Arrays
(chapter 2)

- General array ADT
- Example data types represented by arrays:
  - Polynomial
  - Sparse matrix
- What to learn from the examples:
  - Complexity analysis on real algorithms
  - How representation affects complexity
  - How to improve algorithm performance by reducing redundant operations

Array ADT

- An array is a set of pairs (correspondences) of the form:

  (index → value)

- An index can contain one value (one-dimensional array) or several values in a particular order (multi-dimensional array)
  - Example 1-D indices: 0, 1, 2, …
  - Example 2-D indices: (0,0), (0,1), (0,2), (1,0), …

GeneralArray Class

```
class GeneralArray {
/* objects: A set of pairs < index, value> where for each value of index in
    IndexSet there is a value of type float. IndexSet 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. */
public:
    GeneralArray(int j; RangeList list, float initValue = defatultValue);
    /* The constructor GeneralArray creates a j dimensional array of floats; the
    range of the kth dimension is given by the kth element of list. For each index
    i in the index set, insert <i, initValue> into the array. */
    float Retrieve(index i);
    /* if (i is in the index set of the array) return the float associated with i in the
    array; else signal an error */
    void Store(index i, float x);
    /* if (i is in the index set of the array) delete any pair of the form <i, y>
    present in the array and insert the new pair <i, x>; else signal an error. */
    … /* additional operations */
}; // end of GeneralArray
```

GeneralArray vs. C Array

What additional features can we have in *GeneralArray* beyond the standard C array?

- Index range checking
- Flexible index set (indices to not have to be consecutive integers starting from zero)
- Re-sizing
- Assignment operator
- …

Arrays, Representations

- The *GeneralArray* ADT does not specify a representation.
  - When we mention arrays, we assume that the array elements are stored *sequentially*, as in C/C++.
- We often use a (*more basic*) data type as the representation of a (*more advanced*) data type.
  - Example: We will use arrays as the representation for *ordered lists*, *polynomials*, and *matrices*.

Ordered List

- **Ordered list**: A set of items in a particular order
- Examples of ordered lists:
  - (MONDAY, TUEDSAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAYY, SUNDAY)
  - (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)
  - (1941, 1942, 1943, 1944, 1945)
  - ()
  - …

Operations of Ordered Lists

(1) Find the length, n , of the list.

(2) Read the items from sequentially.

(3) Retrieve the i^{th} element.

(4) Store a new value at the i^{th} position.

(5) Insert a new element at the position i, causing elements numbered i, $i+1$, …, n to become numbered $i+1$, $i+2$, …, $n+1$.

(6) Delete the element at position i, causing elements numbered $i+1$, …, n to become numbered i, $i+1$, …, $n-1$.

Is array (sequential mapping) a good representation here?

Good for operations:

Bad for operations:

Try to estimate the complexities of these operations.

Polynomials

Next, we will deal with polynomials as a special type of ordered lists: Each item is a pair **$<e_i, a_i>$**.

$$p(x) = a_1 x^{e_1} + \ldots + a_n x^{e_n}$$

Polynomials

$$p(x) = a_1 x^{e_1} + \ldots + a_n x^{e_n}$$

class *Polynomial* {
// **objects:** a set of ordered pairs of <e_i, a_i>
// where a_i is a non-zero coefficient
// and e_i is a non-negative integer exponent
public:
Polynomial();
// return the polynomial $p(x) = 0$

Polynomial Add(*Polynomial poly*);
// return the sum of the polynomials ***this** and *poly*

Polynomial Mult(*Polynomial poly*);
// return the product of the polynomials ***this** and *poly*

float *Eval*(**float** *f*);
// evaluate the polynomial ***this** at *f* and return the result

}; // end of Polynomial

Representing Polynomials

- Representation #1: Fixed-size array

```
int degree;
float coef[MaxDegree+1];  // coef[i] = a_{n-i} (n: degree)
```

This representation is very simple.

Complexity:

space

time (Addition)

time (Multiplication)

time (Print-out)

Problems:

How large should we set **`MaxDegree`**?

What to do if degree > **`MaxDegree`**?

Representing Polynomials

- Representation #2: Variable-size array

```
int degree;
float *coef;  // coef[i] = a_{n-i} (n: degree)
```

- Set the size at the constructor:

```
Polynomial::Polynomial(int deg);
{
  degree = deg;
  coef = new float[deg+1];
}
```

No need to worry about degree getting too large.

