Linear Lists – Array Representation

Slides and figures have been collected from various publicly available Internet sources for preparing the lecture slides of IT2001 course. I acknowledge and thank all the original authors for their contribution to prepare the content.

Introduction

- Data Object
  - a set of instances or values
  - Examples:
    - Boolean = {false, true}
    - Digit = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    - Letter = {A, B, C, …, Z, a, b, c, …, z}
    - String = {a, b, …, aa, ab, ac,…}
  - An individual instance is either primitive or composite.
  - An element is the individual component of a composite instance.

Introduction

- Data Structure
  - data object together with the relationships among instances and elements that comprise an instance
  - Among instances of integer

    369 < 370

    280+4 = 284
  - Among elements that comprise an instance

    369

    3 is more significant than 6

    3 is immediately to the left of 6

    9 is immediately to the right of 6

Introduction

- Abstract Data Type (ADT)
  - mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. (WiKipedia)
  - ADT is a collection of data and a set of operations that can be performed on the data.
  - It enables us to think abstractly about the data.
  - Typically, we choose a data structure and algorithms that provide an implementation of an ADT.

Data structures: Classification

- Data structures are classified into 2 categories:
  - Linear
  - Non-linear
- Linear data structures organize their data elements in a linear fashion, where data elements are attached one after the other.
- Data elements in a liner data structure are traversed one after the other and only one element can be directly reached while traversing.
- Consider the examples: ordered lists
  - Days of the week: (mon, tue, wed, thu, fri, sat, sun)
  - Student's roll number in a class
  - E.g., Array, Linked List, Stack, Queue

Data structures: Classification

- In nonlinear data structures, data elements are not organized in a sequential fashion.
- A data item in a nonlinear data structure could be attached to several other data elements to reflect a special relationship among them and all the data items cannot be traversed in a single run.
- E.g., Tree, Graphs

Linear List

- Definitions
  - Linear list is a data object whose instances are of the form $(e_1,e_2,\ldots,e_n)$
  - e_i is an element of the list.
  - e_1 is the first element, and e_n is the last element.
  - n is the length of the list.
  - When $n = 0$, it is called an empty list.
  - e_1 comes before e_2, e_2 comes before e_3, and so on.
- Examples
  - student names order by their alphabets
  - a list of exam scores sorted by descending order

ADT for Linear List

AbstractDataType LinearList {

instances

ordered finite collections of zero or more elements

operations

Create(): create an empty linear list

Destroy(): erase the list

IsEmpty(): return true if empty, false otherwise

Length(): return the list size

Find(k,x): return the k^{th} element of the list in x

Search(x): return the position of x in the list

Delete(k,x): delete the k^{th} element and return it in x

Insert(k,x): insert x just after the k^{th} element

Output(out): put the list into the output stream *out*

}

Implementations of Linear List

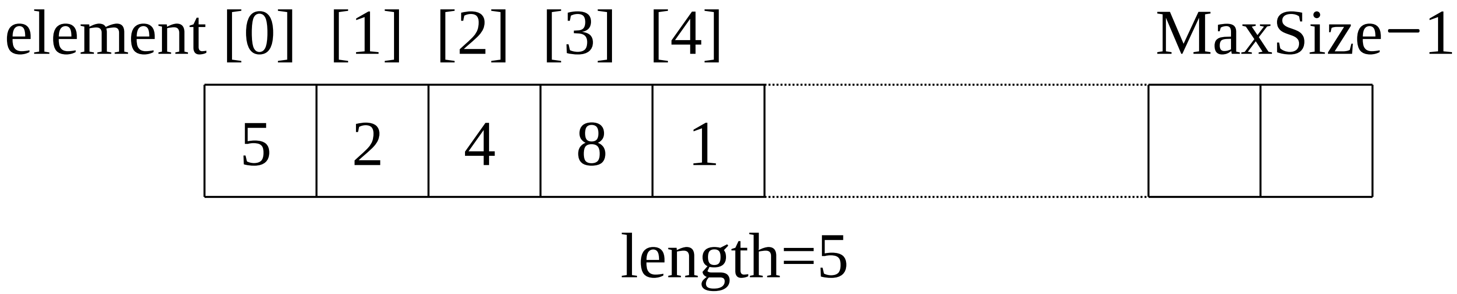
- Array-based (Formula-based)
  - Uses a mathematical formula to determine where (i.e., the memory address) to store each element of a list
- Linked list (Pointer-based)
  - The elements of a list may be stored in any arbitrary set of locations
  - Each element has an explicit pointer (or link) to the next element
- Indirect addressing
  - The elements of a list may be stored in any arbitrary set of locations
  - Maintain a table such that the *i*th table entry tells us where the *i*th element is stored
- Simulated pointer
  - Similar to linked representation but integers replace the C++ pointers

