# Data structure and Algorithms

## Array and List

Thanh-Hai Tran

Electronics and Computer Engineering
School of Electronics and Telecommunications

Hanoi University of Science and Technology
1 Dai Co Viet - Hanoi - Vietnam

## Array

- Introduction to arrays
- Sequential storage structure (SSS)
- Implementing arrays by SSS
- Address functions
- Array in C

## Introduction

- **Array is a data structure consisting of a fixed number of same type elements.**
- **Characteristics:**
  - Number of dimensions: each array has at least one dimension.
  - Size of each dimension: size of an array (number of its elements) is a product of all its sizes of dimensions.
  - Data type of all elements: All elements of an array have the same data type.
- **Declaration of arrays:**
  - **ARRAY** : <*name*>[*dimension*, $len_1$, $len_2$,..., $len_n$] **OF** *datatype*;
- **Size of array *name*: *LEN(name)* calculated by:**

  LEN(name) = $len_1 \times len_2 \times \ldots \times len_n = \Pi(len_i)$ (với i=1,2,..,n)

## Introduction

- **Exp:**
  - Declaration of one dimension array:
    - ARRAY: vector [1, N] OF integer ;
  - Declaration of two dimension array:
    - ARRAY: matrix[2, M, N] OF integer; or
    - ARRAY: matrix[1, M] OF vector;
  - Declaration of N dimension array:
    - ARRAY : a[N, $L_1$, $L_2$,..., Ln] OF integer;

$\rightarrow$ **N dimension array is one dimension array of N-1 dimension arrays**.

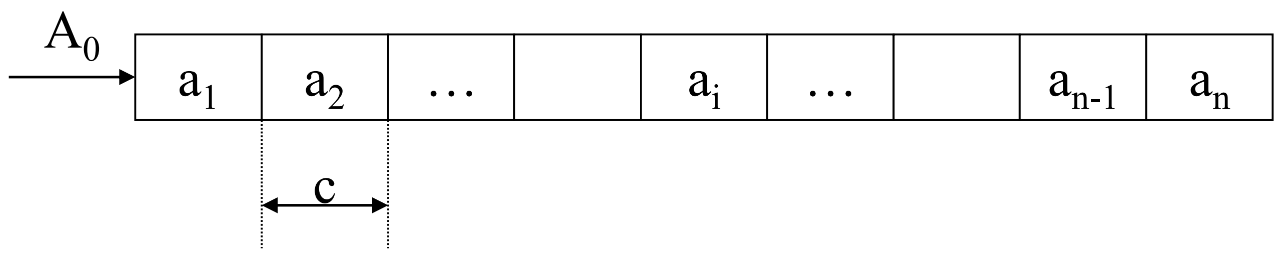
ARRAY: $a_n$[N, $L_1$, $L_2$,..., Ln] OF datatype ; $\Leftrightarrow$

ARRAY: $a_{n-1}$[N-1, $L_2$,..., Ln] OF datatype ; AND

ARRAY: $a_n$[1, $L_1$] OF $a_{n-1}$ ;

# Sequential storage structure

## Description



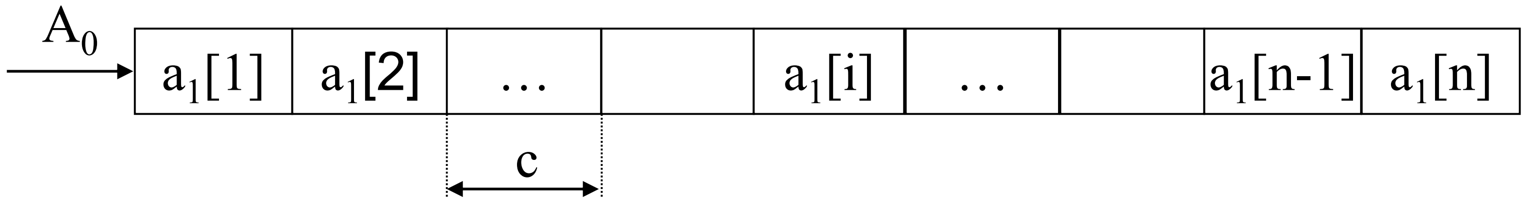
- $A_0$ is beginning address of SSS, also address of the first memory cell.
- Sizes of all cells are identical, a constant called `c` (in byte).
- Address function:
  - Address of $a_i$: `Loc (ai) = A`$_0$ `+ c* (i-1)`
  - Address function: `f(i) = c * (i-1)`

## Sequential storage structure

- **Characteristics:**
  - Simple structure and easy use
  - Fixed memory space. Allocation/Deallocation of memory is performed only once.
  - Quick access with same speed to all elements, because access is performed directly by address functions.
  - Perfectly suitable for implementing arrays with fixed size and identical data type elements.

