Chapter 3. Linked Lists - Continued

Special Linked Lists : Stacks and Queues

Dr. Abderrahmane Kefali

Senior Lecturer Class A,

Department of Computer Science,

University of May 8, 1945 - Guelma

kefali.abderrahmane@univ-guelma.dz

This document is prepared for printing two pages per sheet

1) Introduction

In the previous chapter, we saw that linked lists are dynamic data structures used to store collections of elements. However, there are special linked lists, called ***stacks*** and ***queues***, which are commonly used to solve specific problems in computer science. These two data structures are examples of *abstract data types* that allow storing elements following specific rules and provide operations to add and remove elements.

In this chapter, we will explore in detail stacks and queues as abstract data structures[1] implemented by linked lists. We will begin by describing their functioning and the possible operations for adding or removing elements. Then, we will see how these structures can be implemented using linked lists by providing the implementation of their basic operations (primitives).

2) Stacks

2.1) Presentation

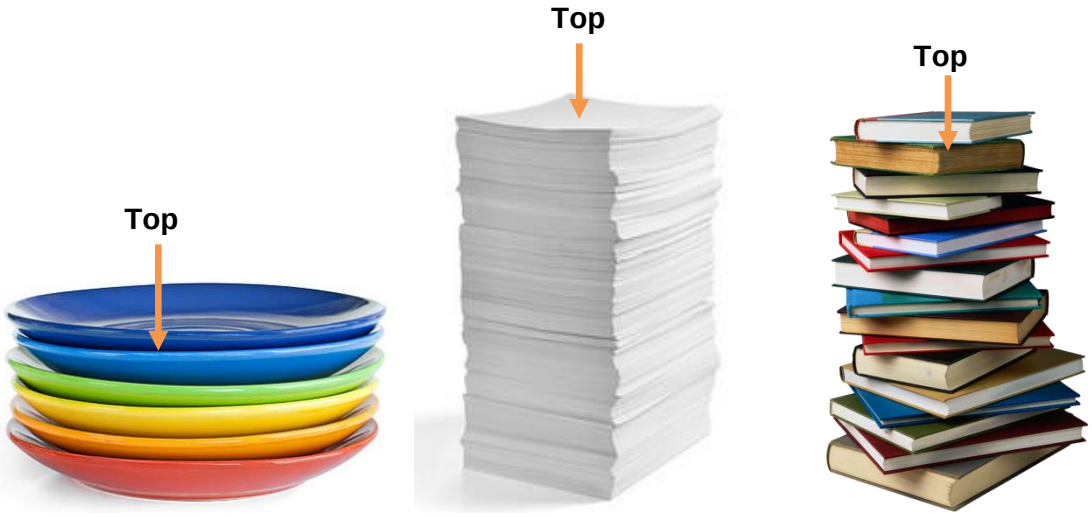
A stack is a data structure in which elements (of the same data type) are added and removed according to the Last In First Out (***LIFO***) rule. This means that the last element added to the stack is the first one to be removed.

The addition and removal of an element in a stack are therefore only allowed at one end, referred to as the stack's **top** (ST).

In other words, elements are ***deposited*** onto the top and ***removed*** from the top.

[1]An abstract data structure represents a data type and the operations that can be performed on that data, without considering how the data is stored or implemented.

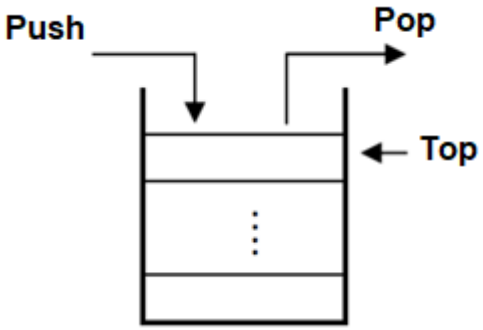
To grasp the basic mechanisms, we can envision a stack of plates, sheets, or books (see the following figure). It is at the top of the stack where we would take or place a plate, sheet, or book.



