# LINKED LIST

Tran Thanh Tung

# Array review

- Arrays have some disadvantages
  - Insertion is slow in ordered arrays
  - Deletion is slow (ordered and unordered)
  - Size of the array can’t be changed after creation

# Introduction to linked list

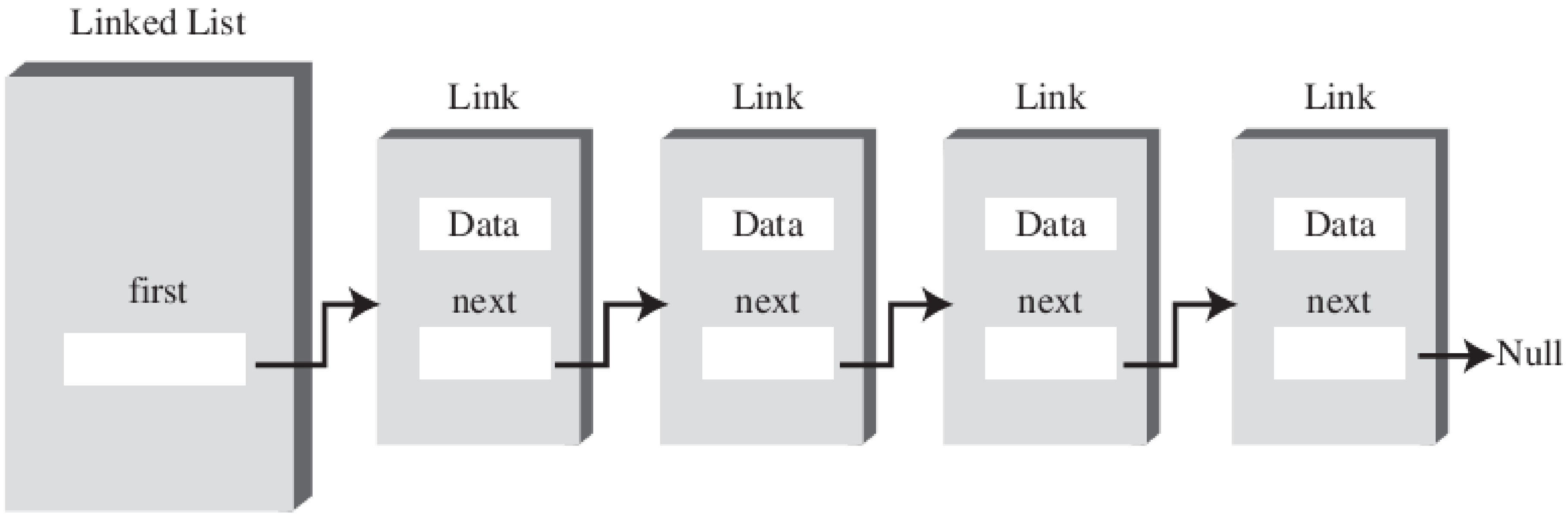
- Is the second widely used data structure
- Is suitable for many general-purpose databases
- Can replace an array in the implementation of Stack, Queue, etc

# Content

- Simple linked list
- Double-ended list
- Sorted list
- Doubly linked list
- List with iterators

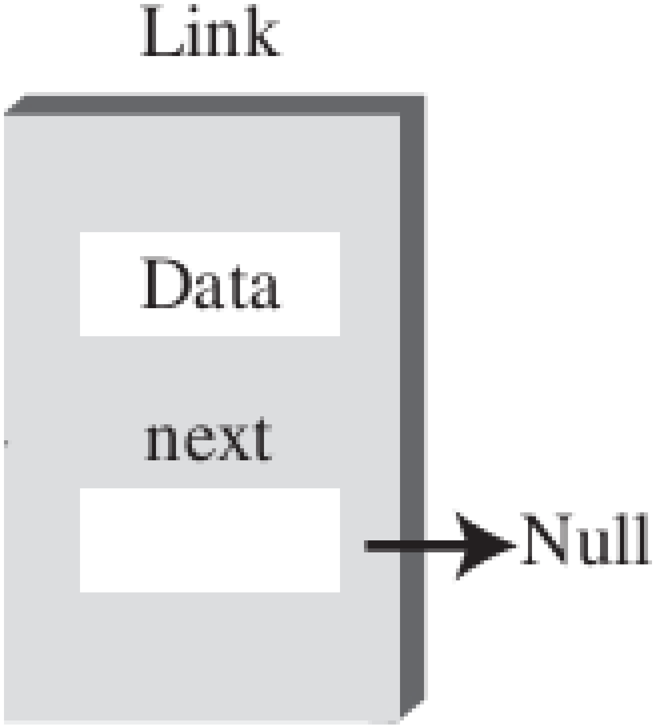
# Simple linked list

# Simple linked list

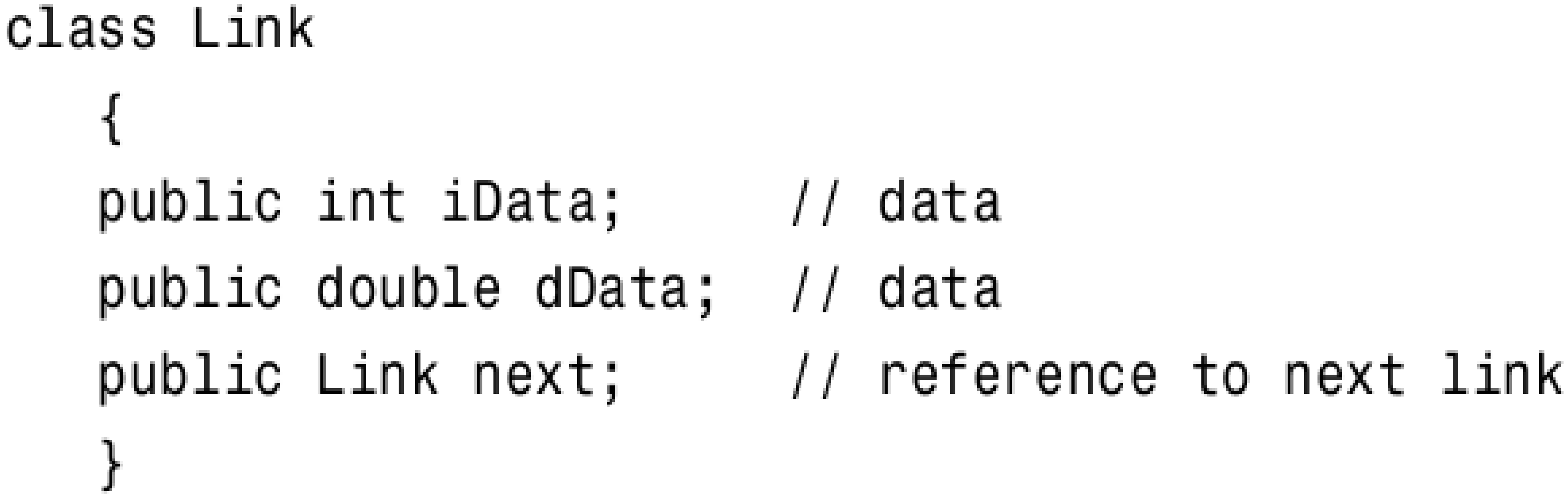


# Link

- A link contains
  - Data and
  - A reference to next link
    - ‘Next’



