Arrays and Structures

Data Structures

Ching-Fang Hsu
Department of Computer Science and Information Engineering
National Cheng Kung University

The Array As An Abstract Data Type

- An array is a set of pairs, <index, value>, such that each index that is defined has a value associated with it.
  - A *correspondence* or a *mapping*
  - A homogeneous aggregate of data elements
- Standard operations provided by most languages (p.52, ADT 2.1)
  - Array creation
  - Value retrieval
  - Value setting

The Array As An Abstract Data Type (contd.)

- The implementation of one-dimensional arrays in C
  - When the compiler encounters an declaration for an array with type `t` and size n, it allocates n aconsecutive memory locations, where each one is large enough to hold a type `t` value.
  - The base address α -- the address of the first element of an array
    - The address of the i-th element = $\alpha + (i\text{-}1) * \text{sizeof}$ (`t`)
    - In C, we do not multiply the offset i and sizeof (`t`) to get the appropriate element of the array.

The Array As An Abstract Data Type (contd.)

- `list[i]` ≡ `*(list + i)`
- Dereferencing -- the pointer is interpreted as an indirect reference
  - p. 54, Program 2.2

The Polynomial Abstract Data Type

- Ordered / linear lists
  - ($item_0$, $item_1$, …, $item_{n-1}$)
  - Operations on lists (p. 65)
- Representing an ordered list as an array
  - Associate $item_i$ with the array index i. ➪ a sequential mapping
  - Sequential mapping works well for most operations listed in page 65 in constant time, except insertion and deletion.
    - A motivation that leads us to consider nonsequential mappings

The Polynomial Abstract Data Type (contd.)

- Example: Build a set of functions for manipulation of symbolic polynomials
  - ADT (p.67, ADT 2.2)
- For simplifying operations, exponents are arranged in decreasing order.
  - Operation Add can be achieved by comparing terms from the two polynomials until one or both of the polynomials becomes empty.
    - Initial version of *padd* function (p. 68, Program 2.5)

The Polynomial Abstract Data Type -- Representation

- Option 1 (p. 66~68)
  - Maximum degree is restricted by `MAX_DEGREE`.

```
#define MAX_DEGREE 101
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
        } polynomial;
```

  - The main drawback : lower flexibility on space requirement
    - Wasting a lot of space when the degree of the polynomial is much less then `MAX_DEGREE` or the polynomial is sparse

The Polynomial Abstract Data Type -- Representation (contd.)

- Option 2 (p. 68~69)
  - Representing $a_i x^i$ as a structure and using only one global array of this structure to store all polynomials (p. 68~69)

```
#define MAX_TERMS 100
typedef struct {
        float coef;
        int expon;
        } polynomial;
polynomial terms[MAX_TERMS];
int avail = 0;
```

The Polynomial Abstract Data Type -- Representation (contd.)

$A(x) = 2x^{1000} + 1 \qquad B(x) = x^4 + 10x^3 + 3x^2 + 1$

	startA	*finishA*	*startB*			*finishB*	*avail*
	↓	↓	↓			↓	↓
coef	2	1	1	10	3	1	
expon	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

The Polynomial Abstract Data Type -- Representation (contd.)

- No limit on the number of polynomials stored in the global array
- The index of the first (last) term of polynomial A is given by `starta` (`finisha`).
  - `finisha` = `starta` + *n* - 1, if A has n nonzero terms
- The index of the next free location in the array is given by `avail`.
- The main drawback: About twice as much space as option 1 is needed when all the terms are nonzero.
- The revised function *padd* (p. 70, Program 2.6)

The Polynomial Abstract Data Type -- Representation (contd.)

- Analysis of Program 2.6
  - Each iteration of the while-loop: $O(1)$
  - The number of iterations: bounded by $m + n - 1 \Rightarrow O(n + m)$
    - The worst case (p. 71)
  - The time for two for-loops: bounded by $O(n + m)$

$\Rightarrow$ The asymptotic time of the algorithm for operation Add is $O(n + m)$.

The Sparse Matrix Abstract Data Type

- A matrix containing many zero entries is called a *sparse matrix*.
  - Difficult to determine exactly whether a matrix is sparse or not
- The standard representation of a matrix is a two-dimensional array, but not appropriate for a sparse matrix due to a waste of space.
  - Storing only non-zero elements is a feasible solution for a sparse matrix.

The Sparse Matrix Abstract Data Type (contd.)

- A minimal set of matrix operations
  - Creation
  - Addition
  - Transpose
  - Multiplication
- The ADT of a sparse matrix (p. 74, ADT 2.3)
- Using the triple <*row*, *col*, *value*> to characterize an element within a matrix.
  - A sparse matrix $\equiv$ an array of triples

The Sparse Matrix Abstract Data Type (contd.)