Stack of Plates *Stack of Sheets* *Stack of Books*

2.2) Operation

A stack can be graphically represented as a container containing a series of elements stacked on top of each other, as follows:

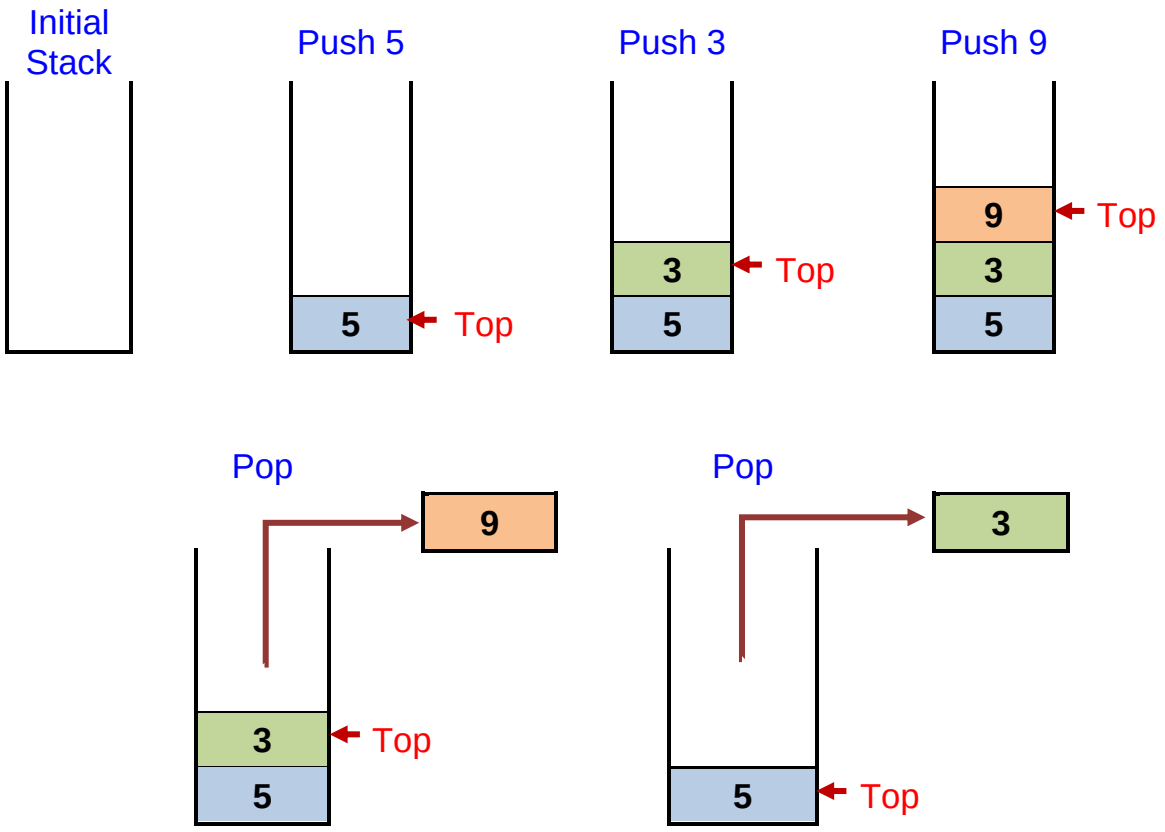


However, the functioning of the stack involves two basic operations:

1. ***Push Operation*:** this operation consists of adding an element to the stack. The element is placed on top of the stack, becoming the new top, that is to say, the only accessible element.
2. ***Pop Operation*:** removing the top element from the stack. The element is deleted from the stack, and the last element added before it (below the top) becomes the new top of the stack. The popped element is returned by the pop function for processing by the program.

To illustrate the functioning of a stack, consider the following example: Suppose we have a stack, and we want to insert three elements (integer values) 5, 3, and 9, and then remove two elements. Each insertion or removal occurs at the top of the stack.

We can represent these operations as follows:



Remark:

A notable property of stacks is that an object can only be popped after popping all the objects that are placed "above" it, causing objects to leave the stack in the reverse order of their arrival.

