CS 2351 Data Structures

Arrays

Prof. Chung-Ta King
Department of Computer Science
National Tsing Hua University

Arrays

- You think that you know *arrays*
  - You know how to define an array in C

```
int a[100];
int *a = malloc(100 * sizeof(int));
```

  - You know the indices are from 0 to 99
  - Given an index, i, you know how to read a value from or write a value into the corresponding entry a[i]

```
a[i] = ...;          ... = a[i];
*(a + i) = ...;      ... = *(a + i);
```

- But, think again …

Arrays

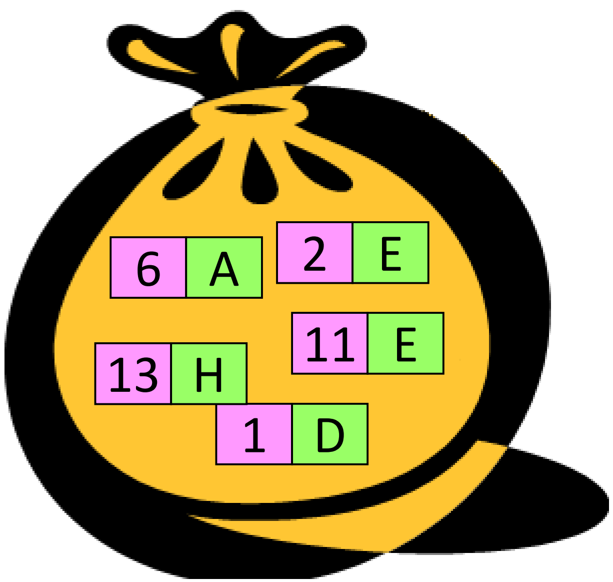
- For an array
  - Why the indices must start from 0?
  - Why the indices must be consecutive?
  - Why the array has to store the same type of data?
- In a very general sense, an array is a set of pairs <index, value>
  - e.g. student id: {(James, #1), (Claire, #2), ..., (Tony, #n)}
- Though general arrays look "general", they can often be implemented efficiently using C-type arrays!
  - We shall study in this chapter how C-type arrays may be extended to support more general arrays and operations

C-Type Arrays vs. General Arrays

- A conceptual C-type array:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
	D	E				A					E		H		

- A conceptual general array:



A possible implementation using C-type arrays

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
Index	6	2	13	11	1		
Value	A	E	H	E	D		

ADT of General Arrays of Floats

ADT *GeneralArray* is

objects: A set of <*index, value*>. Each *index* in *IndexSet* has a *value* of float. *IndexSet* is a finite ordered set of one or more dimensions, e.g. {(0, 0), (0, 1), (1, 0), (1, 1), (1, 2), (2, 1), (2, 2)} for 2-D.

functions:

int SizeOf(); // Return the number of entries in the array

float Retrieve(index i);

/* if (i is in *IndexSet*) return the float associated with i; else signal an error */

void Store(index i, float x);

/* if (i is in *IndexSet*) replace the old pair with <i, x>; else signal an error */

end GeneralArray

Notes on the ADT of General Arrays

- We only define the interface, not implementation
  - We have not specified how the set of <index, value> pairs are organized and structured → often depends on appl.
  - C-type arrays are a special case of general arrays
  - C-type arrays can be used to implement general arrays
- General arrays are more flexible, may use memory more efficiently (depending on implementation), and allow index set checked for validity
- We will see different applications of general arrays in this chapter, which can be implemented efficiently using C-type arrays

How to implement an ADT in C++?

Outline

- C++ class
- From general arrays to ordered list
- Polynomial as an example
  – Space optimization in data structure
- Sparse matrices as another example
  – Time optimization in associated operations

