Sequential Organization

- Organization of data within the computer memory is linear i.e. each element has unique successor.
- Direct or Random access to any element is the major advantage of sequential organization.
- Sequential organization makes it possible to access any element in the data structure randomly in constant time O(1).

Sequential Organization

- Example : Arrays
- Array is an ordered set of objects.
- data structure that stores data of similar type at consecutive memory locations.
- e.g. int A[5]; // A is an array of size 5
- It will allocate 5 consecutive memory locations for the array A. Each element of array can be accessed by referring to its index.

Linear data structure

- The term “linear” means “belonging to a line”
- A Linear data structure consists of elements which are ordered i.e. in a line.
- The elements form a sequence such that there is a first element, second, …… and last element.
- Thus, the elements of a linear data structure have a one-one relationship.
- Examples : array, list.

Arrays

- An array is a finite ordered set of homogenous elements.
  - **Finite :** There are a specific number of elements in the array.
  - **Ordered :** The elements are arranged so that there is a first element, second, and so on.
  - **Homogenous :** All the elements are of the same type.
- Mathematically, an array can be considered as a set of pairs- (index, value). For each index, there is an associated value. This means that, there is a one-to-one mapping or correspondence between the set of indices and the set of values.

Primitive Operations on Arrays

1. **Create :** Creating a new array : This needs the information regarding the type of data objects and the number of data objects to be stored.
2. **Store(array, index, value) :** Store a value at particular position : This needs the name of the array, the position in the array and the value to be stored at that position.
3. **Retrieve (array, index) :** Retrieve a value at a particular position : This operation requires the array name and an index and it returns the corresponding value.

Array as an ADT

Objects: A set of pairs <index, value> where for each value of index there is a value from the set item. Index is a finite ordered set of one or more dimensions, for example, {0, … , n-1} for one dimension, {(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)} for two dimensions, etc.

Methods:

for all A ∈ Array, i ∈ index, x ∈ item, j, size ∈ integer

Array Create(j, list) ::= **return** an array of j dimensions where list is a j-tuple whose kth element is the size of the kth dimension. Items are undefined.

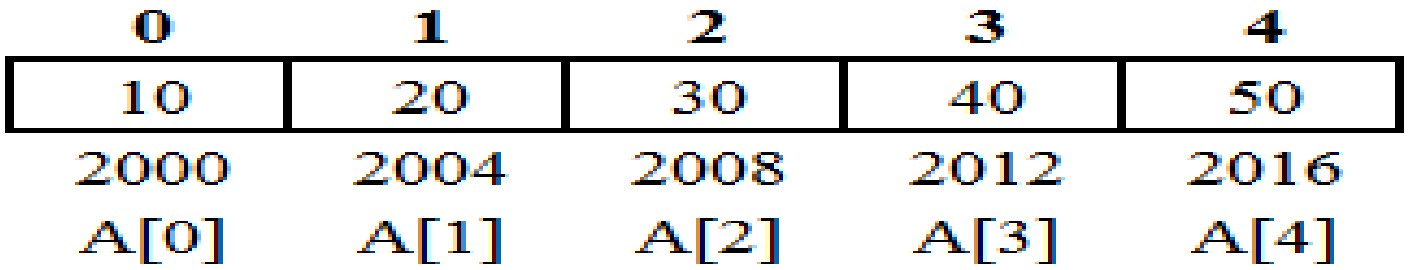
Item Retrieve(A, i) ::= **if** (i ∈ index) **return** the item associated with index value i in array A
else return error

Array Store(A, i, x) ::= **if (**i in index)
return an array that is identical to array A except the new pair <i, x> has been inserted **else return** error

