Linear Data Structures

CE1103 - Algorithms and Data Structures I

Disclaimer / Descargo de Responsabilidad

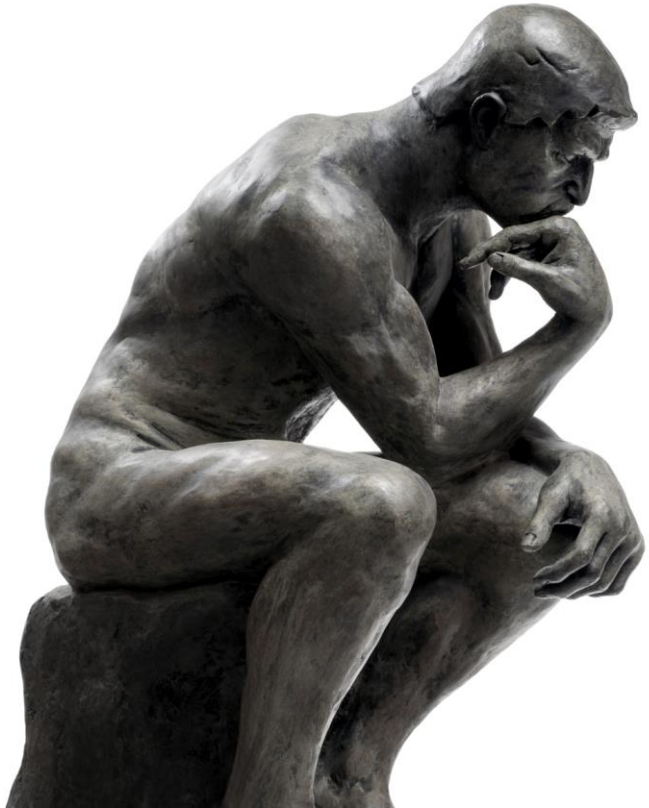
Esta presentación corresponde a una guía usada por el profesor durante las clases. La misma ha sido modificada para ser utilizado en el modelo de cursos asistidos por tecnología. No es una versión final, por lo que la misma podría requerir todavía hacer algunos ajustes. Para aspectos de evaluación esta presentación es solo una guía, por lo que el estudiante debe profundizar con el material de lectura asignado y lo discutido en clases para aspectos de evaluación.

This presentation corresponds to a guide material used by the professor during classes. It has been modified to be used in the model of technology-assisted courses. It is not a final version, so it may still require some adjustments. For evaluation aspects, this presentation is only a guide, so the student should delve with the assigned reading material and what has been discussed in class.

Before we begin...

➔ What is an **A**bstract **D**ata **T**ype (ADT)?

➔ It is a key concept to understand what is a data structure.



Abstract Data Types

➔ It is a set of objects together with a set of operations.

➔ ADTs are mathematical abstractions. Their definition does not include any way to implement the operations.

Abstract Data Types

➔ When we say data type, we think of:
- Integers
- Reals
- Booleans

➔ Each of those have operations associated with them.

Abstract Data Types

➔ Thinking on ADTs we have:
- Lists
- Queues
- Sets
- Graphs

Abstract Data Types

➔ Just like any other data type, ADTs also have operations associated:

- Add
- Remove
- Contains

Abstract Data Types

➔ How can we implement an ADT in Java?
- ◆ Classes can be used to implement ADTs because they allow to hide the implementation details.
- ◆ Doing operations on the ADTs is just calling methods of the class.

➔ ADTs is not an OO concept.

➔ In the OO context, an ADT can be the same as an interface or a class with some characteristics.

Abstract Data Types

➔ Which operations can be should be inside an ADT?

- That is a design decision of the programmer.
- When implementing an ADT be careful of not exposing implementation details.

Arrays

Arrays

➔ They are objects to a very limited extent.

➔ There is no keyword with which all arrays are declared.

➔ They may be considered instances of an understood array class.

Arrays

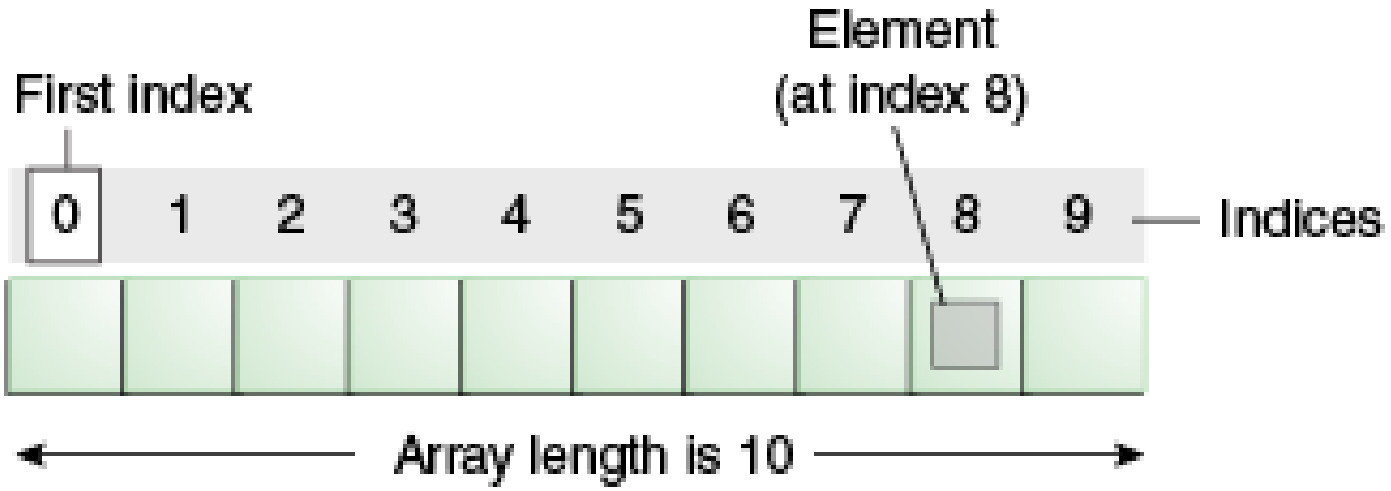
➔ How to declare

```
int[] a;
int a[];

int[] a = new int[10];
int[] b = {5, 4, 2, 1};
```

Arrays

➔ How to access elements



Arrays

LIMITATION

- ➔ Changing the size of the array is expensive
  - ◆ Create a new array
  - ◆ Copying all data from the array with the old size to the array with the new size.

Arrays

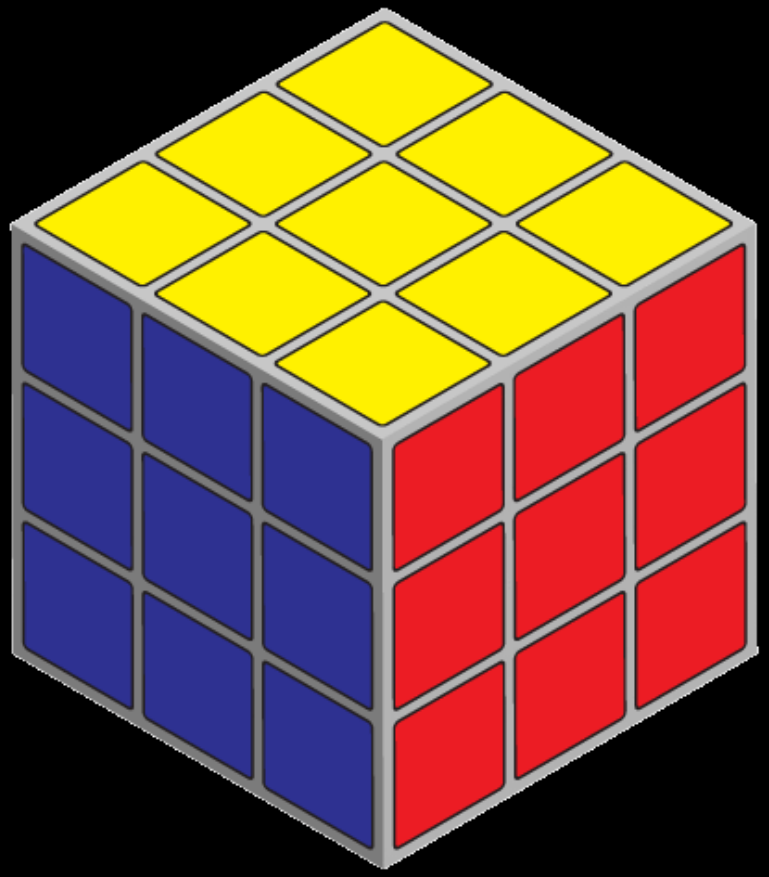
LIMITATION

- ➔ Changing the size of the array is expensive
  - ◆ Create a new array
  - ◆ Copying all data from the array with the old size to the array with the new size.

OTHER LIMITATION

- ➔ The data in the array are next to each other sequentially in memory
  - ◆ Inserting an item inside the array requires shifting some other data in this array.

Matrix



Matrix

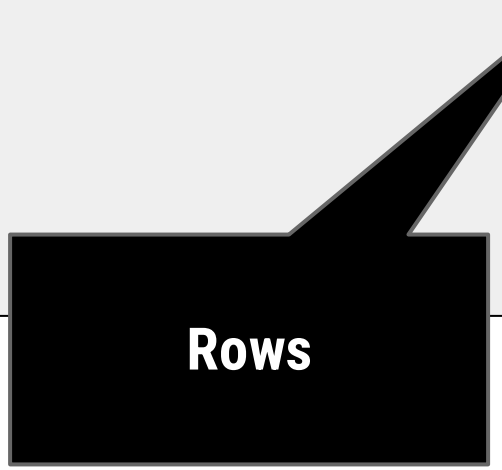
➔ A matrix can be thought as a two-dimensional array or an array of arrays

```
int [][] nums = new int[5][4];
```

Matrix

➔ A matrix can be thought as a two-dimensional array or an array of arrays

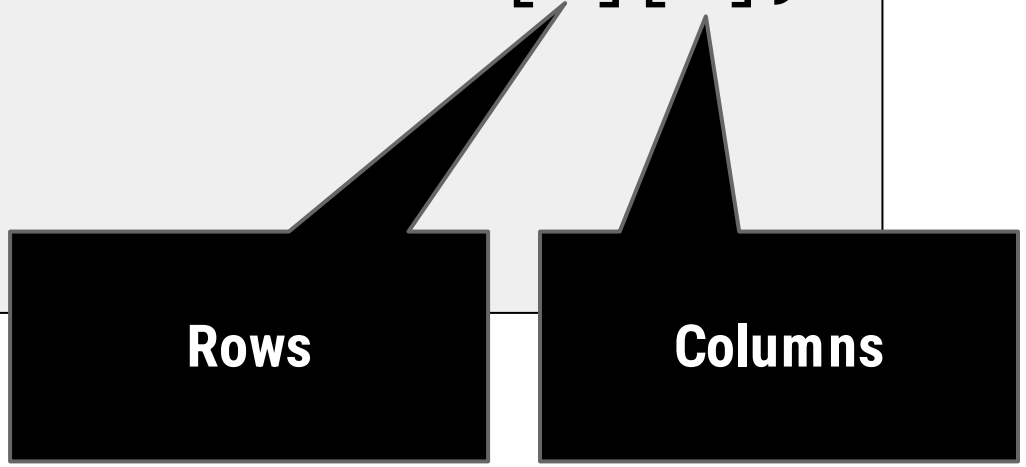
```
int [][] nums = new int[5][4];
```



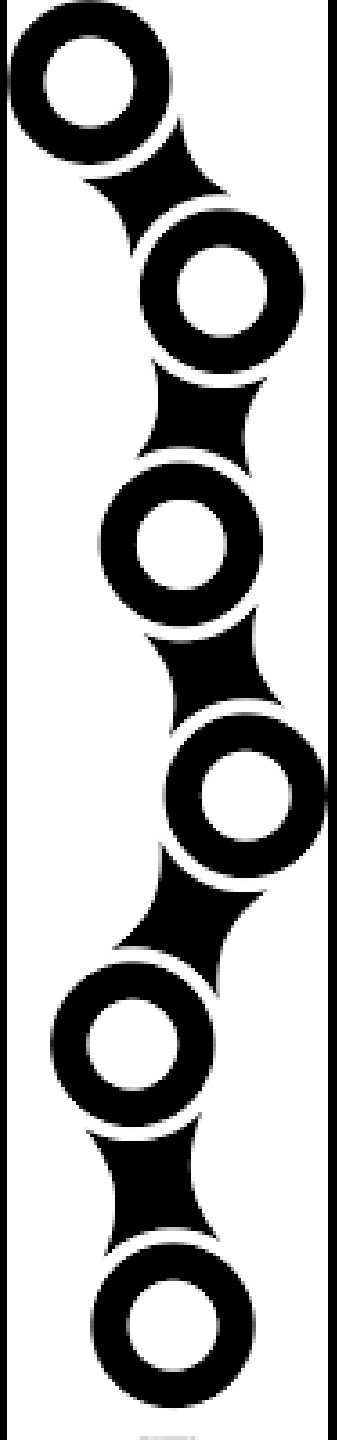
Matrix

➔ A matrix can be thought as a two-dimensional array or an array of arrays

```
int [][] nums = new int[5][4];
```



Linked Lists



Linked List

➔ A linear data structure where each element (node) is a separate object and **not necessarily adjacent in memory**.

➔ A dynamic data structure.

- The number of nodes is not fixed.
- A list can grow and shrink on demand.
- Any application which has to deal with an unknown number of objects will need to use a linked list.

Linked List

➔ A node contains a data field that is a reference to another node. **This is how the nodes are linked**.

➔ So each node holds is composed of data and a reference to another node in the list.

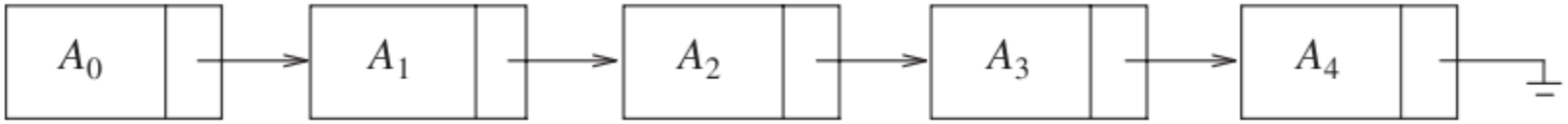
➔ The last node of the list “points” to null in the node reference field.

Linked List

➔ A node contains a data field that is a reference to another node. **This is how the nodes are linked**.

➔ So each node holds is composed of data and a reference to another node in the list.

➔ The last node of the list “points” to null in the node reference field.



