More on Arrays and Structures

Hsuan-Tien Lin

Dept. of CSIE, NTU

March 15–16, 2011
What We Have Done

- Why Correctness Proof?
- Sequential and Binary Search
- Space Complexity
- Time Complexity
- Asymptotic Notations
- Practical Complexity
- Arrays
- Dense Array versus Sparse Array
- Concrete versus Abstract Data Type
- Reading Assignment:
  Some Examples of Space/Time Complexity, Dynamic 1-D Array
Arrays: from Implementation to Abstraction

C Implementation View

(One-dimensional) array is a block of consecutive memory that
- holds a list of $N$ elements
- allows users to retrieve the $k$-th element
- allows users to store to the $k$-th location

An Abstract View

Abstract (one-dimensional) array
- holds a list of $N$ elements
- allows users to retrieve the $k$-th element
- allows users to store to the $k$-th location

different implementations:
different space/time complexity
Dense Array versus Sparse Array

one abstract array, two possible implementations

```c
1 int dense[10] = {1, 3, 0, 0, 0, 0, 0, 0, 0, 2};
2 int sparse[3][2] = {{0, 1}, {1, 3}, {9, 2}};
```

dense array: store everything (consecutively), needs 10 positions
- space: $O(N)$ for a length-$N$ array
- retrieving: $O(1)$
- storing: $O(1)$
- creating: $O(1)$

sparse array: store only non-zero (index, element) pairs, needs 3 pairs
- space: $O(E)$ for $E$ elements, better than $O(N)$ if $E$ small
- retrieving: $O(\log E)$ if index ordered (HOW?)
- storing: ???
- creating: ???

note: often use `array` to mean dense array only
Concrete Data Type (Sec. 1.4)

array consists of ...

- objects: a set of \((index, element)\) pairs (== a list of elements)
- actions: retrieve, store, create which sets/gets the objects

concrete data type: the **actual outcome** of the type

- object representation + action implementation
- for actual coding, per-platform optimization, etc.

(dense 1-D) array in C

- object representation: a block of consecutive memory, with a chunk representing each \(element\) element for each \(index\)
- action implementation: \([\cdot]\) for retrieving and storing, \texttt{malloc} for creating, etc.
Abstract Data Type (Sec. 1.4)

**array consists of ...**
- objects: a set of \((index, element)\) pairs (== a list of elements)
- actions: retrieve, store, create which sets/gets the objects

- abstract data type: the **pseudo essence** (**contract**) of the type
  - object specification + action specification
  - for illustration, high-level analysis, etc.

**abstract 1-D array**
- object specification: \((index, element)\) pairs with \(index \in \{0, \cdots, N - 1\}\)
- action specification:
  - retrieve(index) returns the element associated with index;
  - store(index, element) sets element to be associated with index;
  - create(N) creates the objects, etc.
  - (sometimes with time/space constraints)

will usually look at abstract data type first before going concrete
2-D Array (Subsec. 2.2.2): by 1-D Array

abstract rectangular 2-D array

- object specification: \((index, element)\) pairs with
  \(index \in \{(0, 0), (0, 1), \ldots, (N-1, M-1)\}\)
- action specification:
  \(\text{retrieve}(index); \text{store}(index, element); \text{create}(N, M), \text{etc.}\)

2-D array by 1-D array in C

- object representation: a block of consecutive memory of size \(N \times M\), with a chunk representing each \(element\) for each \(index\)
- action implementation:
#define N (100)
#define M (200)
int* twodim = (int*)malloc(sizeof(int)*N*M);

int get(int* arr, int n, int m)
{
    return arr[n*M + m];
}
2-D Array: by 1-D Array with Constant Folding

**abstract rectangular 2-D array**

- **object specification:** \((index, element)\) pairs with \(index \in \{(0, 0), (0, 1), \ldots, (N - 1, M - 1)\}\)
- **action specification:**
  - retrieve\((index)\); store\((index, element)\); create\((N, M)\), etc.

**2-D array by 1-D array with constant folding in C**

- **object representation:** a block of consecutive memory of size \(N \times M\), with a chunk representing each \(element\) for each \(index\)
- **action implementation:**
#define N (100)
#define M (200)
int twodim[N][M];

int get(int arr[][M], int n, int m)
{
    return arr[n][m];
}
2-D Array: by Array of Arrays

abstract rectangular 2-D array
- object specification: \((index, element)\) pairs with \(index \in \{(0, 0), (0, 1), \ldots, (N - 1, M - 1)\}\)
- action specification:
  - retrieve\((index)\)
  - store\((index, element)\)
  - create\((N, M)\)
  etc.