Q: Which complexities are different from representation #1?

Problem: Wasting space for ***sparse*** polynomials.

Example sparse polynomial: $p(x) = 3x^{1000} + 1$

Representing Polynomials

- Representation #3: Array of terms (index-value pairs)

```
Term *termArray; // array of terms
int capacity;   // size of termArray (pre-allocation)
int terms;      // number of non-zero terms
```

- C++ class for Term:

```
class Term
{
  friend Polynomial;
private:
  float coef; // coefficient
  int exp;    // exponent
};
```

Representation of

$$p(x) = 3x^{1000} + 1$$

coef	3	1		
exp	1000	0		

terms: 2;   capacity: 4

"Sparse Arrays"

Operations for Sparse Polynomials

- Need a function for adding a new term
  - If **`terms == capacity`**
    - Allocate a new **`termArray`** of twice the capacity
    - Copy the polynomial terms to the new array
    - Deallocate the original **`termArray`**
  - Put the new term in;
  - **`terms++`**;

What's the space complexity of storing the array?

What's the time complexity used in adding new terms?

It's always a pain to handle arrays that need to change sizes.
Linked lists (chapter 4) is another solution.

Operations for Sparse Polynomials

- Example of adding two polynomials:

$p_A(x) = 3x^{1000} + 2x^2 + 5$ $p_B(x) = x^2 + x$ result polynomial

3	2	5
1000	2	0

1	1
2	1

a=0

3	2	5
1000	2	0

b=0

1	1
2	1

A.exp(a)>B.exp(b)

3
1000

a++

a=1

3	2	5
1000	2	0

b=0

1	1
2	1

A.exp(a)==B.exp(b)

3	3
1000	2

a++
b++

a=2

3	2	5
1000	2	0

b=1

1	1
2	1

A.exp(a)<B.exp(b)

3	3	1	
1000	2	1	

b++

a=2

3	2	5
1000	2	0

b=2

1	1
2	1

B is done.

3	3	1	5
1000	2	1	0

Just copy the remaining terms in A into result.

What's the time complexity of adding two polynomials?

Sparse Matrix

- A matrix is usually represented as a 2-D array.
- However, similar to the case of polynomials, if there are many zero terms, there is a waste of space.
- Example matrices, one of which is sparse:

$$\begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix} \qquad \begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

- Example sparse matrices in real life:

Sparse Matrix Representation

To save space, this is very similar to the representation #3 (pairs of coefficients and exponents) of polynomials:

- Use a triple **<row, column, value>** for each term.
- Must know the number of rows and columns and the number of nonzero elements.
- Store the triples row by row.
- For all the triples within a row, their column indices are in ascending order.

Sparse Matrix Representation

$$
\begin{array}{c|c}
 & \begin{array}{cccccc} \text{col} \; 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} \text{row} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} &
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
\end{array}
\quad \Rightarrow
$$

0	0	15
0	3	22
0	5	-15
1	1	11
1	2	3
2	3	-6
4	0	91
5	2	28

The terms are ordered first by the row index and then by the column index.

Sparse Matrix ADT

class *SparseMatrix* {
/* objects: A set of triples **<row, column, value>**; all **<row, column>** pairs are unique; row, column, and value are integers; row ≧ 0; col ≧ 0 */

public:
SparseMatrix(**int** *r*, **int** *c*; **int** *t*);
/* Constructor of a sparse matrix of *r* rows, *c* columns; *t* is the capacity*/

SparseMatrix Transpose();
/* standard matrix transpose */

SparseMatrix Add(*SparseMatrix b*);
/* return (*this)+*b* if they have the same size, otherwise throw an exception */

SparseMatrix Multiply(*SparseMatrix b*);
/* return (*this)**b* if the number of columns in (*this) is the same as the number or rows in b, otherwise throw an exception */

}; // end of *SparseMatrix*

See textbook listing for more complete comments.

Sparse Matrix Class

- Class members:

```
MatrixTerm *smArray; // array of terms
int capacity;      // size of smArray (pre-allocation)
int rows, columns;  // size of matrix
int terms;          // number of non-zero terms
```

- C++ class for **MatrixTerm**:

```
class MatrixTerm {
  friend class SparseMatrix
private:
  int row, col, value;
};
```

Transposing a Sparse Matrix

Think about the time complexities of the following:

- Transposing a matrix stored as a 2-D (non-sparse) array.
- Transposing a sparse matrix without placing the elements in the correct row and column order.

