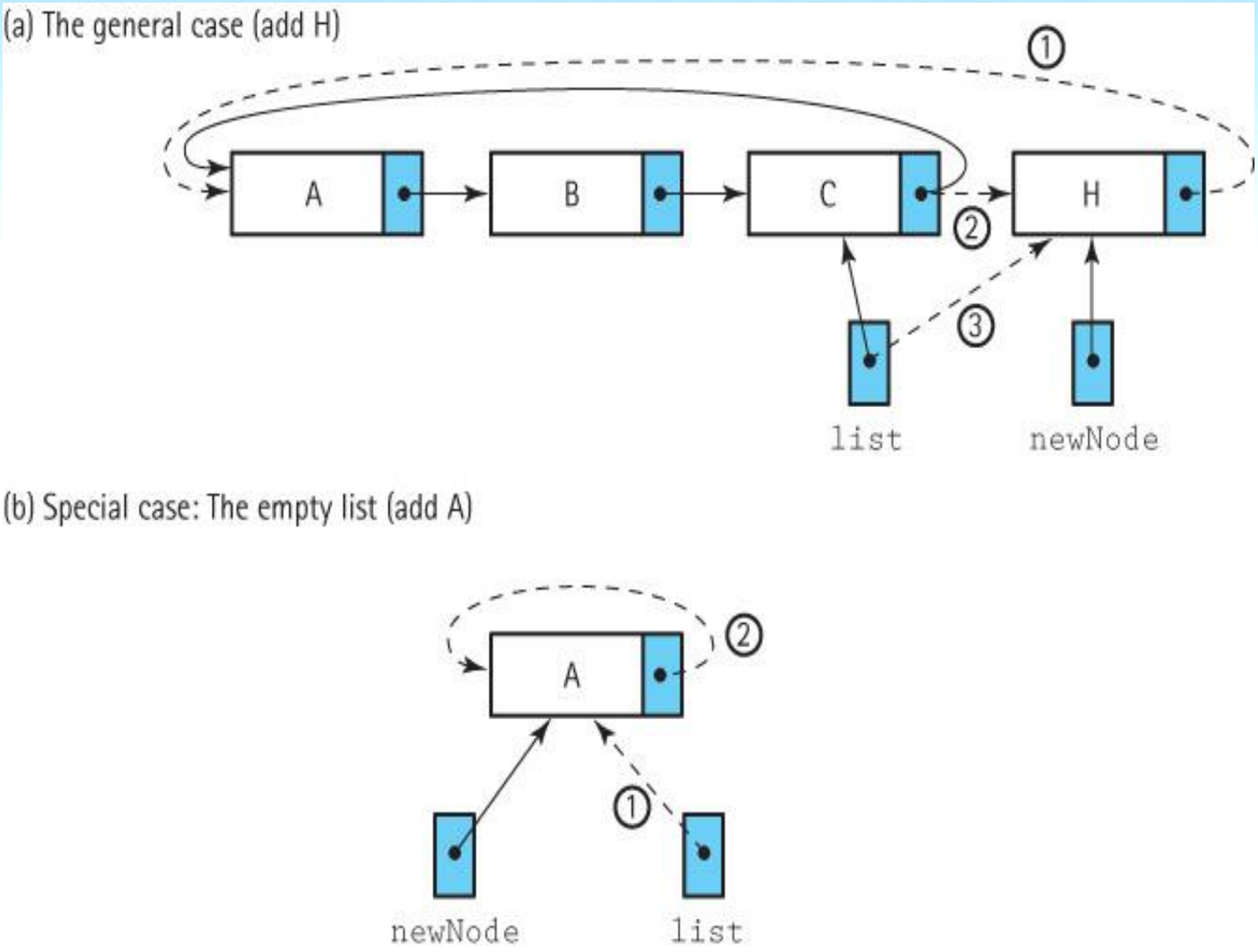
The `remove` method

```
public boolean remove (Object 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 `CRefUnsortedList` Class

- To create the `CRefUnsortedList` class and complete our implementation of the unsorted circular list we just need to extend the `CRefList` class with an `add` method.
- To implement the add method:
  - create the new node using the `LLObjectNode` constructor
  - if adding to an empty list set the list variable to reference the new element and link the new element to itself
  - otherwise, add the element to the list in the most convenient place – at the location referenced by `list`
  - increment `numElements`.

Adding a node

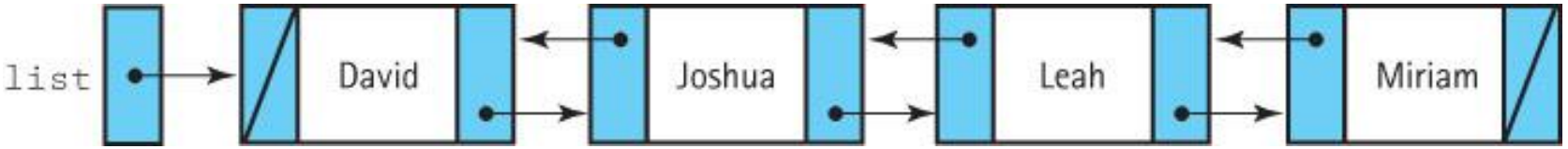


The `add` method

