Vietnam National University of HCMC

International University

School of Computer Science and Engineering

Data Structures and Algorithms
★ Linked List ★

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn

https://vichithanh.github.io

SCAN ME

Week by week topics (*)

1. Overview, DSA, OOP and Java
2. Arrays
3. Sorting
4. Queue, Stack
5. List
6. Recursion

Mid-Term

7. Advanced Sorting
8. Binary Tree
9. Hash Table
10. Graphs
11. Graphs Adv.

Final-Exam

10 LABS

Content

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.

List Data Structures

- A list is a sequential data structure, i.e., it is a sequence of items of a given base type, where items can be added, deleted, and retrieved from any position in the list.
- A list can be implemented as an array, or as a dynamic array to avoid imposing a maximum size.
- An alternative implementation is a linked list, where the items are stored in nodes that are linked together with pointers. These two implementations have very different characteristics.
- The possible values of this type are sequences of items of type BaseType (including the sequence of length zero).

Operations

- getFirst(), getLast(), getNext(p), getPrev(p), get(p), set(p,x), insert(p,x), remove(p),removeFirst(), removeLast(), removeNext(p), removePrev(p), find(x),size()

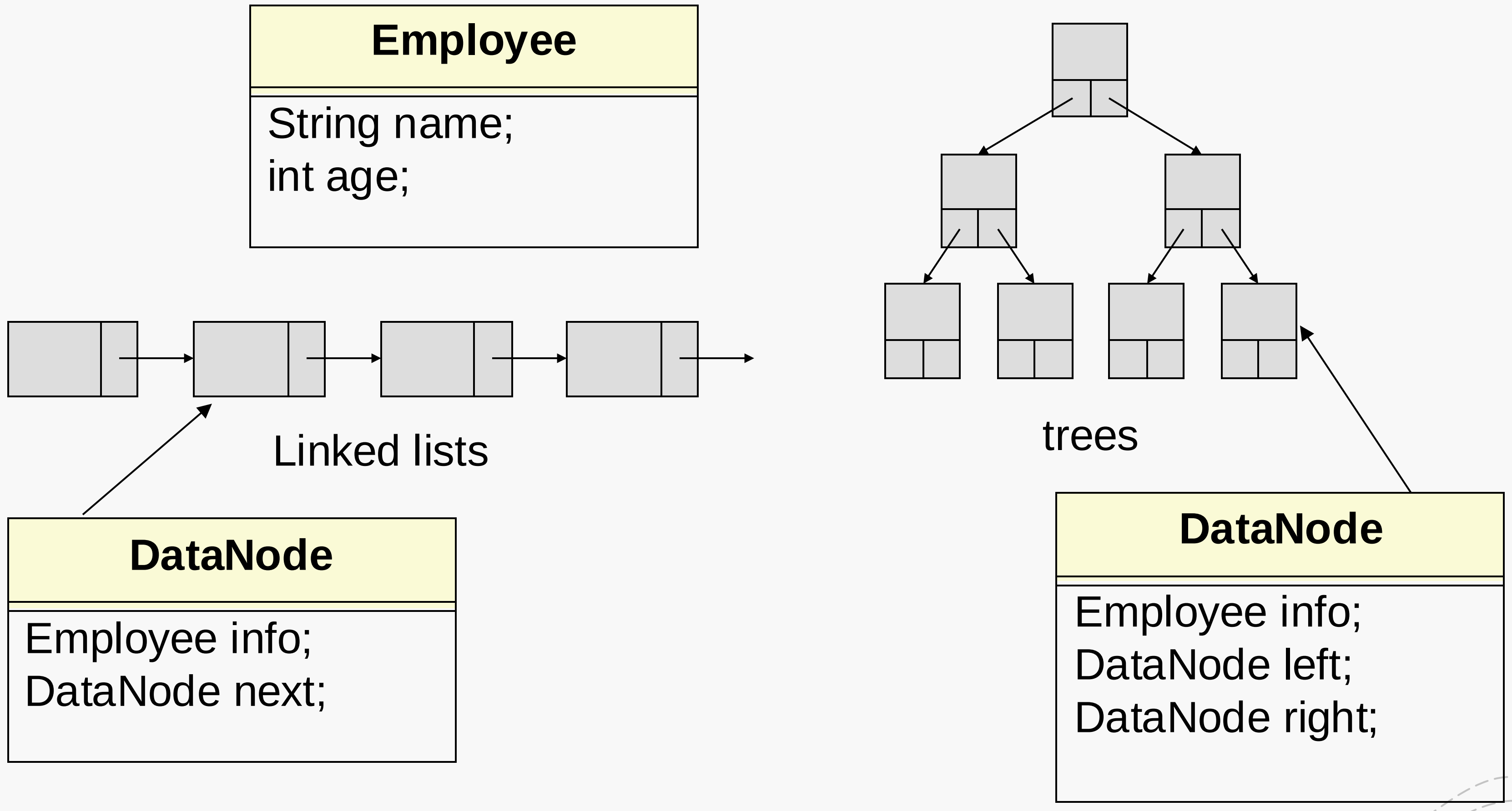
Operations

- getFirst(), getLast(), getNext(p), getPrev(p), get(p), set(p,x), insert(p,x), remove(p),removeFirst(), removeLast(), removeNext(p), removePrev(p), find(x),size()

Self-Referential Structures

- Many dynamic data structures are implemented through the use of a self-referential structure.
- A self-referential structure is an object, one of whose elements is a reference to another object of its own type.
- With this arrangement, it is possible to create ‘chains’ of data of varying forms

Self-Referential Structures



Employee

```
String name;
int age;
```

DataNode

```
Employee info;
DataNode next;
```

DataNode

```
Employee info;
DataNode left;
DataNode right;
```

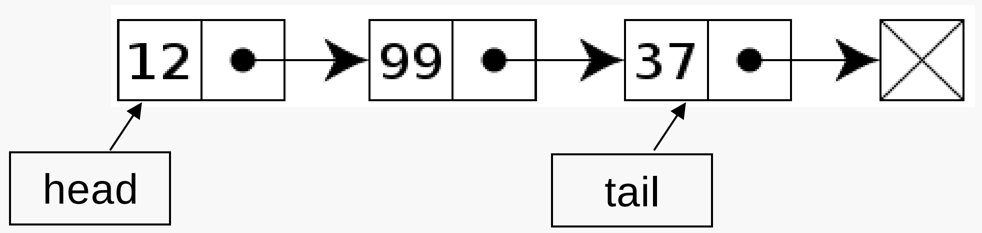
Self-Referential Structures

Linked Lists

- A **linked structure** is a collection of nodes storing data and links to other nodes
- A **linked list** is a linear data structure composed of nodes, each node holding some information and a reference to another node in the list
- Types of linked lists:
  - Singly-Linked List
  - Circular Lists
  - Double-Ended List
  - Doubly-Linked List

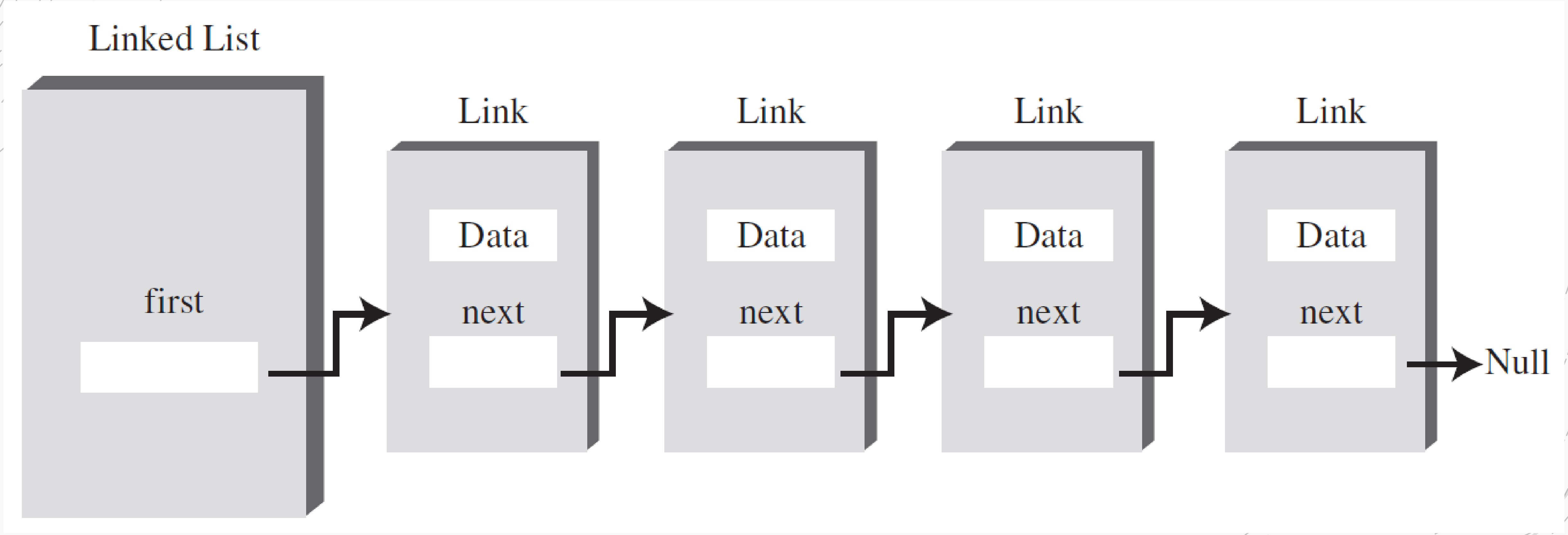
Singly Linked Lists

- A singly linked list is a list whose node includes two data fields: info and next. The info field is used to store information, and this is important to the user. The next field is used to link to its successor in this sequence
- The following image depicts a simple integer linked list.