Array Implementation

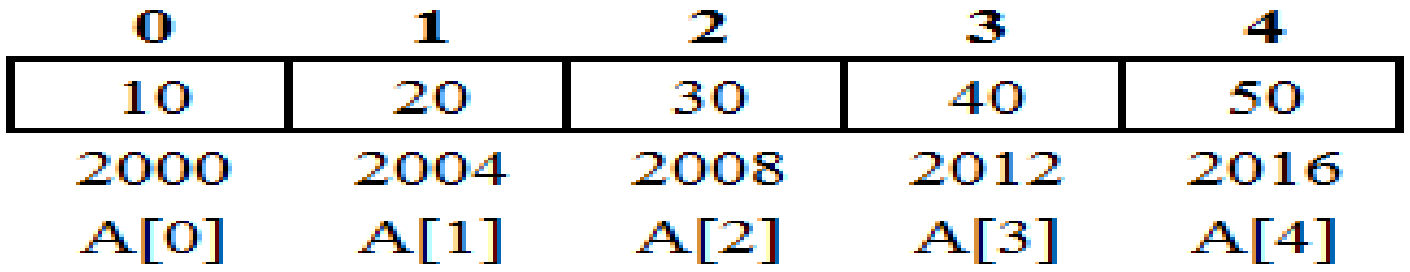
- A single dimension array is created as :
  - <data type> <array name> [size] ;
- Example
  - int A[5];
- The array is implemented as a constant pointer to the array elements.
- The elements are stored in consecutive memory locations. The address of the first element is called the base address.

Memory Representation



- The lower bound of array starts at 0 and upper bound is (size-1)
- The operation store is implemented as :
  - arrayname[position] = value ;
  - e.g. A[2] = 30;
- The operation extract is implemented as :
  - arrayname[position]
  - e.g. num = A[2];
- Any reference to element A[i] is converted to a pointer notation as
  - A[i] = i[A] = *(A + i)

Address Calculation



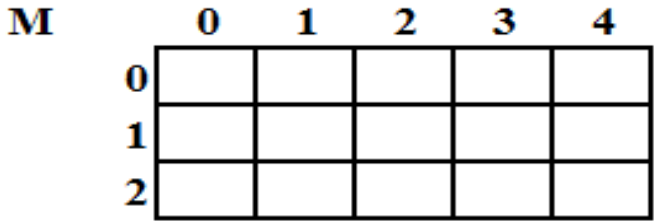
- Address of A[i] = Base Address + i * element size
- E.g.
- Address of A[3] = 2000 + 3 * sizeof(int)
  = 2000 + 3 * 4
  = 2012

Operations on Arrays

- Inserting an element into an array
- Deleting an element from the array
- Inserting an element into an sorted array
- Merging two sorted arrays

Multidimensional Arrays

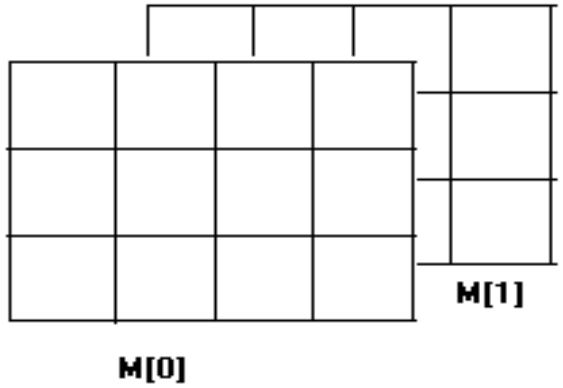
- When each array element is another array, it is called a multidimensional array.
- Example:
  - int M[3][5];
  - declares M as an array containing 3 elements each of which is an array containing 5 elements.



- An element of this array is accessed by specifying two indices. Such an array is called a two-dimensional array.
- M is said to have 3 rows and 5 columns. The rows are numbered 0 to 2 and columns are numbered 0 to 4.