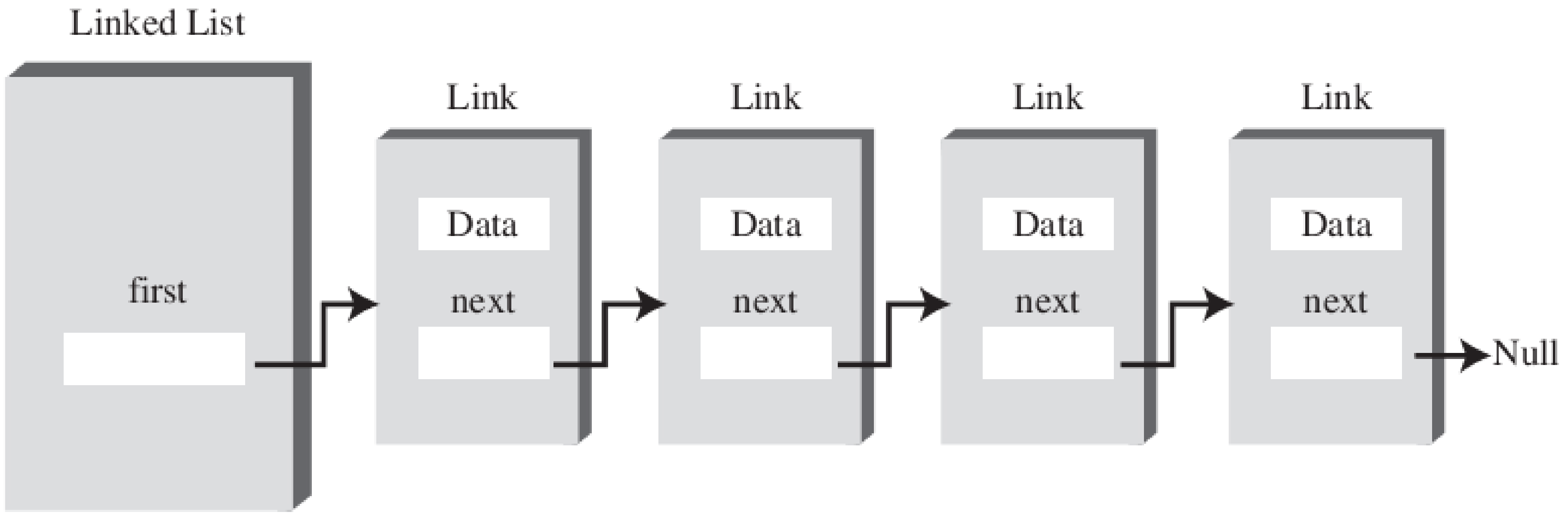
# Link class



```
class Link
   {
   public int iData;        // data
   public double dData;     // data
   public Link next;        // reference to next link
   }
```

# Relationship, not Position

- Can not access a data item directly.
- Must follow the chain from ‘First’ item



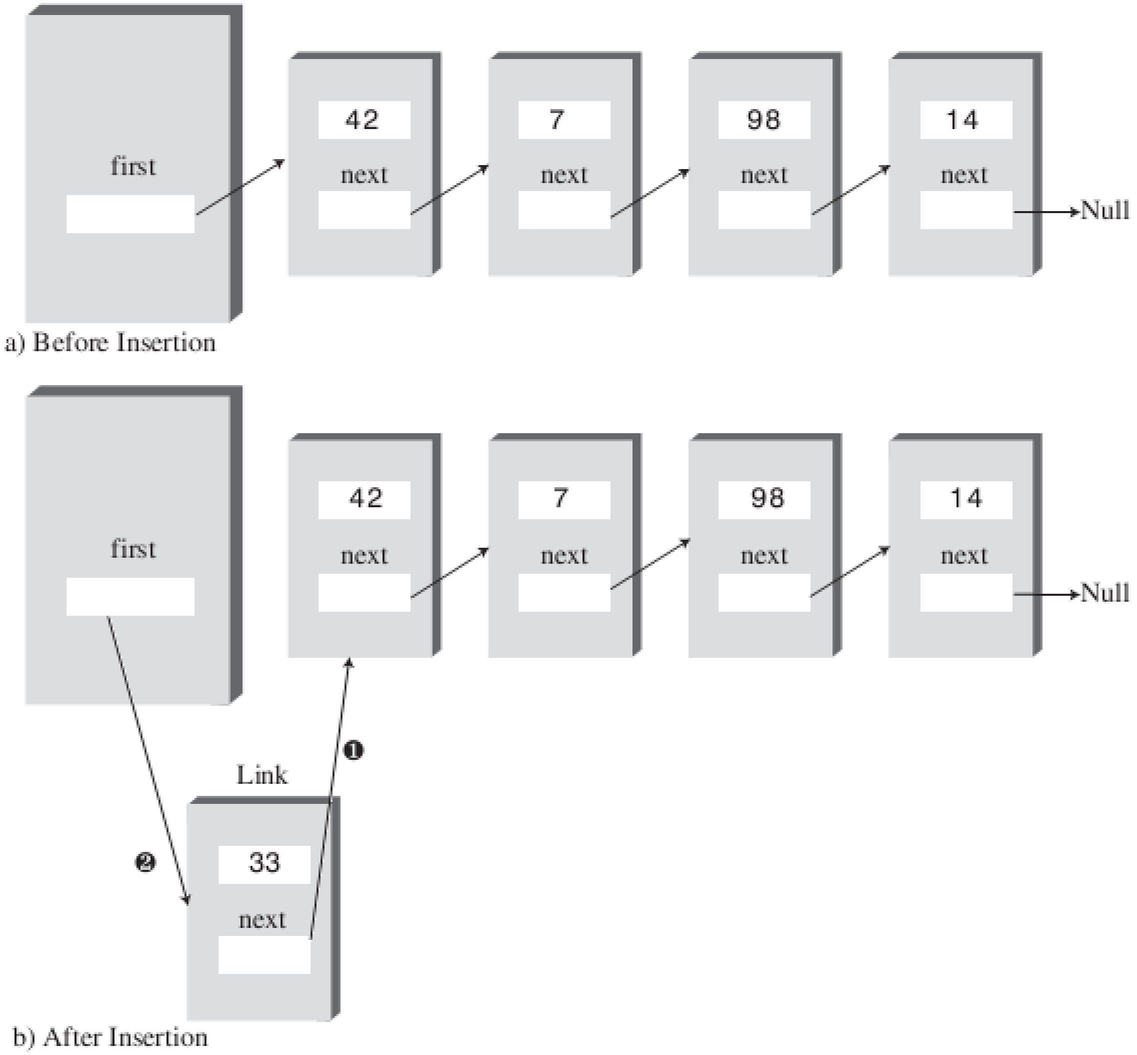
# Action on simple linked list

- Insertion
- Deletion
- Searching

→ Use the workshop for more information

# How would you do that - Insertion

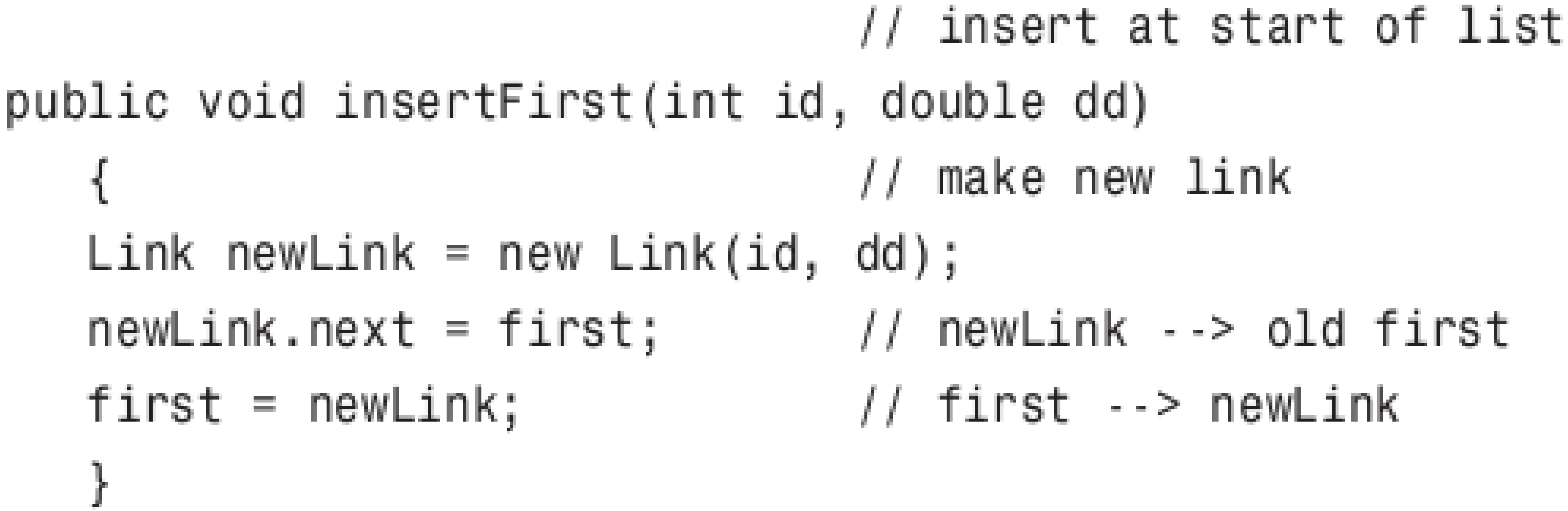
- □ InsertFirst?
- □ InsertLast?