```
public void add(Object element)
// Adds element to this list.
{
  LLObjectNode newNode = new LLObjectNode(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

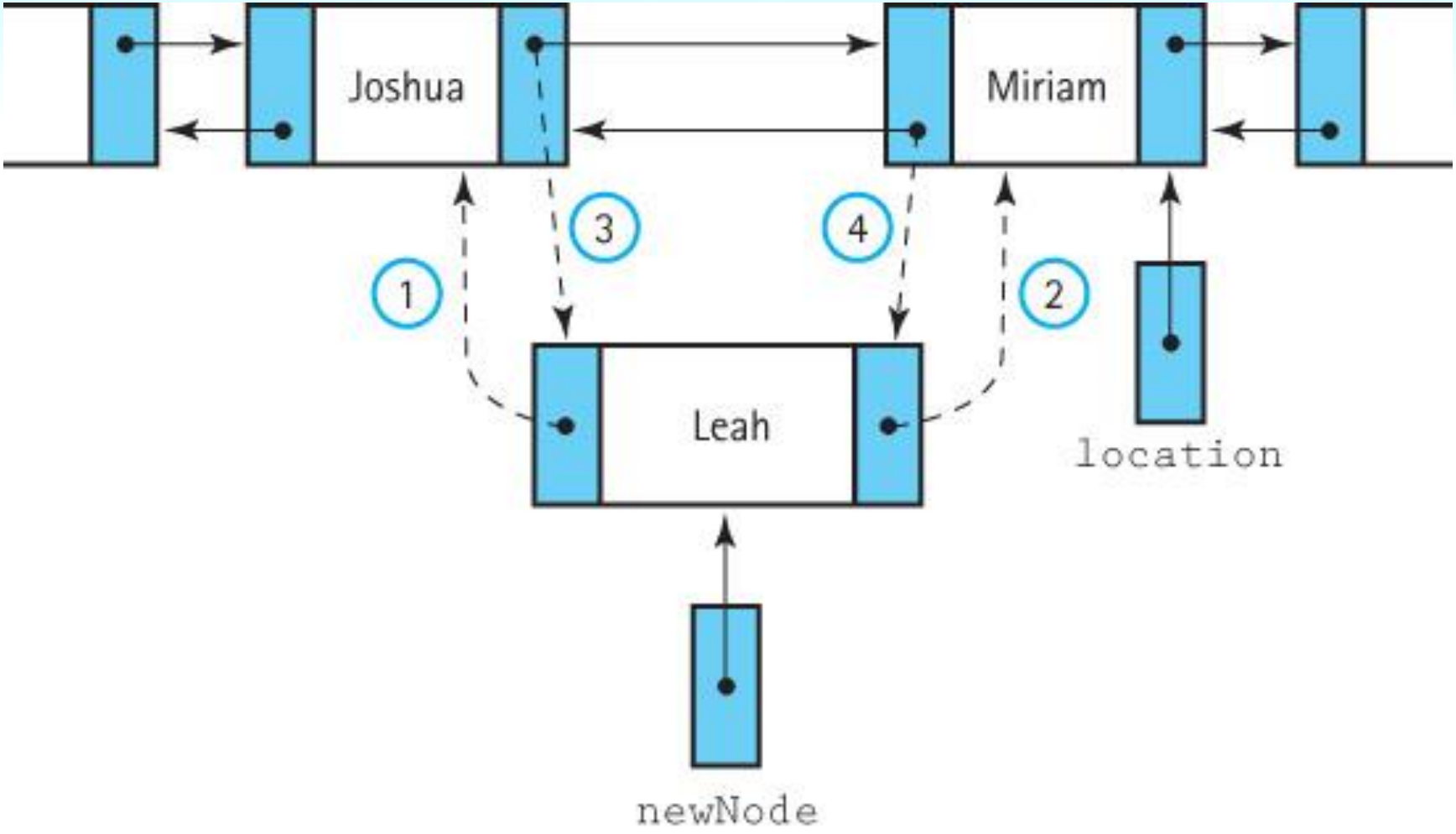
- **Doubly linked list** A linked list in which each node is linked to both its successor and its predecessor