## Arrays implemented by 3S

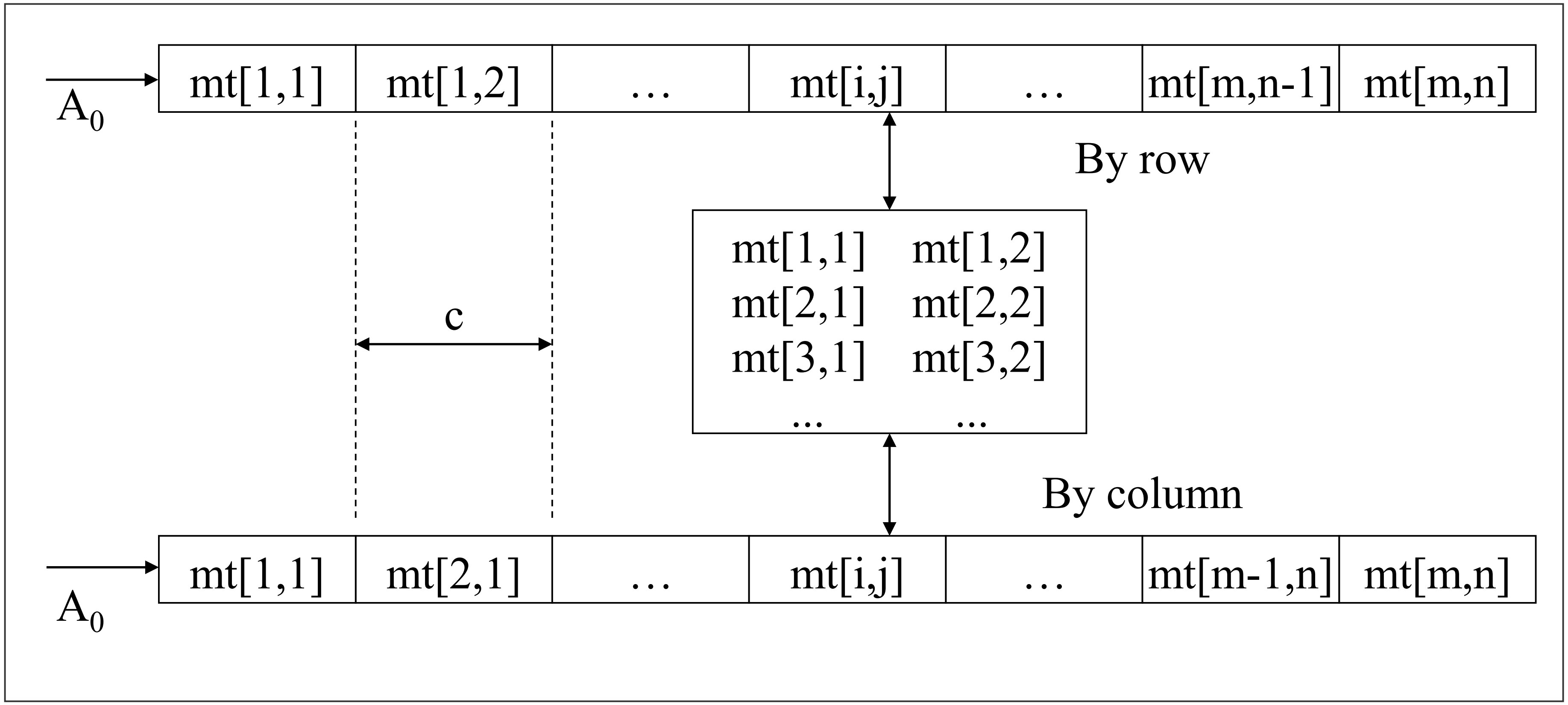
- **Implementation of one dimension arrays:**
  - `ARRAY : a`$_1$`[1, N] OF` *`datatype`* `;`
  - Step 1: determine suitable 3S:
    - Number of cells: **`N`**, number of elements (size) of array.
    - Size of each cell: called **`c`** which is the size of **`datatype`** of the array.
    - Allocation of a block of memory with size of **`c.N`**, and its first address $A_0$ for the array.
  - Step 2: arrangement of elements of array into selected 3S:
    - Sequential arrangement of elements into 3S, means that ith of array will be placed in ith cell of 3S ($1 \le i \le N$).
    - Address of ith element: $a_1[i]$: `Ai = A0 + c(i-1)`