2.3) Uses of Stacks

A stack is primarily used to store data that cannot be processed immediately because the program has a more urgent or prerequisite task to accomplish beforehand.

Some of the most common uses of stacks include:

- Managing subprogram calls,
- Undoing operations,
- In a web browser,
- Syntax analysis.

2.4) Stack Operations

The only operations allowed with a stack are:

- Initialize a Stack.
- Push (Insert): always at the top of the stack.
- Pop (Remove): always from the top of the stack.
- Peek (Consult): the last element on the stack (the top) without removing it.
- Check if the Stack is Empty.

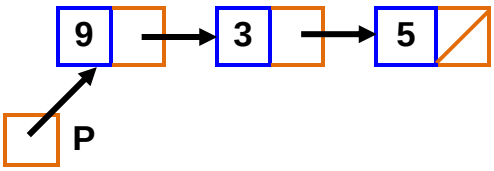
2.5) Implementation of a Stack

Generally, there are two ways to represent a stack. The first is static and can be done using an array. The second is dynamic and relies on the use of linked lists.

Here, we opt for a linked representation. In this case, the stack is essentially a singly linked list. The only difference is that the addition and removal of an element should only occur at the top of the stack (Head of the linked list).

Example:

The stack **P** from the previous example is represented in linked form as follows:



2.5.1) Declaration

The implementation of a linked stack first requires the definition of the structure representing an element of the stack. The declaration is similar to that of a linked list. For example, for a stack of integer values, the declaration would be as follows:

a) In algorithmics

The type describing a stack is called **`Stack`**, and it is defined as follows:

```
Type  Stack = ^node ;
      node = Record
           Begin
           val : integer ;
           next : Stack ;
           End ;
```

Once the type **`Stack`** is defined, a variable of this type can be declared as follows:

```
Var s : Stack ;
```

b) In C language

A stack of integers is declared in the C language as follows:

```
typedef struct node* Stack;
typedef struct node{
           int val;
           Stack next;
      }node;
```

A stack variable is then declared as follows:

```
Stack s ;
```

2.5.2) Stack Manipulation

As mentioned earlier, stacks support five fundamental operations or primitives. These primitives are implemented through subprograms. Manipulating a stack involves calling these subprograms, defined once and used as many times as needed.

a) Initializing a Stack

Initializing a stack means creating an empty stack ready for use in an algorithm or computer program. It is accomplished using the following procedure:

a.1) In algorithmics

```
Procedure initializeStack(Var s: Stack);
Begin
s ← Nil;
End;
```

a.2) In C language

```
void initializeStack(Stack* s){
   *s = NULL;
}
```

b) Checking if a Stack is Empty

A stack is considered empty when the head pointer is set to **`Nil`**. To confirm this, a test is performed on the value of the head pointer.

b.1) In algorithmics

```
Function isStackEmpty(S: Stack): Boolean;
Begin
If stack = Nil Then
        isStackEmpty ← True
Else isStackEmpty ← False;
End;
```

b.2) In C language

```
int isStackEmpty(Stack s){
    if(s == NULL) return 1;
    else return 0;
}
```

c) Pushing an Element

Pushing an element onto the stack involves placing it at the top of the stack. However, as the stack is implemented as a linked list, pushing is equivalent to inserting at the head of the list.

c.1) In algorithmics

```
Procedure push(e:integer; Var s:Stack);
Var p:Stack;
Begin
Allocate(p);
p^.val ← e;
p^.next ← s;
s ← p;
End;
```

c.2) In C language

```
void push(int e,Stack* s){
   Stack p;
   p = malloc(sizeof(node));
   p->val = e;
   p->next = *s;
   *s = p;
}
```

d) Popping an Element

If the stack is not empty, popping involves removing the element at the top of the stack and returning it. As the stack is implemented as a linked list, popping entails deleting the element at the head of the list and storing it in a variable.