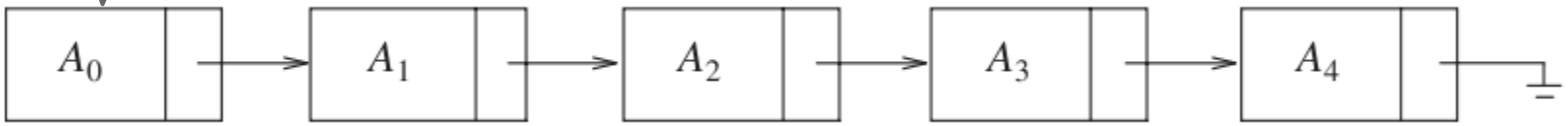
Linked List

➔ A node contains a data field that is a reference to another node. **This is how the nodes are linked**.

➔ So each node holds is composed of data and a reference to another node in the list.

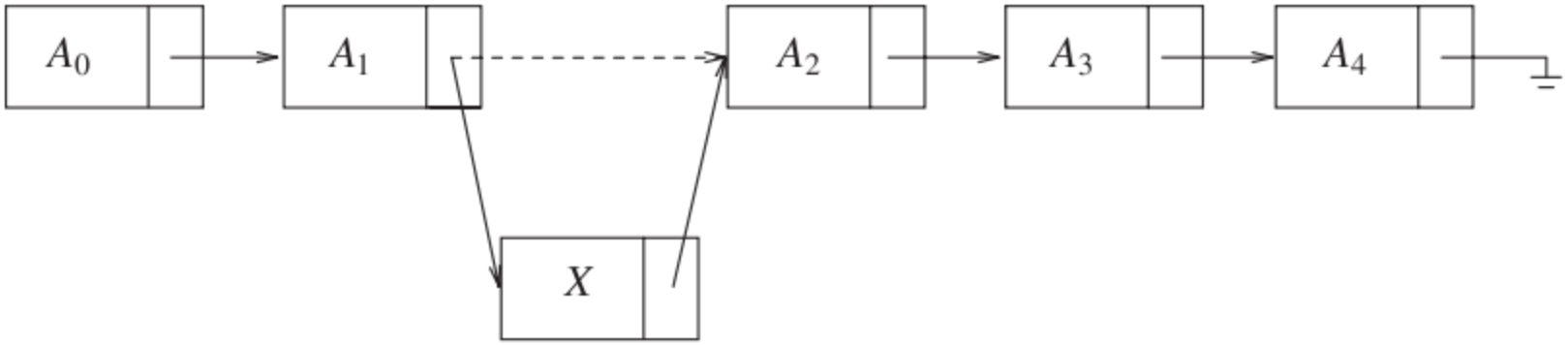
Always do the drawing!

➔ The last node of the list "points" to null in the node reference



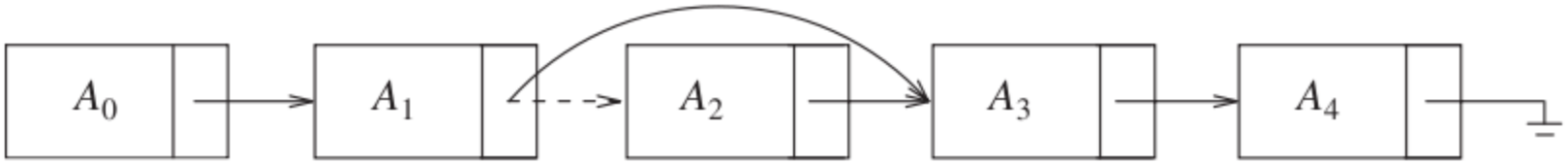
Linked List

➔ Inserting an element to a list

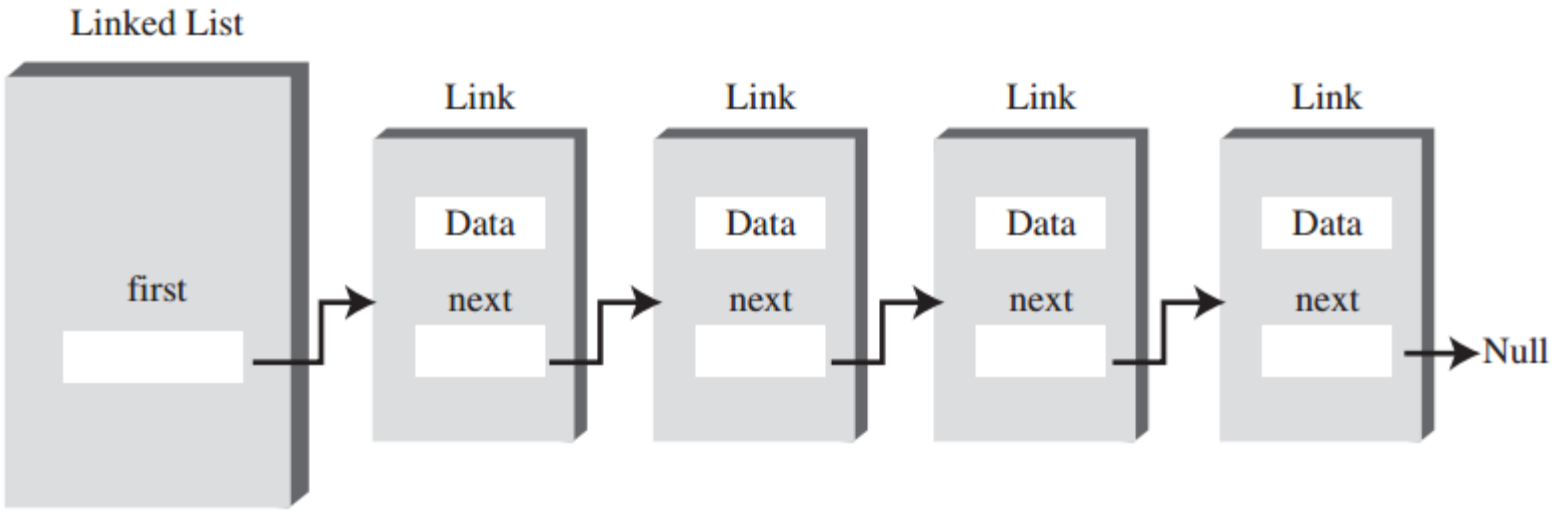


Linked List

➔ Deleting an element from a list



Linked List


Linked List
first
Link
Data
next
Link
Data
next
Link
Data
next
Link
Data
next
Null

Linked List

➔ Implementing a linked list

- There is **more than one specific way to implement a linked list**.
- Just like most things in computing, there is more than one way of solving problems.

Linked List

➔ Implementing a linked list

```
01  class Node {
02    private Object data;
03    private Node next;
04
05    public Node(Object data) {
06      this.next = null;
07      this.data = data;
08    }
09
10    public Object getData() {
11      return this.data;
12    }
13
14    public void setData(Object data) {
15      this.data = data;
16    }
```

Linked List

➔ Implementing a linked list

```
01    public Node getNext() {
02      return this.next;
03    }
04
05    public void setNext(Node node) {
06      this.next = node;
07    }
08  }
09
10
11
12
13
14
15
16
```

Linked List

➔ Implementing a linked list

```
01  class LinkedList {
02    private Node head;
03    private int size;
04
05    public LinkedList() {
06      this.head = null;
07      this.size = 0;
08    }
09
10    public boolean isEmpty() {
11      return this.head == null;
12    }
13
14    public int size() {
15      return this.size;
16    }
```

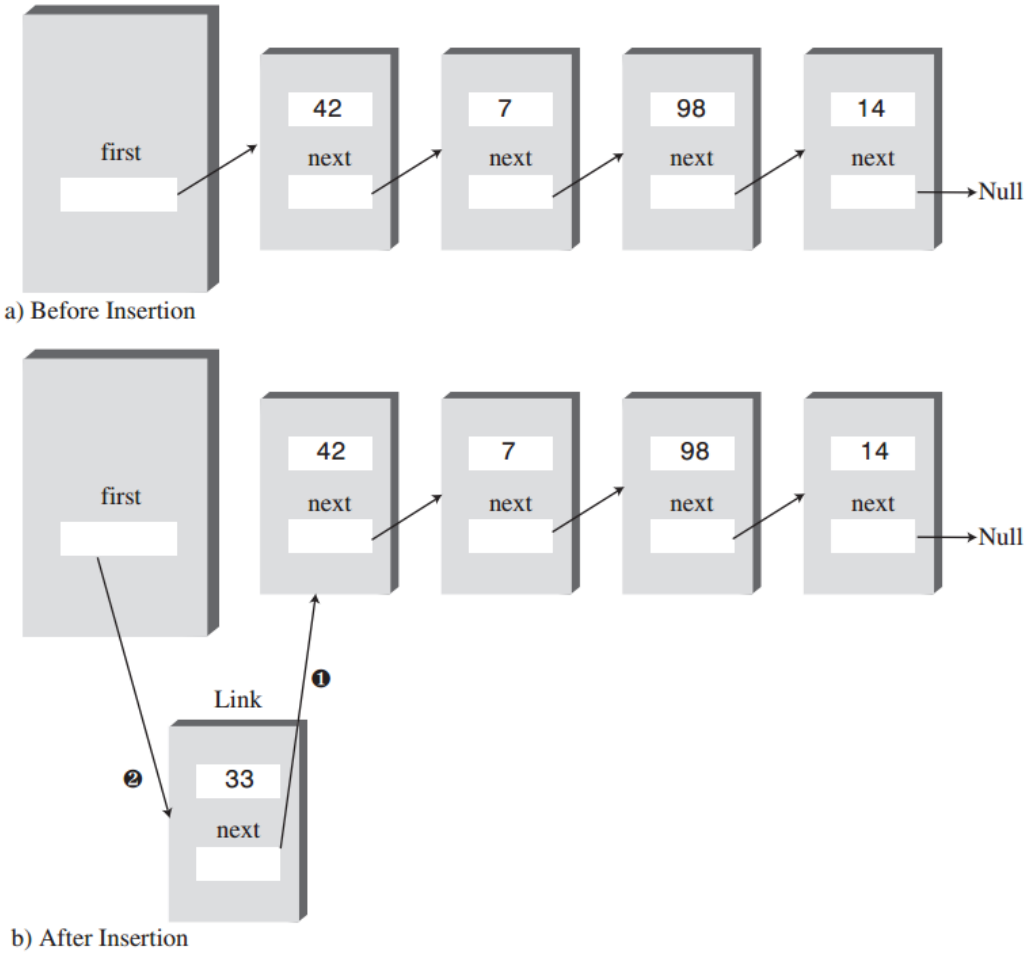
Linked List

➔ Implementing a linked list

```
01    public void insertFirst(Object data) {
02      Node newNode = new Node(data);
03      newNode.next = this.head;
04      this.head = newNode;
05      this.size++;
06    }
07
08    public Node deleteFirst() {
09      if (this.head != null) {
10        Node temp = this.head;
11        this.head = this.head.next;
12        this.size--;
13        return temp;
14      } else {
15        return null;
16      }
17    }
```