## Implementation of 2D arrays

- `ARRAY  a`$_2$`[2,M,N] OF datatype;`
- **1st step: determine characteristics of suitable SSS:**
  - Number of memory cells: equal $M*N$, size if the array.
  - Size of each cell: equal $c$, size of *datatype*.
  - A block of memory with size equal to $c.M.N$, and its first address is $A_0$.
- **2nd step: arrangement of elements of array in the selected SSS. There are 2 arrangement methods:**
  - By rows (Theo thứ tự ưu tiên hàng): by this method, elements will be placed sequentially row by row.
  - By columns (Theo thứ tự ưu tiên cột): by this method, elements will be placed sequentially column by column.

# Implementation of 2D arrays



## Implementation of 2D arrays

- **Address function:**

  `ARRAY` $a_2$`[2,M,N] OF` *`datatype`*`;`

  - *By row method:* element $a_2[i,j]$ stored in the cell $z$ computed by:

    $$z = N*(i-1) + j \qquad (2.3)$$

  - This formula is called *address function* for 2D array $a_2$ by row arrangement method.
  - Vice versa, the cell z storing the element $a_2[i,j]$ with $i$ and $j$ computed by formula:

    ```
    i = (z -1) DIV N +1
    j = (z-1) MOD N +1
    ```

# Implementation of 2D arrays

- **Address function:**

  `ARRAY : a`$_2$`[2,M,N] OF` *`datatype`*`;`

  - *By column method:* element `a`$_2$`[i,j]` stored in the cell `z` computed by:

    `z = M*(j-1) + i` (2.5)

  - This formula is called *address function* for 2D array `a`$_2$ by column arrangement method.
  - Vice versa, the cell z storing the element `a`$_2$`[i,j]` with `i` and `j` computed by formula:

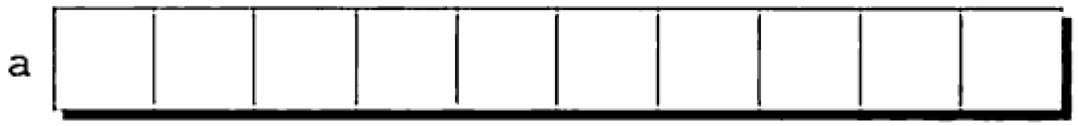
```
i = (z -1) MOD M +1
j = (z-1) DIV M +1
```

## Multidimensional array

- **Ideas:**
  - Recall that a N dimension array can be transformed to 1D array of (N-1)D arrays.
  - For example, 3D array is a 1D array of many 2D arrays. Because the implementations of 1D and 2D arrays are known, we can induct the implementation of 3D array.
  - The same process can be applied to more than 3D arrays.

## Array in C

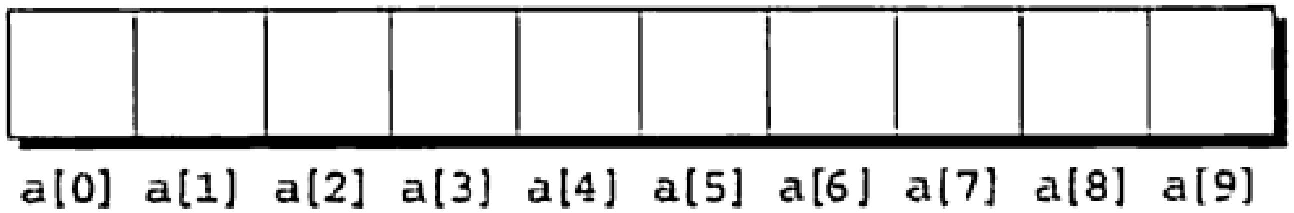
- An array is a data structure containing a number of data values, all of which have the same type
- These values, known as **elements**, can  be selected by their position within the array
- 1D array is a simplest kind of array, its elements are arranged one after another in a single row (or column)



- Declaration:
  - `int a[10];`
  - `#define N 10`
  - `int a[N];`
  - `int a[]; //error`

## Indexing and element access

- Array numbers are numbered starting from 0
- Elements of an array with N elements are indexed from 0 to N-1