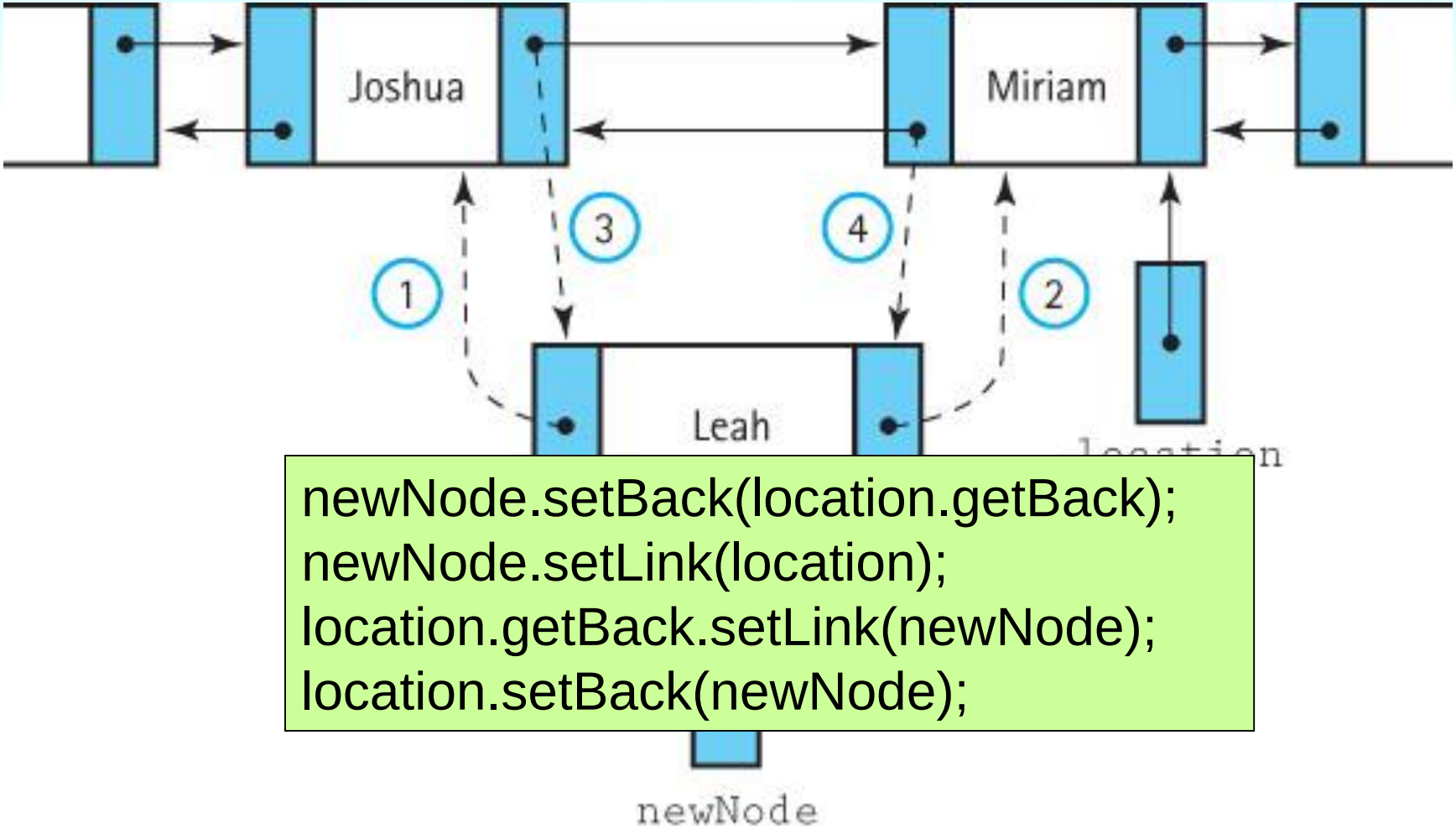
DLLObjectNode Class

```
package support;

public class DLLObjectNode extends LLObjectNode
{
  private DLLObjectNode back;

  public DLLObjectNode(Object info)
  {
    super(info);
    back = null;
  }

  public void setBack(DLLObjectNode back)
  // Sets back link of this DLLObjectNode.
  {
    this.back = back;
  }

  public DLLObjectNode getBack()
  // Returns back link of this DLLObjectNode.
  {
    return back;
  }
}
```