d.1) In algorithmics

```
Procedure pop(Var e: Integer; Var s: Stack);
Var p: Stack;
Begin
e ← s^.val;
p ← s;
s ← s^.next;
Free(p);
End;
```

d.2) In C language

```
void pop(int* e,Stack* s){
   Stack p;
   *e = (*s)->val;
   p = *s;
   *s = (*s)->next;
   free(p);
}
```

e) Accessing the Top of a Stack

The following **`peekStack`** function allows access to the top of a non-empty stack and returns its value.

e.1) In algorithmics

```
Function peekStack(s: Stack): Integer;
Begin
peekStack ← s^.val;
End;
```

e.2) In C language

```
int peekStack(Stack s){
    return s->val;
}
```

Remark:

In contrast to popping, accessing the top of a stack simply involves returning the value of the top element without removing it.

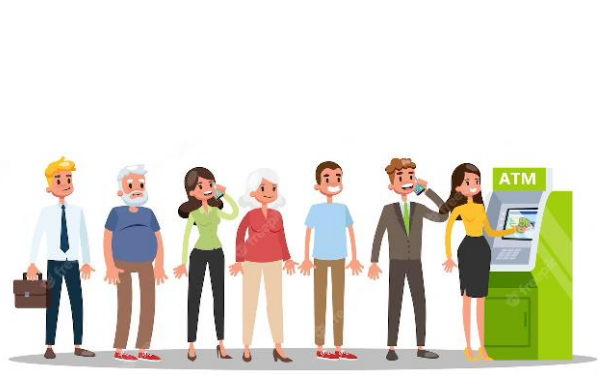
3) Queues

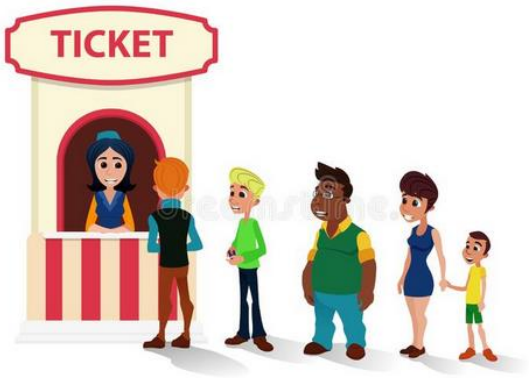
3.1) Presentation

Queues are data structures that implement the ***FIFO*** (First In First Out) strategy: the first item to be added is the first to be removed.

Unlike stacks, queues have two distinct ends for insertion and removal. Elements are added to one end of the queue, often called the ***back***, ***tail***, or ***rear***, and removed from the other end, often called the ***front*** or ***head***.

Thus, a queue in programming behaves exactly like a real-life queue. To understand the basic mechanisms of a queue, you can think of a queue in everyday life, such as those found in stores, cinemas, etc. In such a queue, people arrive one after the other and are served in the order they arrived (see the following figure).

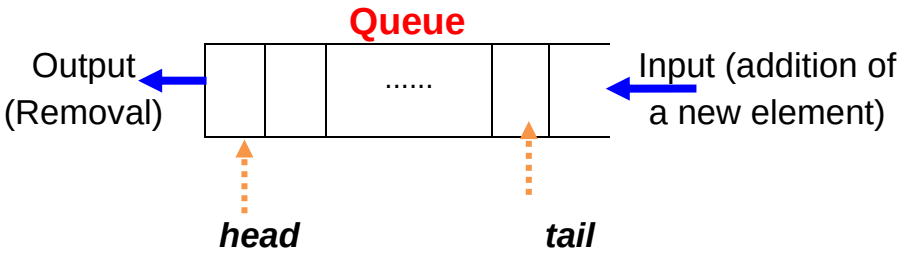




3.2) Operation

Graphically, a queue can be represented as a horizontal container where elements are arranged sequentially one after the other. The addition of elements takes place