a) Before Insertion

b) After Insertion

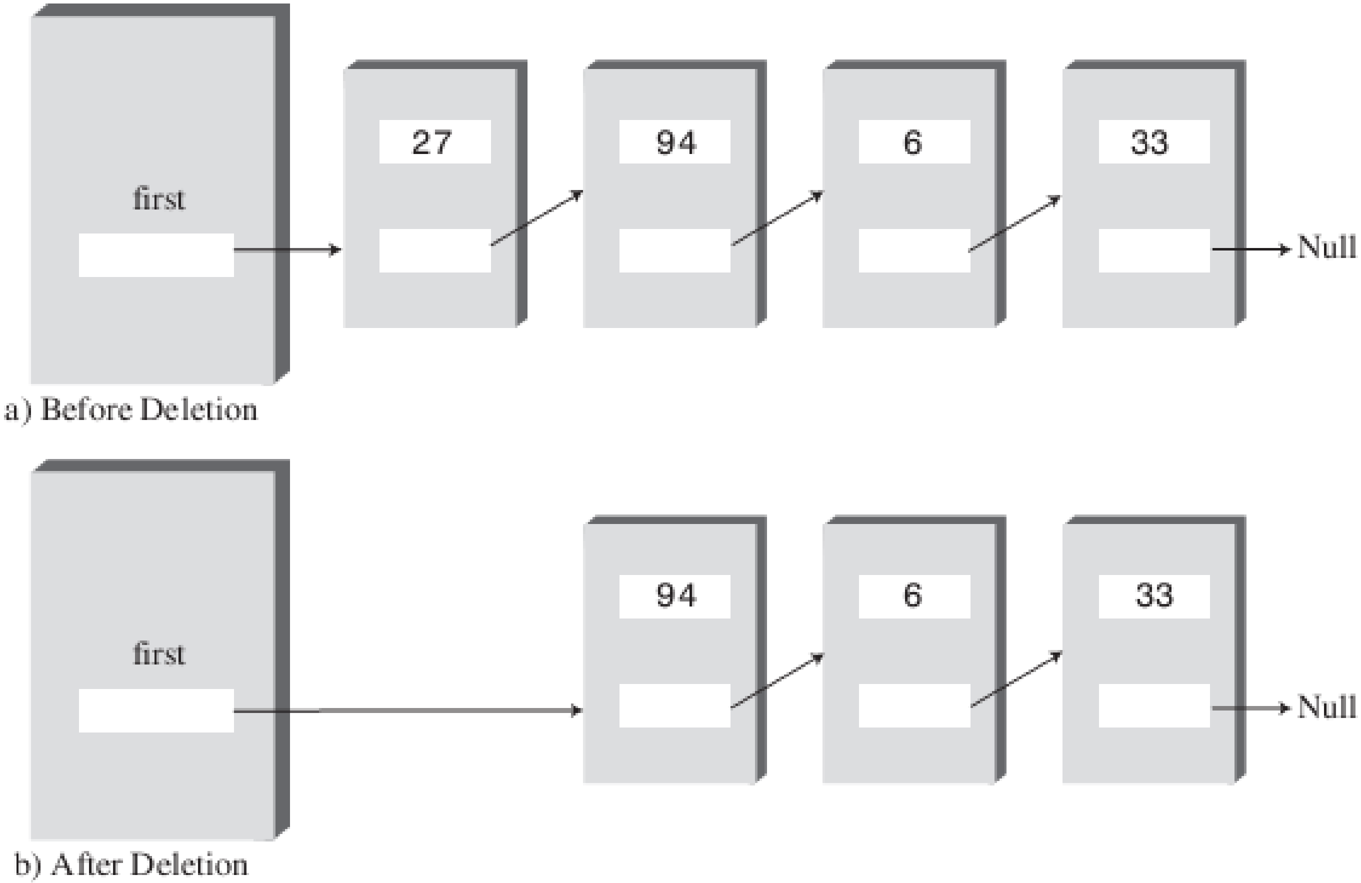
# In Java



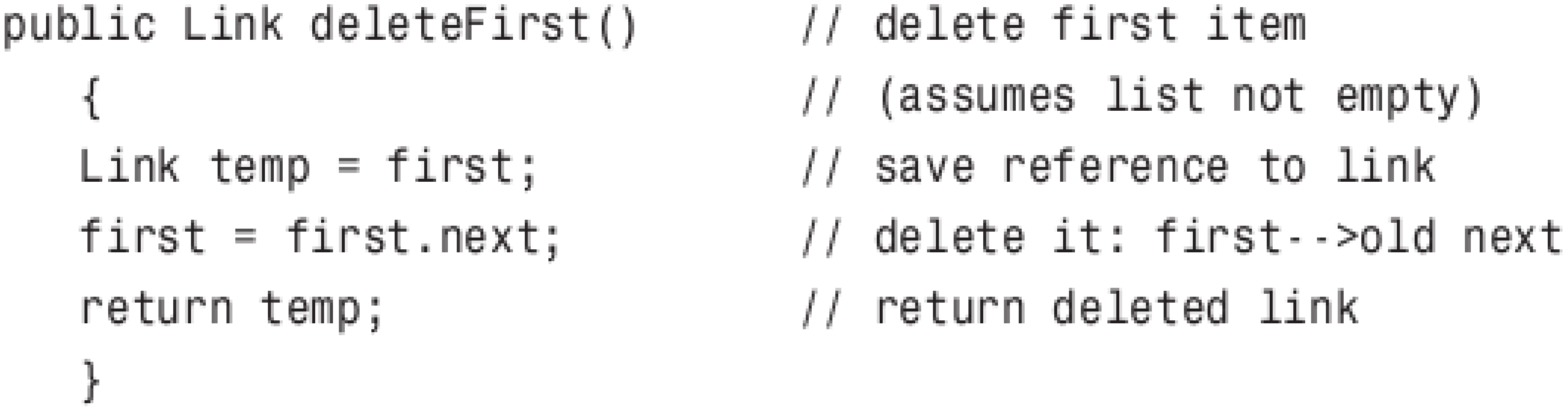
```
                               // insert at start of list
public void insertFirst(int id, double dd)
   {                           // make new link
   Link newLink = new Link(id, dd);
   newLink.next = first;       // newLink --> old first
   first = newLink;            // first --> newLink
   }
```

# How would you do that - Deletion

- Delete first? Delete last?