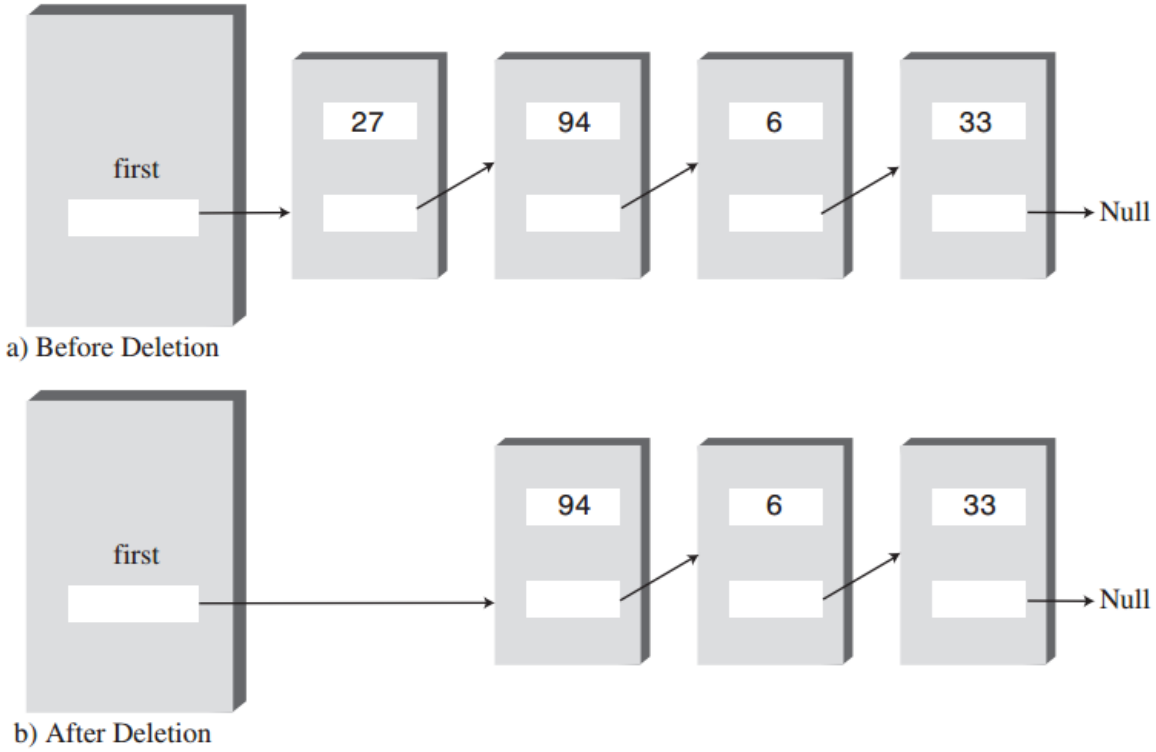
Linked List

➔ Implementing a linked list



Linked List

➔ Implementing a linked list



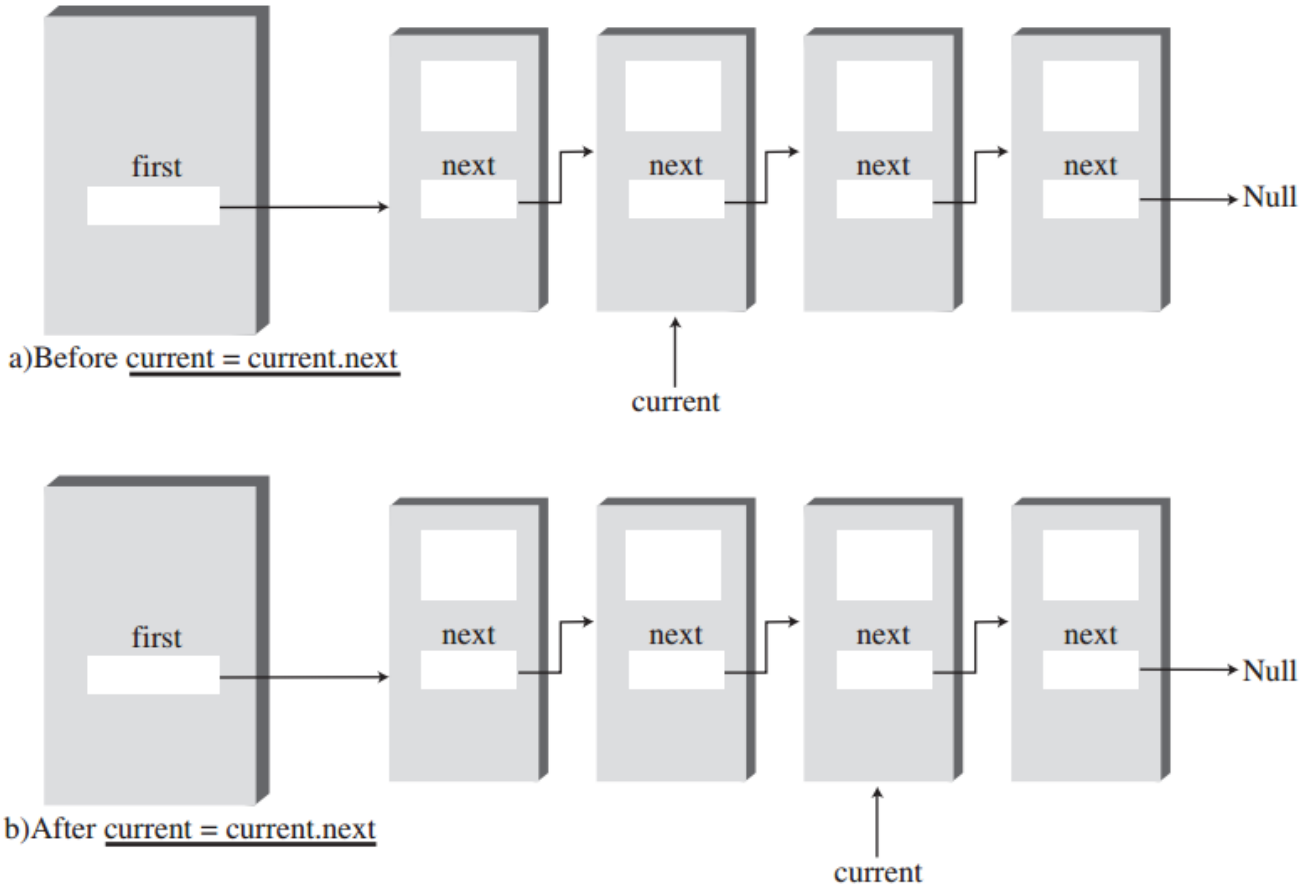
Linked List

➔ Implementing a linked list

```
01    public void displayList() {
02      Node current = this.head;
03      while (current != null) {
04        System.out.println(current.getData());
05        current = current.getNext();
06      }
07    }
08
09
10
11
12
13
14
15
16
```

Linked List

➔ Implementing a linked list



Linked List

➔ Implementing a linked list

```
01    public Node find(Object searchValue) {
02      Node current = this.head;
03      while (current != null) {
04        if (current.getData().equals(searchValue)) {
05          return current;
06        } else {
07          current = current.getNext();
08        }
09      }
10      return null;
11    }
12
13
14
15
16
```

Linked List

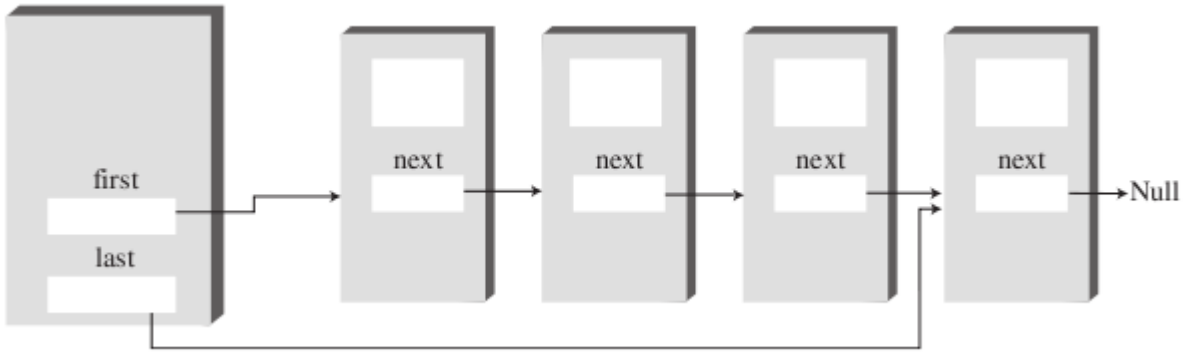
➔ Implementing a linked list

```
01    public Node delete(Object searchValue) {
02      Node current  = this.head;
03      Node previous = this.head;
04
05      while (current != null) {
06        if (current.getData().equals(searchValue)) {
07          if (current == this.head) {
08            this.head = this.head.getNext();
09          } else {
10            previous.setNext(current.getNext());
11          }
12          return current;
13        } else {
14          previous = current;
15          current = current.getNext();
16        }
17      }
18      return null;
19    }
```

Linked List

➔ Implementing a linked list

- To increase performance, specially when inserting at the end, you can **add a reference to the last node (double-ended list)**.



Linked List

➔ Implementing a linked list

```
01  class DoubleEndedList {
02    private Node head;
03    private Node last;
04    private int size;
05
06    public LinkedList() {
07      this.head = null;
08      this.last = null;
09      this.size = 0;
10    }
11
12    public boolean isEmpty() {
13      return this.head == this.last == null;
14    }
15
16    public int size() {
17      return this.size;
18    }
```

Linked List

➔ Implementing a linked list

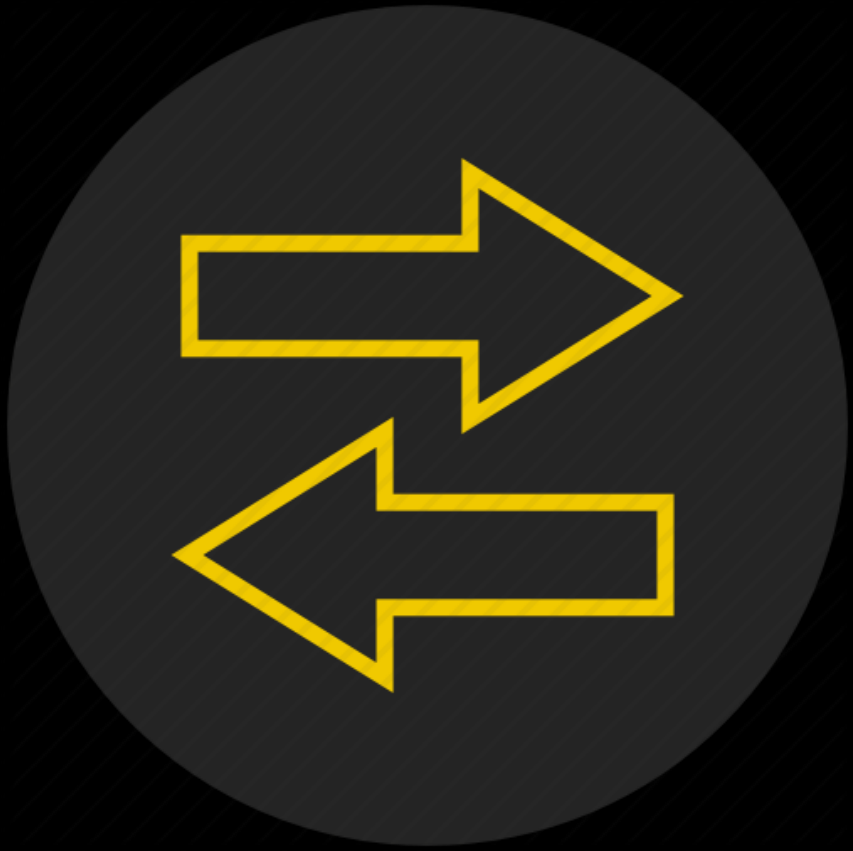
```
01    public void insertFirst(Object data) {
02      Node newNode = new Node(data);
03
04      if (this.isEmpty()) {
05        this.head = this.last = newNode;
06      } else {
07        newNode.setNext(this.head);
08        this.head = newNode;
09      }
10
11      this.size++;
12    }
13
14
15
16
17
18
```

Linked List

➔ Implementing a linked list

```
01    public void insertLast(Object data) {
02      Node newNode = new Node(data);
03
04      if (this.isEmpty()) {
05        this.head = this.last = newNode;
06      } else {
07        this.last.setNext(newNode);
08        this.last = newNode;
09      }
10
11      this.size++;
12    }
13
14
15
16
17
18
```

Doubly Linked Lists



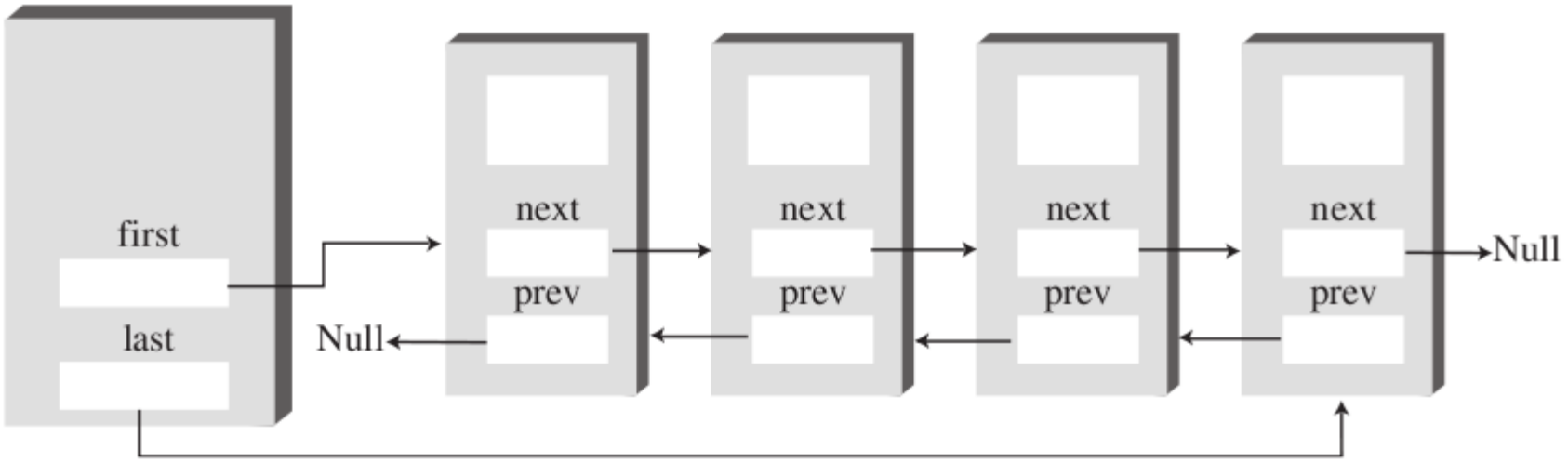
Doubly Linked Lists

➔ Is a variation of the simple list.

➔ The advantage is that ease backward navigation of the list.

➔ Each node has two references to other nodes instead of one. The first is a link to the next node, like a simple list. The second is the previous node.

Doubly Linked Lists



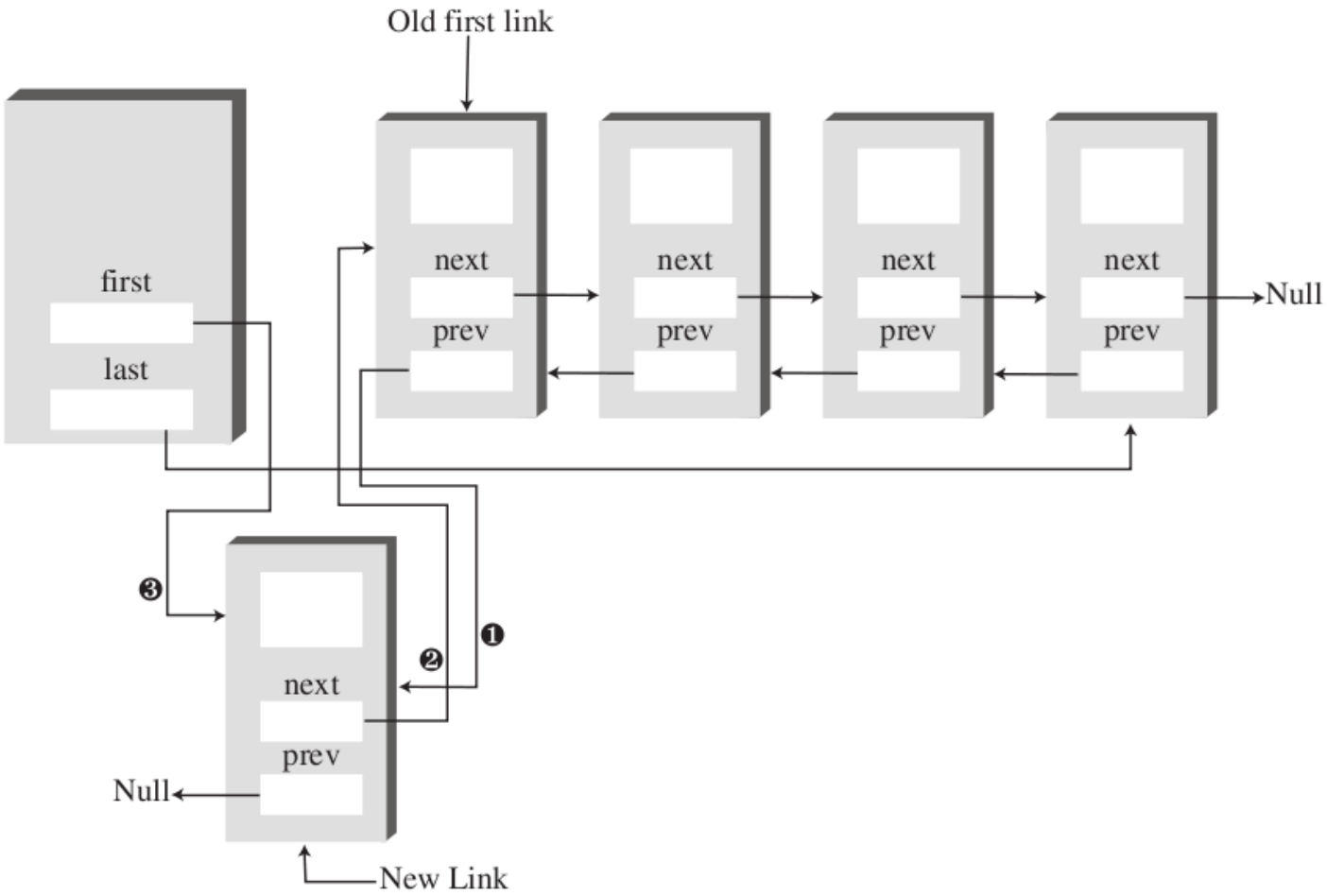
Doubly Linked Lists

➔ The downside of doubly linked list is dealing with more links while inserting / deleting nodes.