- For efficient transpose operation, the triples are ordered by rows and within rows by columns.
- With the triple definition, the number of rows and columns, and the number of nonzero elements, the Create operation can be derived (p. 75).

The Sparse Matrix Abstract Data Type -- Transposing a Matrix

- A simple algorithm for transposing

```
for all elements in column j
    place element <i, j, value> in
    element <j, i, value>
```

  - p. 77, Program 2.8
  - Time complexity: $O(columns \cdot elements)$
  - cf. $O(rows \cdot columns)$ with a two-dimensional array representation

```
for (j = 0; j < columns; j++)
  for (i = 0; i < rows; i++)
     b[j][i] = a[i][j];
```

$$A = \begin{bmatrix} 15 & X & X & 22 & X & -15 \\ X & 11 & 3 & X & X & X \\ X & X & X & -6 & X & X \\ X & X & X & X & X & X \\ 91 & X & X & X & X & X \\ X & X & 28 & X & X & X \end{bmatrix} \qquad A^T = \begin{bmatrix} 15 & X & X & X & 91 & X \\ X & 11 & X & X & X & X \\ X & 3 & X & X & X & 28 \\ 22 & X & -6 & X & X & X \\ X & X & X & X & X & X \\ -15 & X & X & X & X & X \end{bmatrix}$$

	row	col	value
$a[0]$	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

	row	col	value
$b[0]$	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

	[0]	[1]	[2]	[3]	[4]	[5]
$rowTerms$ =	2	1	2	2	0	1
$startingPos$ =	1	3	4	6	8	8

The Sparse Matrix Abstract Data Type -- Transposing a Matrix (contd.)

- The $O(columns \cdot elements)$ time becomes $O(columns^2 \cdot rows)$ when the number of elements is of the order *columns* · *rows*.
  - An improved version: *fast_transpose* (p. 78, Program 2.9) with $O(columns + elements)$ complexity

The Sparse Matrix Abstract Data Type -- Transposing a Matrix (contd.)

- Analysis of Program 2.9
  - The 1st for-loop: $O(columns)$
    - `row_terms` initialization
  - The 2nd for-loop: $O(elements)$
    - calculating # of non-zero elements within each column
  - The 3rd for-loop: $O(columns)$
    - starting positions calculations
  - The 4th for-loop: $O(elements)$
    - value setting for array b

$\Rightarrow$ The time complexity of *fast_transpose* is $O(columns + elements)$.

The Sparse Matrix Abstract Data Type -- Matrix Multiplication

- **Definition**: Given A and B where A is $m \times n$ and B is $n \times p$, the <i, j> element of the product matrix D is

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

- Step 1: Compute the transpose of B.
- Step 2: Do a merge operation similar to that used in the polynomial addition.
- p. 81~82, Program 2.10, 2.11
  - The for-loop: $O(\sum_{row} (colsB \bullet termsRow + totalB))$

$$= O(colsB * totalA + rowsA * totalB)$$

$$
\begin{array}{cc}
 & \begin{array}{cccccc} i & & & i & & i \end{array} \\
\text{row} \rightarrow & \\
A = & \begin{bmatrix} 15 & X & X & 22 & X & -15 \\ X & 11 & 3 & X & X & X \\ X & X & X & -6 & X & X \\ 91 & X & X & X & X & X \\ X & X & 28 & X & X & X \end{bmatrix}
\end{array}
\qquad
\begin{array}{cc}
 & \begin{array}{cccccc} j & j & & & j & \end{array} \\
\text{column} \rightarrow & \\
B^T = & \begin{bmatrix} X & X & X & X & X & X \\ X & 7 & X & X & -9 & X \\ -1 & X & X & 13 & X & X \\ X & X & 23 & X & X & 2 \\ X & X & X & X & X & X \\ X & -5 & X & 12 & 5 & 3 \\ X & X & 6 & X & X & X \end{bmatrix}
\end{array}
$$

rowBegin

Representation of Multidimensional Arrays

- Two common ways
  - Row major order
    - Storing multidimensional arrays by rows
  - Column major order
- Assume that α is the starting address of a n-dimensional array $A[upper_0][upper_1]\ldots[upper_{\text{n-1}}]$.
  - The address for $A[i_0][i_1]\ldots[i_{n\text{-}1}]$ is:

$$\alpha + \sum_{j=0}^{n-1} i_j a_j \ \text{where} \ a_j = \begin{cases} \prod_{k=j+1}^{n-1} upper_k & 0 \le j < n-1 \\ 1 & j = n-1 \end{cases}$$