- Access to an element of an array: a[i]
- Each element of array behave like a variable of type T

**idiom**
```
for (i = 0; i < N; i++)
  a[i] = 0;                  /* clears a */
```

**idiom**
```
for (i = 0; i < N; i++)
  scanf("%d", &a[i]);        /* reads data into a */
```

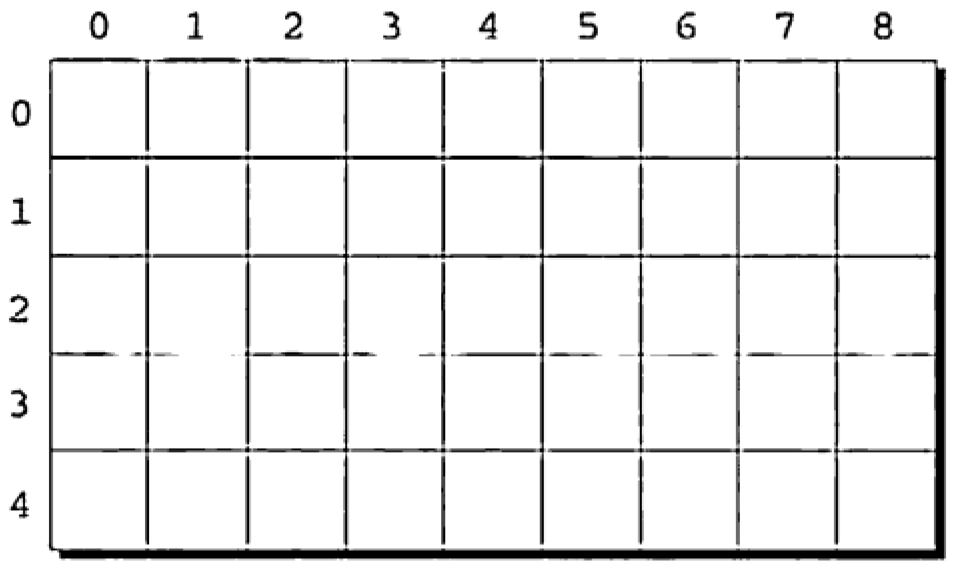
**idiom**
```
for (i = 0; i < N; i++)
  sum += a[i];               /* sums the elements of a */
```

## Initialization of array

- `int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`
- `int a[10] = {1, 2, 3, 4, 5, 6}; // {1,2,3,4,5,6,0,0,0,0}`
- `int a[10] = {0};//{0,0,0,0,0,0,0,0,0,0}`
- `int a[] = {1,2,3,4,5,6,7,8,9,10}; //10 elements`

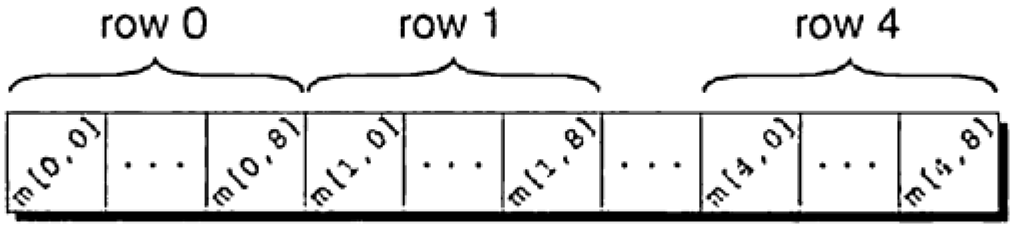
## Multidimensional array

- **An array may have any number of dimensions**
- **2D array:**
  - int m[5][9]; the array a has 5 rows and 9 columns
  - Both rows and columns are indexed from 0
  - To access the element of m in row I and column j: m[i][j]



## Multidimensional array

- C stores arrays in row-major order, with row 0 first, then row 1 and so forth



- We can create an initializer for a 2D array by nesting 1D initializers

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

## Multidimensional arrays

- The initializer will fill only the first three rows of m, the last two rows will contain 0

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0}};
```

- If an inner list isn't long enough to fill a row, the remaining elements in the row are initialized to 0

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1},
               {0, 1, 0, 1, 1, 0, 0, 1},
               {1, 1, 0, 1, 0, 0, 0, 1},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

## Initialization

- We even omit the inner braces, once the compiler has seen enough values to fill one row, it begins filling the next