The difficulty here is in having the elements of the transpose in the correct order.

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

transpose

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

Transposing a Sparse Matrix

Idea: First copy only the elements in the first column of the original matrix, and then the elements in the second column of the original matrix, and so on.

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

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

Transposing a Sparse Matrix

```
SparseMatrix SparseMatrix::Transpose()
{
  SparseMatrix b(cols, rows, terms);
  if (terms > 0)    // nonzero matrix
  {
    int CurrentB = 0;
    for (int c = 0; c < cols; c++)    // transpose by columns
      for (int i = 0; i < terms; i++)
        // find elements in column c
        if (smArray[i].col == c) {
          b.smArray[CurrentB].row = c;
          b.smArray[CurrentB].col = smArray[i].row;
          b.smArray[CurrentB].value = smArray[i].value;
          CurrentB++;
        }
  }
  return b;
}
```

Q: What's the time complexity?

Transposing a Sparse Matrix

- Time complexity: $O(cols*terms)$.
- Source of waste: We need to look at all the terms "*cols*" times.
- Q: Can we look at each term only once?
- Idea: We can do this if we know where to place a term in the transpose without looking at the terms after it in the original matrix.
- To do this, we just need to know the starting location of each row in the transpose.

Fast Matrix Transposing

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

For the transpose:

row:	0	1	2	3	4	5
rowSize =	2	1	2	2	0	1
rowStart =	0	2	3	5	7	7

number of terms in a row

starting location of a row in the transpose

rowStart[0]=0	b[0] ← <0, 0, 15>	rowStart[0]++
rowStart[3]=5	b[5] ← <3, 0, 22>	rowStart[3]++
rowStart[5]=7	b[7] ← <5, 2, 28>	rowStart[5]++
rowStart[1]=2	b[2] ← <1, 1, 11>	rowStart[1]++
rowStart[2]=3	b[3] ← <2, 1, 3>	rowStart[2]++
rowStart[3]=6	b[6] ← <3, 2, -6>	rowStart[3]++
rowStart[0]=1	b[1] ← <0, 4, 91>	rowStart[0]++
rowStart[2]=4	b[4] ← <2, 5, 28>	rowStart[2]++

Fast Matrix Transposing

```
SparseMatrix SparseMatrix::Transpose()
{
  int *rowSize = new int[cols];
  int *rowStart = new int[cols];
  SparseMatrix b(cols, rows, terms);
  int i;
  if (terms > 0)     // nonzero matrix
  {
    // compute rowSize[i] = number of terms in row i of b
    for ( i = 0; i < cols; i++) rowSize[i] = 0;  // initialize
    for ( i = 0; i < terms; i++) rowSize[smArray[i].col]++;

    // rowStart[i] = starting position of row i in b
    rowStart[0] = 0;
    for (i = 1; i < cols; i++)
      rowStart[i] = rowStart[i-1] + rowSize[i-1];
```

(code to be continued)

Fast Matrix Transposing

(code continued)

```
    for (i =0; i < terms; i++)    // scan all elements in *this
    {
      int j = rowStart[smArray[i].col];
      b.smArray[j].row = smArray[i].col;
      b.smArray[j].col = smArray[i].row;
      b.smArray[j].value = smArray[i].value;
      rowStart[smArray[i].col]++;
    }
  }

  delete [] rowSize;
  delete [] rowStart;
  return b;
}
```

Q: What's the time complexity?

Sparse Matrix Multiplication

- Definition: $$D_{ij} = \sum_{k=0}^{A.cols-1} A_{ik} B_{kj}$$
- Idea: For row i in A, go through B to find all the terms with column index i.
- We need to compute all the terms in D in the order of increasing row index so that they are stored in the correct order.

A faster algorithm is in the textbook. However, it's too complicated so we will skip it here.

Extra Reading Assignments

- From the textbook: Sections 2.5, 2.6.1.
- (Optional) From the textbook: Sections 2.6.2. The KMP algorithm is a nice example of reducing the complexity by identifying and avoiding redundant operations. Focus on understanding the source of redundancy and the role of *failure function*.