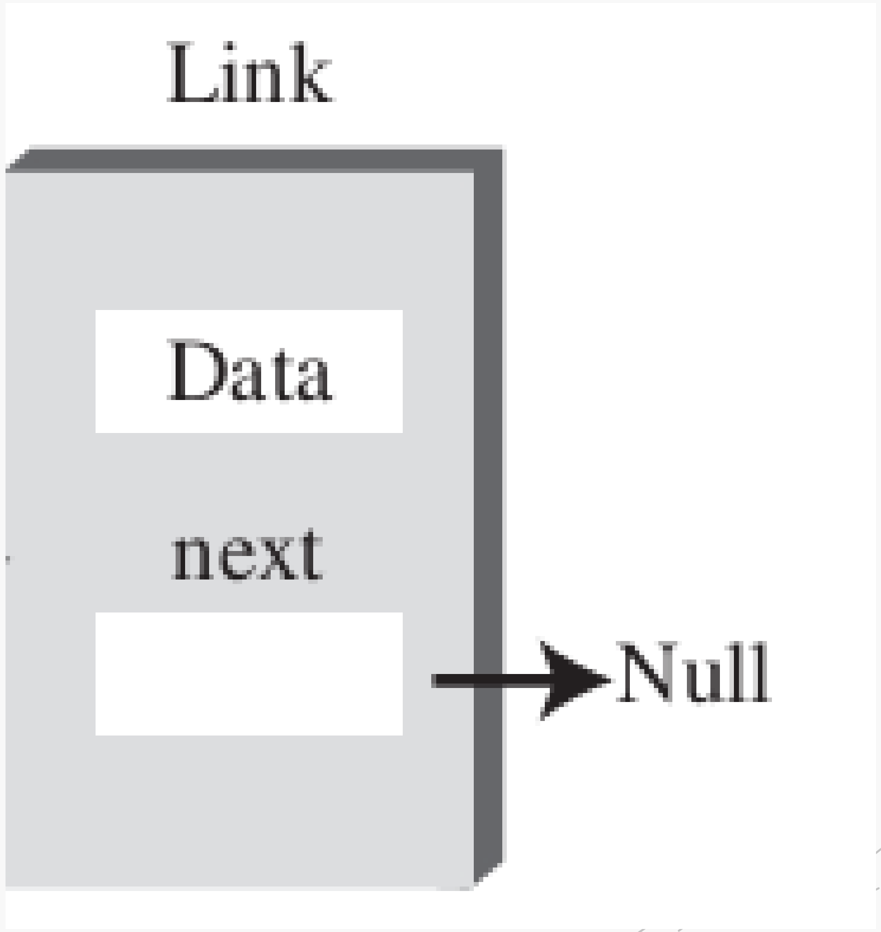
Singly Linked List

Simple linked list

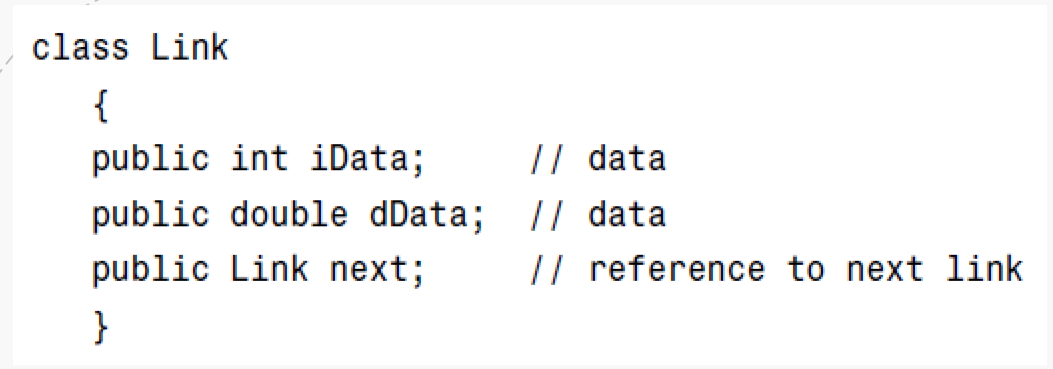


Link

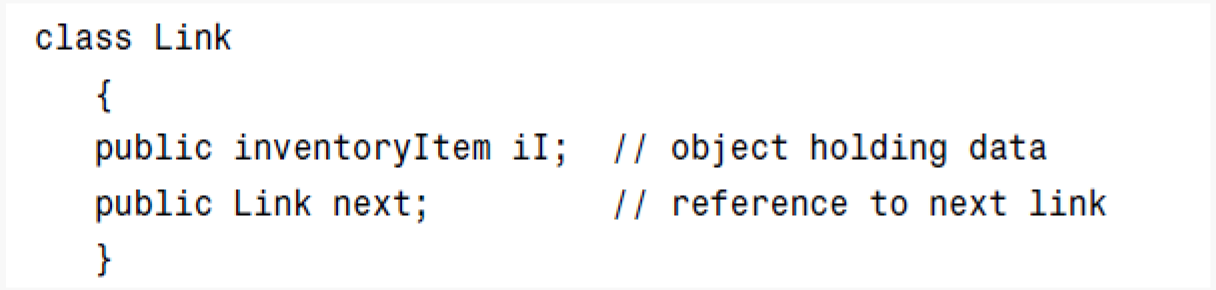
- A link contains
  - Data
  - A reference to next link ('Next')