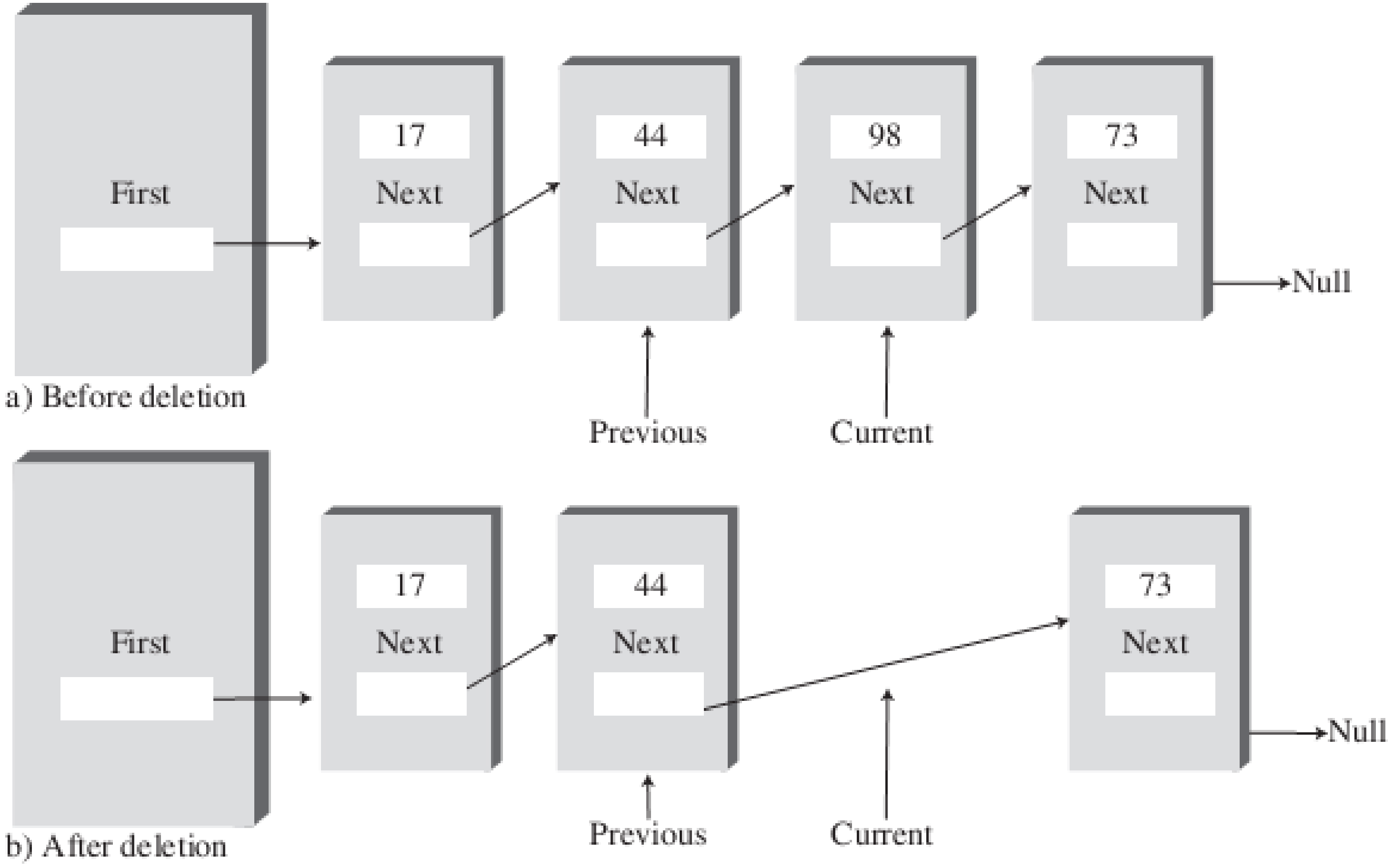
# In Java



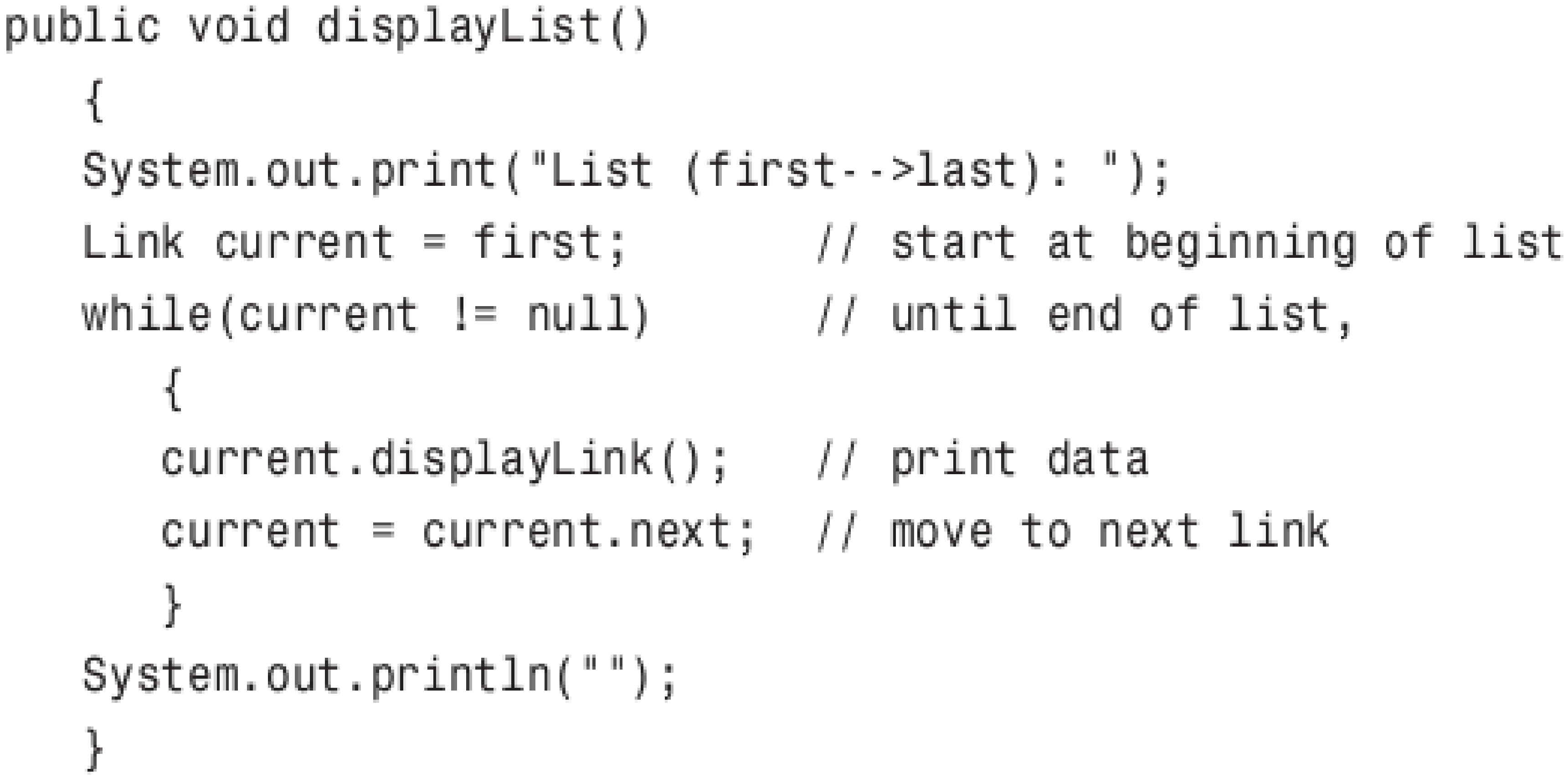
```
public Link deleteFirst()      // delete first item
   {                           // (assumes list not empty)
   Link temp = first;          // save reference to link
   first = first.next;         // delete it: first-->old next
   return temp;                // return deleted link
   }
```

# How would you do that - Deletion

- Delete a link in the middle of the list



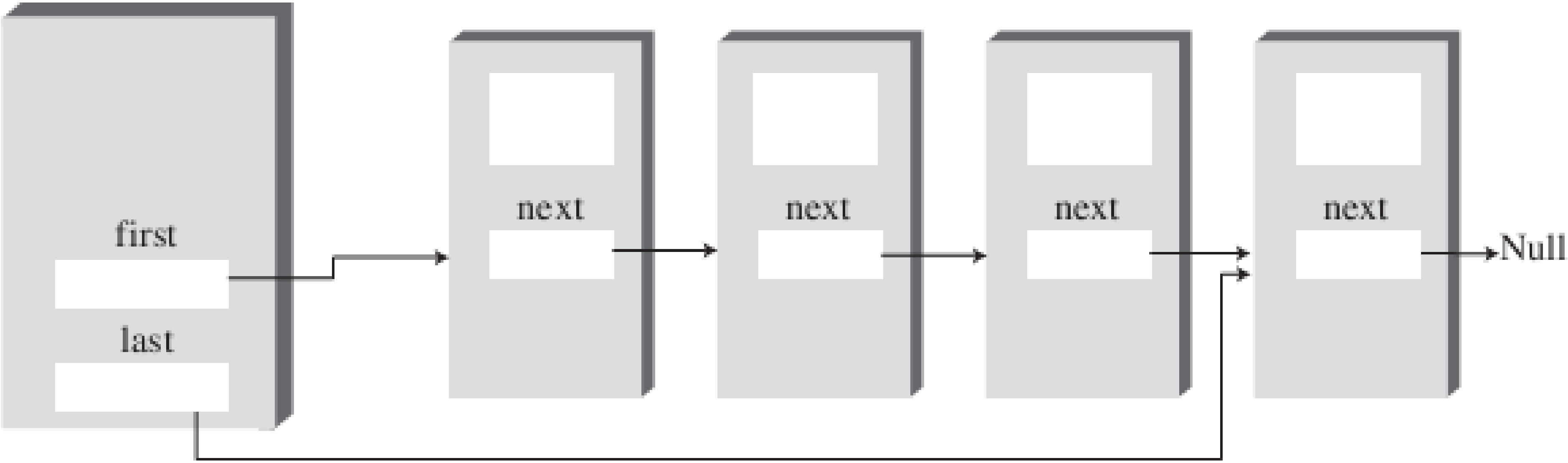
# How would you do that - Display



```
public void displayList()
    {
    System.out.print("List (first-->last): ");
    Link current = first;         // start at beginning of list
    while(current != null)        // until end of list,
        {
        current.displayLink();    // print data
        current = current.next;   // move to next link
        }
    System.out.println("");
    }
```