➔ Nodes are little bigger because they have an extra link

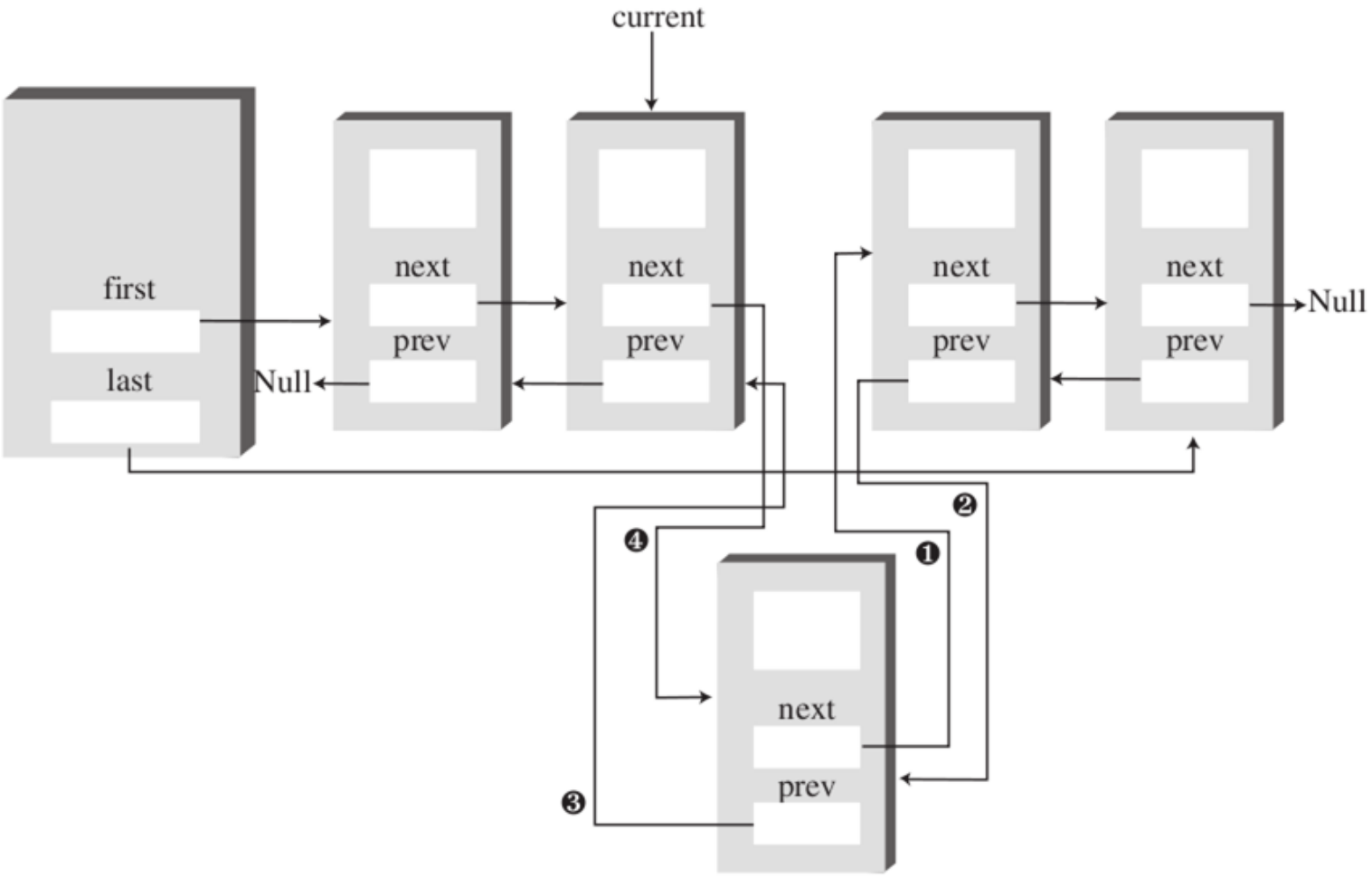
Doubly Linked Lists

➔ Insert at the beginning



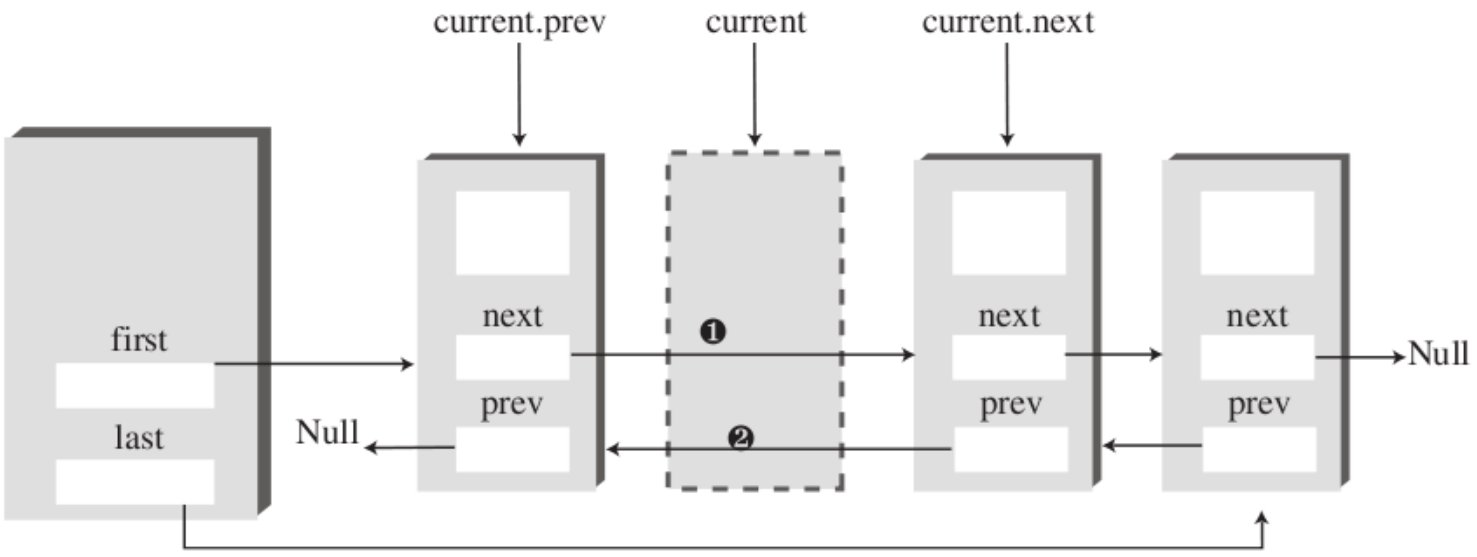
Doubly Linked Lists

➔ Insert at an arbitrary position



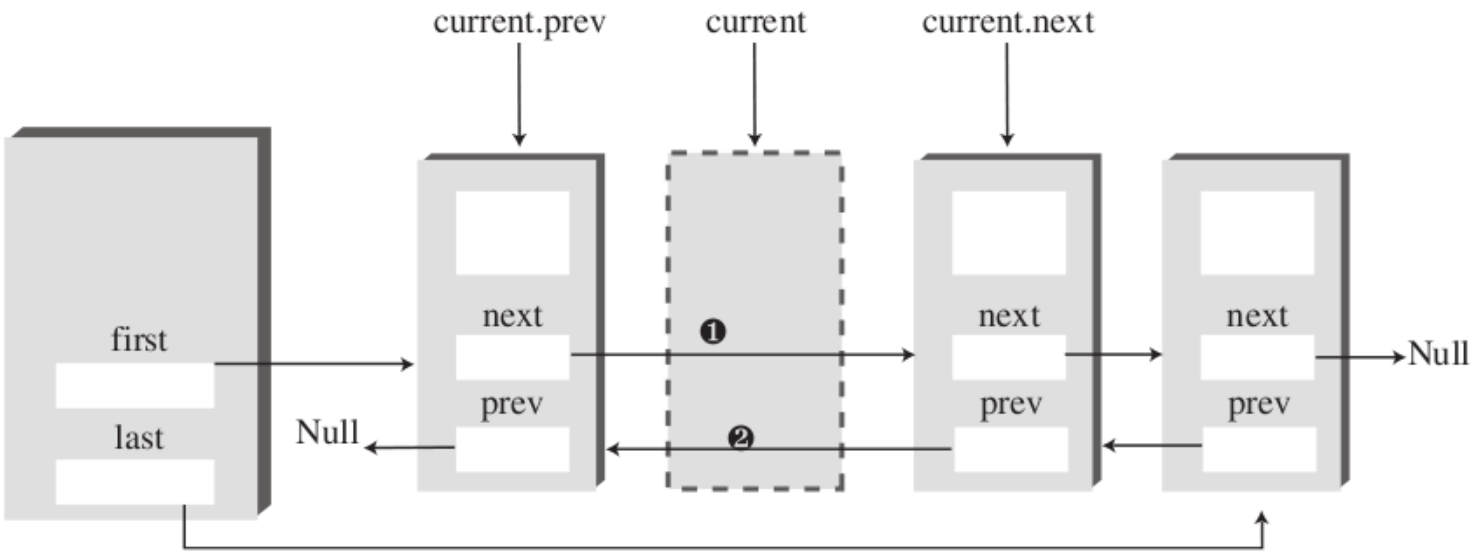
Doubly Linked Lists

➔ Delete at an arbitrary position



Doubly Linked Lists

➔ Delete at an arbitrary position



Circular List

Circular Lists

- ➔ Lists viewed so far have a linear nature: each node has a predecessor and each node has a unique successor.

- ➔ If we change the last node next reference to point to the head node, we will have a circular list.
  - ◆ We can traverse the whole list starting from any node.

- ➔ A circular list is a list in which every node has a successor and the “last” element is succeeded by the “first” element.

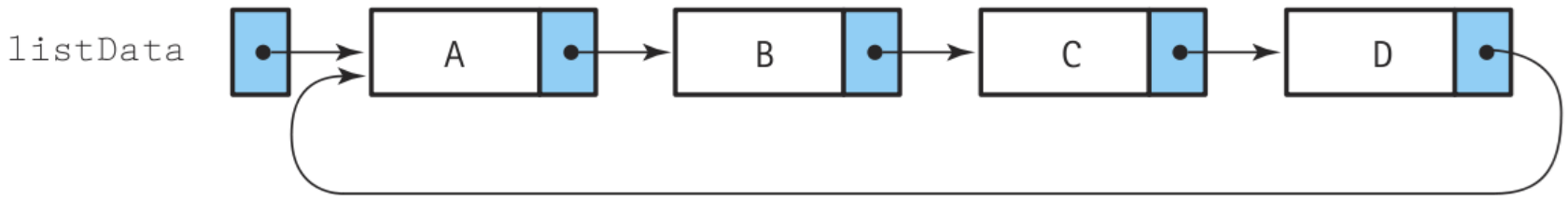
Circular Lists

➔ Lists viewed so far have a linear nature: each node has a predecessor and each node has a unique successor.

➔ If we change the last node next reference to point to the head node, we will have a circular list.

- ◆ We can traverse the whole list starting from any node.

➔ A circular list is a list in which every node has a successor and the “last” element is succeeded by the “first” element.



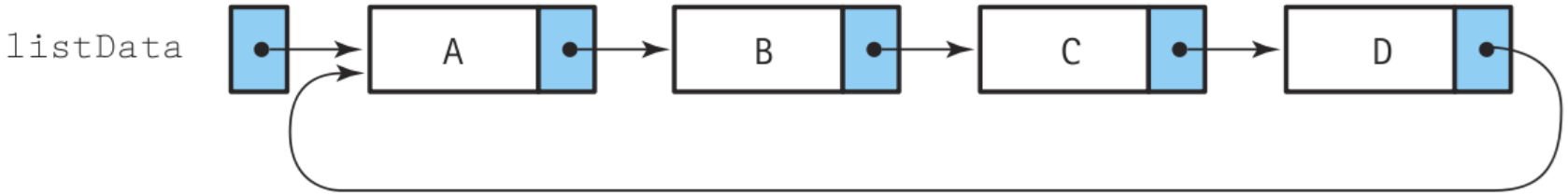
Circular Lists

➔ Lists viewed so far have a linear nature: each node has a predecessor and each node has a unique successor.

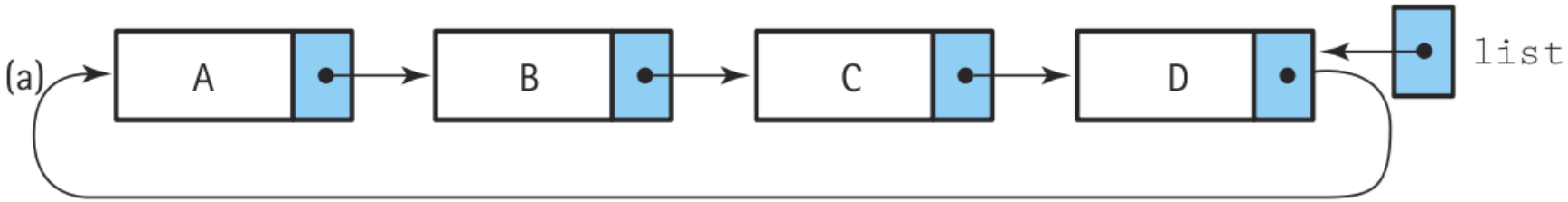
➔ If we change the last node next reference to point to the head

What problems can arise with having the head to the first node?