```
int m[5][9] = {1, 1, 1, 1, 1, 0, 1, 1, 1,
               0, 1, 0, 1, 0, 1, 0, 1, 0,
               0, 1, 0, 1, 1, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 1, 1, 1};
```

## Using sizeof operator with arrays

- sizeof() determines the size of an array
- Example:
  - `int a[10];`
  - `int s = sizeof(a);// s = 40 because each int requires 4 bytes`
  - `int n = size(a)/size(int); // number of elements`

```
for (i = 0; i < (int) (sizeof(a) / sizeof(a[0])); i++)
  a[i] = 0;
```

```
#define SIZE ((int) (sizeof(a) / sizeof(a[0])))

for (i = 0; i < SIZE; i++)
  a[i] = 0;
```

## Constant arrays

- Any array, 1 or many dimensional can be made “constant” by starting its declaration with keyword “constant”

```
const char hex_chars[] =
  {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
   'A', 'B', 'C', 'D', 'E', 'F'};
```

- It documents that the program wont change the array, it helps the compiler catch errors, by informing it that we don’t modify array

## Limitations of array

- The array formed will be **homogeneous**. Thus no array can have values of two data types.
- While declaring the array passing size of the array is compulsory, and the size must be a constant. Thus there is either shortage or wastage of memory.
- Insertion or deletion of elements in an array will require shifting.
- The array does not check its boundaries: In C there is no check to see if the values entered in the array are exceeding the size of the array. Data entered with the subscript exceeding the array size will be simply placed outside the array, probably on the top of the data or the program itself.

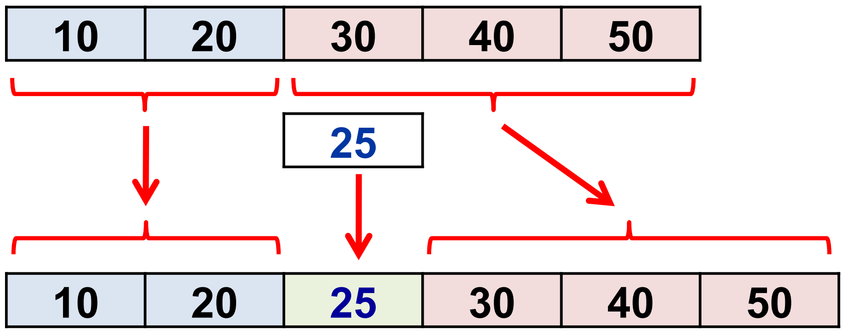
## List structure

- Description
- Last in first out (LIFO) (Stack)
- First in first out (FIFO) (Queue)
- Some applications of stacks and queues

## Introduction

- Conventionally, an array stores its elements consecutively in memory[(*)]
  ➔ Performance issue when invoking operations that involve the array size
- Insert an element into dynamic array:

  -
    ```
    let a = [10, 20, 30, 40, 50];
    a.splice(2, 0, 25);
    console.log(a);
    ```
  - Need to allocate new blocks of memory



- Similar problem for element removal

## Overview

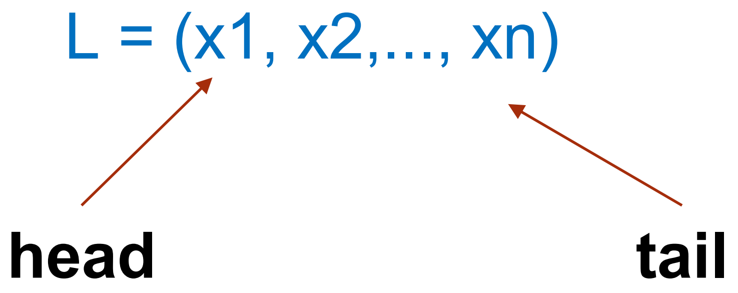
- Above example shows that dynamic arrays are slow in inserting/removing elements because of moving large memory areas, especially when there are many elements → necessity of a more flexible data structures
- Frequently used data structures:
  - Stack
  - Queue
  - Linked list
  - Dynamic array (vector)
  - Map, dictionary, hash table
  - Set
  - Tree
  - Graph

# Linear list

- **Linear list: the data structure consists of one or more elements of the same data type, and a linear order exists between the elements.**
- **Symbol: $L = (x_1, x_2, ..., x_n)$**
  - $n \geq 0$ and $x_1, x_2, ..., x_n$ are list elements,
  - $x_1$ is called the first (first) element of the list
  - $x_n$ is called the last element (tail) of the list
- **Linear order: front-to-back order between elements, that is, for every pair of $(x_i, x_j)$ $(1 \leq i, j \leq n$ and $i \neq j)$ elements in the list, there is always unique before-after order.**
- **Convention: special case when the list has no elements, called the *empty list*, denoted by $\emptyset$ $(L = \emptyset)$.**

# Feature of a list

- **Size or length of the list**: the number of elements in the list. The size of the empty list is 0.
  - Note that the list size is not fixed but changed by some of its operations.
- **Data type of elements**: there is a unique data type for list elements. The data type for the elements is always fixed.
- **The linear order in the list**: a list always has two ends, one end is the beginning (also called head), and the other is the tail of the list. Before-after order is order from head to tail.