Multidimensional Arrays

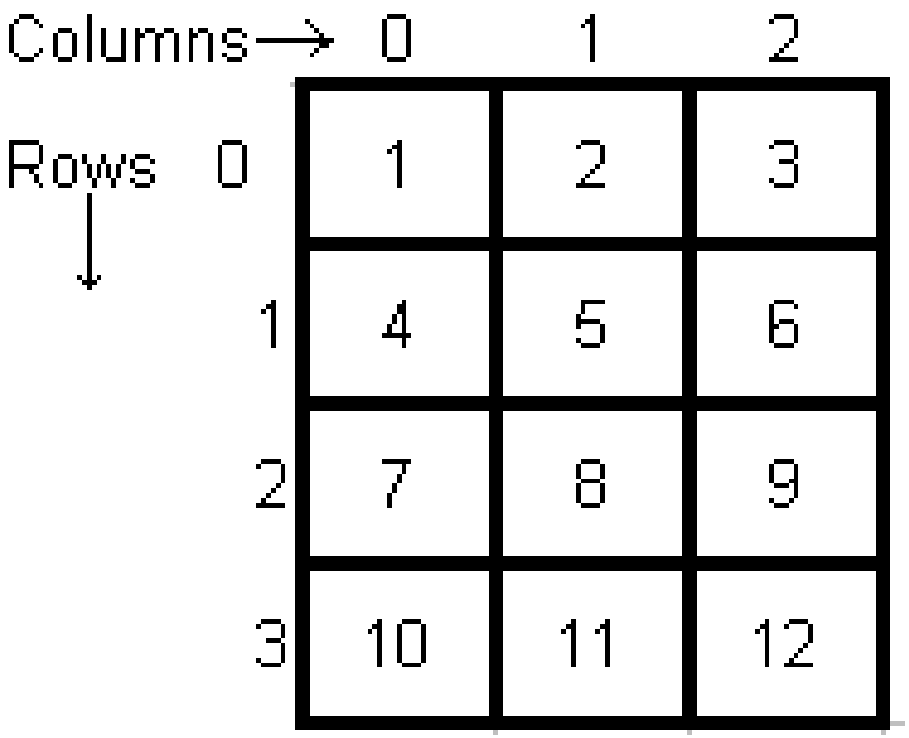
- An array can have more than two dimensions. For example, a three dimensional array may be declared as :
  - int M[2][3][4];
  - M is an array containing 2 two-dimensional arrays, each having 3 rows and 4 columns.



Representation of Multidimensional arrays

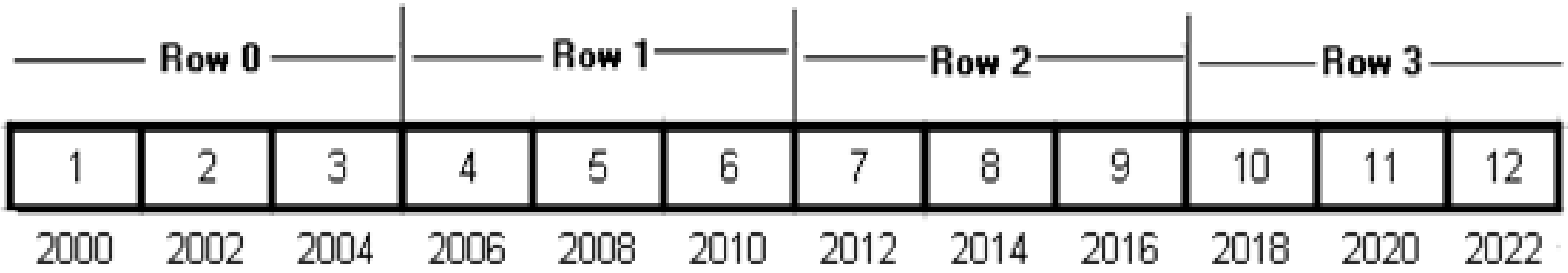
- A two-dimensional array is a logical data structure that is useful in describing an object that is physically two-dimensional such as a matrix or a check board. This is the logical view of data.
- However, computer memory is linear, i.e. it is essentially a one-dimensional array. Thus, the elements of a two-dimensional array have to be stored linearly in memory.
- They can be represented in two ways.
  - **Row-major representation :** In this representation, the elements are arranged row-wise i.e. the 0^{th} row elements are stored first, the next row elements after them and so on.
  - **Column-major representation :** Elements are arranged columnwise i.e. all elements of the 0th column occupy the first set of memory locations, elements of the first column come next and so on.