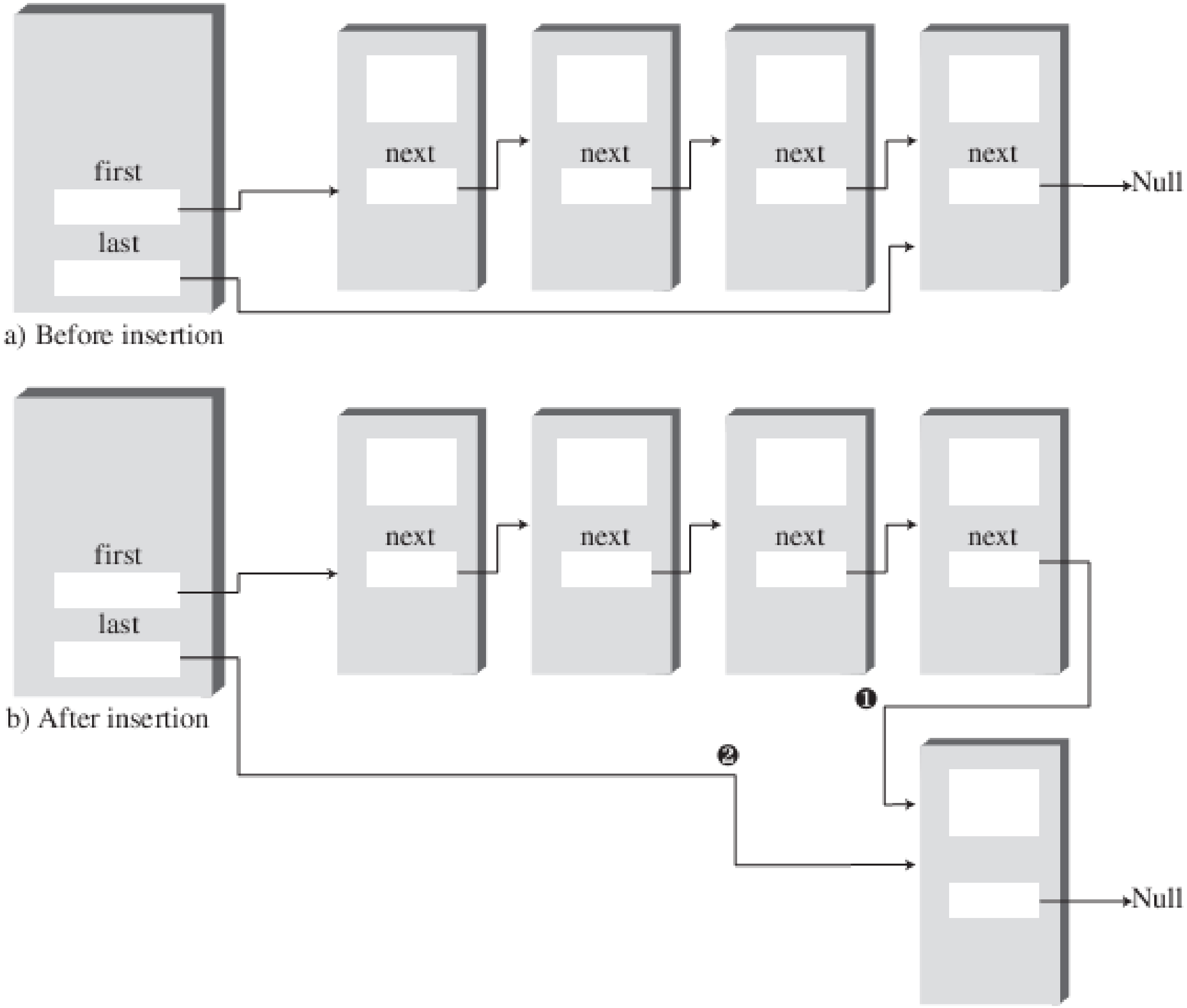
# Double-Ended Lists

# Double-Ended Lists



In compare with Simple linked list, what are the advantages?

# Directly insert to last position



# How about the deletion of last item

- Unfortunately, it doesn’t help.
  - Why?

# Simple linked list efficiency

- Insertion and deletion at beginning of the list are very fast: O(1)
- Finding, deleting, or insert item: O(n)
  - → is it the same as array (O(n) also)?
- In  comparison with array
  - Don’t have to shift items to delete or insert.
  - Uses exactly as much memory as it needs
  - Size can be changed

# Abstract Data Type

# ADT

- Is the way of looking at data structure focusing on
  - WHAT it does
  - NOT HOW it does
- Example:
  - Stack: Pop, Push, Peek
  - Queue: Enqueue, Dequeue
  - → We can implement these data structure by Array or Linked List

# Implement Stack & Queue

- Implement Stack using Linked List, any idea?
  - Push: InsertFirst
  - Pop: DeleteFirst
  - Peek: First
- Implement Queue using Linked List
  - Enqueue: InsertLast
  - Dequeue: DeleteFirst
- Stack/ Queue from the view of End-User: nothing change

# Data Types and Abstraction

- “Abstract”:
  **data description** is **considered apart** from **implementation**

- Classes vs Objects
  - Classes are abstractions - Abstraction
  - Individual objects – instantiations of those classes

# Data Types and Abstraction

- In OOP, we have ADTs.
  - Have **descriptions** of fields and methods
  - Contain **NO details** regarding the implementations.
  - A client has access to the methods and how to invoke them, and what to expect in return.
  - A client DO NOT know how the methods are implemented

# Data Types and Abstraction and Interface

- Client knows that stack operations include a
  - push(), pop(), isEmpty() and isFull().
- But have no knowledge as to how the data are stored (array, linked list, tree, etc.) or accessed / processed in logical data structures.
- Client has no knowledge as to how
  - push(), pop(), insert() and remove() are implemented.
  - Client has no knowledge about the underlying implementing data structure.

# Interface in OOP

- The ADT specification: **Interface**.
- It provides what the client needs to see
- Example:
  - **public interface IStack**
    - void push(long value)
    - long pop()

# ADTs as a Design Tool

- You are decoupling the specification of the ADT from its implementation.
  - Can change the implementation later!
  - This is its beauty.
- Naturally the underlying data structure must make the specified operations as efficient as possible.
  - Sequential access?  Perhaps a linked list.
  - Random access? An array does if you know the index of the desired array element.

# Sorted Lists

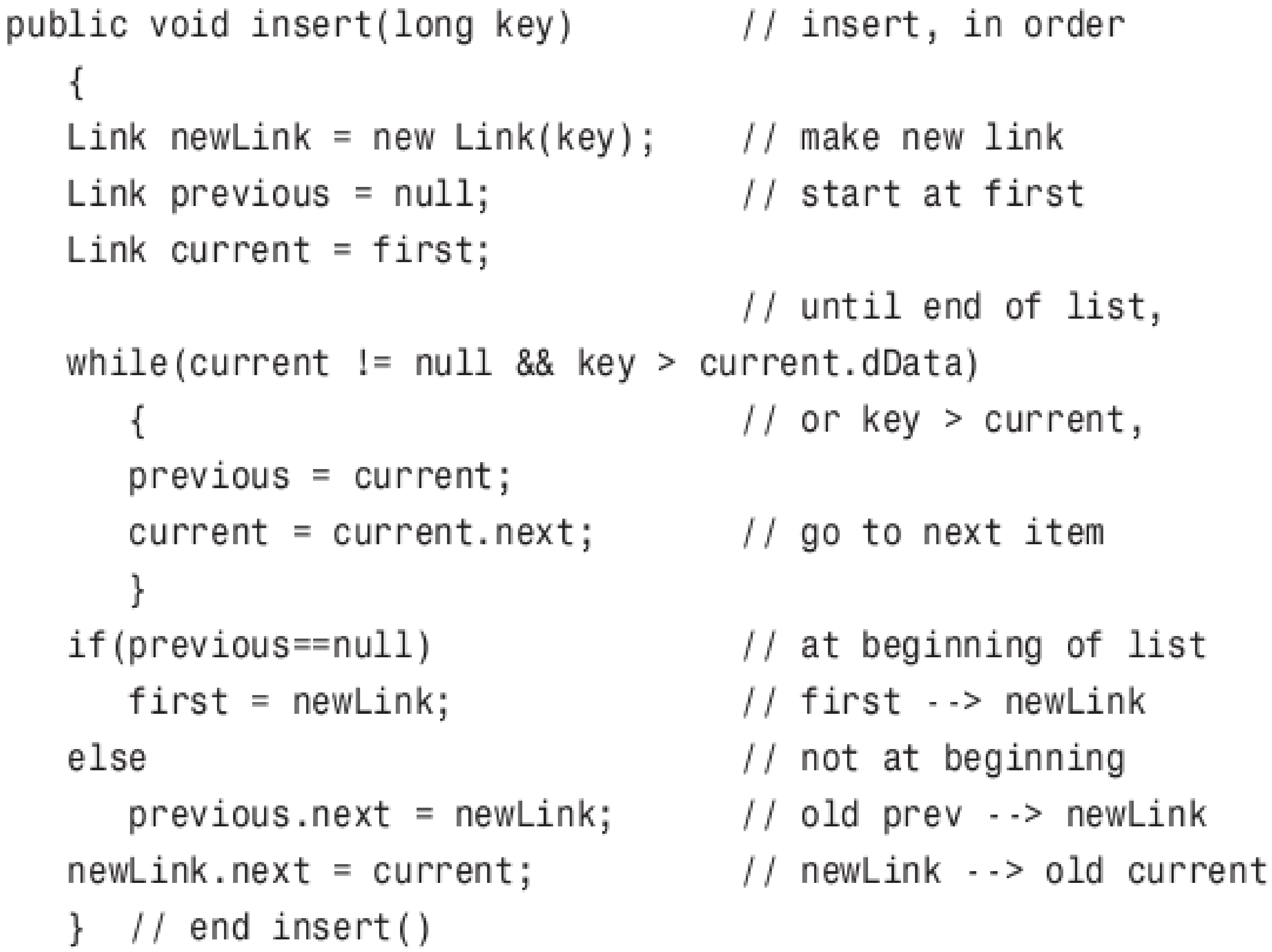
# Sorted list

- We need to store data in order
- Operations
  - Insert
  - DeleteSmallest, DeleteLargest
  - Delete(key)
- Can used to replace Array
  - Insertion speed is faster
  - Size of the list can expand

