3 Elementary (Linear) ADTs

In chapter 2 we defined an abstract data type as a model with a set of operations. We can broadly classify ADTs according to the structure of their models. The first class of ADTs includes those whose models have a linear structure. A linear structure is one in which each element (except the last) may have exactly one successor element and each element (except the first) one predecessor element. The ADT stack falls into this category. The model for a stack is a set of elements where the elements are time-stamped on their arrival into the set. According to the time-stamps, the elements can be organised into a linear structure. This is why there can be no more than one pair with similar time-stamps in a stack.

Trees and graphs are examples of non-linear data types. In trees, each element is allowed to have a maximum of one 'previous' element but many 'next' elements. In graphs, an element can have many 'next' and 'previous' elements. In this chapter we shall study three linear ADTs, namely stacks, queues and lists.

Figure 3.1 illustrates linear and non-linear structures.

![Linear and non-linear structures](image)

3.1 The ADT Stack

The structure and operations of stacks were described in detail in chapter 2. An implementation of bounded stacks was given where the maximum number of elements in the stack was known in advance. For unbounded stacks, an implementation using pointers and dynamic storage is preferred.
Pointer Implementation of Stacks

Using pointers a stack can be implemented using a linked list of cells. Figure 3.2 shows the pointer representation of a stack \( S \) with elements \( a, b, c \) and \( d \) pushed into the stack in that order. Thus, we can declare the following types:

```plaintext
type cellptr = ↑cell;
cell = record
    element : elemtype;
    next : cellptr
end;
stack = cellptr;
```

The elements of the stack are given the generic type 'elemtype' which can be replaced with any data type as required.

![Figure 3.2 Pointer representation of a stack \( S \).](image)

An empty stack \( S \) is thus represented by a \( nil \) pointer. Using the above declaration the operations on stacks can be defined accordingly:

```plaintext
procedure initialise (var \( s \) : stack);
begin
    \( s \) := nil
end; (* of initialise *)

procedure push (var \( s \) : stack; \( a \) : elemtype)
var \( p \) : cellptr;
begin
    new (\( p \)); (* create a new cell for \( a \) *)
    \( p \).element := \( a \);
    \( p \).next := \( s \);
    \( s \) := \( p \)
end; (* of push *)
```