2-D array by array of arrays in C
- object representation: \(N\) blocks of consecutive memory of size \(M\)
- action implementation:
```c
#define N (100)
#define M (200)
int** twodim = (int**)malloc(sizeof(int*)*N);
for (int n=0;n<N;n++)
    twodim[n] = (int*)malloc(sizeof(int)*M);
int get(int** arr, int n, int m)
    { return arr[n][m];}
```
### Comparison of Three Implementations

1. \[ \text{int* twodim = (int*)malloc(sizeof(int)*N*M);} \]
2. \[ \text{int twodim[N][M];} \]
3. \[ \text{int** twodim = (int**)malloc(sizeof(int*)*N);} \]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>( N \times M ) integers</td>
<td>( N \times M ) int.</td>
<td>( N \times M ) int. + ( N ) pointers</td>
</tr>
<tr>
<td>type</td>
<td>int*</td>
<td>int*[M]</td>
<td>int**</td>
</tr>
<tr>
<td>create</td>
<td>constant</td>
<td>constant</td>
<td>( O(N) )</td>
</tr>
<tr>
<td>retrieve</td>
<td>arithmetic+dereference</td>
<td>arith.+deref.</td>
<td>deref.+deref.</td>
</tr>
</tbody>
</table>

- Method 2 for static allocating; method 1 or 3 for dynamic allocating (your choice)
A Tale between Two Programs

```
int rowsum()
{
    int i, j;
    int res = 0;
    for (i=0; i<MAXROW; i++)
        for (j=0; j<MAXCOL; j++)
            res += array[i][j];
}
```

```
int colsum()
{
    int i, j;
    int res = 0;
    for (j=0; j<MAXCOL; j++)
        for (i=0; i<MAXROW; i++)
            res += array[i][j];
}
```
Reading Assignment

be sure to go ask the TAs or me if you are still confused
typedef struct {
    int degree;
    double* coef;
} densepoly;

typedef struct {
    int nTerms;
    int* expo; /* expo[i] ordered */
    double* coef;
} sparsepoly;

- **densepoly** versus **sparsepoly**: like dense array versus sparse
- a simple polynomial adding algorithm
  - allocate the resulting poly
  - fill in the values
- trivial for **densepoly** (HW1), slightly harder for **sparsepoly**
typedef struct{
    int nTerms;
    int* expo; /* expo[i] ordered */
    double* coef;
} sparsepoly;

add $x^{100} + 2x^3 + 3$ and $4x^4 + 5x^3 + 6x + 7$

- allocate the resulting poly

- fill in the values
Abstract Polynomials (Subsec. 2.4.1) and Other Parts of Sec. 2.4

Reading Assignment

be sure to go ask the TAs or me if you are still confused
Sparse Matrix (Sec. 2.5)

\[
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0 \\
\end{bmatrix}
\]

**Specialty**

a rectangular 2-D array that contains many common elements (0) that we may not want to repeatedly store
matrix consists of ...

- objects: a set of \((row, column, value)\) triples where \(value\) is numerical
- actions: create, transpose, add, multiply, (retrieve, store)

- dense implementation: as 2D dense arrays
- array of array implementation:
  - \("(dense 1D) of (sparse 1D)"\)
  - \("(sparse 1D) of (sparse 1D)"\)
- ordered triples implementation: our next topic
Ordered Triples Implementation

\[
\begin{pmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{pmatrix}
\]

Ordered (-by-row-then-by-col) Triples

<table>
<thead>
<tr>
<th>row</th>
<th>col</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>-15</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>-6</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>91</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>28</td>
</tr>
</tbody>
</table>

- space complexity? \(O(E)\)
- time complexity for retrieve? \(O(\log E)\)

Simple exercise: compare to unordered triple implementation
Transposing A Sparse Matrix

\[
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0
\end{bmatrix}^T =
\begin{bmatrix}
15 & 0 & 0 \\
0 & 11 & 0 \\
0 & 3 & 0 \\
22 & 0 & -6 \\
0 & 0 & 0 \\
-15 & 0 & 0
\end{bmatrix}
\]