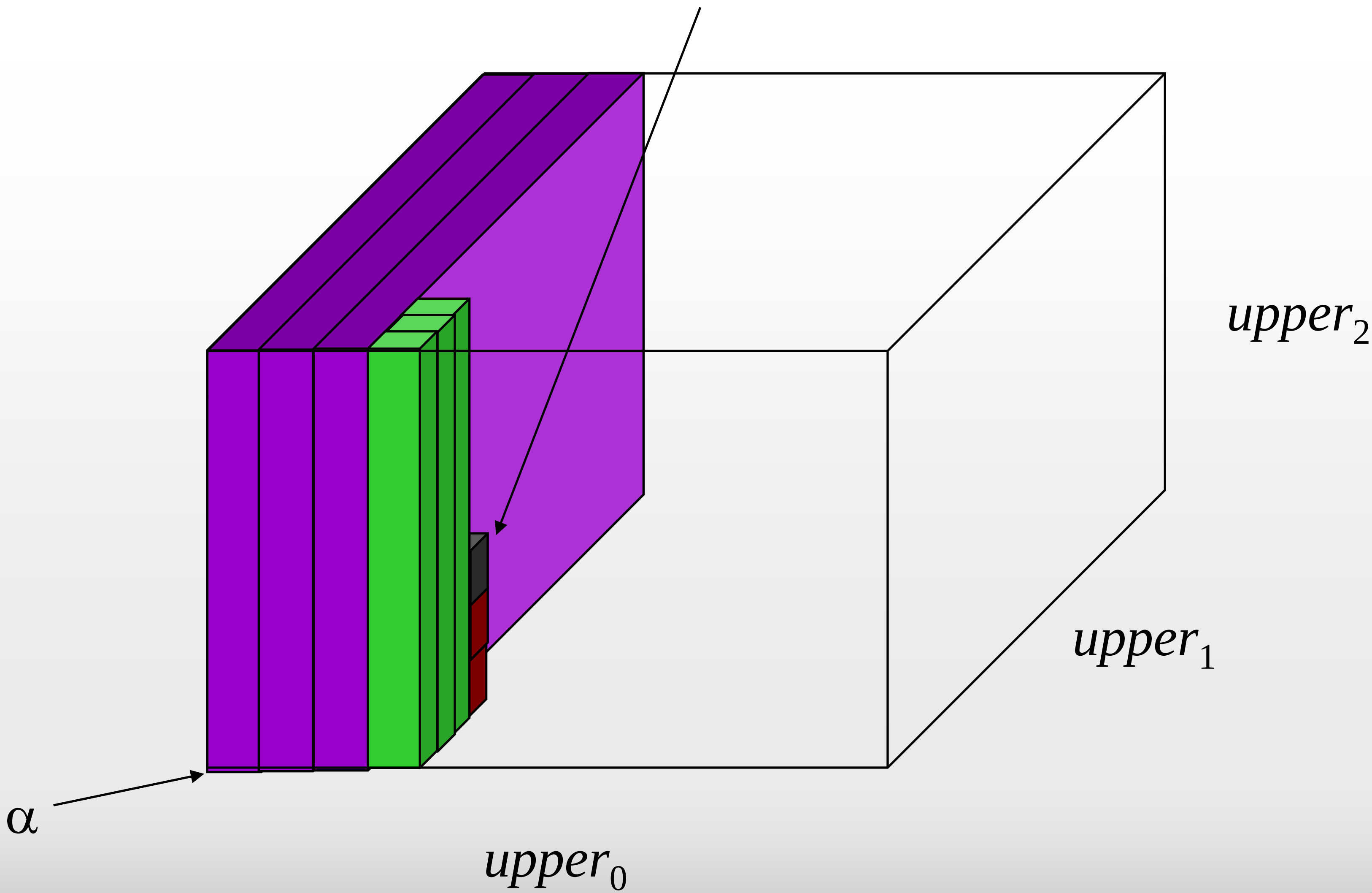
Row major

α

$$upper_0 \left\{ \underbrace{\begin{bmatrix} \# & \# & \# & \# & \# & \# \\ \# & \# & \# & \# & \# & \# \\ \# & \# & * & \# & \# & \# \end{bmatrix}}_{upper_1} \right.$$

$$A[i_0][i_1] \rightarrow \alpha + i_0 * upper_1 + i_1 * 1$$

$$A[i_0][i_1][i_2] \rightarrow \alpha + i_0 * upper_1 * upper_2 + i_1 * upper_2 + i_2 * 1$$



Representation of Multidimensional Arrays (contd.)

- A compiler will initially take the declared bounds (i.e., $upper_k$, $0 \le k \le n\text{-}1$) and use them to compute the constants a_j, $0 \le j \le n\text{-}2$.
- The computation of the address of $A[i_0][i_1]\ldots[i_{n\text{-}1}]$ requires $n\text{-}1$ more multiplications and n additions.