



UNIVERSITY OF LONDON  
External System

# Correction

---

## **2910226 Software engineering, algorithm design and analysis – Volume 2**

BSc in Computing and Information Systems

December 2009: First correction to the 2006 edition of the subject guide

---

## Chapter 2 – Abstract data types I: lists and hashing tables

### p.28, Algorithm 2.2

Algorithm 2.2 should be corrected as follows.

---

**Algorithm 2.2** boolean isEmpty()

---

1: return(head=null)

---

### p.31, section 2.7.7 Implementation

Since the attributes of Node class (see Example 2.7, p.26) have private access, it is not possible for the List class to access them directly. Hence in section 2.7.7 the explicit mentions of the private variables should be replaced as below with calls to the getter and setter methods defined in 2.7.2.

```
class List{
    Node head;

    public void initialise(){
        head=null;
    }

    public boolean empty() {
        return(head=null);
    }

    public void addOneNodeAfter(Node p, Node newNode){
        newNode.setNext(p.getNext());
        p.setNext(newNode);
    }

    public void addOneNodeBefore(Node previous, Node p, Node newNode) {
        previous.setNext(newNode);
        newNode.setNext(p);
    }
}
```

```

public void addOneNodeBeforeHead(Node newNode) {
    newNode.setNext(head);
    head = newNode;
}

public void deleteOneNodeAfter(Node p) {
    p.setNext(p.getNext().getNext());
}
}

```

**pp.37-38, Algorithms 2.7 to 2.11**

In the given implementation, the pointer variable *top* has been initialised incorrectly. Please replace algorithms 2.7 to 2.11 in the subject guide with the following.

---

**Algorithm 2.7** initialise()

---

1: top  $\leftarrow$  0

---



---

**Algorithm 2.8** boolean isEmpty()

---

1: return(top=0)

---



---

**Algorithm 2.9** boolean isFull()

---

1: return(top=max) //assume that *max* is defined in stack class

---

---

**Algorithm 2.10** push(Object x)

---

```
1: if not isFull() then
2:   top  $\leftarrow$  top+1
3:   stackArray[top]  $\leftarrow$  x
4: end if
```

---

---

**Algorithm 2.11** pop()

---

```
1: if not isEmpty() then
2:   x  $\leftarrow$  stackArray[top]
3:   top  $\leftarrow$  top-1
4: end if
5: return x
```

---

**pp.40-42, section 2.9.1 Operations on queues**

In diagrams 2.19, 2.20 and 2.21, the *Rear* pointer is pointing to the wrong location. The *Rear* pointer should be pointing to the last occupied location, rather than the empty location after the last element. Thus the diagrams should be as follows.

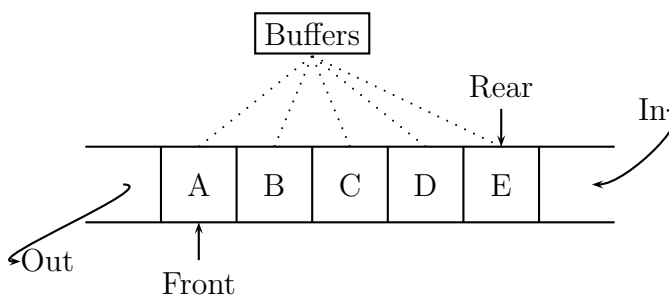


Figure 2.19: A queue

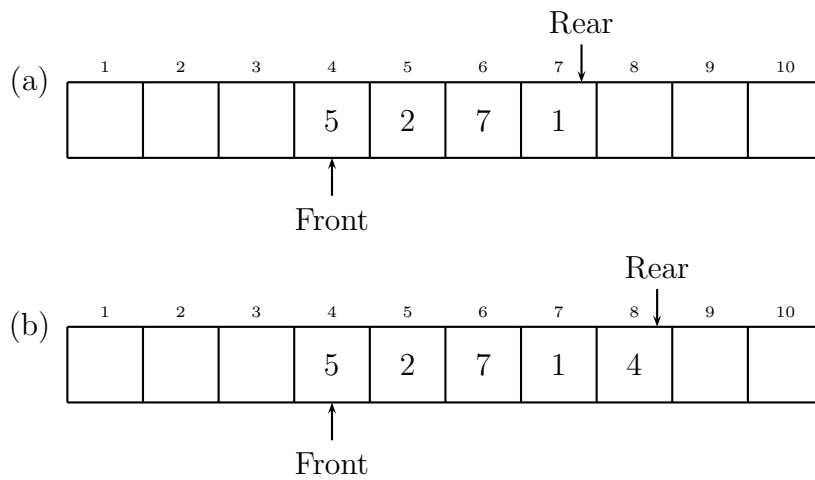


Figure 2.20: enqueue('4')

The text after figure 2.20 should be replaced with the following:

As we can see, in (a) the rear of the queue  $R$  pointed to address 7, the last occupied cell. In (b), rear is updated to point to index address 8 and data '4' is added at the rear position.