Ordered(-by-row-then-by-col) Triples

<table>
<thead>
<tr>
<th>row</th>
<th>col</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>-15</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>-6</td>
</tr>
</tbody>
</table>

\[
\begin{bmatrix}
1 & 1 & 11 \\
2 & 3 & 3 \\
3 & 5 & 0 \\
5 & 0 & -15
\end{bmatrix}
\]
set up a sparse matrix \(res\) of attributes \((col, row, elements)\)

\[
\text{for } j \leftarrow 0 \text{ to } col - 1 \text{ do}
\]

\[
\text{for } e \leftarrow 0 \text{ to } elements \text{ do}
\]

\[
\text{if } e.col == j
\]

\[
\text{append } (e.col, e.row, e.value) \text{ to } res
\]

\[
\text{end if}
\]

\[
\text{end for}
\]

\[
\text{end for}
\]

- space complexity: \(\Theta(elements)\) for \(res\), constant for others
- time complexity: \(\Theta(col \times elements)\) (so \(O(col \times elements)\))

**Netflix competition:**

\[
row = 17700, \ col = 480189, \ elements = 100480507
\]

\[
\text{col} \times \text{elements} \approx 5 \cdot 10^{13} \ (> 5 \text{ hr on 2.8 GHz CPU})
\]
to save time on transposing, we want to scan only once

set up a sparse matrix $res$ of attributes $(col, row, elements)$

for $j \leftarrow 0$ to $col - 1$

    for $e \leftarrow 0$ to $elements$

        if $e.col = j$
            append $(e.col, e.row, e.value)$ to the $(e.col)$-th pile
        end if
    end for
end for
where’s the \( j \)-th pile? pre-compute pile size and starting locations

set up two arrays \( \text{pilesize} \) and \( \text{pilestart} \), each of length \( \text{col} \)

\[
\text{for } e \leftarrow 0 \text{ to } \text{elements do}
\]

\[
\begin{align*}
\text{if } e.\text{col} &= j \\
\text{pilesize}[j] &\leftarrow \text{pilesize}[j] + 1
\end{align*}
\]

\[
\text{end if}
\]

\[
\text{end for}
\]

\[
\text{for } j \leftarrow 0 \text{ to } \text{col} - 1 \text{ do}
\]

\[
\text{pilestart}[j] \leftarrow \text{pilestart}[j - 1] + \text{pilesize}[j]
\]

\[
\text{end for}
\]
- space complexity: $\Theta(\text{elements})$ for res, $\Theta(\text{col})$ for the helping arrays
- time complexity: $\Theta(\text{col})$ for pre-computing, $\Theta(\text{elements})$ for scanning

Netflix competition:

row = 17700, col = 480189, elements = 100480507

col * 2 + elements $10^8$ (0.04 * constant sec. on 2.8 GHz CPU)
Concrete Implementation of the SparseMatrix Data Structure

- unordered triples: simpler transpose $O(\text{elements})$, time-consuming retrieval $O(\text{elements})$
- ordered triples: harder transpose $O(\text{col} + \text{elements})$, efficient retrieval $O(\logrow \times \logcol)$

The Transpose Algorithm

- scanning each column: smaller space $O(\text{elements})$, time-consuming algorithm $O(\text{col} \times \text{elements})$
- scanning once: bigger space $O(\text{col} + \text{elements})$, efficient algorithm $O(\text{col} + \text{elements})$

Good Programmer (a.k.a. you):
understand the trade-off clearly and make wise choices!
do a temporary transpose of $B$

for $i \leftarrow 0$ to $\text{row}A - 1$ do
  for $j \leftarrow 0$ to $\text{col}B - 1$ do
    compute $C[i, j]$ by multiplying $A[i, :]$ and $B[:, j]^T$
  end for
end for

- multiplying $A[i, :]$ and $B[:, j]^T$: similar to (sparse) polynomial adding—keep it in your toolbox
- time complexity:
  - each multiplying takes $O(\#A_i + \#B_j)$
  - total (careful counting): $O(\text{col}B \times \#A + \text{row}A \times \#B)$
Reading Assignment

be sure to go ask the TAs or me if you are still confused
Reading Assignment

be sure to go ask the TAs or me if you are still confused