## Basic operation of list

- Initialize a list
- Add an element into list
- Remove an element from list
- Search for elements in list
- Sort elements in list

## Initialize a list

- Define list structure, identify list characteristics.
- After the initialization, we usually get an empty list (the list has no element, but only the storage structure).
- In programming languages, the initialization operation is usually the declaration of the appropriate storage structure to represent the requested list structure.
- Suppose we want to have a list of at most N integers. We can use 1D array structure as storage structure and make initialization for the list as follows (two implementation methods):

```
//definition of list
struct {
   int ss[N];
   int size;
} List;
```

```
//first method
List L;
L.size = 0;
```

```
//second method
void init(List *L){
   L->size = 0;
}
List L;
init(&L);
```

## Add a new element to the list

- First, it is necessary to locate in the list the new element to be included.
- Usually, added positions are usually the beginning or the end of the list. However, we can also insert new elements into the middle of the list.
- Each of these additional operations increases the list size by 1.
- Note that the first condition (or pre-condition) to perform the additional operation is the list is not "full" or unsaturated.

## Remove an element from the list

- This is the reverse of the additional operation.
- This removal reduces the size of the list by 1.
- Note that the first condition for removing an element is that the list must be non-empty, that is, it must have at least one element.
- In general, determining the empty state of the list is a simple task, because this is the state of the list after the initialization operation.

## Other operations

- **Search for items in the list:** search for the occurrence or not of one or several elements in the list according to a search condition, and if present, where are found ones in the list? Search operations are often accompanied by additional operations and removals. A chapter "Search algorithms" devoted to talk about this operation.
- **Sorting list:** is the operation of arranging list elements in a certain order. The sorting algorithms will be presented in detail in chapter "Sorting algorithms".
- **Join two or more sub lists** to form a list.
- **Split** the list into two or more sub lists.

## Common list structure

- Last in first out (Last In, First Out - LIFO) or stack structure - Stack
- First in first out (First In, First Out - FIFO) or queue structure - Queue

## Implementation Methods for Lists

- **Implementation by sequential storage structures (or one-dimension arrays):**
  - This is the simplest and fastest way to implement.
  - However, this implementation also has some limitations due to the basic difference between the list structure and the array structure, between a dynamic and a static structure.
- **Implementation by linked storage structure:**
  - this is a dynamic storage structure with flexible storage size and organization according to the requirements of the data structure,
  - so it is suitable for organizing and storing dynamic data structures.

## General principles

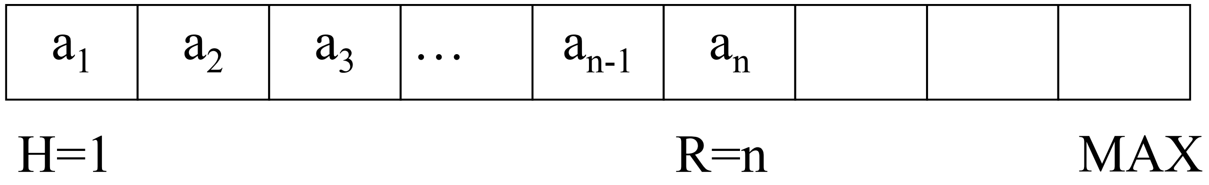
- **When using a storage structure to implement a data structure, we should follow the following principles:**
  - *Sufficiency:* storage structure must have the appropriate characteristics to fully represent the characteristics and capabilities of data structure. It manifests itself in two aspects
    - **Storage capacity:** is the size of the storage structure. This quantity is determined by the value domain of the data structure.
    - **Processing capability:** specified by a set of operations implemented on the selected storage structure.
  - *Optimal:* highest efficiency in storage and processing.
  - *Extensible:* the implementation is easy to extend.
  - *Encapsulation:* some unnecessary detail implementation can be hidden from users.