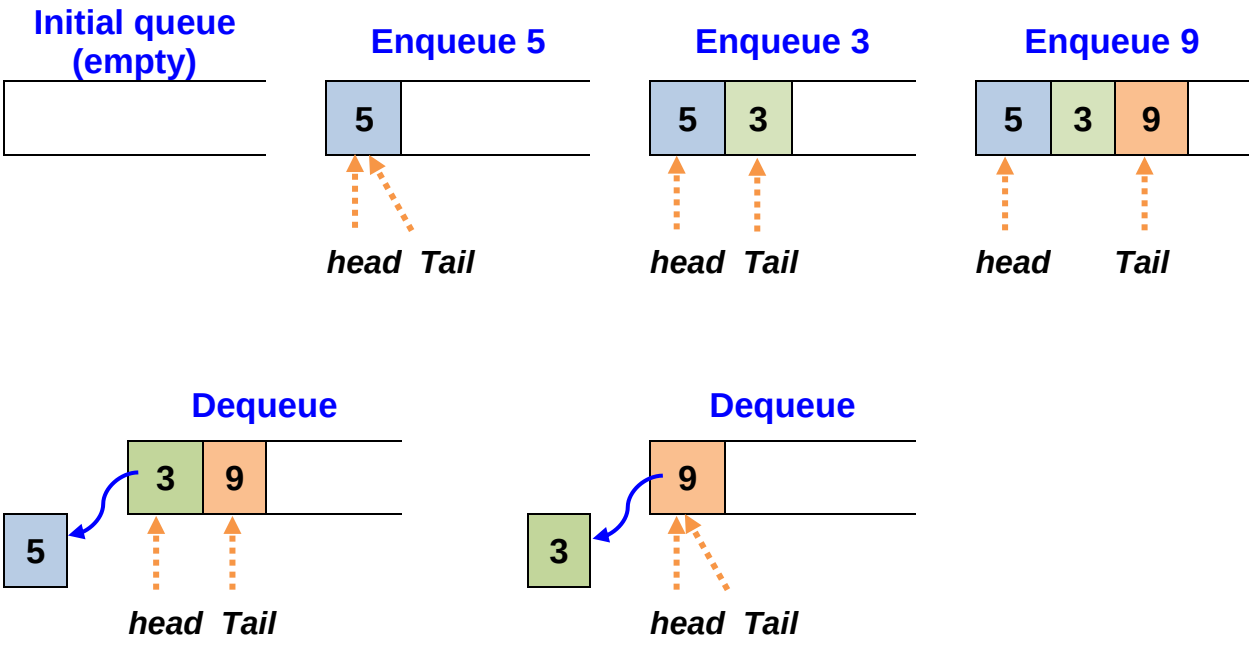
at one end (the back or tail of the queue), while removal occurs at the opposite end (the front or head of the queue), following the order in which elements arrived:



However, the functioning of the queue involves two basic operations:

1. ***Enqueue***: Involves adding a new element to the end of the queue.
2. ***Dequeue***: Involves removing the element at the front of the queue.

To illustrate the functioning of a queue, let's take the following example: Suppose we have a queue, and we want to insert three elements (integer values) 5, 3, and 9, and then remove two elements. Each insertion is done at the end, and each removal is done from the front. These operations can be schematized as follows:



3.3) Uses of queues

Queues are widely used in computer science to queue up information in the order it was received. There are numerous applications of queues in computer science, including:

- Document printing
- Scheduling tasks in operating systems
- Managing messages in a telephone network switch
- Handling database queries

3.4) Operations on Queues

The only operations allowed on a queue are:

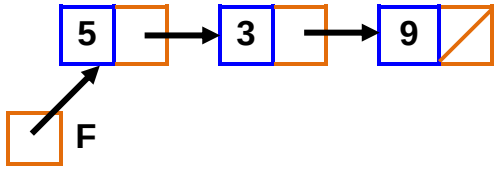
- Initialize a Queue.
- Enqueue: Add an element to the rear of the queue.
- Dequeue: Remove an element from the front of the queue.
- Peek: Examine the first element in the queue (the front), without dequeuing it.
- Check if the queue is empty or not.

3.5) Implementation of Queues

We can implement a queue as needed using either an array or a linked list. Just like with stacks, we have chosen linked list implementation. In this case, the elements of the queue form a singly linked list where insertions are made at the end and deletions are made at the head.