The add operation

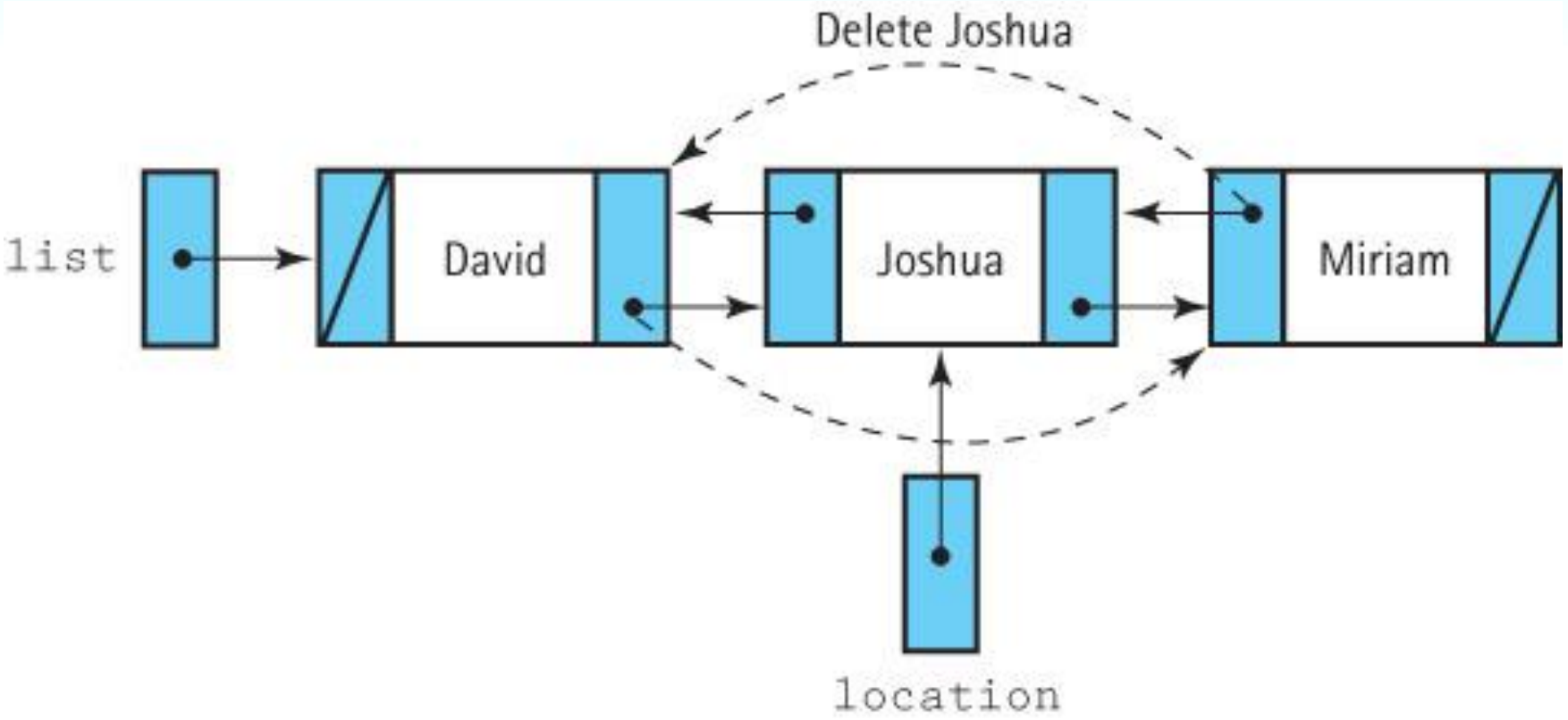


The add operation



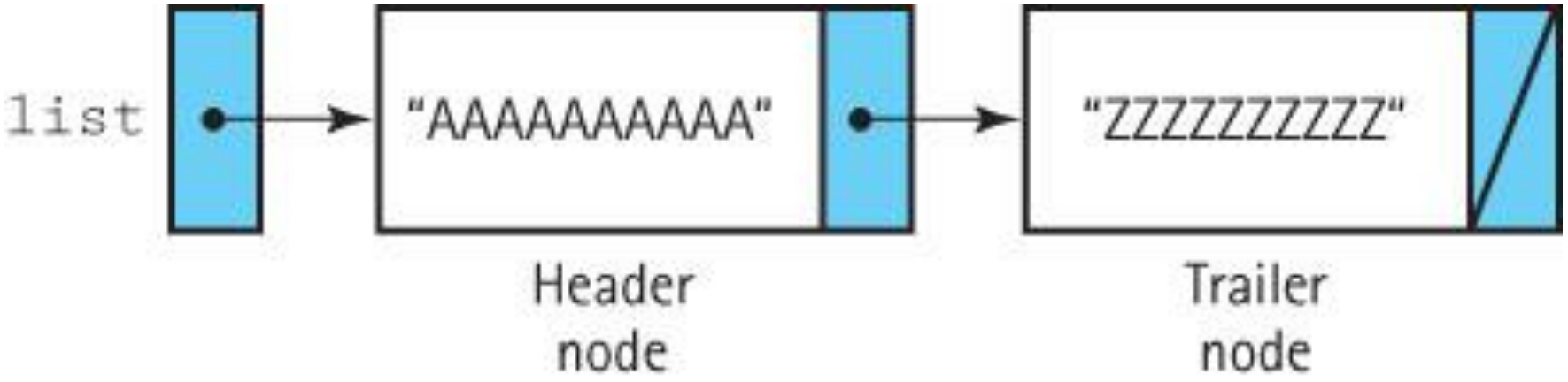

```
newNode.setBack(location.getBack);
newNode.setLink(location);
location.getBack.setLink(newNode);
location.setBack(newNode);
```