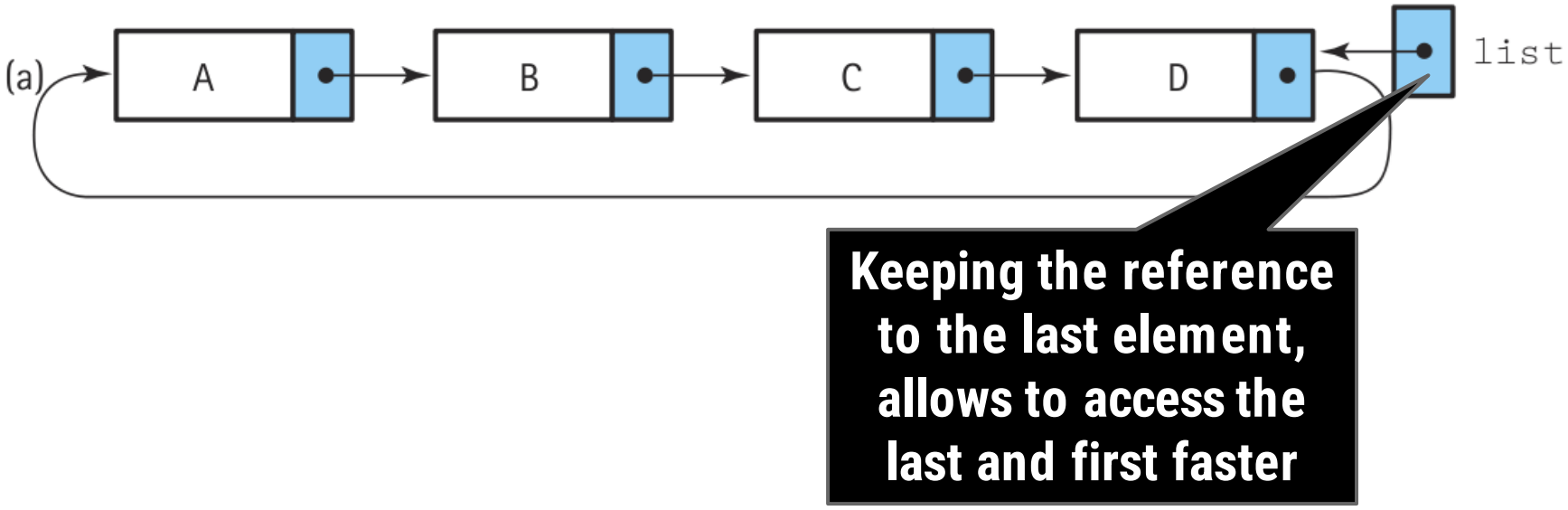
Deleting the first node, means traversing the whole list to update the last...



Circular Lists

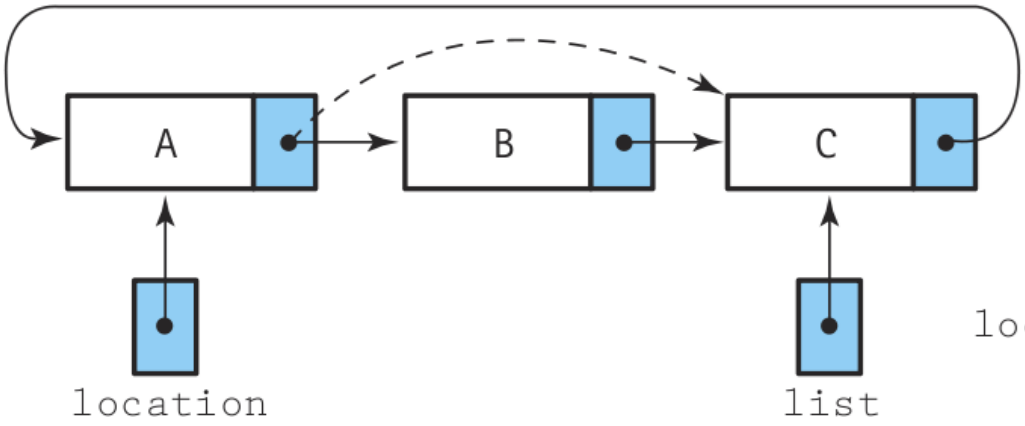


Circular Lists



Circular Lists

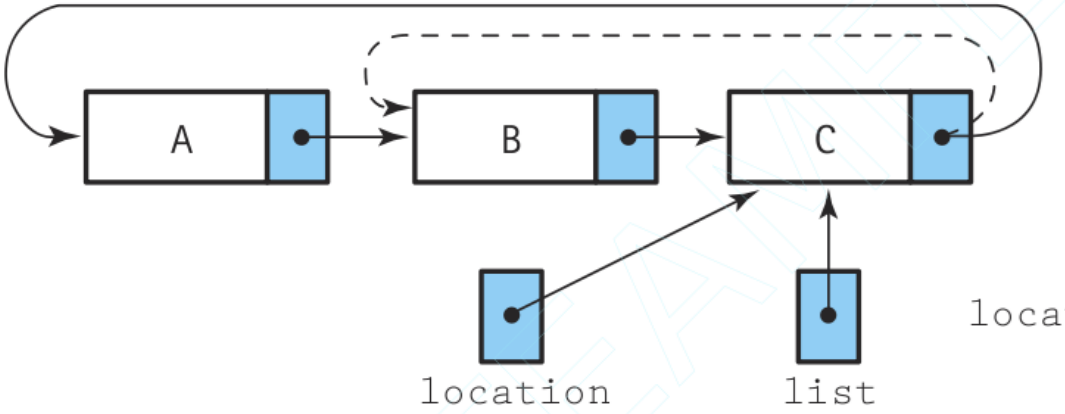
➔ Deleting




```
location.next = location.next.next;
```

Circular Lists

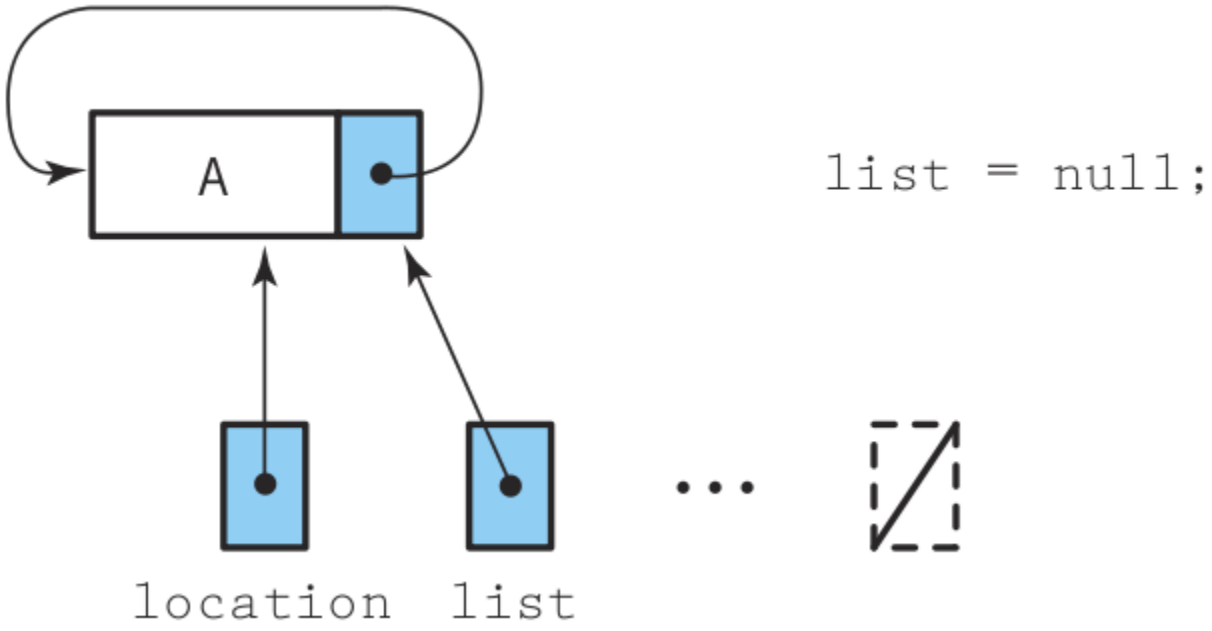
➔ Deleting




```
location.next = location.next.next;
```