Logical View

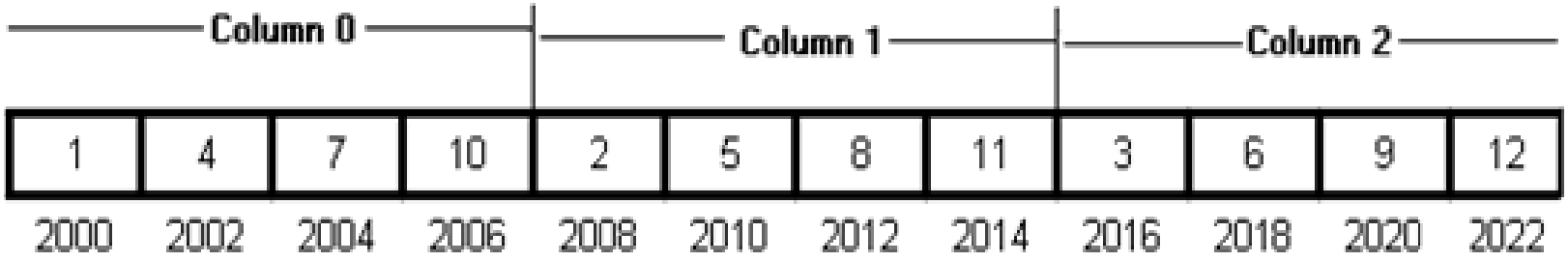


Physical View

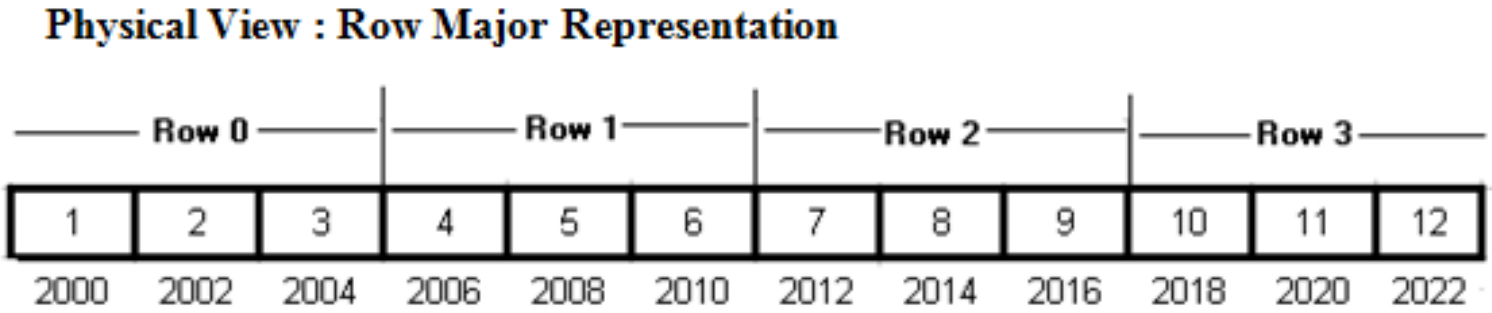
Physical View : Row Major Representation



Physical View : Column Major Representation



Address Calculation



For an array int M[R][C]; where R = number of rows and C = number of columns , the location of an element M[i][j] in the array can be calculated as :

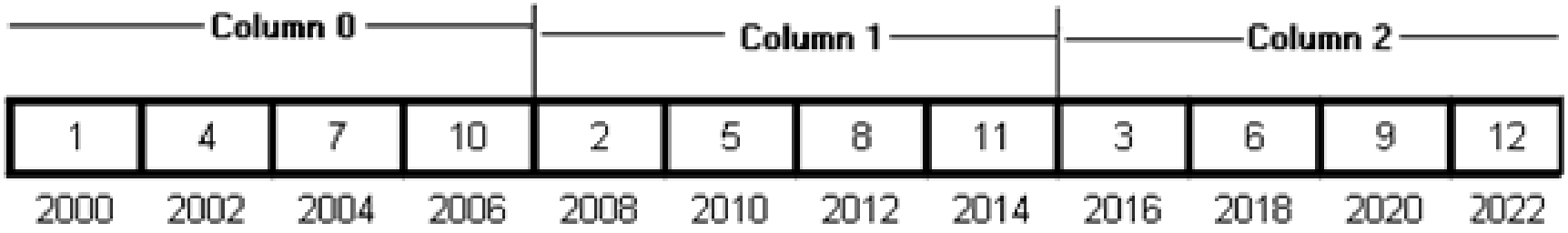
Address of M[i][j] = Base Address + i * C * Element size + j * Element size
= Base Address + (i * C + j) * Element size

Example :

Address of M[1][2] = 2000 + (1 * 3 + 2) * sizeof (int)
= 2000 + 5 * 2
= 2010

Address Calculation

Physical View : Column Major Representation



For an array int M[R][C]; where R = number of rows and C = number of columns , the location of an element M[i][j] in the array can be calculated as :

Address of M[i][j] = Base Address + j * R * Element size + i * Element size
= Base Address + (j * R + i) * Element size

Example :

Address of M[1][2] = 2000 + (2 * 4 + 1) * sizeof (int)
= 2000 + 9 * 2
= 2018