Link class



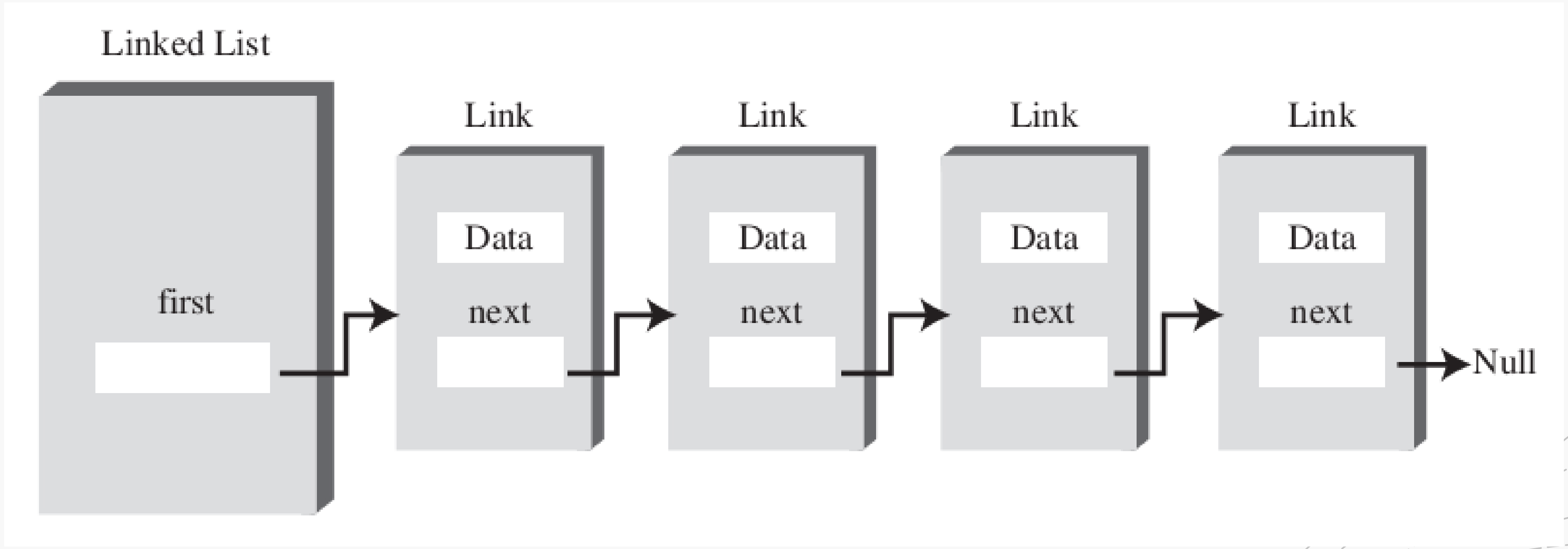
```
class Link
   {
   public int iData;     // data
   public double dData;  // data
   public Link next;     // reference to next link
   }
```



```
class Link
   {
   public inventoryItem iI;  // object holding 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



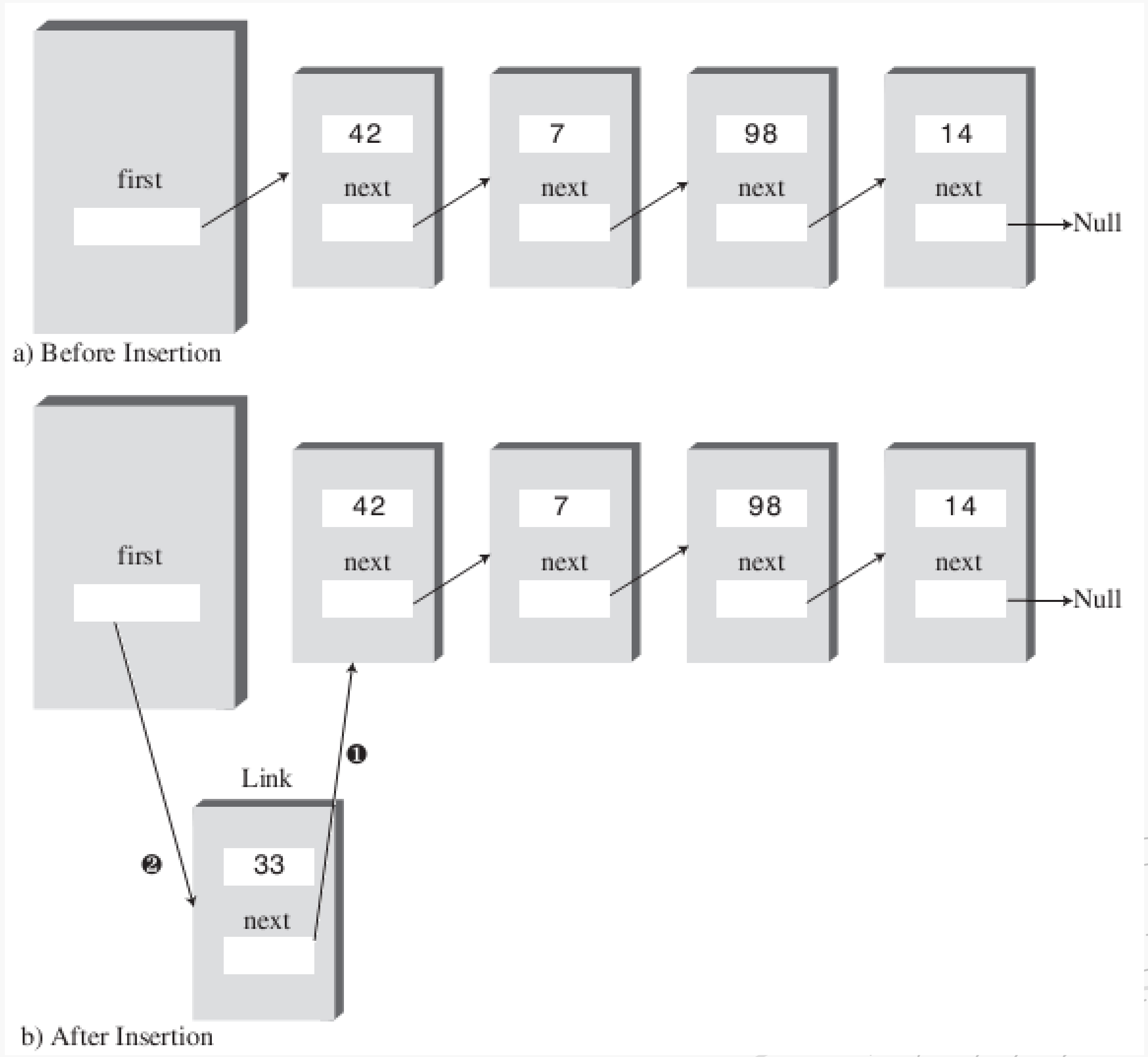
Action on simple linked list

- Insertion
- Deletion
- Searching

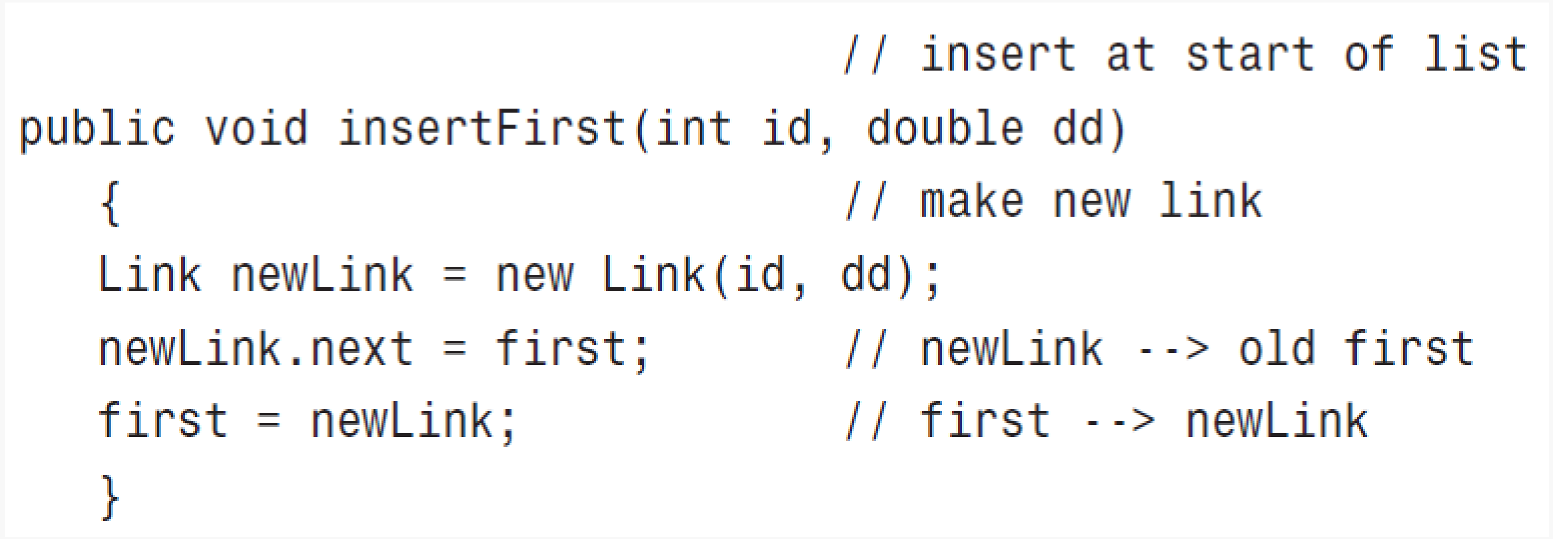
How would you do that – Insertion

- InsertFirst?

- InsertLast?



In Java

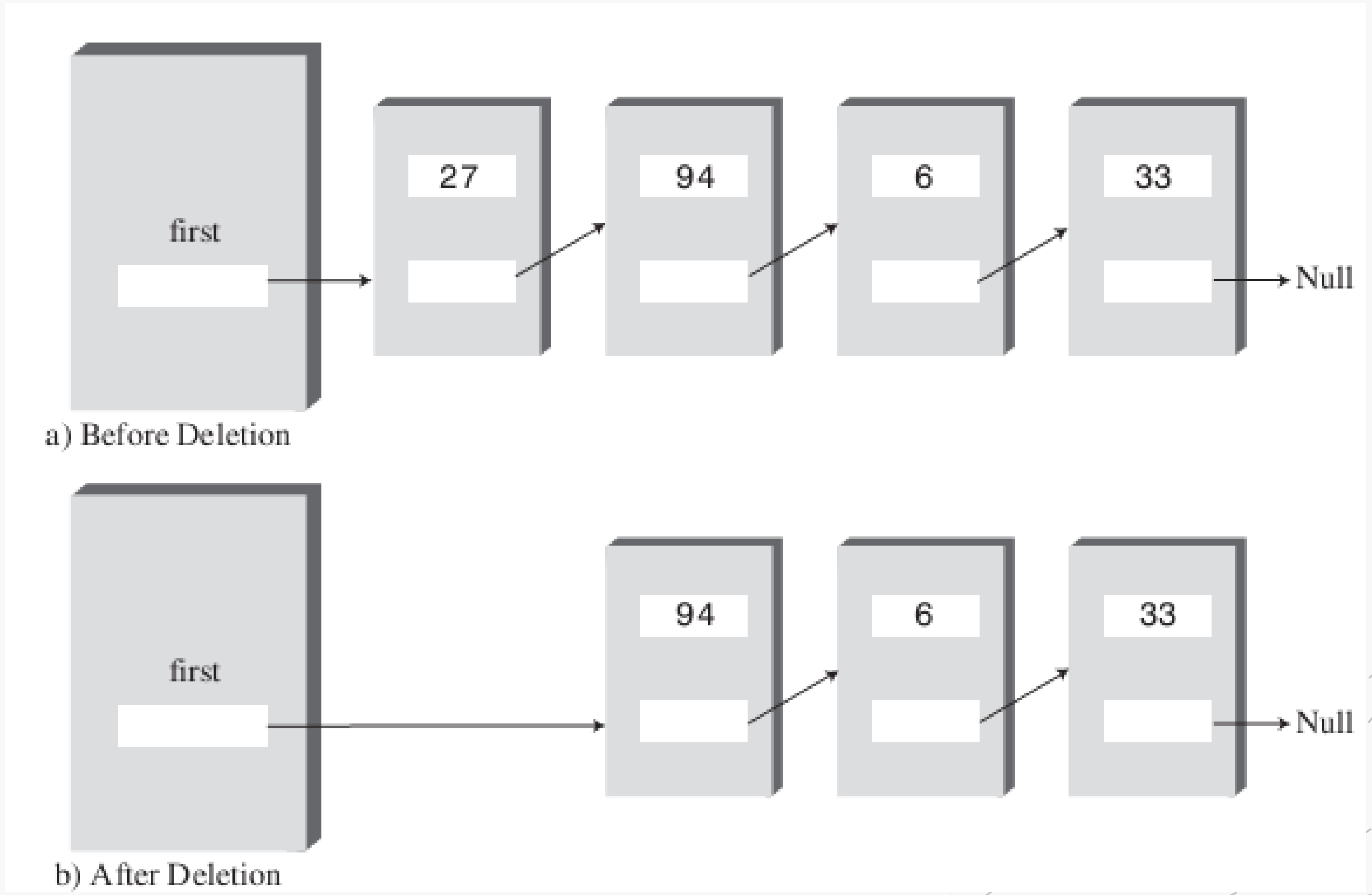


```
                               // insert at start of list
public void insertFirst(int id, double dd)
   {                           // make new link
   Link newLink = new Link(id, dd);
   newLink.next = first;       // newLink --> old first
   first = newLink;            // first --> newLink
   }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

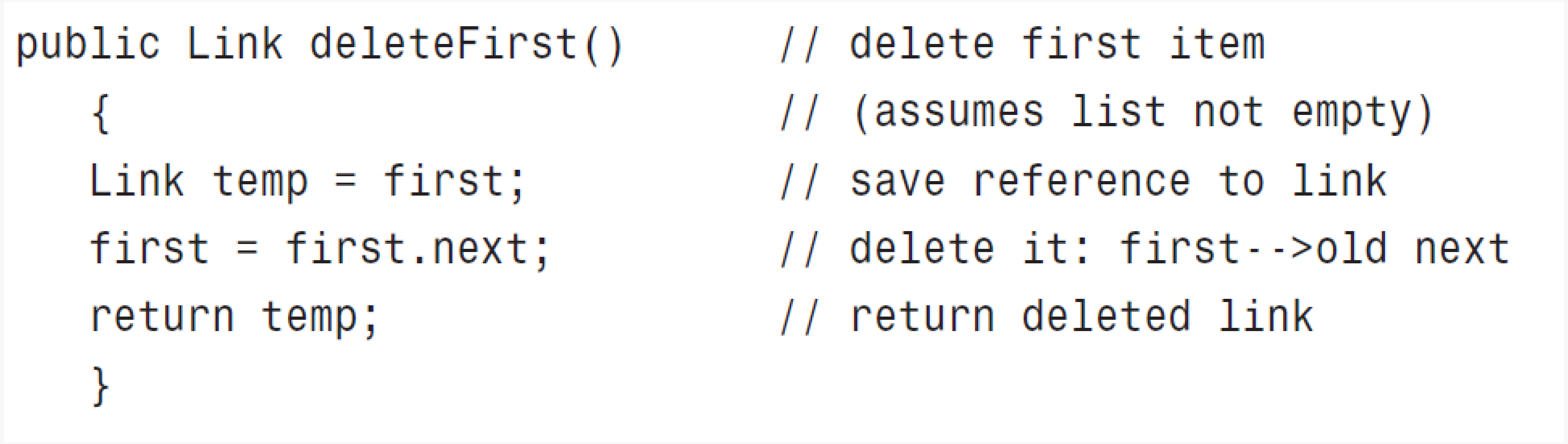
How would you do that – Deletion

- Delete first?