# How would you do that

- Operations
  - Insert
  - DeleteSmallest
  - DeleteLargest
  - Delete(key)

# Insert data to sorted list



```
public void insert(long key)        // insert, in order
   {
   Link newLink = new Link(key);    // make new link
   Link previous = null;            // start at first
   Link current = first;
                                    // until end of list,
   while(current != null && key > current.dData)
      {                             // or key > current,
      previous = current;
      current = current.next;       // go to next item
      }
   if(previous==null)               // at beginning of list
      first = newLink;              // first --> newLink
   else                             // not at beginning
      previous.next = newLink;      // old prev --> newLink
   newLink.next = current;          // newLink --> old current
   }  // end insert()
```

# Efficiency of sorted list

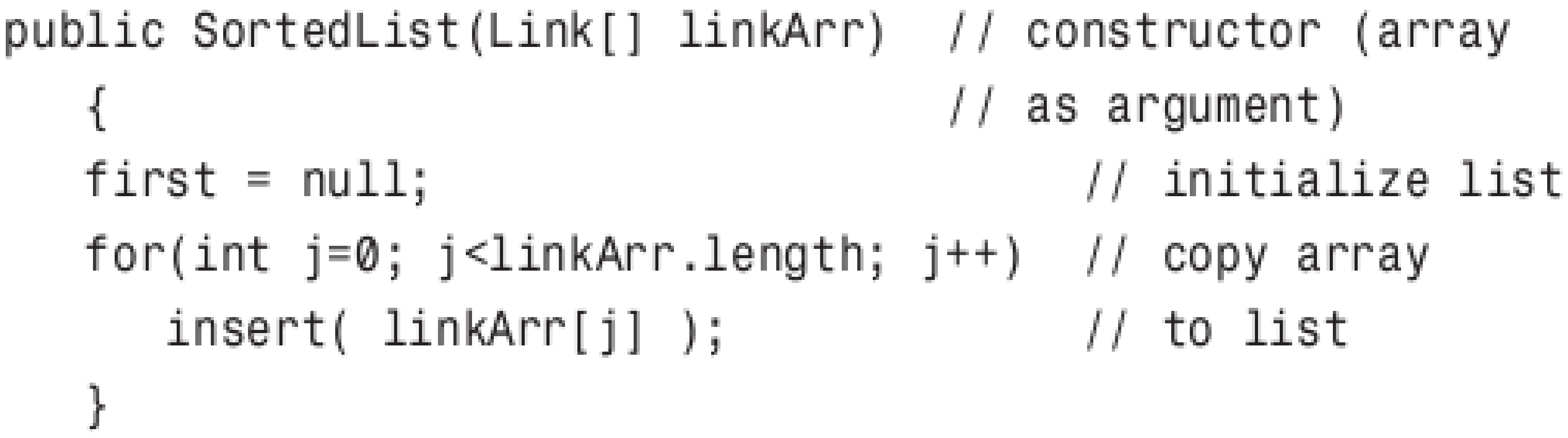
- Find/ Insertion / Deletion of arbitrary item: O(n)
- Find/ Insertion / Deletion of smallest/largest item: O(1)

- → For frequently access the minimum/maximum item application (Priority queue)

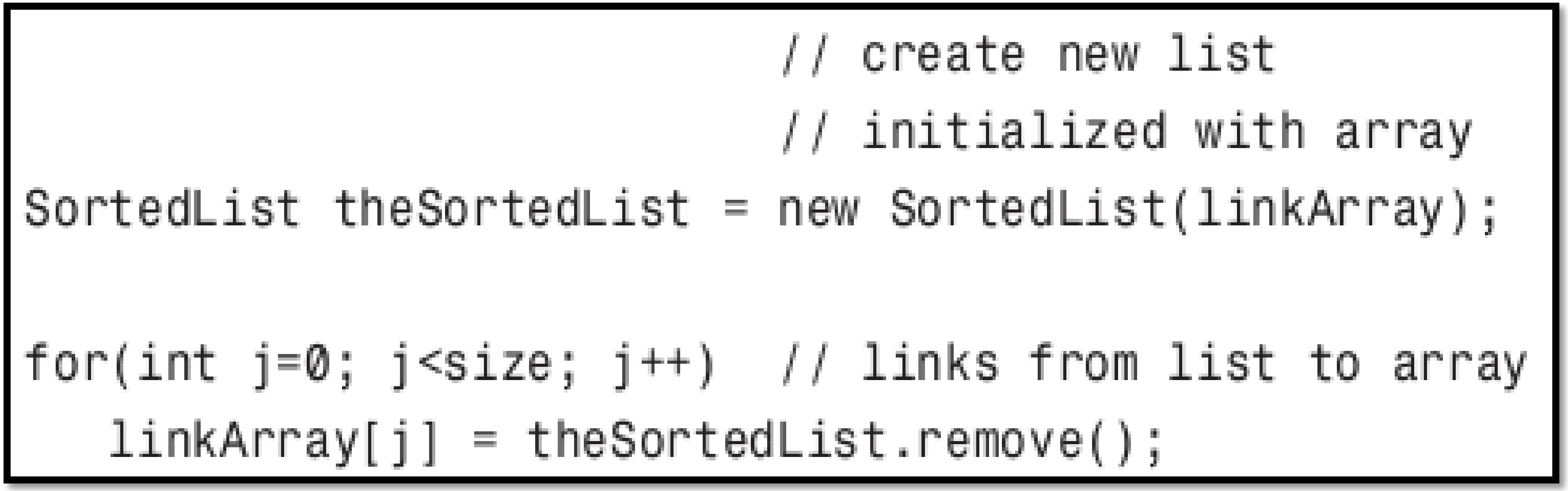
# Application

- Sort an array
  - Insert item from array to sorted list
  - Get item from list and insert back to array
- → Still O($n^2$)
- But
  - Fewer copy/shift operation

# Some code



```
public SortedList(Link[] linkArr)  // constructor (array
   {                               // as argument)
   first = null;                       // initialize list
   for(int j=0; j<linkArr.length; j++) // copy array
      insert( linkArr[j] );            // to list
   }
```



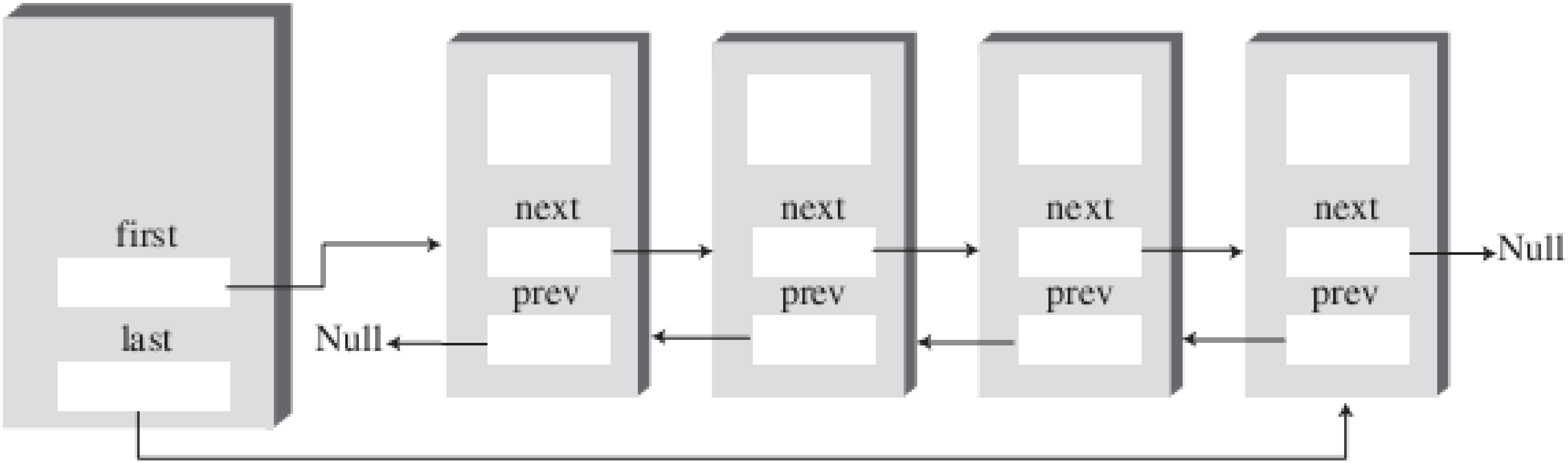
```
                             // create new list
                             // initialized with array
SortedList theSortedList = new SortedList(linkArray);

for(int j=0; j<size; j++)  // links from list to array
   linkArray[j] = theSortedList.remove();
```