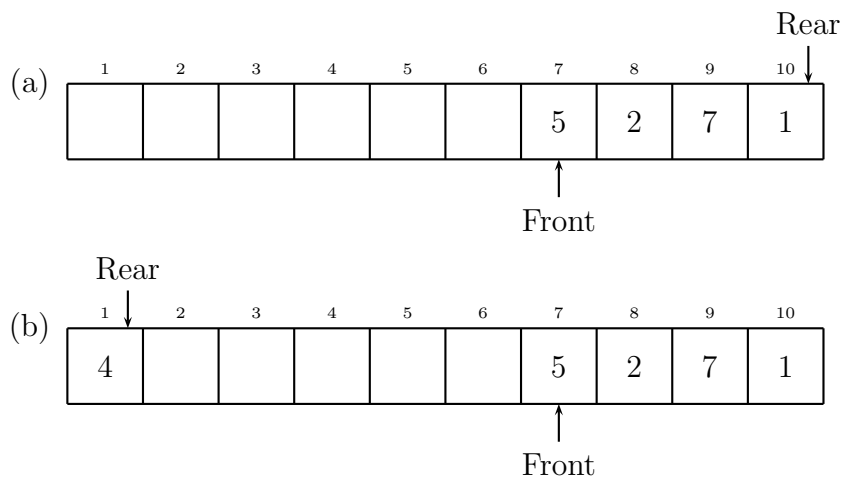


Figure 2.21:  $R$  has reached  $max$  and is cycled to the beginning of the queue array

**p.43, Algorithm 2.23**

There is no need to pass in a Node object to the dequeue algorithm. However

the method should have a return value of *Object* type. If the queue is empty then the *null* value is returned. Algorithm 2.3 should be as follows.

---

**Algorithm 2.23** dequeue()

---

```
1: if front  $\neq$  null then
2:   x  $\leftarrow$  front.data
3:   front  $\leftarrow$  front.next
4:   if front = null then
5:     rear  $\leftarrow$  null
6:   end if
7:   return x
8: end if
9: else return null
```

---

## Chapter 3 – Algorithm design techniques

### p.63, Algorithm 3.8

The parameter list is incorrect. Algorithm 3.8 should be as follows.

---

**Algorithm 3.8** linkedTraverse(object head)

---

```
1: if head  $\neq$  null then
2:   processNode(head.data)
3:   linkedTraverse(head.next)
4: end if
```

---

### p.63, Algorithms 3.9 and 3.10

These algorithms should include a return value as follows.

---

**Algorithm 3.9** find(node l,object x)

---

```
1: if l ≠ null then
2:   if l.data ≠ x then
3:     find(l.next,x)
4:   end if
5: end if
6: return l
```

---

---

**Algorithm 3.10** find(node l,object x)

---

```
1: if (l ≠ null) and (l.data ≠ x) then
2:   find(l.next,x)
3: end if
4: return l
```

---

**p.64, Algorithms 3.13 and 3.14**

Algorithm 3.13 which incorrectly uses a *while* loop within a recursive procedure should be as follows.

---

**Algorithm 3.13** display(node p)

---

```
1: if (p ≠ null) then
2:   writeln(p.data)
3: else
4:   while p ≠ null do
5:     display(p.next)
6:   end while
7: end if
```

---

Algorithm 3.14 which is the *correction* should then be as follows.

---

**Algorithm 3.14** display(node p)

---

```
1: if (p ≠ null) then
2:   writeln(p.data)
3:   display(p.next)
4: end if
```

---

## Chapter 4 – Abstract data types II: trees, graphs and heaps

### p.80, Algorithm 4.3

The object name should be included in step 2 as follows.

---

**Algorithm 4.3** treeNode combine(l,r)

---

```
1: t ← null
2: t.setLeft(l),t.setRight(r)
3: return t
```

---

### p.88, Algorithms 4.8 and 4.9

In Algorithm 4.8, the **if** loop started on line 3 should end after line 4 as follows.

---

**Algorithm 4.8** addRoot(index i,n)

---

```
1: j ← 2 × i
2: if j ≤ n then
3:   if (j ≤ n) and (A[j] ≤ A[j+1]) then
4:     j ← j+1
5:   end if
6:   if A[j] ≤ A[j+1] then
7:     tmp ← A[i]
8:     A[i] ← A[j]
9:     A[j] ← tmp
```

```
10:     addRoot(j,n)
11:   end if
12: end if
```

---

Algorithm 4.9 should have a descending for loop as follows.

---

**Algorithm 4.9** buildHeap()

---

```
1: for k ← (length(A) div 2, k ≥ 1, k ← k-1) do
2:   addRoot(k,length(A))
3: end for
```

---

## Chapter 6 – Sorting

### p.132, Algorithm 6.15

This algorithm should also have a descending for loop as follows.

---

**Algorithm 6.15** heapSort(heapArray A)

---

```
1: item tmp, index k
2: buildHeap() //construct a heap
3: for k ← length(A), k > 2, k ← k-1 do
4:   tmp ← A[k]
5:   A[k] ← A[1]
6:   A[1] ← tmp //swap A[1] with A[k]
7:   addRoot(1,k-1)
8: end for
```

---

### p.132, Algorithms 6.16 and 6.17

Corresponding to Algorithms 4.8 and 4.9 on page 88, Algorithms 6.16 and 6.17 have the same error and should be corrected as follows.

---

**Algorithm 6.16** addRoot(index i,n)

---

```
1: j ← 2 × i
2: if j ≤ n then
3:   if (j < n) and (A[j] > A[j+1]) then
4:     j ← j+1
5:   end if
6:   if A[j] > A[j+1] then
7:     tmp ← A[i]
8:     A[i] ← A[j]
9:     A[j] ← tmp
10:    addRoot(j,n)
11:  end if
12: end if
```

---

---

**Algorithm 6.17** buildHeap()

---

```
1: for k ← (length(A) div 2, k ≥ 1, k ← k-1) do
2:   addRoot(k,length(A))
3: end for
```

---