- Delete last?



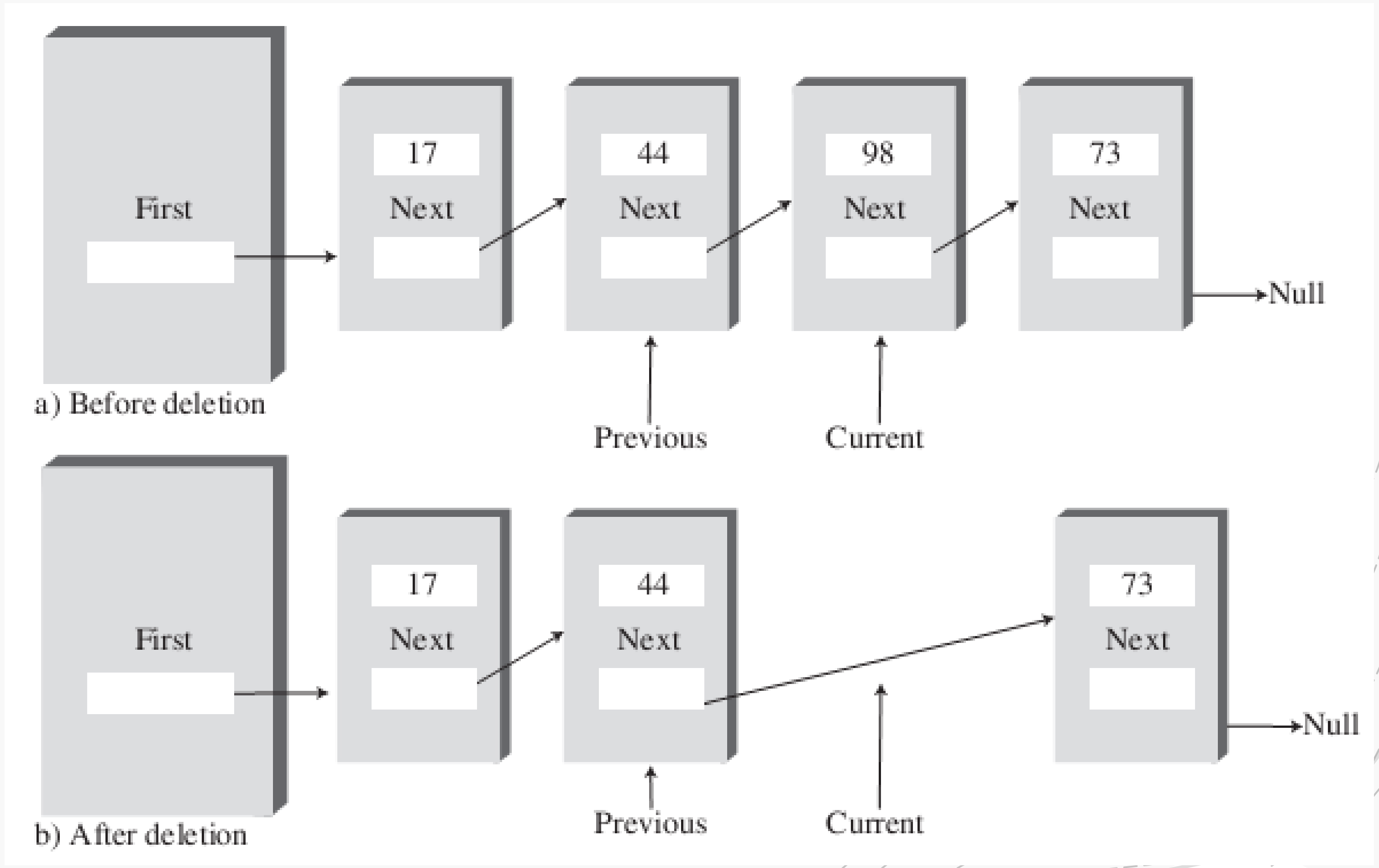
In Java



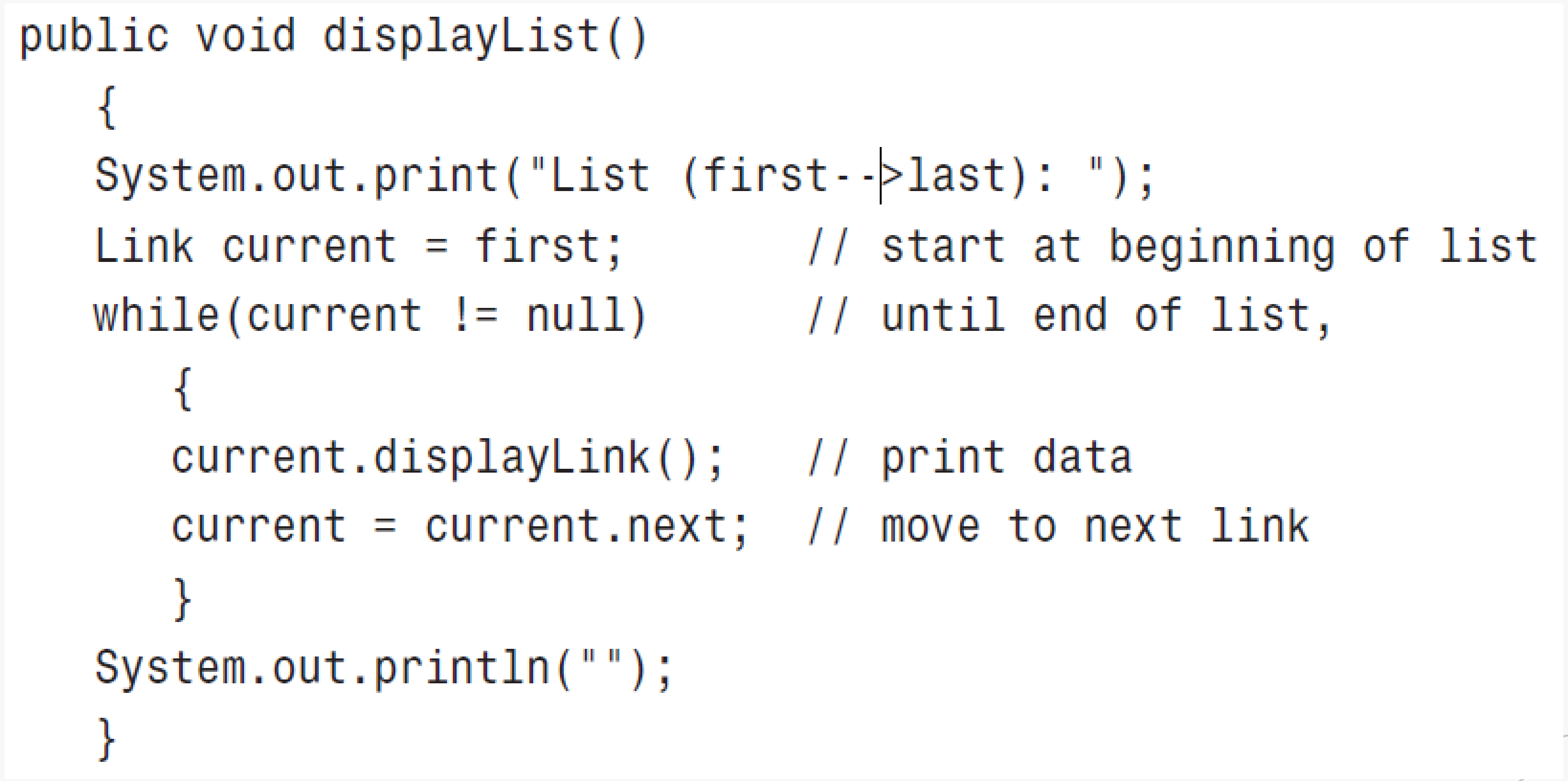
```
public Link deleteFirst()      // delete first item
   {                           // (assumes list not empty)
   Link temp = first;          // save reference to link
   first = first.next;         // delete it: first-->old next
   return temp;                // return deleted link
   }
```