Array-based Representation of Linear List

- It uses an array to store the elements of linear list.
- Individual element is located in the array using a mathematical formula.
- typical formula

  location(i) = i – 1

  → *i*th element of the list is in position *i-1* of the array



Construct ‘LinearList’

```
Void LinearList(int MaxSize)
{   // Construction of array-based linear list
    int element[MaxSize];
}
```

- The time complexity is: **$\Theta(1)$**

Operation ‘Find’

```
#define true 1
#define false 0
typedef int bool;
bool Find(int k, int x)
{   // Set x to the  kth  element in the list if it exists
    if (k < 1 || k > length)
            return false;
    x = element[k-1];
    return true;
}
```

- The time complexity is: Θ**(1)**

Operation ‘Search’

```
int Search( int element[ ], int x )
{   // Locate x and return the position of x if found
    for (int i = 0; i < length; i++)
            if (element[i] == x)
                        return ++i;
    return 0;
}
```

- The time complexity is: **O(length)**

Operation ‘Delete’

```
Delete(int element[ ], int k, int x)
{   // Delete the k’th element if it exists.
    if (Find(k, x)) {
            for (int i = k, i < length; i++)
                    element[i-1] = element[i];
            length--;
            }
}
```

- The time complexity is **O(length)**

Operation ‘Insert’

```
Insert(int k, x)
{   // Insert x after the k’th element.
    //length: # of elements in the list
    if (k < 0 || k > length)  error
    if (length == MaxSize) error
    for (i = length-1; i >= k; i--)
          element[i+1] = element[i];
    element[k] = x;
    length++;
    }
```

- The time complexity is **O(length)**

Operation ‘Output’

```
Output(element  [])
{  // print out the list
   for (int i = 0; i < length; i++)
         printf(“%d” , element[i] );
}
```

- The time complexity is **Θ(length)**

Arrays and Matrices

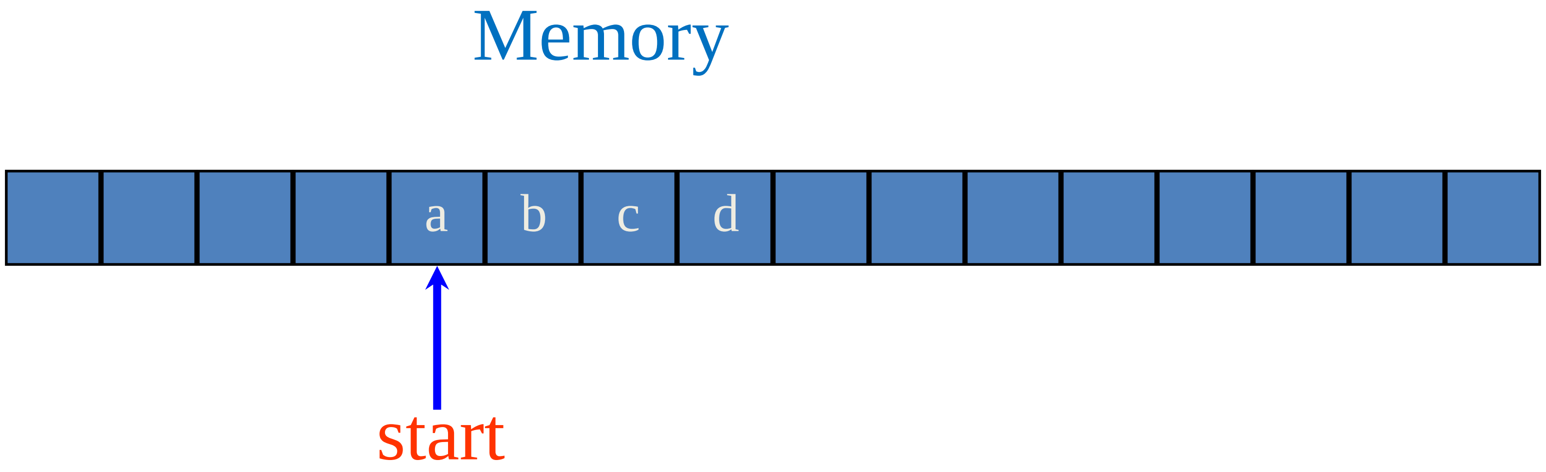
Introduction

- Data is often available in tabular form
- Tabular data is often represented in arrays
- Matrix is an example of tabular data and is often represented as a 2-dimensional array
  - Matrices are normally indexed beginning at 1 rather than 0
  - Matrices also support operations such as **add**, **multiply**, and **transpose**, which are NOT supported by C/C++'s 2D array