## Implementation of lists using 3S

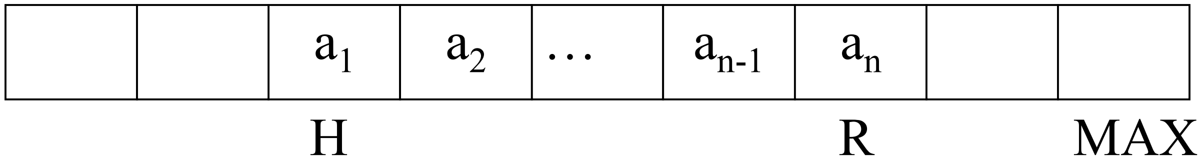
- **Choose a suitable storage structure.**
  - Determine the maximum size MAX of the list
  - Specify the data type for each element of the list
  - Select storage structure as a one-dimension array of  selected data type.
- **Logically arrange the elements of the list in the selected storage structure.**
  - Arrange each element in a memory compartment
  - Arrange the elements in their linear order, respectively
  - Arrange elements in contiguous positions to ensure the least amount of information to manage the list.
  - Choose the appropriate arrangement so that the basic operations performed on the list are most effective.

## Implementation of lists using 3S

- **Some arrangements of list elements in an array**
  - Arrange immediately from the beginning of the array, with H = 1, R = n (H for Head, R for Rear of the list), only one information n needed.

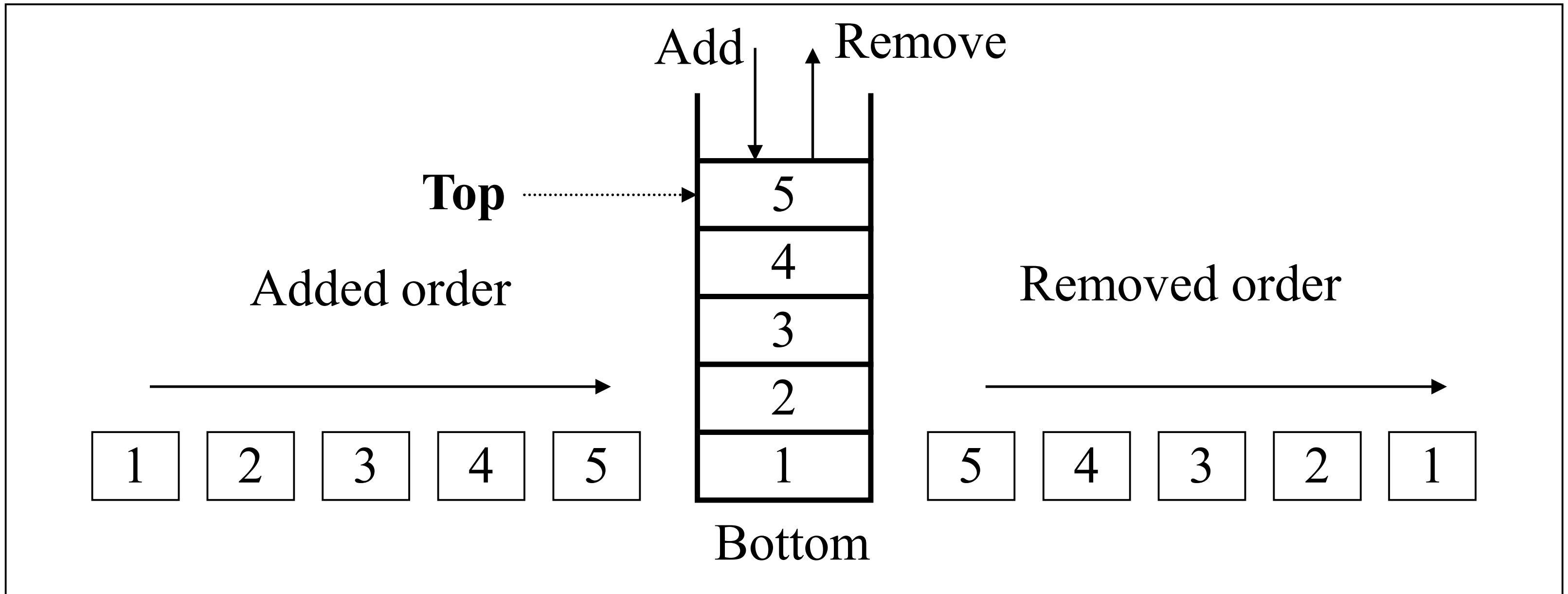


  - Arrange the sequence of elements continuously. This requires two of the three information H, R, n (as n = R – H + 1).