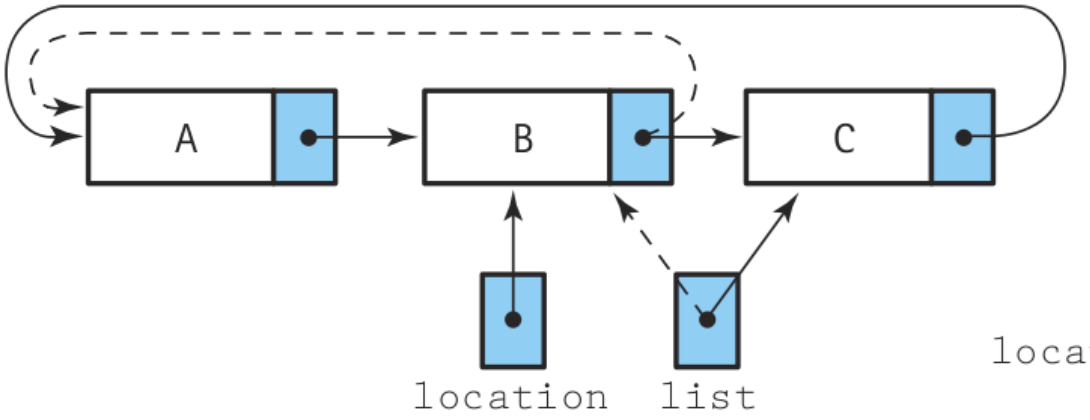
Circular Lists

➔ Deleting



Circular Lists

➔ Deleting



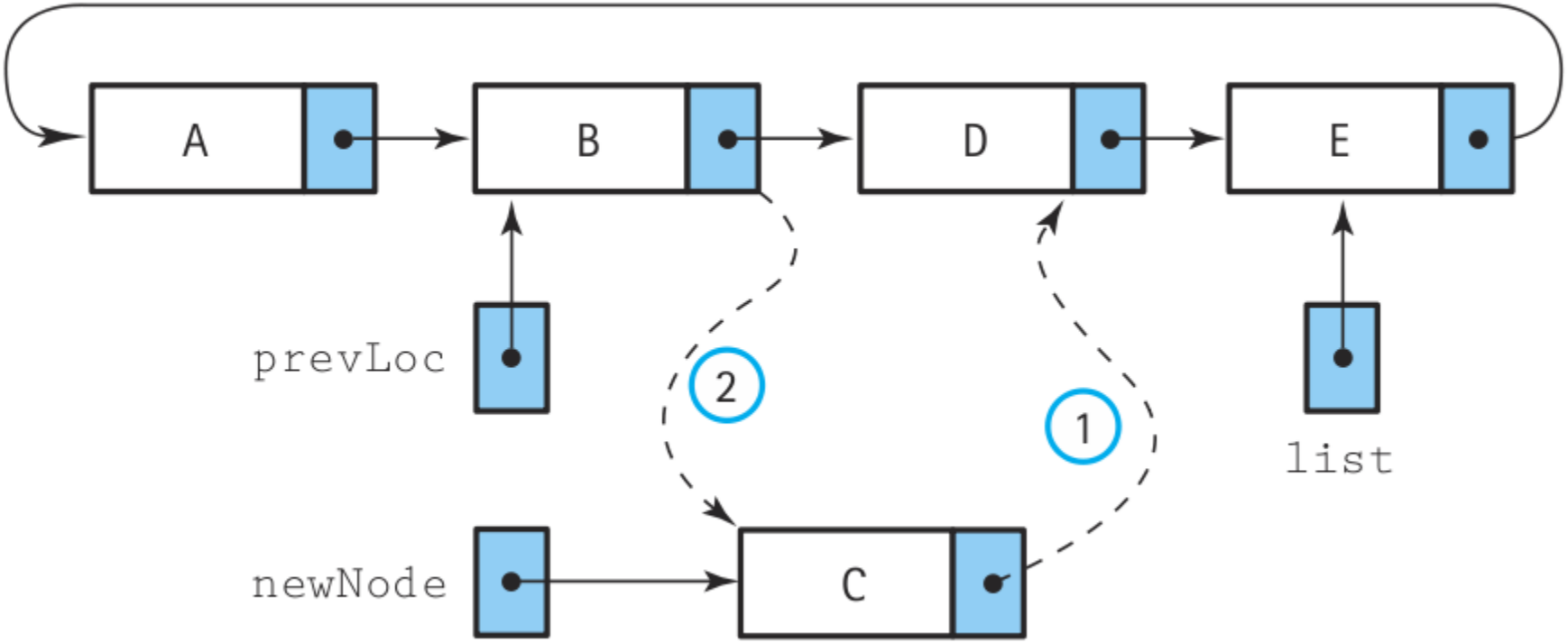

```
list = location;
```

(the general case PLUS:)

```
location.next = location.next.next;
```

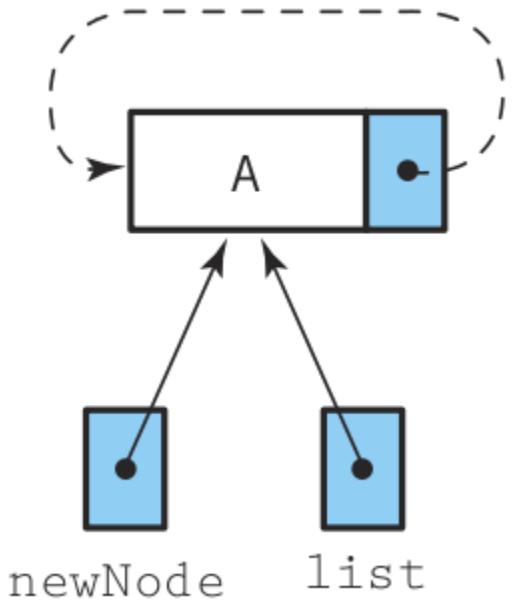
Circular Lists

➔ Inserting



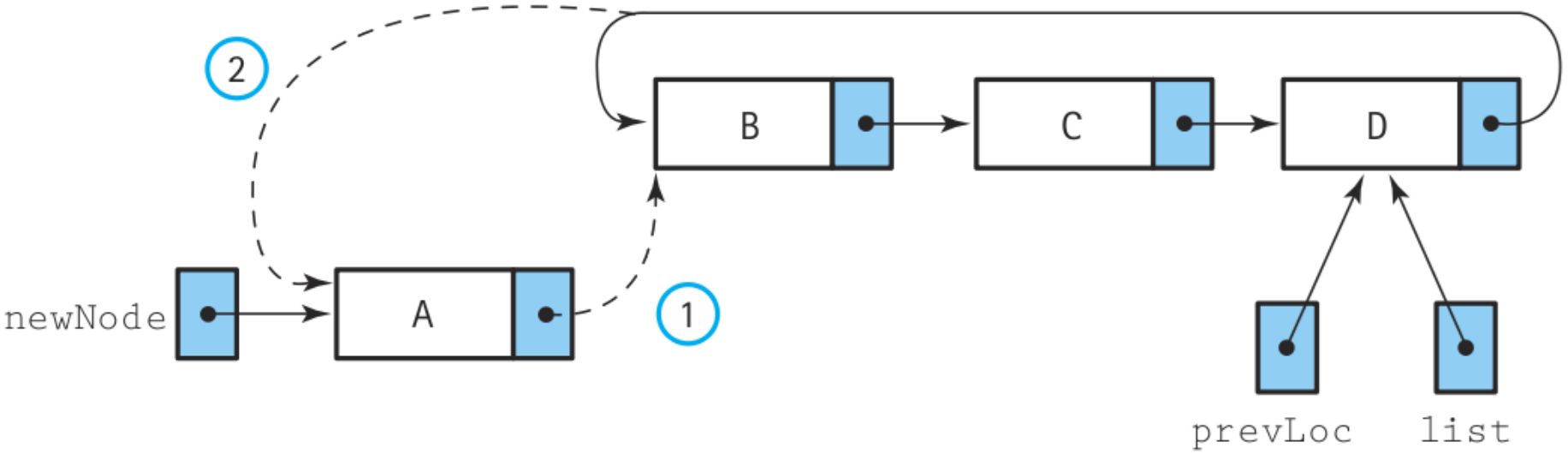
Circular Lists

➔ Inserting



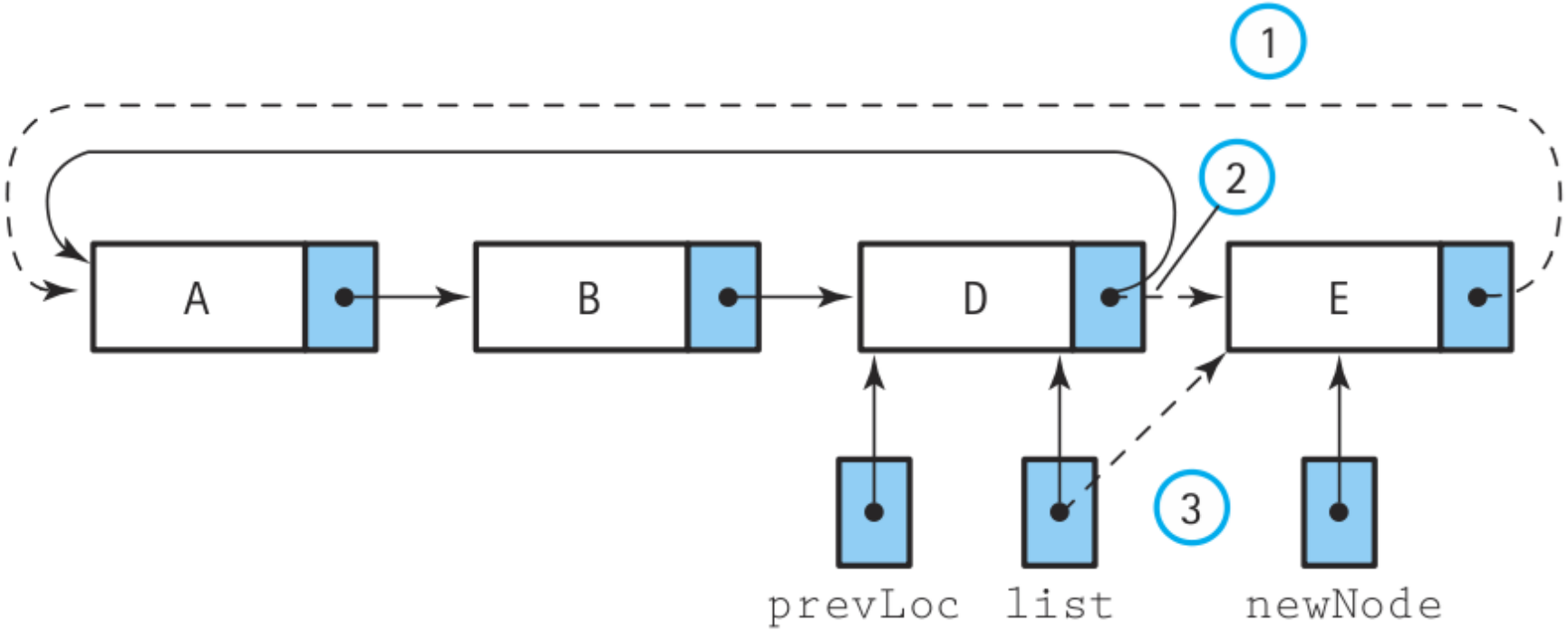
Circular Lists

➔ Inserting



Circular Lists

➔ Inserting



Circular Lists

➔ When to use a circular list

- Good for applications that needs to access both ends of the list.
- If the list is ordered, is easier to find the largest element and doing operations like inBetween().
- Can be more efficient for some specific tasks.

Stack

Stack

➔ A stack allows access to only one data item: **the last item inserted**.

➔ If you remove this item, you can access the next-to-last element inserted, and so on.

➔ **LIFO** (**L**ast **I**n, **F**irst **O**ut) nature.

Stack

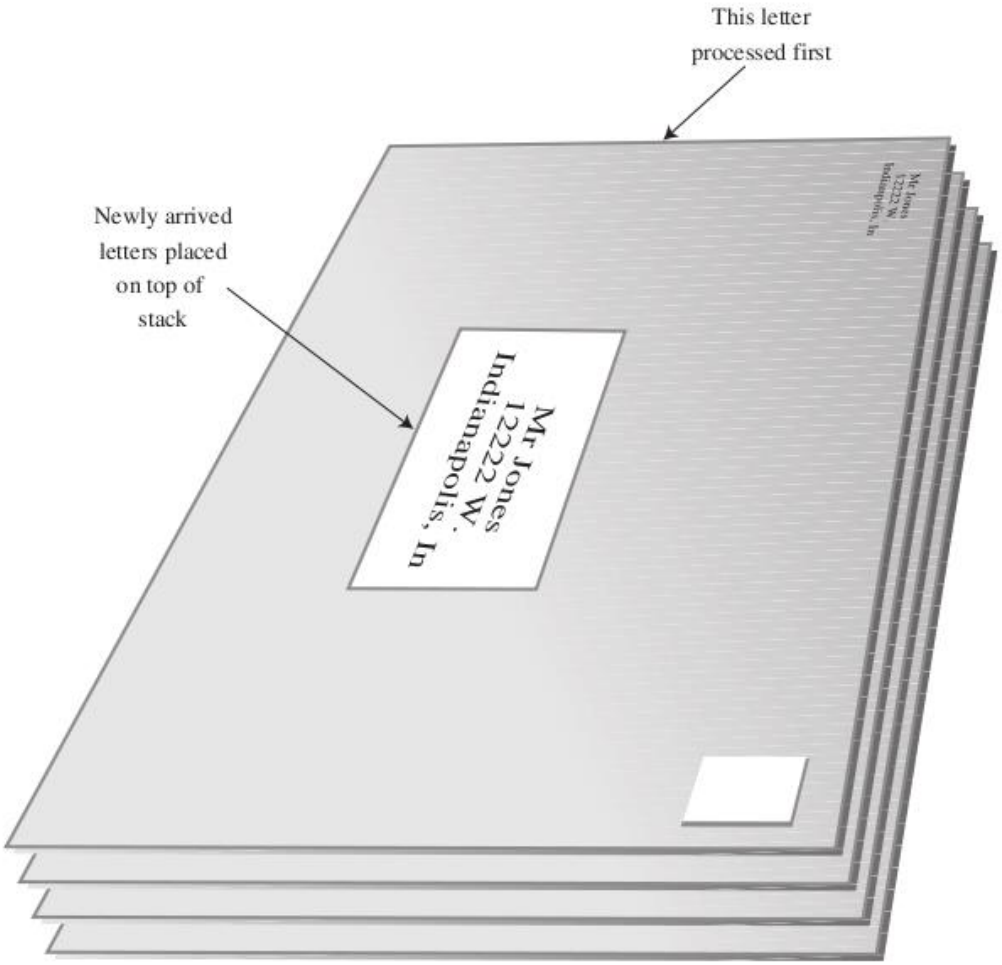
➔ Stacks play vital role in many situations.

➔ Processors use stacks intensively.

➔ Compilers use stacks to interpret the programming language expressions.

➔ Many algorithms need to use stacks.

Stack


This letter
processed first
Newly arrived
letters placed
on top of
stack
Mr Jones
12222 W.
Indianapolis, In

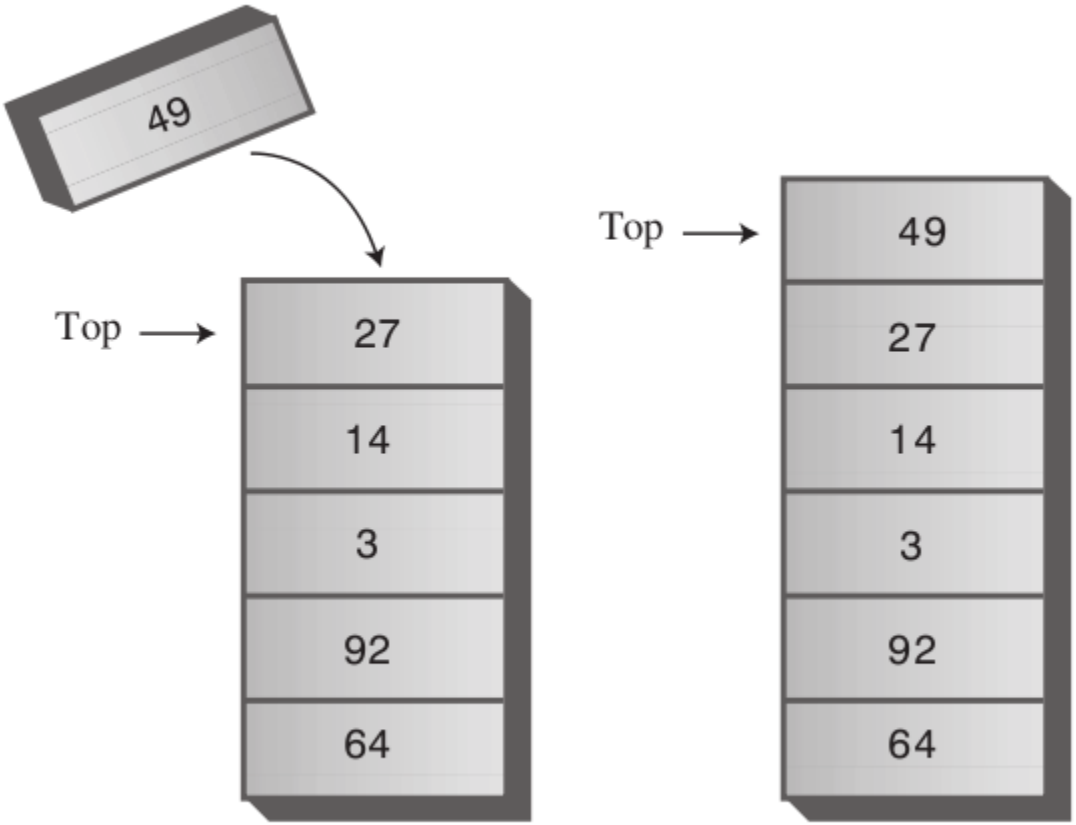
Stack

➔ Stack have three main operations:

- **Pop**: removes the element on the top of the stack.
- **Push:** adds a new element on the top of the stack.
- **Peek:** gets the element on the top without removing it.

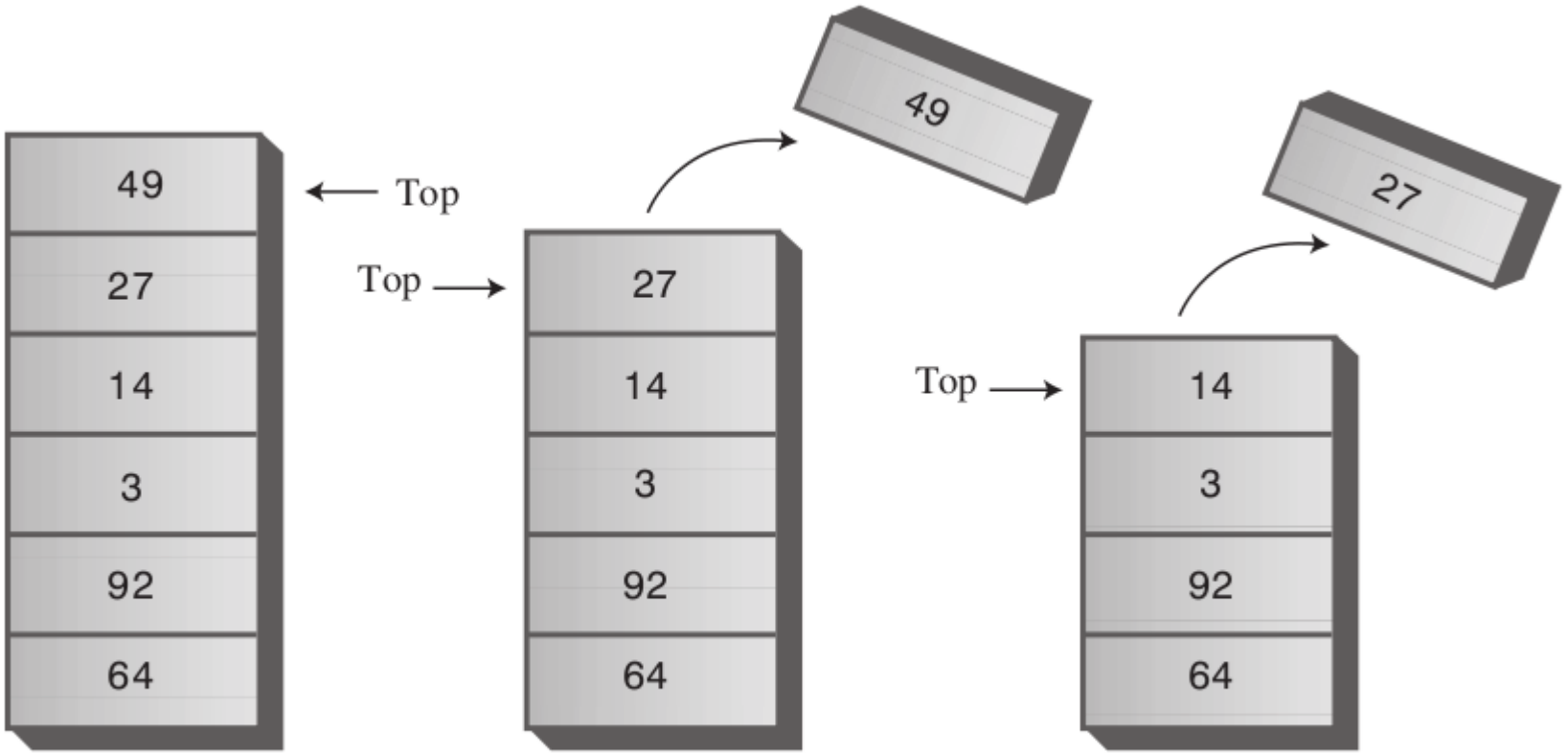
➔ There is no other way to traverse the stack, only through push and pop operations.