Introduction

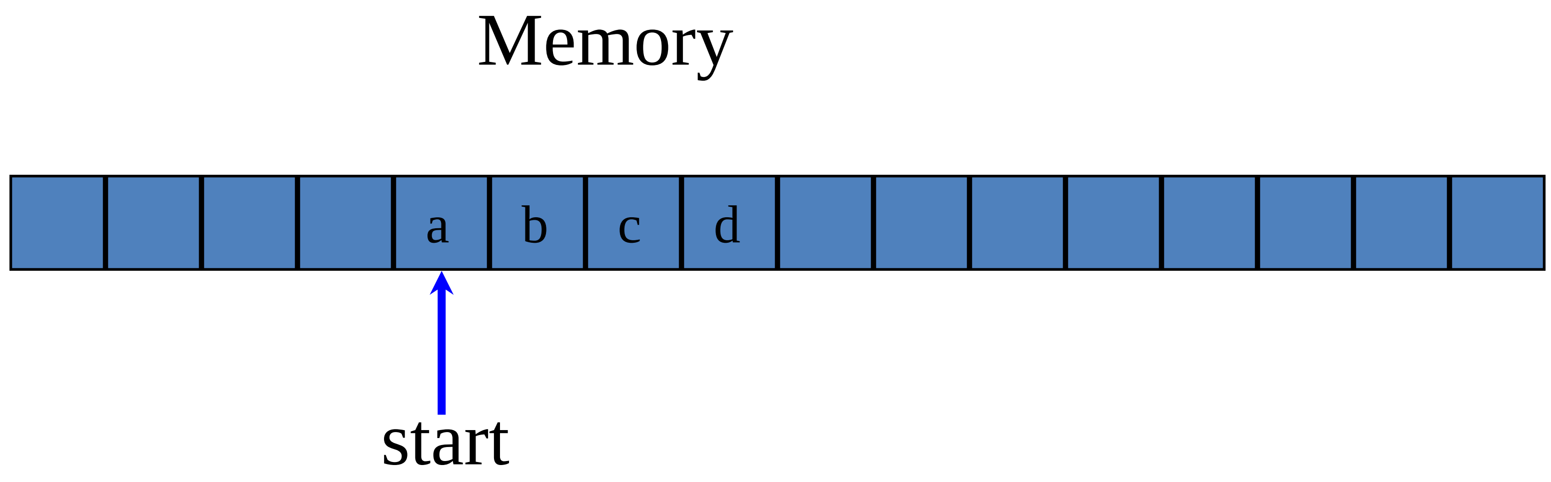
- It is possible to **reduce time and space** using a **customized representation** of multidimensional arrays
- Focus on
  - Row- and column-major ordering and representations of multidimensional arrays
  - Special matrices
    - Diagonal, tridiagonal, triangular, symmetric, sparse

1D Array Representation in C/C++



- 1-dimensional array x = [a, b, c, d]
- map into <u>contiguous memory</u> locations
- location(x[i]) = start + i

Space Overhead



space overhead = 4 bytes for start

(excludes space needed for the elements of array x)

2D Arrays

The elements of a 2-dimensional array a declared as:

```
int a[3][4];
```

may be shown as a table

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Rows of a 2D Array

a[0][0]	a[0][1]	a[0][2]	a[0][3]	row 0
a[1][0]	a[1][1]	a[1][2]	a[1][3]	row 1
a[2][0]	a[2][1]	a[2][2]	a[2][3]	row 2

Columns of a 2D Array

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]
column 0	column 1	column 2	column 3

2D Array Representation in C/C++

2-dimensional array x

a, b, c, d

e, f, g, h

i, j, k, l

view 2D array as a 1D array of rows

x = [row0, row1, row 2]