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("");
   }
```

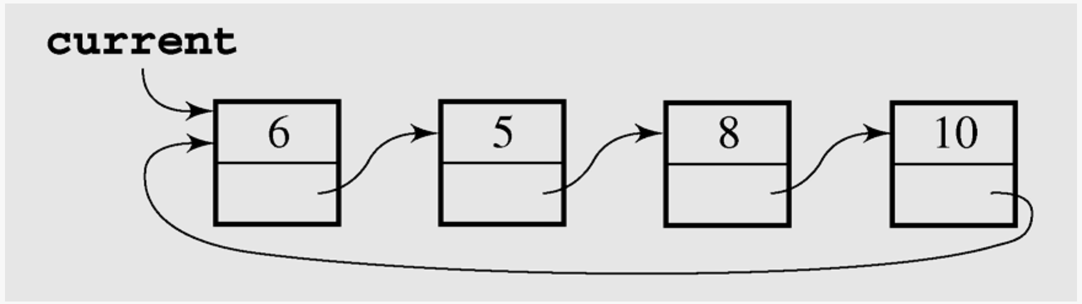
Practice

- LinkList2App.java
- Complete the functions:
  - insertFirst
  - find
  - delete

Circular List

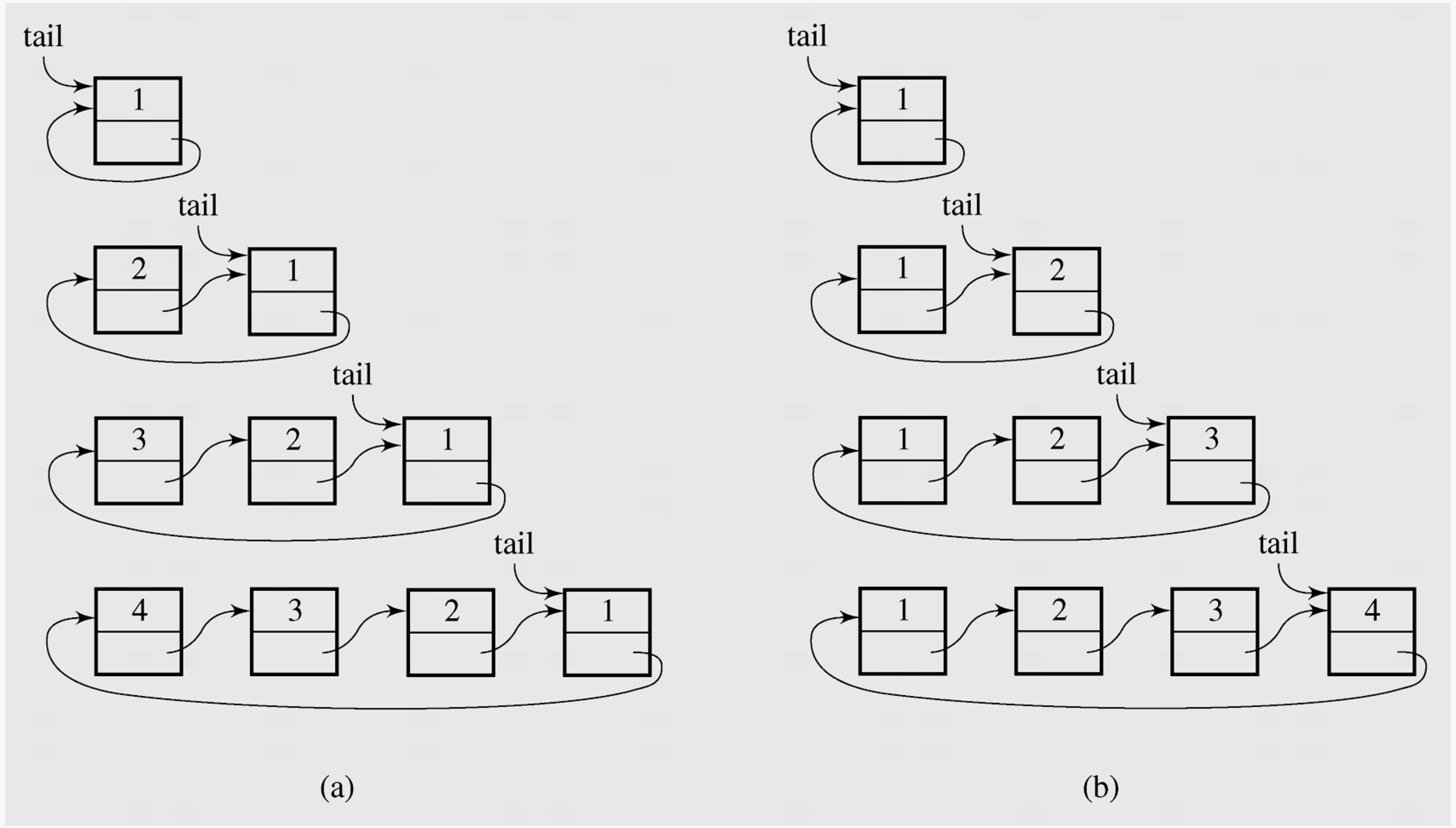
Circular Lists – 1

- A **circular list** is when nodes form a ring: The list is finite, and each node has a successor



Circular Singly Linked List

Circular Lists - 2
Inserting nodes



Inserting nodes at the front of a circular singly linked list (a) and at its end (b)

Circular List application

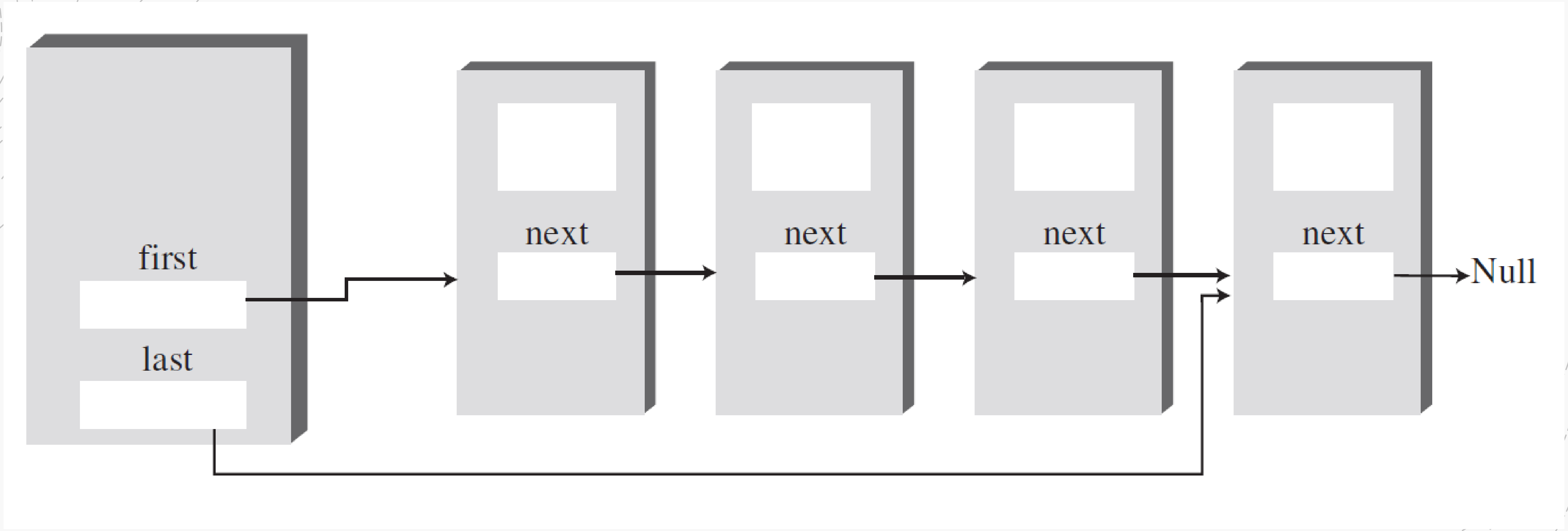
- **1. Round-Robin Scheduling**
- One of the most important roles of an operating system is in managing the many processes that are currently active on a computer, including the scheduling of those processes on one or more central processing units (CPUs).
- To support the responsiveness of an arbitrary number of concurrent processes, most operating systems allow processes to effectively share use of the CPUs, using some form of an algorithm known as ***round-robin scheduling***.
- A process is given a short turn to execute, known as a ***time slice***, but it is interrupted when the slice ends, even if its job is not yet complete. Each active process is given its own time slice, taking turns in a cyclic order.

Circular List application

- **2. Using circular linked list to implement Round-Robin Scheduling**
- We can use circular linked list to implement Round-Robin Scheduling by the following method: rotate( ): Moves the first element to the end of the list.
- With this new operation, round-robin scheduling can be efficiently implemented by repeatedly performing the following steps on a circularly linked list C:
  1. Give a time slice to process C.first( )
  2. C.rotate( )

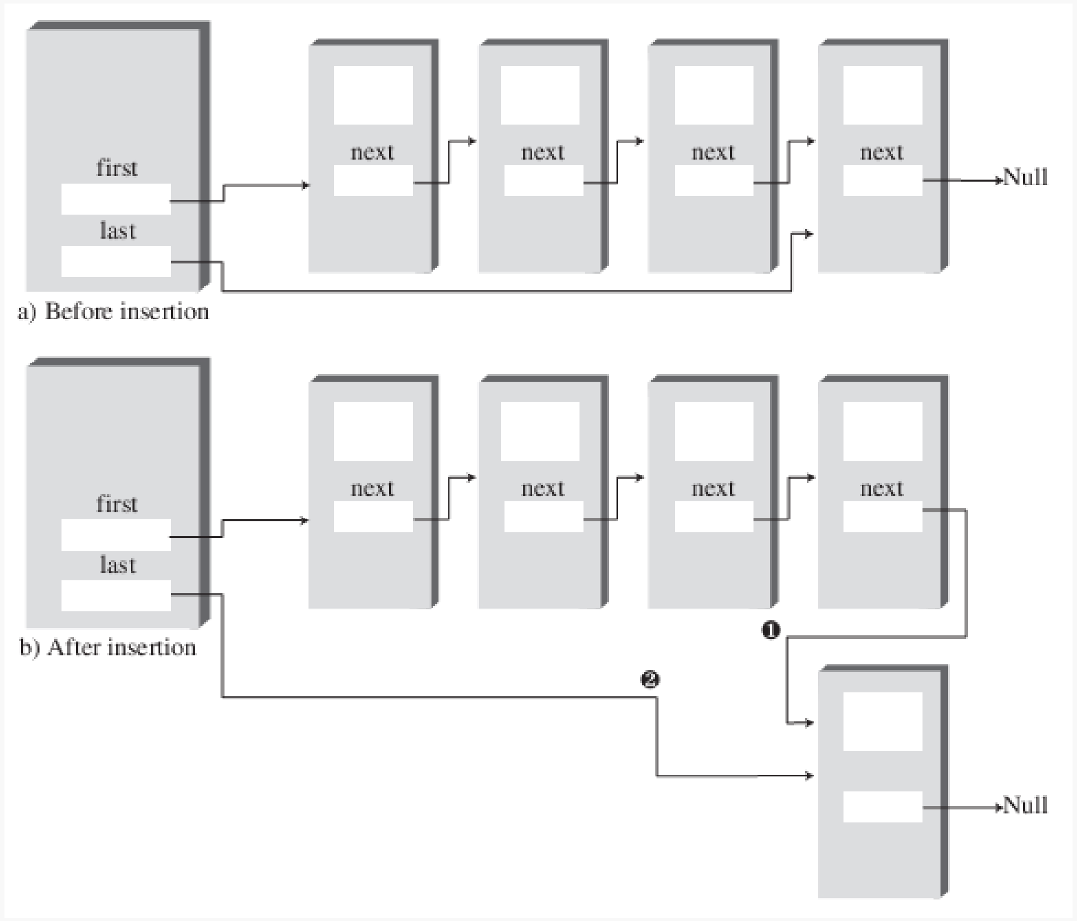
Double-Ended Lists

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)

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 detailed specifications or implementation*
- Abstract Data Type is a class considered without regard to its 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