Example:

The queue **F** from the previous example is represented in linked form as follows:



Remark:

In fact, to work efficiently with a queue, it is appropriate to keep track of both the head and the tail (unlike stacks where only the head is needed). This allows for direct insertion after the element at the end of the list without the need to traverse the entire list. In this case, the queue will be considered as a record composed of two pointer fields:

1. The first (called ***Head***) points to the first element of the list,
2. The second (called ***Tail***) points to the last element of the list.

3.5.1) Declaration

To define the queue, we need to start by defining the type of each element in the queue. The declaration is similar to that of a simple linked list. Here is the declaration of a queue of integers:

a) In algorithmics

The type **`Queue`** is defined as follows:

```
Type  Queue = ^node ;
      node = Record
           Begin
           val : integer ;
           next : Queue ;
           End ;
```

Once the type **Queue** is defined, a variable of this type can be declared as follows:

```
Var q : Queue ;
```

b) In C language

A queue of integers is declared in the C language as follows:

```
typedef struct node* Queue;
typedef struct node{
          int val;
          Queue next;
      }node;
```

A queue variable is then declared as follows:

```
Stack q ;
```

3.5.2) Queue Manipulation

The primitives or basic operations on queues described earlier are materialized by the following subprograms:

a) Initializing a Queue

Initializing a queue means creating an empty queue ready to be used in a computer program. This is done using the following procedure:

a.1) In algorithmics

```
Procedure initializeQueue(Var q: Queue);
Begin
q ← Nil;
End;
```

a.2) In C language

```
void initializeQueue(Queue* q){
   *q = NULL;
}
```

b) Checking if a Queue is Empty

To check if a queue is empty, it suffices to test the **Head** pointer. If it is **Nil**, then the queue is empty. The following function **isQueueEmpty** returns **True** if the queue is empty and returns **False** othersiwe.

b.1) In algorithmics

```
Function isQueueEmpty(q: Queue): Boolean;
Begin
If q = Nil Then isQueueEmpty ← True
Else isQueueEmpty ← False;
End;
```

b.2) In C language

```
int isQueueEmpty (Queue q){
 if(q == NULL) return 1;
 else return 0;
}
```

c) Enqueuing an Element

Enqueueing translates to inserting at the end of the list. To add a new element, we simply reach the last node and attach the new element to it, making it the new last element. If the queue is empty, the newly inserted element is considered both the first and the last.

c.1) In algorithmics

```
Procedure enqueue(e: Integer; Var q: Queue);
Var p,r: List;
Begin
Allocate(p);
p^.val ← e;
p^.Next ← Nil;
If q = Nil Then
        q ← p
Else Begin
     r ← q;
     While r^.next ≠ Nil Do
          r ← r^.next;
     r^.next ← p;
     End;
End;
```

c.2) In C language

```
void enqueue(int e,Queue* q){
    Queue p,r;
    p = malloc(sizeof(node));
    p->val = e;
    p->next = NULL;
    if((*f) == NULL) (*q) = p;
    else{
          r = *q;
          while(r->next != NULL)
               r = r->next;
          r->next = p;
     }
}
```

d) Dequeuing an Element

Dequeuing involves removing the first element from the queue and storing it in a variable. When a queue contains only one element, removing that element involves setting the **`Head`** pointer of the queue to **`Nil`**.

d.1) In algorithmics

```
Procedure dequeue(Var e: Integer; Var q: Queue);
Var p: Queue;
Begin
e ← q^.val;
p ← q;
q ← q^.next;
Free(p);
End;
```

Remark:

Dequeuing cannot be performed when the queue is empty.

d.2) In C language

```
void dequeue(int* e,Queue* q){
    Queue p;
    *e = (*q)->val;
    P = *q;
    (*q) = (*q)->next;
    free(p);
}
```

e) Access the First Element of a Queue

The following **`peekQueue`** function allows access to the first element of a non-empty queue and returns its value.

e.1) In algorithmics

```
Fonction peekQueue(q:Queue): integer;
Begin
peekQueue ← q^.val;
End;
```

e.2) In C language

```
int peekQueue(Queue q){
   return q->val;
}
```