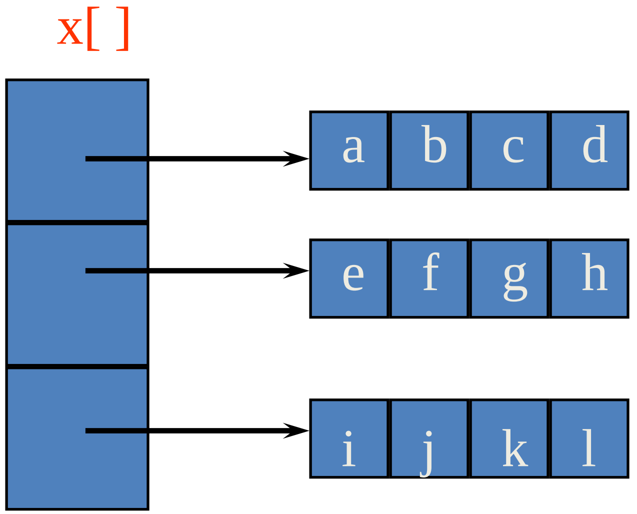
row 0 = [a, b, c, d]

row 1 = [e, f, g, h]

row 2 = [i, j, k, l]

and store as 4 1D arrays

2D Array Representation in C/C++



4 separate

1-dimensional arrays

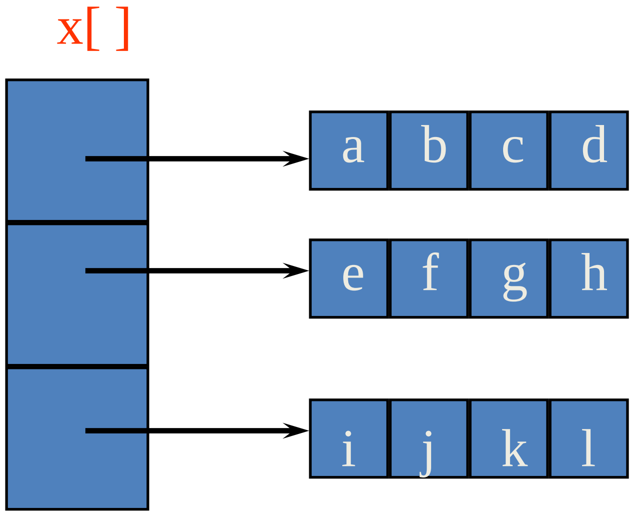
- space overhead = overhead for 4 1D arrays

  = 4 * 4 bytes

  = 16 bytes

  = (number of rows + 1) x 4 bytes

Array Representation in C/C++



- This representation is called the array-of-arrays representation.
- Requires contiguous memory of size 3, 4, 4, and 4 for the 4 1D arrays.
- 1 memory block of size number of rows and number of rows blocks of size number of columns

Row-Major Mapping

- Example 3 x 4 array:

  a b c d

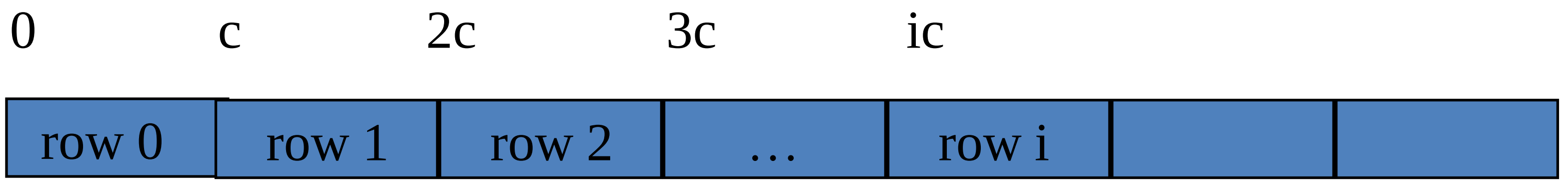
  e f g h

  i j k l

- Convert into 1D array y by collecting elements by rows.
- Within a row elements are collected from left to right.
- Rows are collected from top to bottom.
- We get y[ ] = {a, b, c, d, e, f, g, h, i, j, k, l}

row 0	row 1	row 2	…	row i		

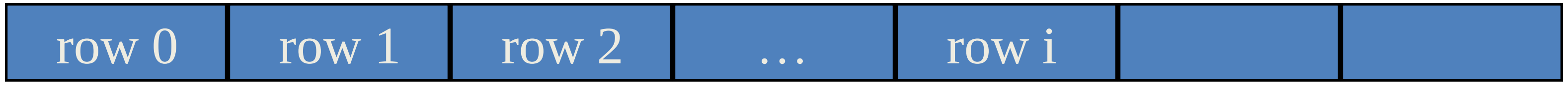
Locating Element x[i][j]



- assume x has r rows and c columns
- each row has c elements
- i rows to the left of row i
- so ic elements to the left of x[i][0]
- x[i][j] is mapped to position

  ic + j of the 1D array

Space Overhead



4 bytes for start of 1D array +

4 bytes for c (number of columns)

= 8 bytes

Note that we need contiguous memory of size rc.