```java
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)

➔ Useful for frequently access the minimum/maximum item application (Priority queue)

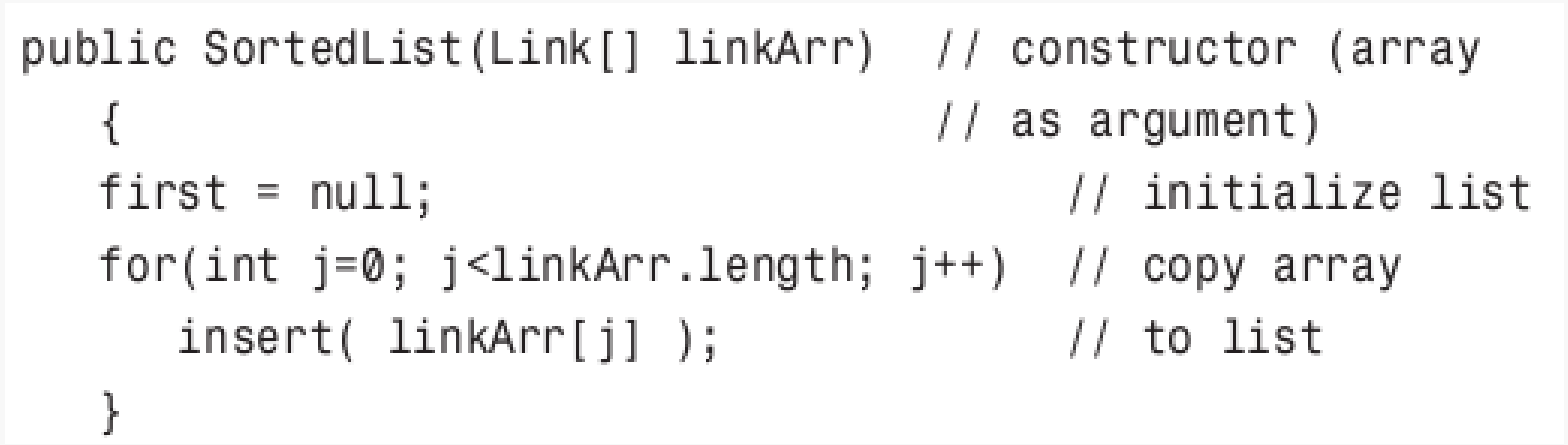
Application

- Sort an array - List Insertion Sort
  - Insert each item of an array to a sorted list
  - Get item from list and insert back to array

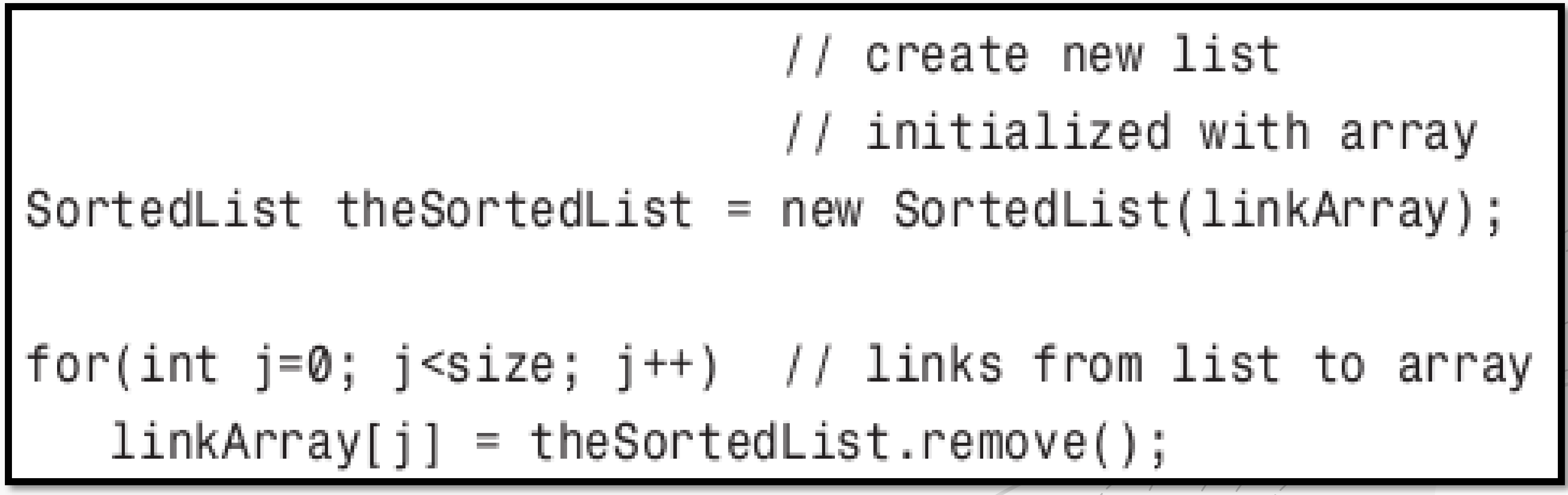
➔ Still O(n^2)

- But
  - Fewer copy/shift operation
  - N*2 vs N^2

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

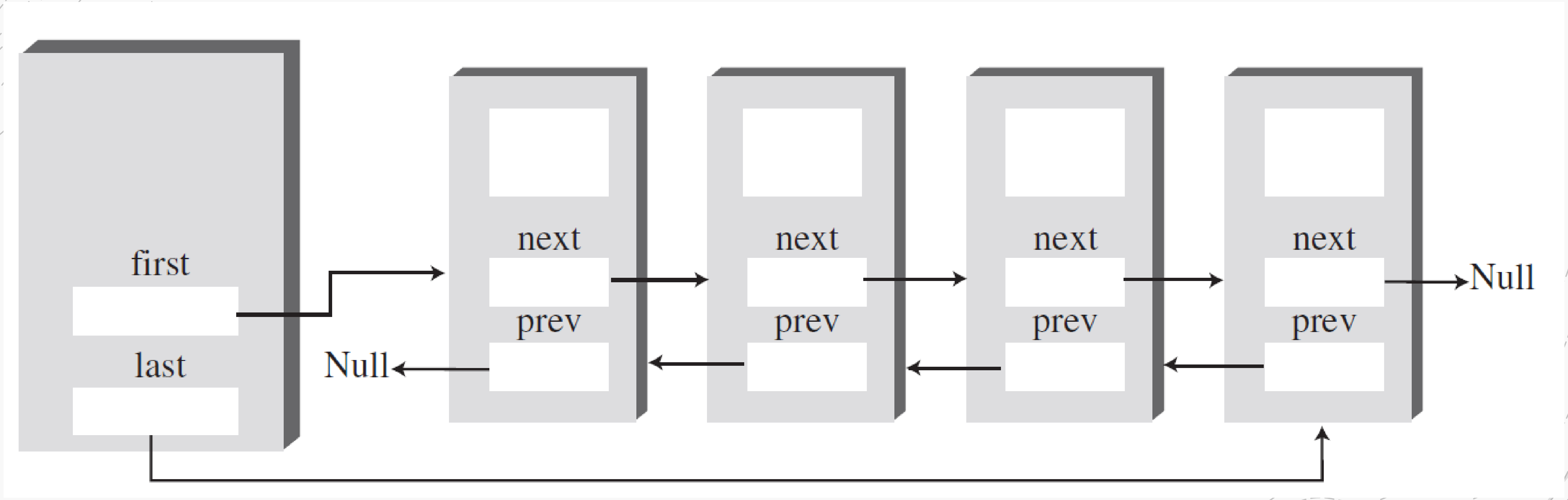
Introduction

- Singly linked list: One way traversing
  - current = current.next

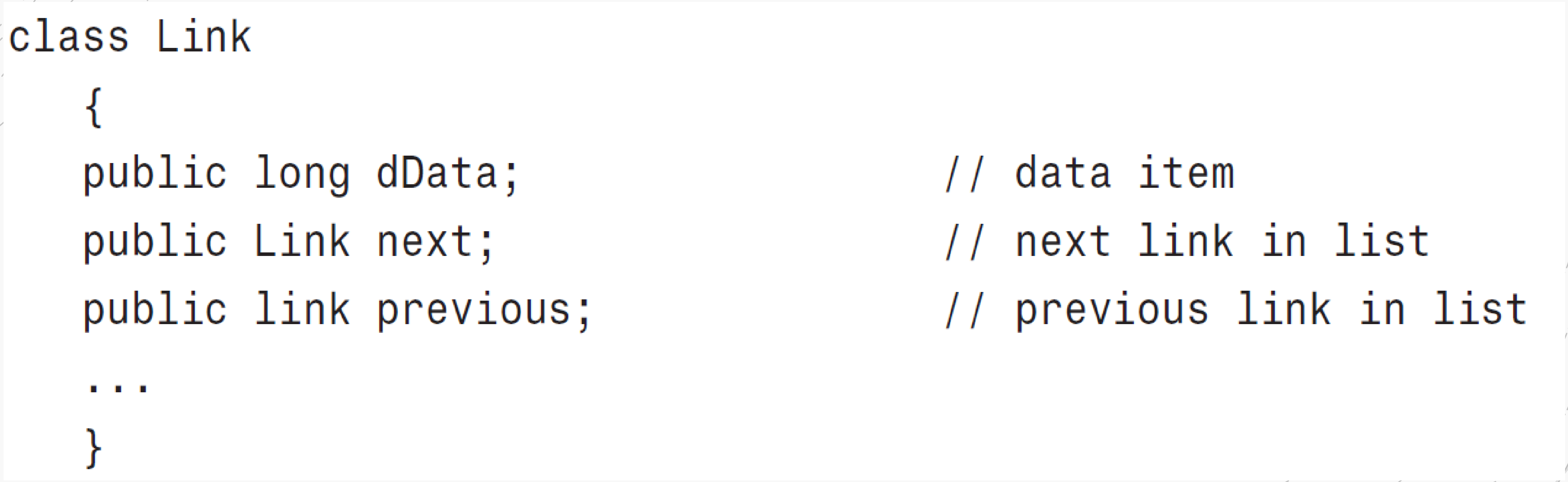
➔need to traverse backward as well as forward through the list