The remove operation



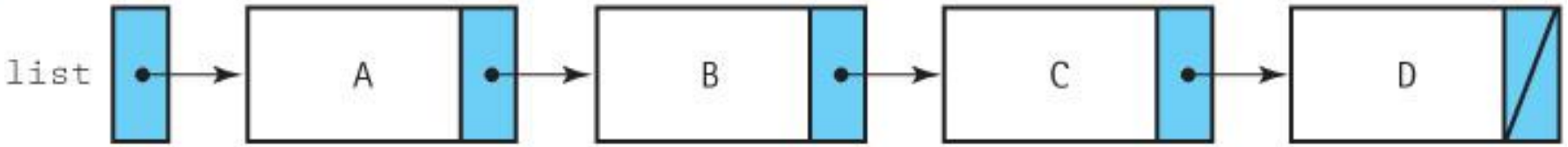
Linked Lists with Headers and Trailers

- **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

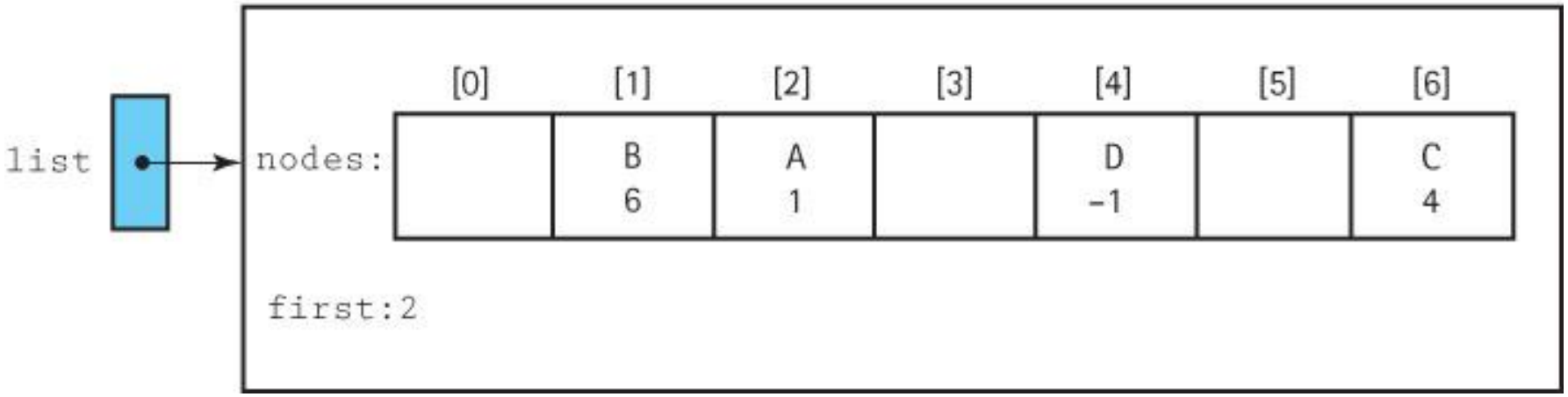


A Linked List as an Array of Nodes

(a) A linked list in dynamic storage



(b) A linked list in static storage



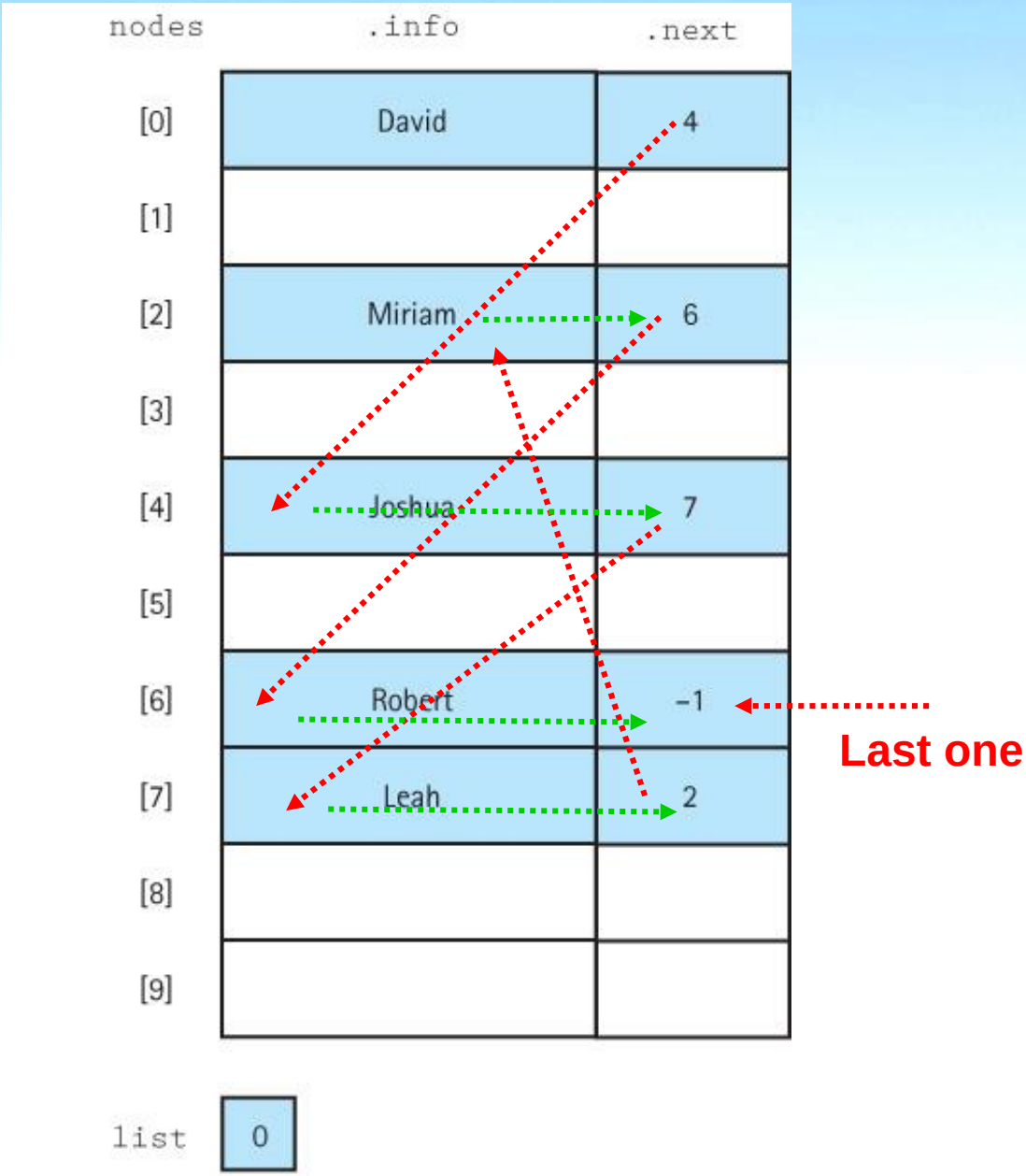
Why Use an Array?

- Sometimes managing the free space ourselves gives us greater flexibility
- 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

Boundedness

- A desire for static allocation is one of the primary motivations for the array-based linked approach
- We drop our assumption that our lists are of unlimited size in this section - our lists will not grow as needed.
- Applications should not add elements to a full list.
- Our list will export an `isFull` operation, in addition to all the other standard list operations