Column-Major Mapping

a b c d

e f g h

i j k l

- Convert into 1D array y by collecting elements by columns.
- Within a column elements are collected from top to bottom.
- Columns are collected from left to right.
- We get y[ ] = {a, e, i, b, f, j, c, g, k, d, h, l}

Row- and Column-Major Mappings

2D Array int a[3][6];

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]	a[0][5]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]	a[1][5]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]	a[2][5]

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17

(a) Row-major mapping

0	3	6	9	12	15
1	4	7	10	13	16
2	5	8	11	14	17

(b) Column-major mapping

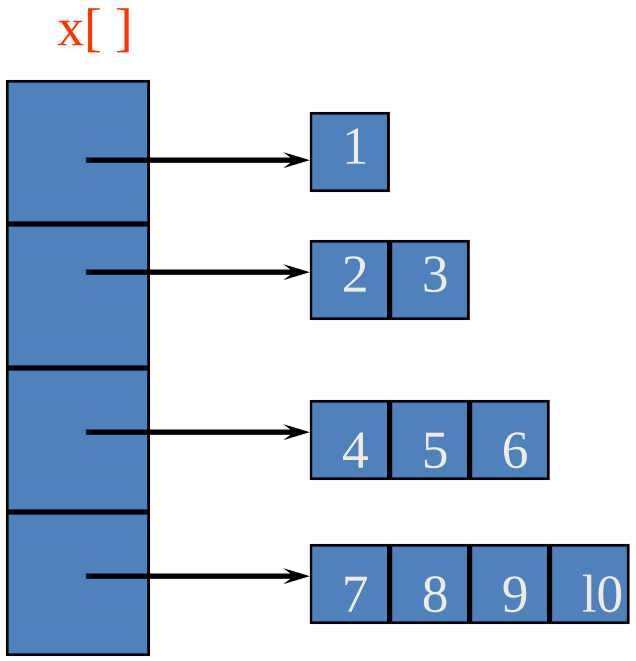
Row- and Column-Major Mappings

- Row-major order mapping functions
  $map(i_1,i_2) = i_1u_2+i_2$ for 2D arrays
  $map(i_1,i_2,i_3) = i_1u_2u_3+i_2u_3+i_3$ for 3D arrays

- What is the mapping function?
  $map(i_1,i_2) = 6i_1+i_2$
  $map(2,3) = ?$

- Column-major order mapping functions
  // do this as an exercise

Irregular 2D Arrays



Irregular 2-D array: the length of rows is not required to be the same.

Matrices

- *m x n* matrix is a table with *m* rows and *n* columns.
- *M(i,j)* denotes the element in row *i* and column *j*.
- Common matrix operations
  - transpose
  - addition
  - multiplication

$$
\begin{array}{c|cccc|}
 & \text{col 1} & \text{col 2} & \text{col 3} & \text{col 4} \\
\text{row 1} & 7 & 2 & 0 & 9 \\
\text{row 2} & 0 & 1 & 0 & 5 \\
\text{row 3} & 6 & 4 & 2 & 0 \\
\text{row 4} & 8 & 2 & 7 & 3 \\
\text{row 5} & 1 & 4 & 9 & 6
\end{array}
$$

Matrix Operations

- Transpose
  - The result of transposing an *m x n* matrix is an *n x m* matrix with property:

    $M^T(j,i) = M(i,j)$, $1 <= i <= m$, $1 <= j <= n$

- Addition
  - The sum of matrices is only defined for matrices that have the same dimensions.
  - The sum of two *m x n* matrices A and B is an *m x n* matrix with the property:

    $C(i,j) = A(i,j) + B(i,j)$, $1 <= i <= m$, $1 <= j <= n$

Matrix Operations

- Multiplication
  - The product of matrices A and B is only defined when the number of columns in A is equal to the number of rows in B.
  - Let A be *m x n* matrix and B be a *n x q* matrix. A*B will produce an *m x q* matrix with the following property:

$$C(i,j) = \Sigma(k=1 \ldots n)\ A(i,k) * B(k,j)$$

where $1 <= i <= m$ and $1 <= j <= q$

Shortcomings of using a 2D Array for a Matrix

- Indexes are off by 1.
- C/C++ arrays do not support matrix operations such as add, transpose, multiply, and so on.
  - Suppose that x and y are 2D arrays. Cannot do x + y,   x –y, x * y, etc. in C/C++.