➔doubly linked list

Doubly linked list



Doubly linked list



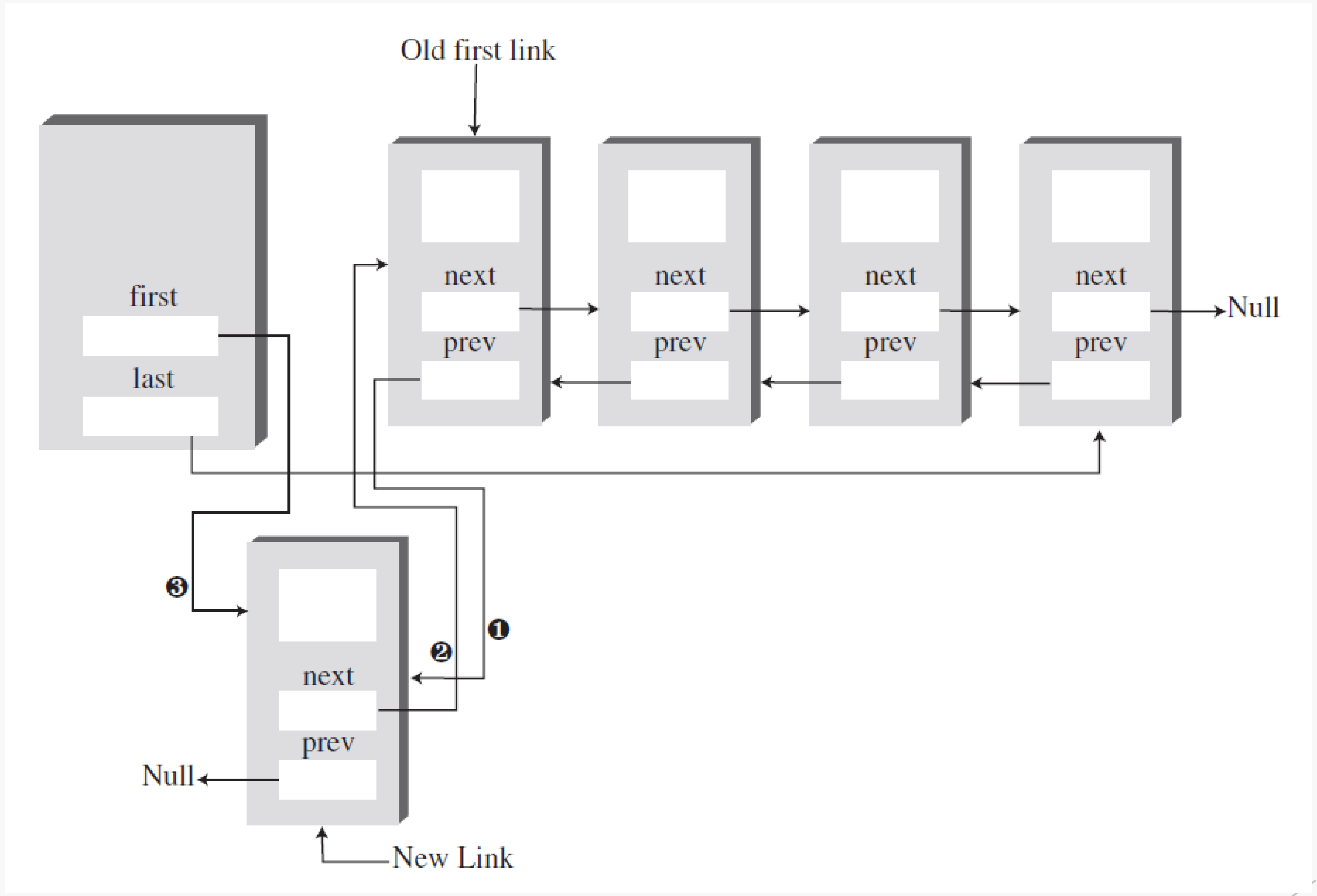
```
class Link
   {
   public long dData;                    // data item
   public Link next;                     // next link in list
   public link previous;                 // previous link in list
   ...
   }
```

Operations

- Insert
    - InsertFirst
    - InsertLast
    - InsertAfter
    - InsertBefore
- Display:
    - DisplayForward
    - DisplayBackward

- Delete
    - DeleteFirst
    - DeleteLast
    - Delete(key)