# Doubly linked lists

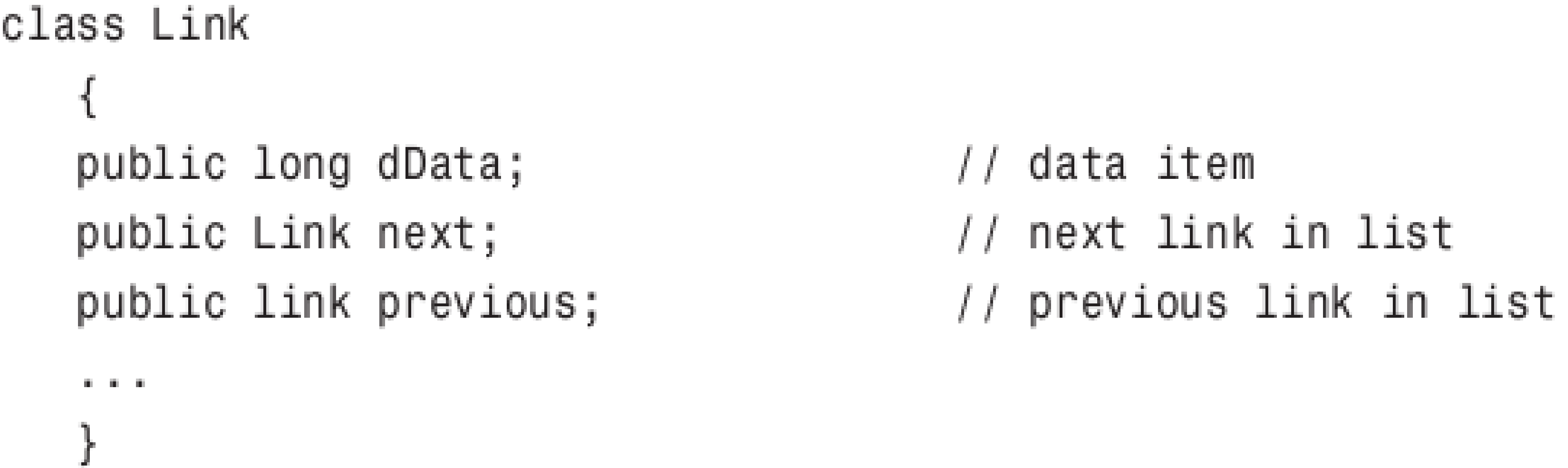
# Introduction

- Singly linked list: One way traversing
  - current = current.next
- → need to traverse backward as well as forward through the list
- → doubly linked list

# Doubly linked list



# Doubly linked list

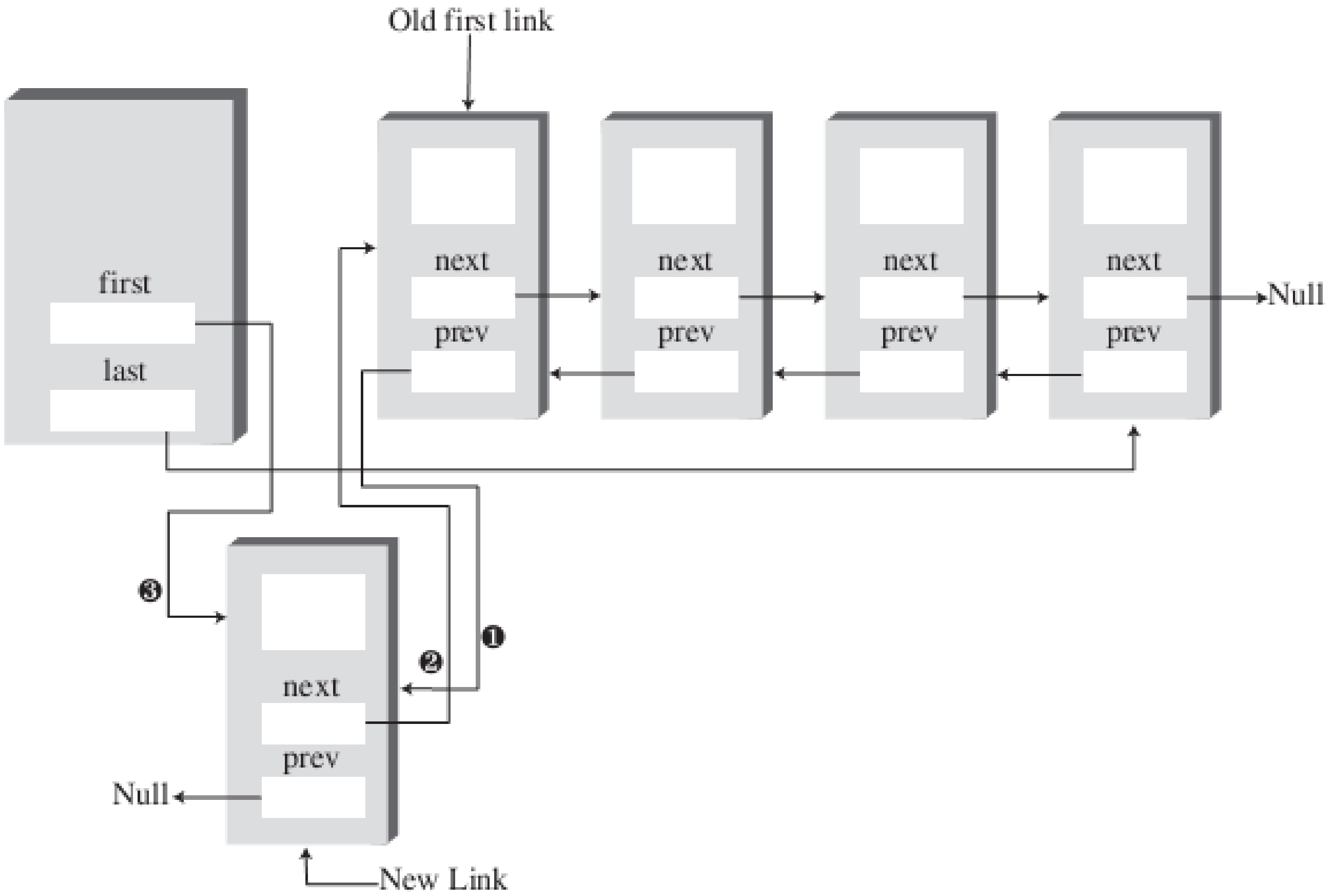


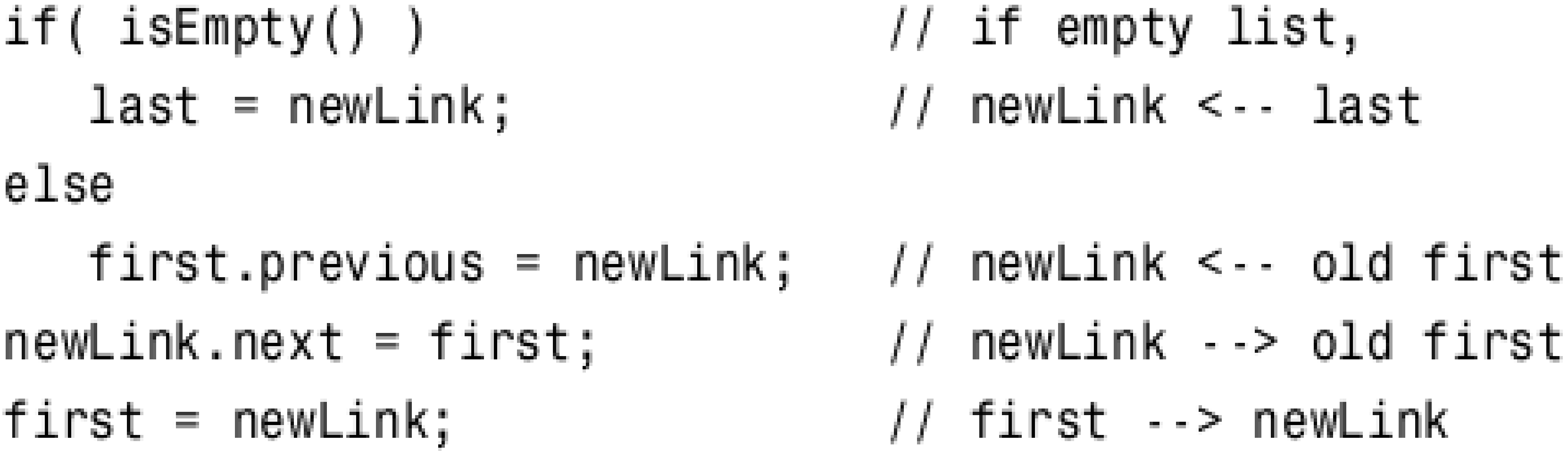
```
class Link
   {
   public long dData;                    // data item
   public Link next;                     // next link in list
   public link previous;                 // previous link in list
   ...
   }
```