Stack



New item pushed on stack

Stack



Two items popped from stack

Stack

➔ Using arrays

```
01  class Stack {
02     private int maxSize;
03     private Object[] stackArray;
04     private int top;
05
06     ...
07
08     public void push(Object newObject) {
09       if (top < maxSize) {
10          this.stackArray[++top] = newObject;
11       } else {
12         throw new Exception("Stack is full");
13       }
14     }
15
16     public Object pop() {
17       return this.stackArray[top--];
18     }
19     ...
```

Stack

➔ Using arrays

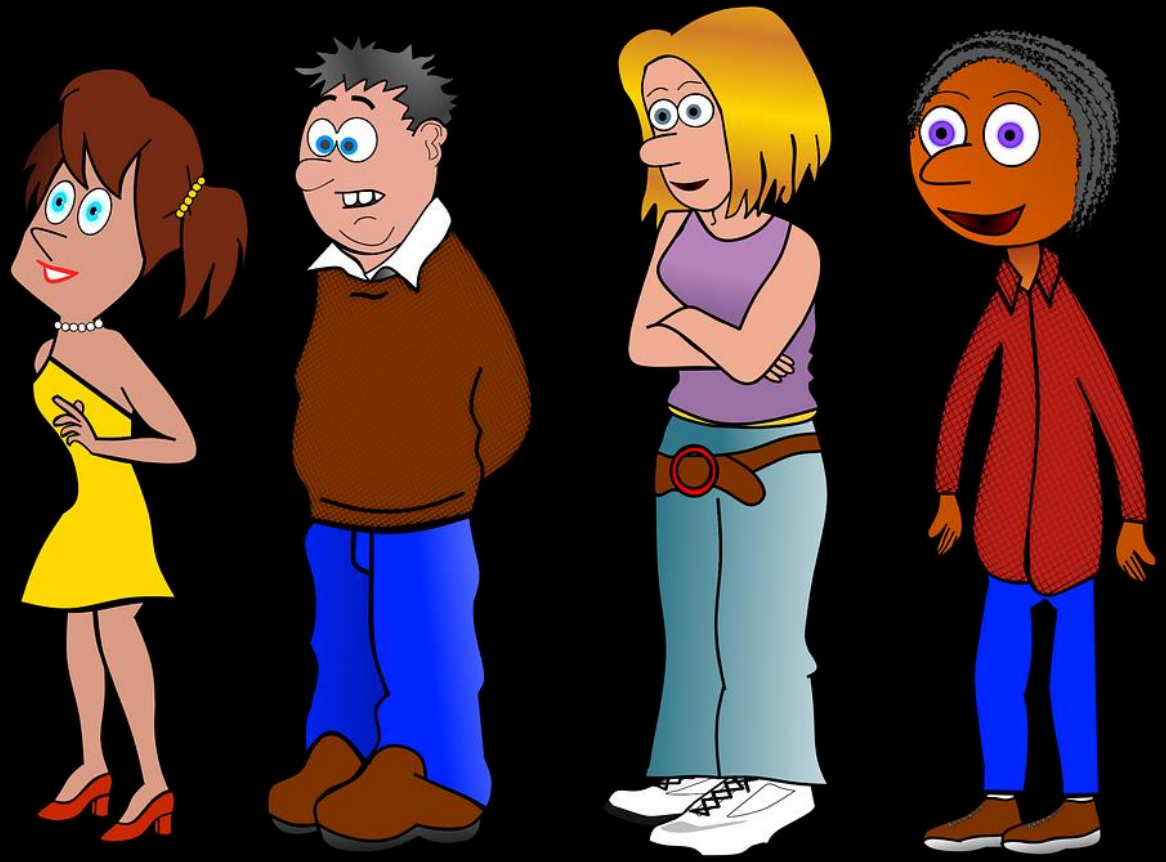
```
01    public Object peek() {
02      return this.stackArray[top];
03    }
04
05
06
07
08
09
10
11
12
13
14
15
16
17
18
19
```

Stack

➔ Using linked lists

```
01  class Stack {
02    private LinkedList stackList;
03
04    ...
05
06    public void push(Object newElement) {
07      this.stackList.insertFirst(newElement);
08    }
09
10    public Object pop() {
11      return this.stackList.deleteFirst();
12    }
13
14    public Object peek() {
15      return this.stackList.getHead();
16    }
17  }
18
19
```

Queues



Queue

➔ Similar to a stack but follows a FIFO (**F**irst **I**n, **F**irst **O**ut) nature.

Queue

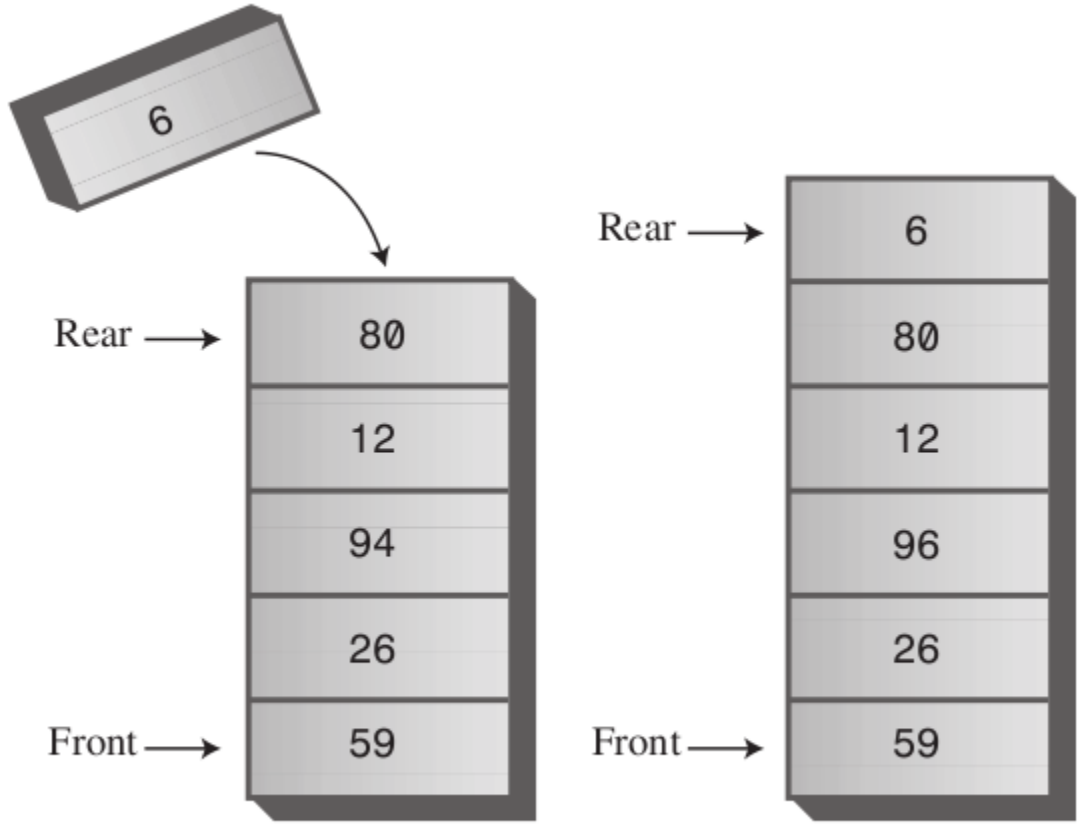
➔ Used to abstract many real life situation.

➔ Internally, the Operating System uses queues intensively (printing queues, keyboard input, so on).

➔ Queue support three main operations:

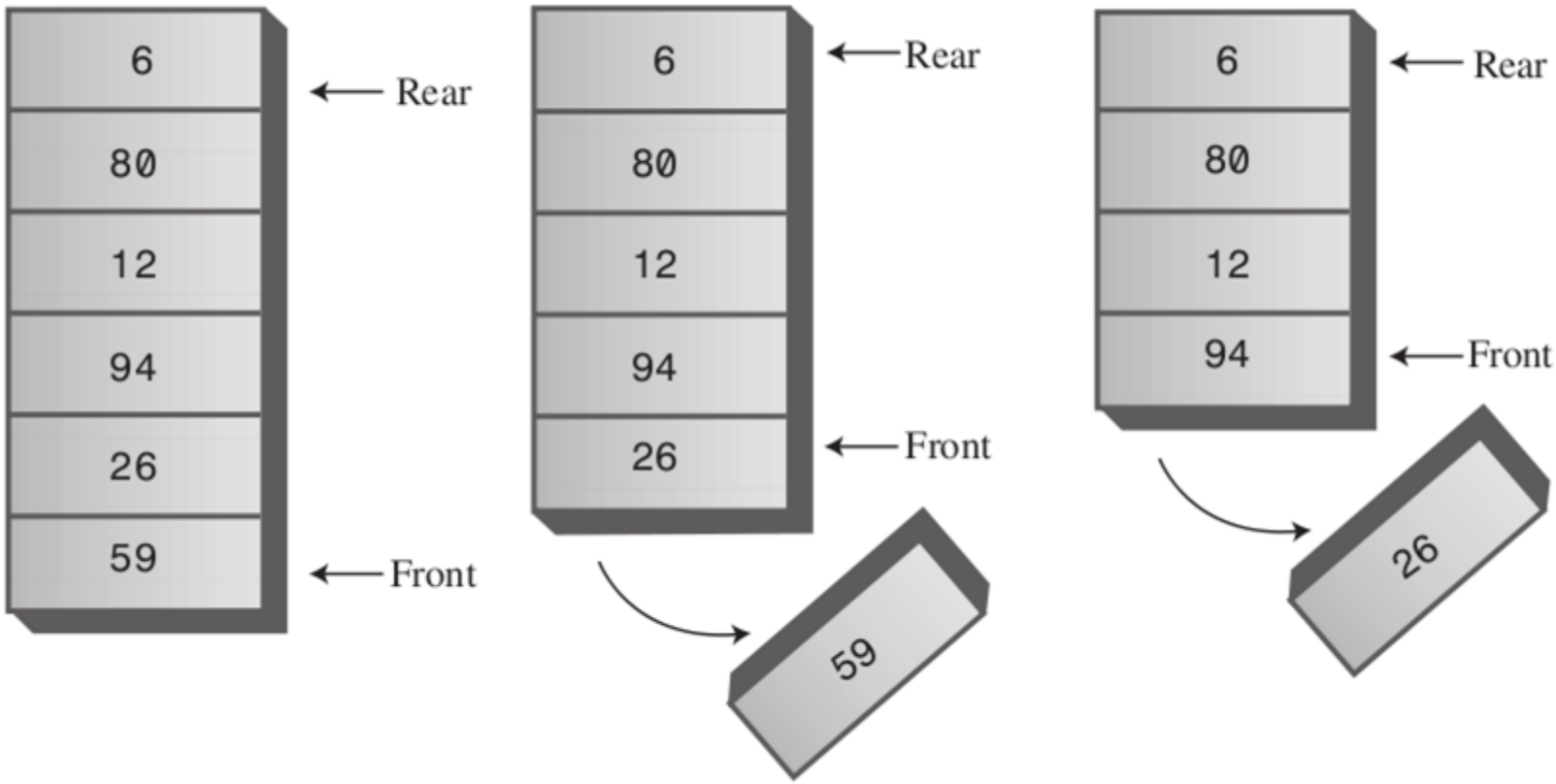
- **Enqueue**: add an element to the rear of the queue.
- **Dequeue**: remove an element from the front.
- **Peek**: get the element in front without removing it.

Queue



New item inserted at rear of queue

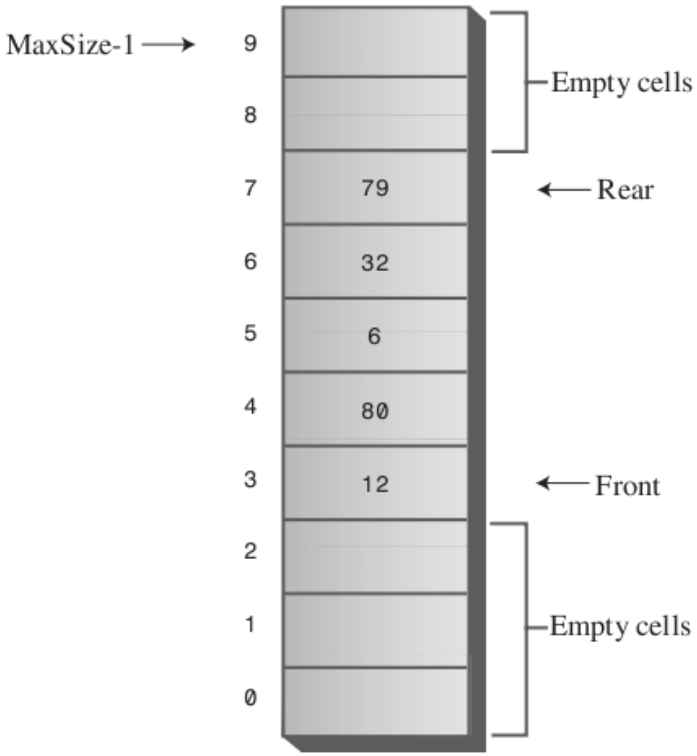
Queue



Two items removed from front of queue

Queue

➔ If the queue is **implemented using an array**, we can end up having the following situation:

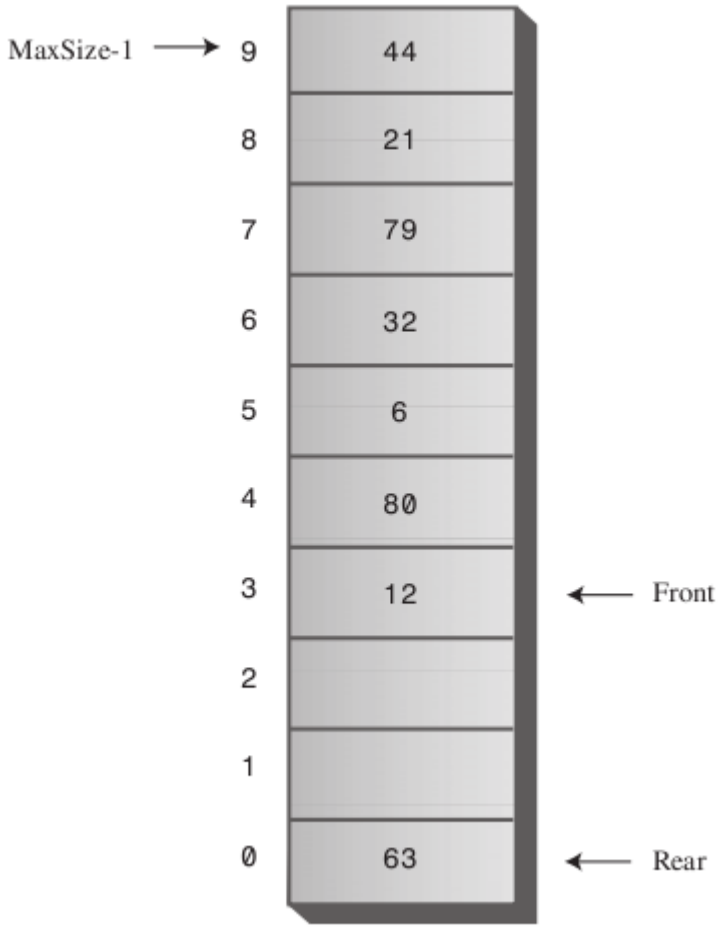


Queue

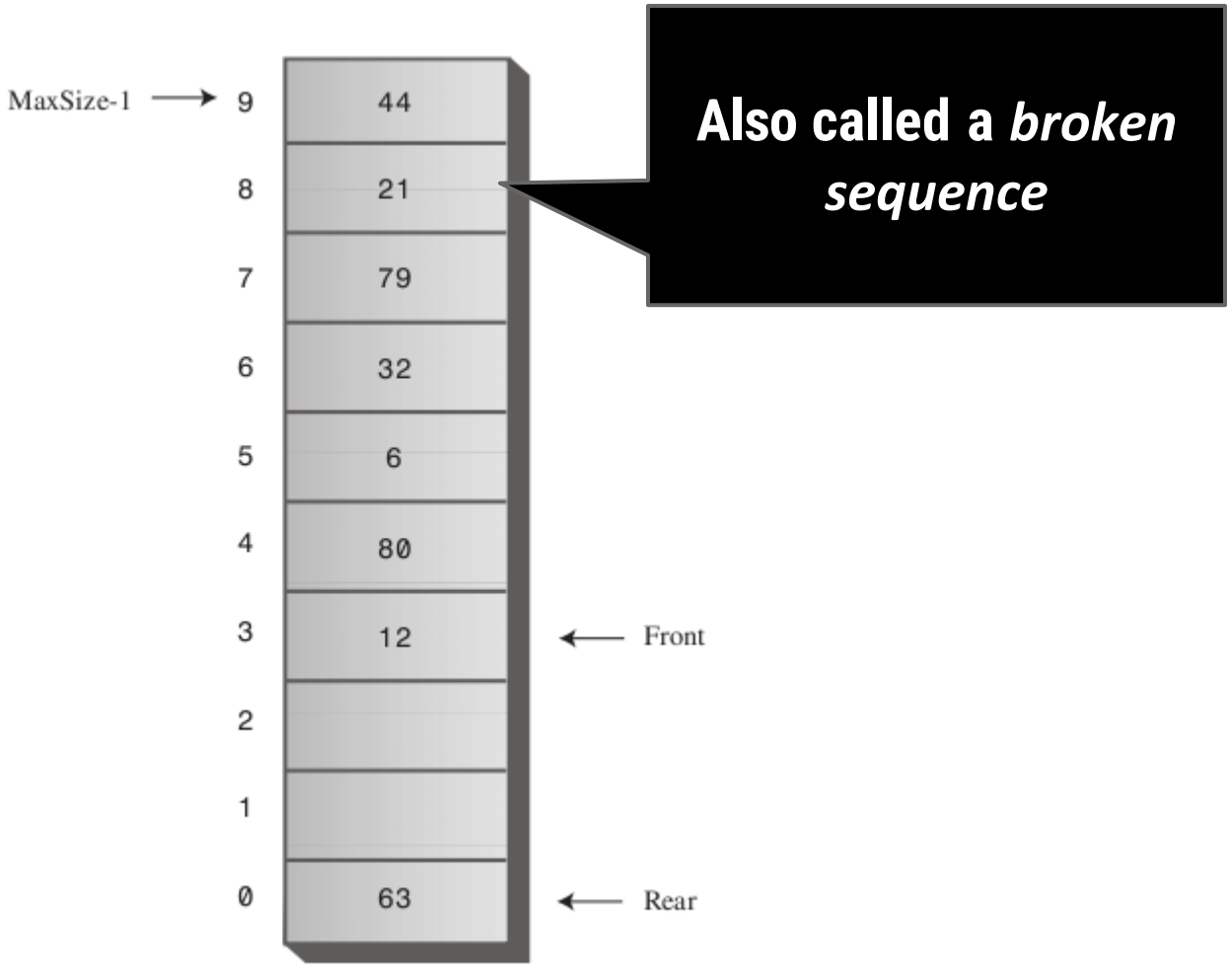
➔ One way to avoid this is moving the whole array after each dequeue operation.

➔ A prefered way is using a wrapping technique.

Queue



Queue


MaxSize-1
9
8
7
6
5
4
3
2
1
0
44
21
79
32
6
80
12
63
Front
Rear
Also called a *broken sequence*

Queue

➔ Using arrays

```
1 class Queue {
2    private int maxSize;
3    private Object[] queueArray;
4    private int front;
5    private int rear;
6    private int nItems;
7
8    public void enqueue(Object element) {
9      this.front = (this.front + 1) % this.maxSize;
10     this.queueArray[this.front] = element;
11     this.nItems++;
12   }
13
14   public Object dequeue() {
15     Object toReturn = this.queueArray[this.rear];
16     this.queueArray[this.rear] = null;
17     this.rear = (this.rear + 1) % this.maxSize;
18     this.nItems--;
19     return toReturn;
20   }
21
```

Queue

➔ Using linked lists

```
01  class Queue {
02     private DoubleEndedList list;
03
04     ...
05
06     public void enqueue(Object element) {
07       this.list.insertLast(element);
08     }
09
10     public Object dequeue() {
11       return this.list.removeFirst();
12     }
13
14
15
16
17
18
19
```

Priority Queue

CHARACTERISTICS

- ➔ Is a more specialized Data Structure than a queue or a stack.
- ➔ It is an useful tool in a surprising number of situations.
- ➔ Like a normal queue, it has a front and a rear. Items are removed from the front.

Priority Queue

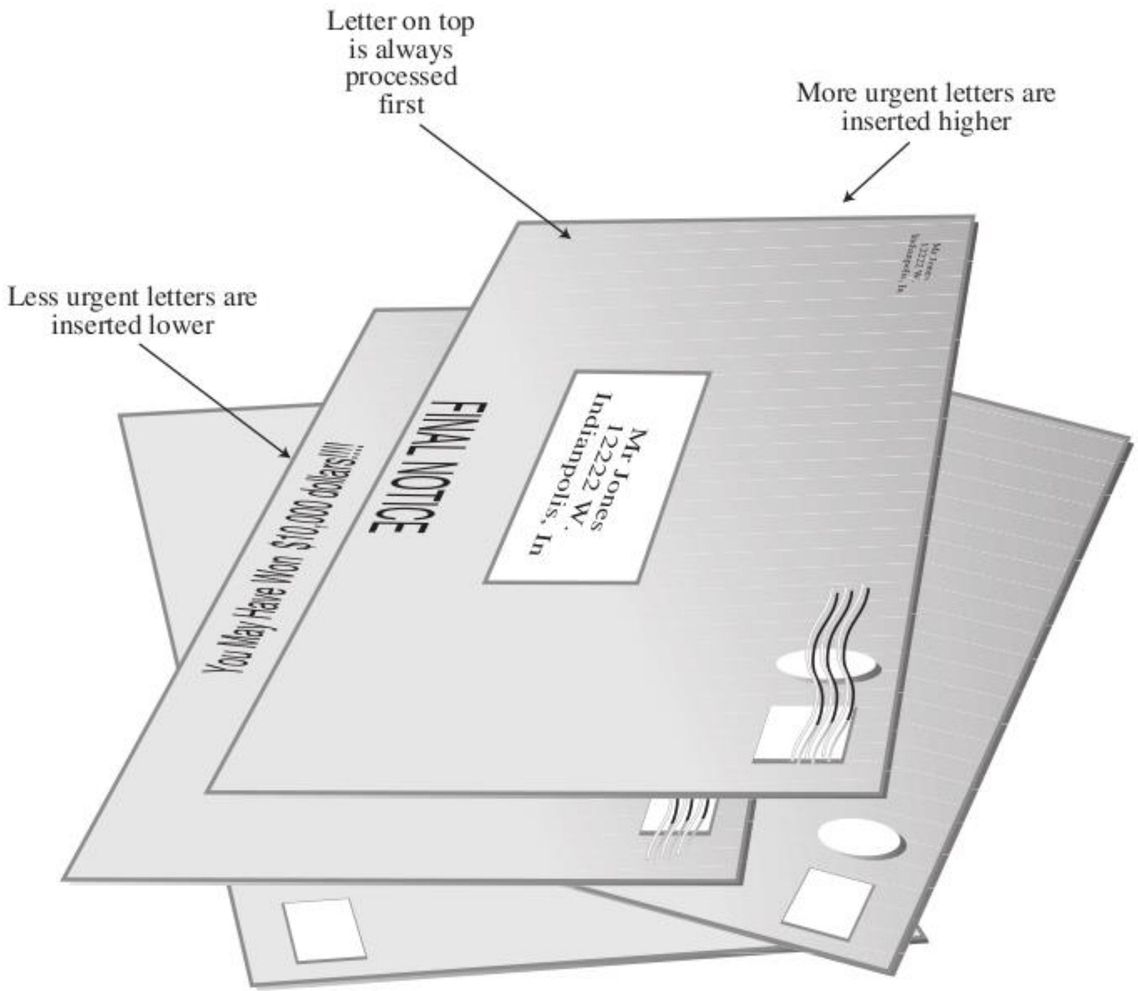
CHARACTERISTICS

- ➔ Is a more specialized Data Structure than a queue or a stack.
- ➔ It is an useful tool in a surprising number of situations.
- ➔ Like a normal queue, it has a front and a rear. Items are removed from the front.

OTHER CHARACTERISTICS

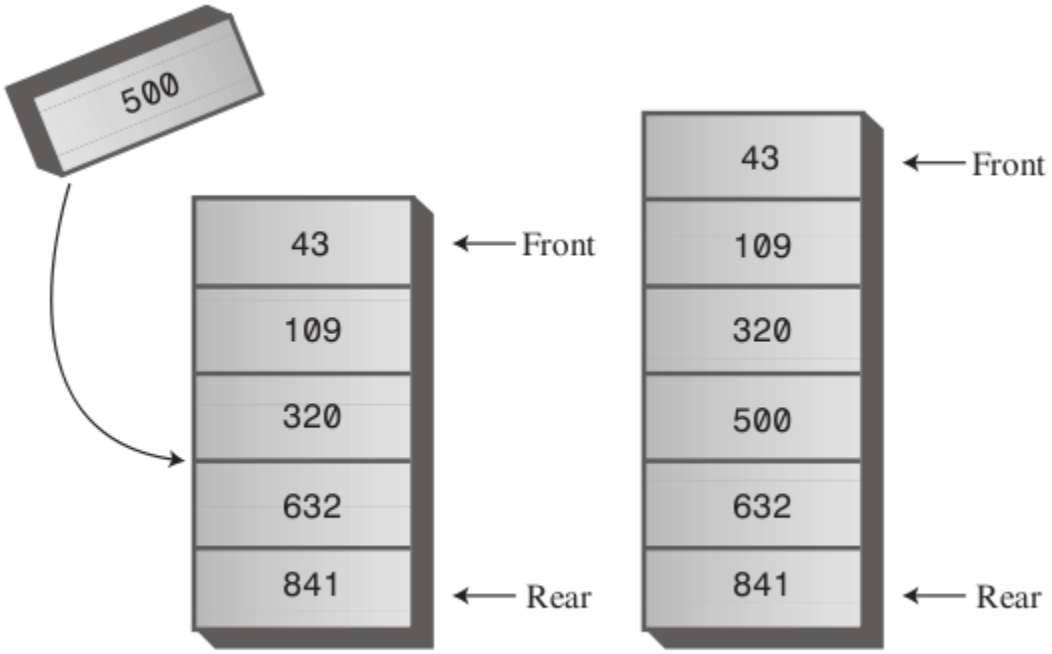
- ➔ Enqueuing an item is different. Items are ordered by a key.
- ➔ An item with the lowest key is always at the front. Items are inserted in the proper position to maintain order.

Priority Queue



Priority Queue

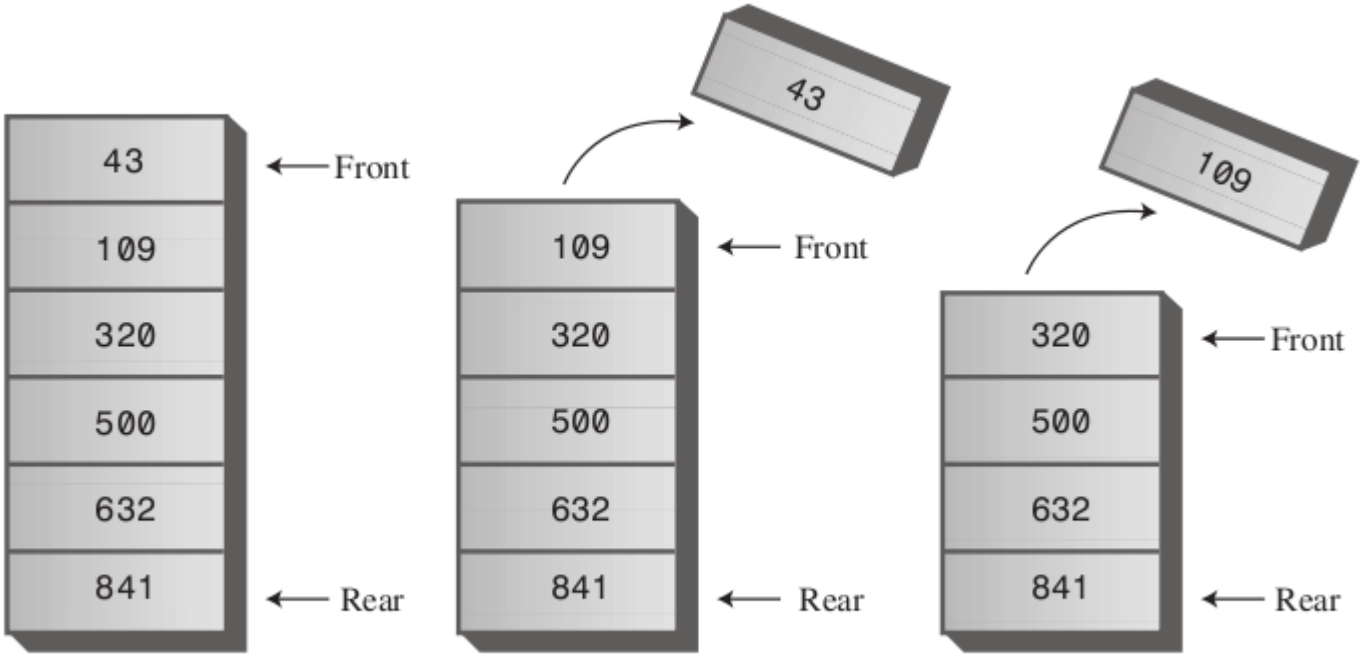
➔ Enqueuing a new element



New item inserted in priority queue

Priority Queue

➔ Dequeuing a new element



Two items removed from front of priority queue

Linear Data Structures

CE1103 - Algorithms and Data Structures I