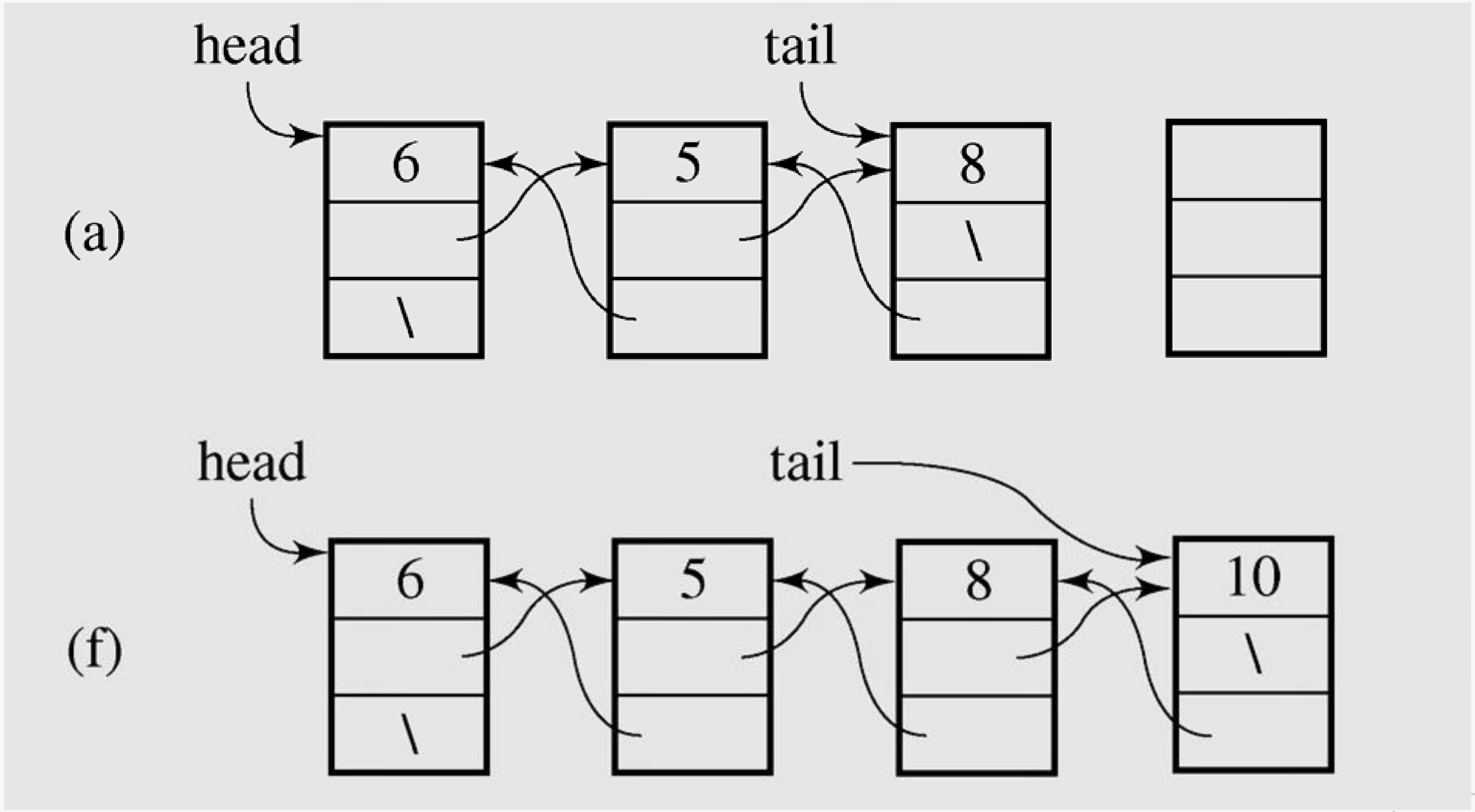
InsertFirst



InsertFirst

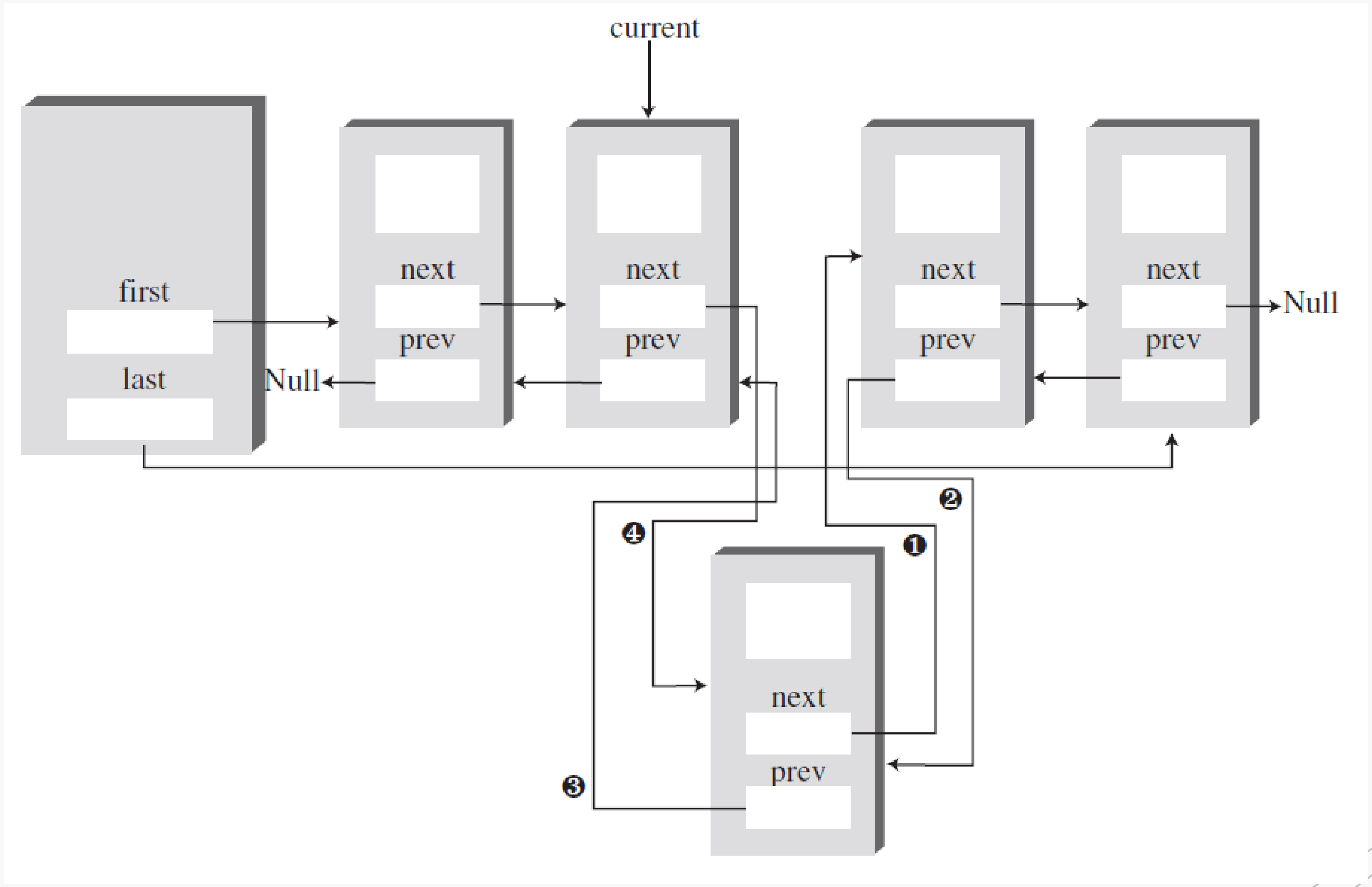
```
if( isEmpty() )                   // if empty list,
   last = newLink;                // newLink <-- last
else
   first.previous = newLink;      // newLink <-- old first
newLink.next = first;             // newLink --> old first
first = newLink;                  // first --> newLink
```

InsertLast

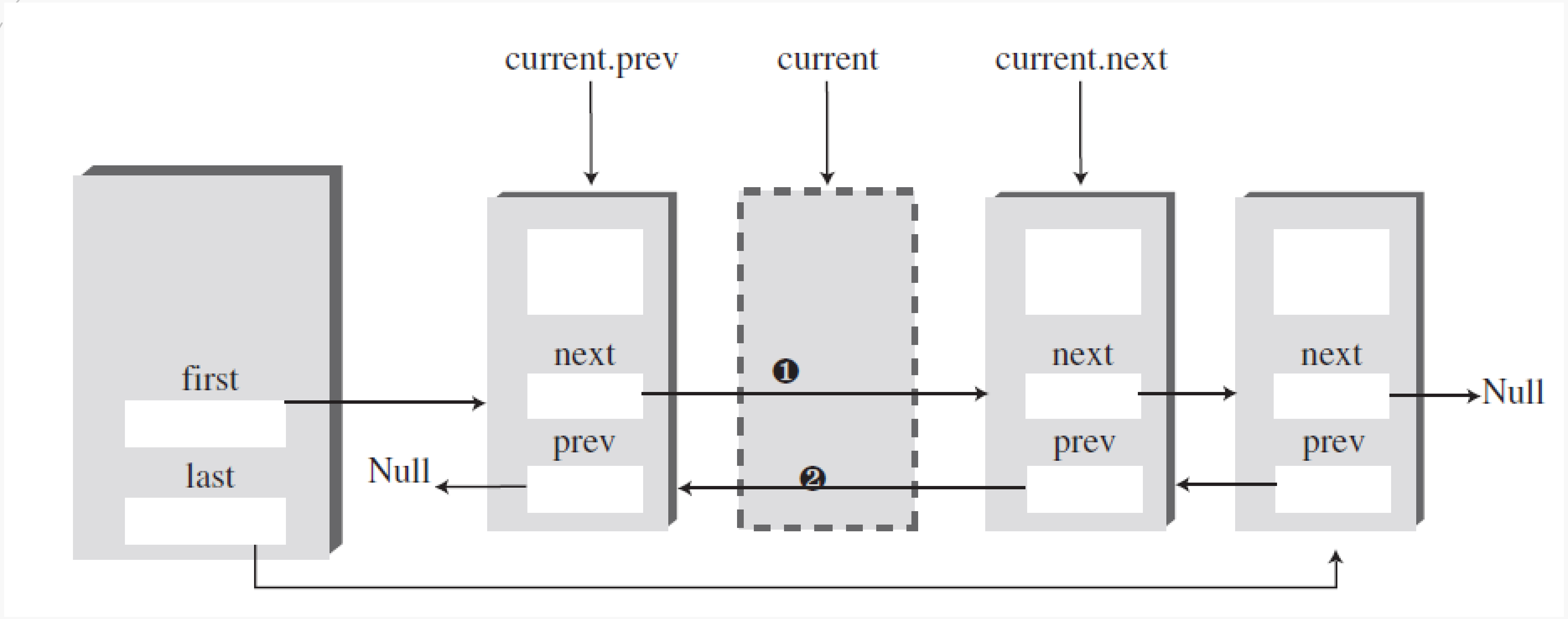


Adding new node at the end of Doubly Linked List

Insert in the middle of list



Delete an item



```
current.previous.next = current.next;
current.next.previous = current.previous;
```

Delete an item

- DeleteFirst?
- DeleteLast?

Application

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

Iterators

- Read "***Data Structure and Algorithm***", Robert Lafore, page 231

Lists in java.util - LinkedList class

boolean	**add**(E o) Appends the specified element to the end of this list.
void	**addFirst**(E o) Inserts the given element at the beginning of this list.
void	**addLast**(E o) Appends the given element to the end of this list.
void	**clear**() Removes all of the elements from this list.
E	**get(int index)** Returns the element at the specified position in this list.
E	**getFirst**() Returns the first element in this list.
E	**getLast**() Returns the last element in this list.
E	**remove(int index)** Removes the element at the specified position in this list.
E	**removeFirst**() Removes and returns the first element from this list.
E	**removeLast**() Removes and returns the last element from this list.
int	**size**() Returns the number of elements in this list.
Object[]	**toArray**() Returns an array containing all of the elements in this list in the correct order.

Lists in java.util
LinkedList class example

```
import java.util.*;
class Node {
      String name;
      int age;
      Node() {}

      Node(String name1, int age1) {
         name=name1; age=age1;
      }

      void set(String name1, int age1) {
        name=name1; age=age1;
      }

      public String toString() {
         String s = name+"  "+age;
           return(s);
      }
}
```

```
class Main
{
     public static  void main(String [] args)
     {
          LinkedList t = new LinkedList();

          Node x; int n,i;

          x = new Node("A01",25); t.add(x);

          x = new Node("A02",23); t.add(x);

          x = new Node("A03",21); t.add(x);

          for(i=0;i<t.size();i++)
            System.out.println(t.get(i));
     }
}
```

Lists in java.util - ArrayList class

boolean	**add**(E o) Appends the specified element to the end of this list.
void	**add(int index,** E o) Inserts the given element at the specified pos.
void	**clear()** Removes all of the elements from this list.
E	**get(int index)** Returns the element at the specified position in this list.
E	**remove(int index)** Removes the element at the specified position in this list.
int	**size()** Returns the number of elements in this list.
void	**ensureCapacity**(int minCapacity) Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
void	trimToSize() Trims the capacity of this ArrayList instance to be the list's current size.
Object[]	**toArray()** Returns an array containing all of the elements in this list in the correct order.

Summary

- A linked list consists of one ***linkedList*** object and a number of ***Link*** objects.
- The ***linkedList*** object contains a reference, often called ***first***, to the first link in the list.
- Each ***Link*** object contains data and a reference, often called ***next***, to the next link in the list.
- A ***next*** value of null signals the end of the list.
- Inserting an item at the beginning of a linked list involves changing the new link's ***next*** field to point to the old ***first*** link and changing ***first*** to point to the new item.

Summary

- Deleting an item at the beginning of a list involves setting ***first*** to point to ***first.next***.
- To traverse a linked list, you start at ***first*** and then go from link to link, using each link's ***next*** field to find the next link.
- A link with a specified key value can be found by traversing the list. Once found, an item can be displayed, deleted, or operated on in other ways.
- A new link can be inserted before or after a link with a specified key value, following a traversal to find this link.

Summary

- A double-ended list maintains a pointer to the last link in the list, often called ***last***, as well as to the ***first***.
- A double-ended list allows insertion at the end of the list.
- An Abstract Data Type (ADT) is a data storage class considered without reference to its implementation.
- Stacks and queues are ADTs. They can be implemented using either arrays or linked lists.

Summary

- In a sorted linked list, the links are arranged in order of ascending (or sometimes descending) key value.
- Insertion in a sorted list takes ***O(N)*** time because the correct insertion point must be found. Deletion of the smallest link takes ***O(1)*** time.
- In a doubly linked list, each link contains a reference to the previous link as well as the next link.
- A doubly linked list permits backward traversal and deletion from the end of the list.

Practice

- DoublyLinkedApp.java
- Complete the functions:
  - insertFirst
  - insertLast
  - deleteFirst
  - deleteLast
  - deleteKey
  - displayForward
  - displayBackward

Vietnam National University of HCMC

International University

School of Computer Science and Engineering

THANK YOU

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn

https://vichithanh.github.io

SCAN ME