# Operations

- InsertFirst, InsertLast, InsertAfter, InsertBefore
- DisplayForward, DisplayBackward
- DeleteFirst, DeleteLast, Delete(key)

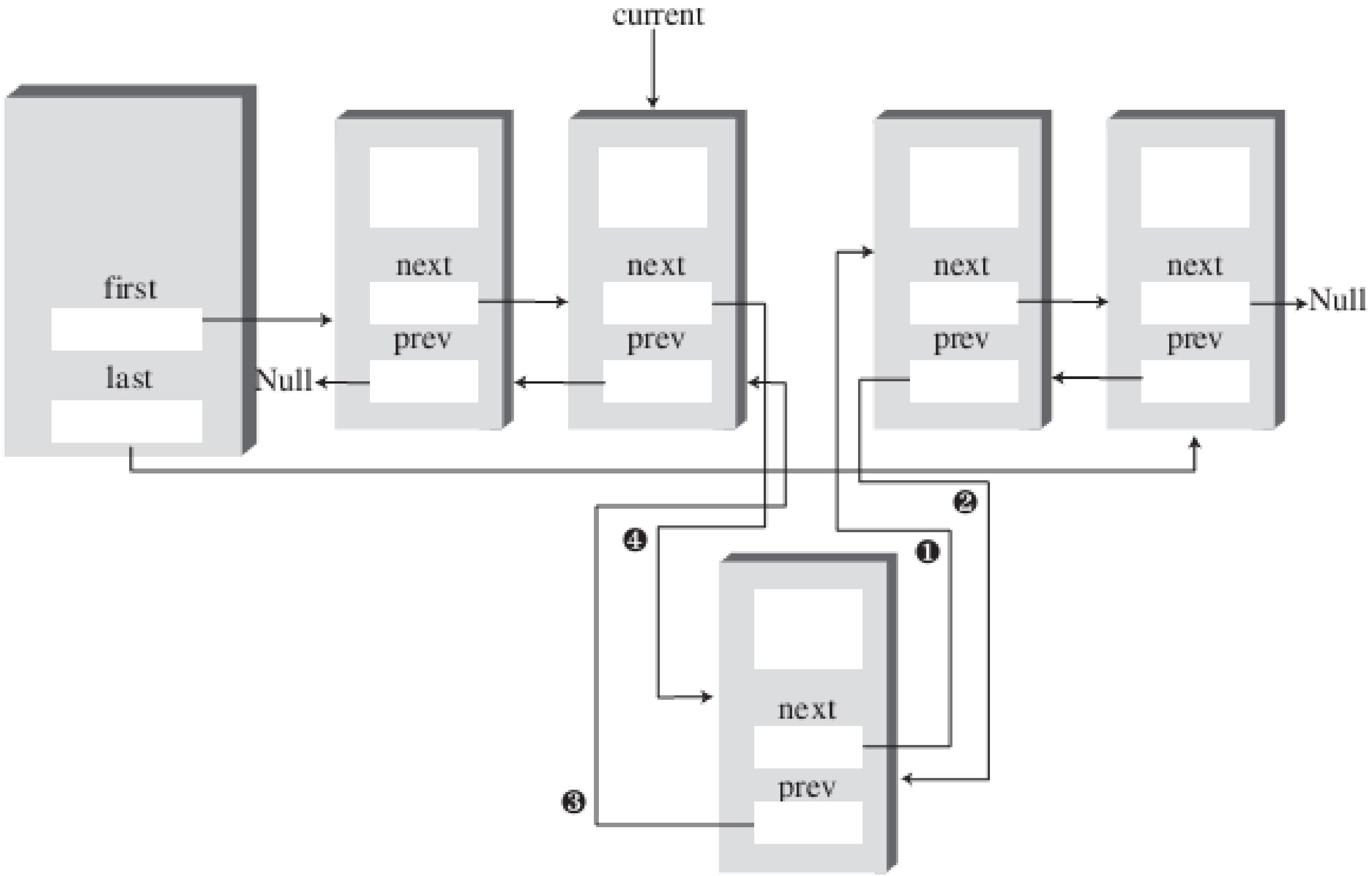
# InsertFirst



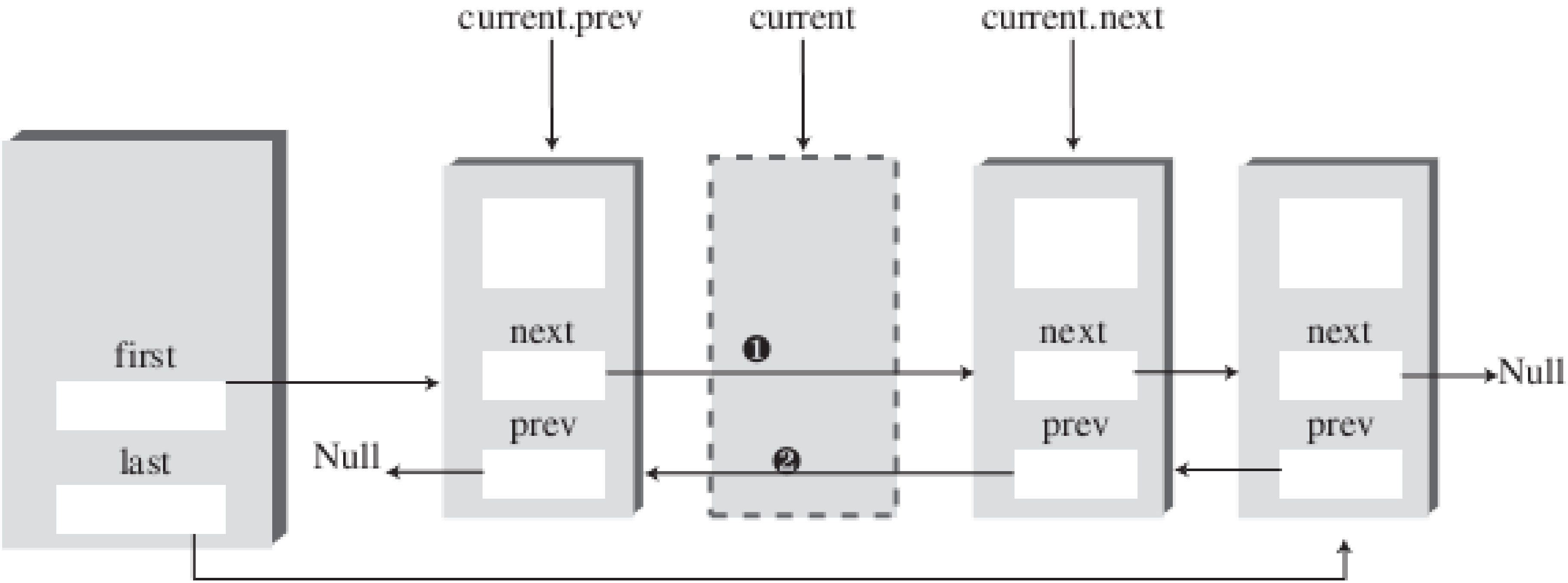


```
if( isEmpty() )                // if empty list,
   last = newLink;             // newLink <-- last
else
   first.previous = newLink;   // newLink <-- old first
newLink.next = first;          // newLink --> old first
first = newLink;               // first --> newLink
```

# Insert in the middle of list



# Delete an item



# Application

- Implement deque
  - Queue that can insert and delete at either end
- Support bi-direction traversing

# Iterators

Data Structure and Algorithm,

Robert Lafore

Page 231

# Homework

- What is Iterator?
- Why do we need Iterator?
- What can we do with Iterator?
- Some application of iterator?