A sorted list



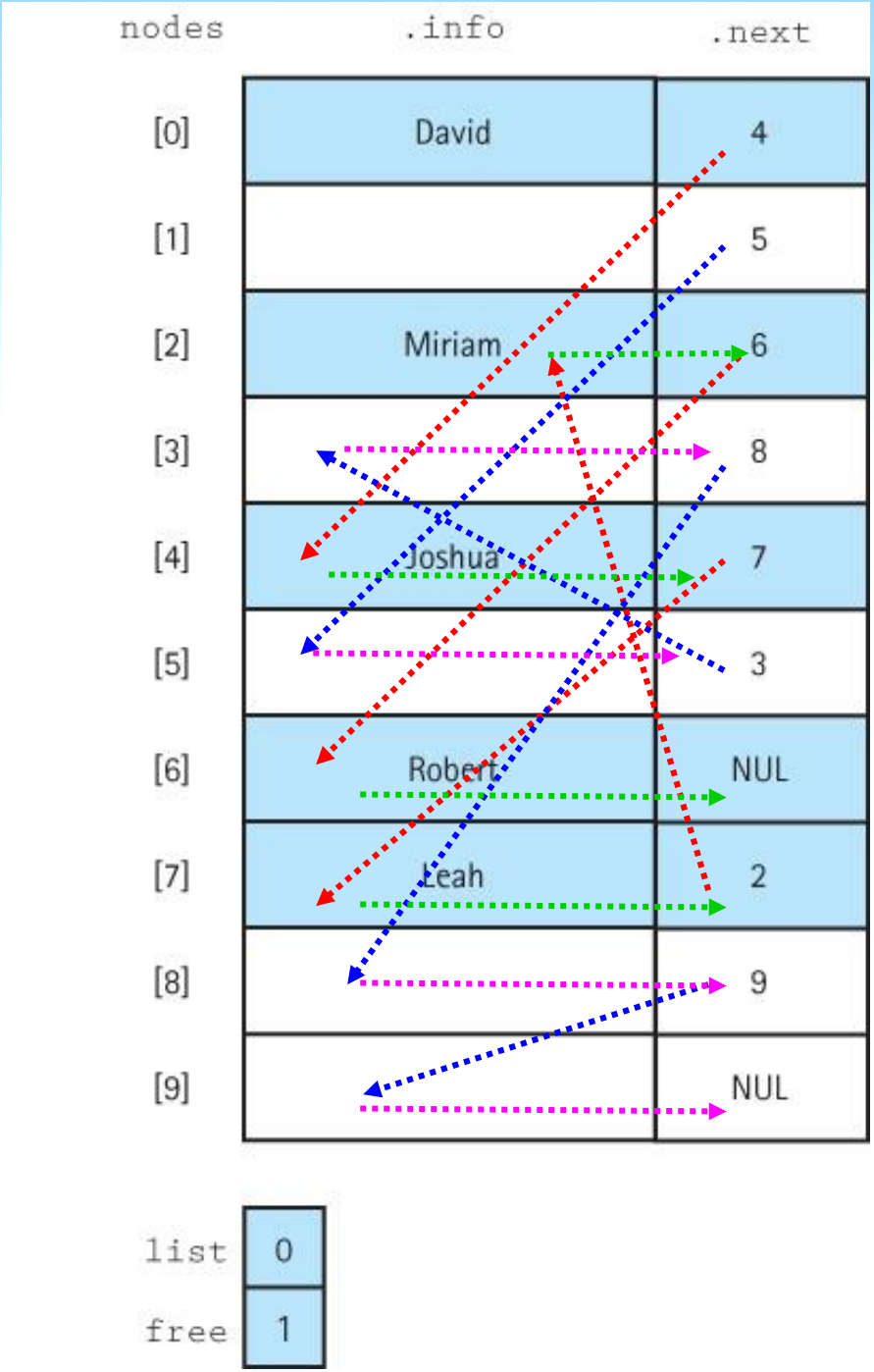
Implementation Issues

- We mark the end of the list with a “null” value
  - the “null” value must be an invalid address for a real list element
  - we use the value –1
  - we use the identifier NUL and define it to be -1

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

- We must directly manage the free space available for new list elements.
  - We link the collection of unused array elements together into a linked list of free nodes.
  - We write our own method to allocate nodes from the free space. We call this method `getNode`. We use `getNode` when we add new elements onto the list.
  - We write our own method, `freeNode`, to put a node back into the pool of free space when it is de-allocated.

A linked list and free space



More than one list