## LIFO (Last In, First Out - Stack)

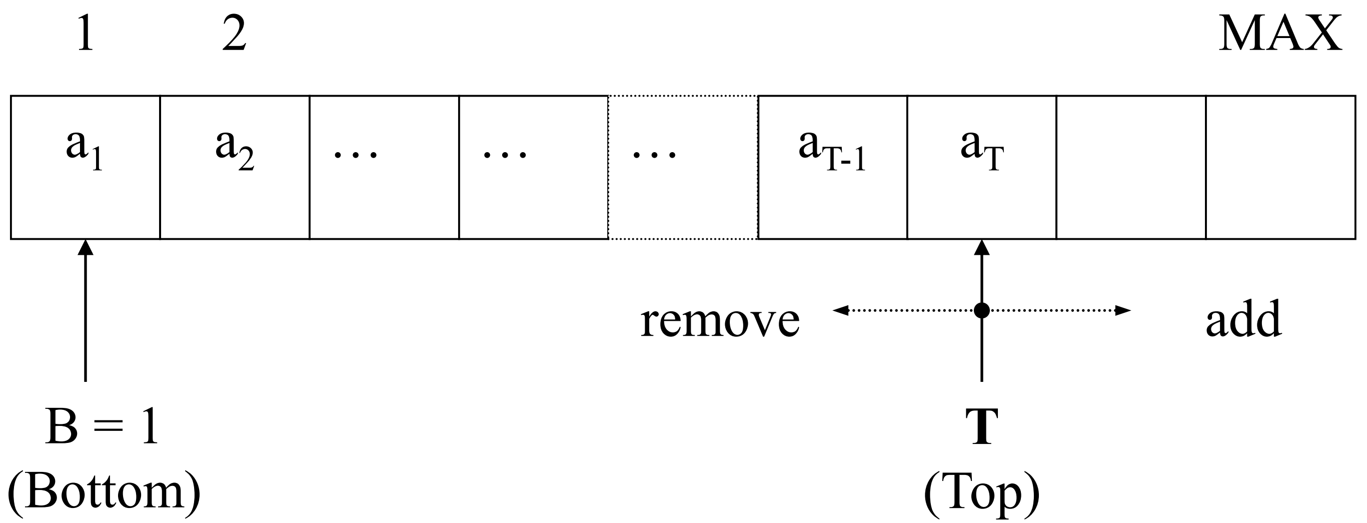
- **LIFO means that the last added element is also the first one removed.**



Structure and operations of the Stack

## LIFO (Last In, First Out - Stack)

- **Implementation technique**



Stack representation by sequential storage structure

## Implementation of Stack in C

- Definition of Stack structure:

```
typedef  struct  {
   Type  info [MAX];
   unsigned int n;
} Stack;
```

- Sequential storage structure is represented by the array info.
- Type is the data type of the list.
- n is the number of elements in the stack.
- The top of the stack will be at n-1 position.

## Implementation of Stack in C

- Definition of Stack structure:

```
typedef  struct  {
   Type  info [MAX];
   unsigned int n;
} Stack;
```

- Current status of the stack:
  - *Empty*: `n = 0`
  - *Full*: usually, the value `MAX` is the maximum size of the stack (if it can be determined correctly) or the maximum size that this sequential storage structure is allowed, so we will convention that the stack is full when `n = MAX`.
  - *Normal*: the state is neither empty nor full. When the stack is in this state, it can perform adding action and remove.

## Implementation of Stack in C

```
//Input: No; Output: stack S
void Initialize (Stack & S){
    S.n =0;
}
```

```
//Input: stack S; Output: true/false if S is empty/not empty
bool IsEmpty (Stack S){
    if (S.n == 0) return true;
    else return false;
}
```

```
//Input: stack S; Output: true/false if S is full/not full
bool IsFull (Stack S){
    if (S.n == MAX) return true;
    else return false;
}
```

# Implementation of Stack in C

```
// Returns the element at the top of the stack S (do not remove it)
//Input: S; Output: the top element
Type Top (Stack  S){
   if (IsEmpty(S)) return NULL;
   return S.info[S.n - 1];

}
```

## References

- Slide, C/C++ programming technique, Tran Thi Thanh Hai
- Slide, Data structure and algorithm, Nguyen Thanh Binh