An Example C++ Class

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
// In the header file Rectangle.h
class Rectangle {
public:
    Rectangle();     // constructor
    ~Rectangle();    // destructor
    int GetHeight();
    int GetWidth();
private:
    int xLow, yLow, height, width;
};
#endif
```

C++ Class Definition

- A C++ class consists of 4 components
  - Class name: *Rectangle*
  - Data members: *xLow, yLow, height, width*
  - Member functions: *GetHeight(), GetWidth()*
  - Levels of program access: *public, protected, private*
- Program access of data members/member functions
  - **Public**: can be accessed from any where in the program
  - **Private**: can be accessed from within its class or by a **friend** class or function
  - **Protected**: can be accessed from within its class or from its subclasses or by a **friend**

C++ Constructors and Destructors

- Constructor: a member function to initialize data members of an object, e.g. **`Rectangle()`**
  - Has same name as class, must be public, no return value
  - If defined, automatically executed when object is created
  - Can define default initial values
- Destructor: a member function to delete data members immediately before object is deleted or goes out of scope , e.g. **`~Rectangle()`**

Data Abstraction & Encapsulation in C++

- Data encapsulation is enforced in C++ by declaring all data members of a class to be **private** or **protected**
  - External access to data members only by member functions
- Data abstraction of classes:
  - Specification: must be in public portion, consist of names of public member functions, type of their arguments and return values (*function prototype*)
    → usually placed in a header file
  - Implementation: usually placed in a source file of the same name

Implementation of a C++ Class

```
// In the source file Rectangle.cpp
#include "Rectangle.h"
Rectangle::Rectangle(int x=0, int y=0,
        int h=0, int w=0):
  xLow(x), yLow(y), height(h), width(w)
{}
int Rectangle::GetHeight() {
   return height;
}
int Rectangle::GetWidth() {
   return width;
}
```

Default values

Declaring and Invoking a C++ Class

```
#include <iostream>
#include “Rectangle.h”
main() {
   Rectangle r,s; //object of class Rectangle
   Rectangle *t = &s; // object pointer
   ...
   if (r.GetHeight()*r.GetWidth() >
     t->GetHeight()*t->GetWidth()) cout<<“r”;
   else cout<<“s”;
   cout<<“ has the greater area” << endl;
};
```

Operator Overloading

- How to check if two Rectangle objects are equal?
  - You may write a function, e.g. **`equal(r,s)`**, to compare, which takes these two objects as arguments, compares the four data members, and returns a true or false
  - Isn't it wonderful if you could just say

    ```
    if (r == s) { ... }
    ```

    So, the operator **`==`** not only compares variables of basic data types, e.g. **`int`**, **`float`**, but also user defined types

    → operator overloading
  - User defined data types can be treated same as basic data types

Overloading == as a Member Function

```
bool Rectangle::operator==(cont Rectangle& s)
{
  if (this == &s) return true;
  if ((xLow == s.xLow) && (yLow = s.yLow)
     && (height == s.height)
     && (width == s.width)) return true;
  else return false;
}
```

- The pointer **`this`** inside a member function points to the class object that invoked it → **`*this`** points to class itself

Overloading << as Non-member Function

```
ostream& operator<<(ostream& os,Rectangle& r)
{ os << "Position is: " << r.xLow << " ";
  os << r.yLow << endl;
  os << "Height is: " << r.Height << endl;
  os << "Width is: " << r.Width << endl;
  return os;
};
```

- This function accesses private data members of class **`Rectangle`** and must be made a **friend** of it (see next page)

Return **`cout`**; thus have **`cout<<endl`**

- With the overloaded operator, we can do

```
cout << r << endl;
```

Class Rectangle

```
class Rectangle {
friend ostream& operator<<(ostream& os,
               Rectangle& r);
public:
  Rectangle(int x = 0, int y = 0,
      int h = 0, int w = 0)
    : xLow(x), yLow(y), height(h), width(w){ }
  bool operator==(const Rectangle& s){
     ...
  }
private:
     int xLow, yLow, height, width;
};
```

ADT of General Arrays in C++

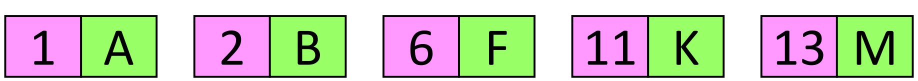
```
class GeneralArray {
private:
/* A set of <index, value>, where IndexSet is a
finite ordered set of one or more dimensions */
public:
  GeneralArray(int j; RangeList list,
               float initValue = defatultValue);
     /* Constructor creates a j-D array of floats.
     Range of k-D is given by kth element of list.
     For each i in IndexSet, insert <i,initValue> */
  float Retrieve(index i);
  void Store(index i, float x);
}; // end of GeneralArray
```

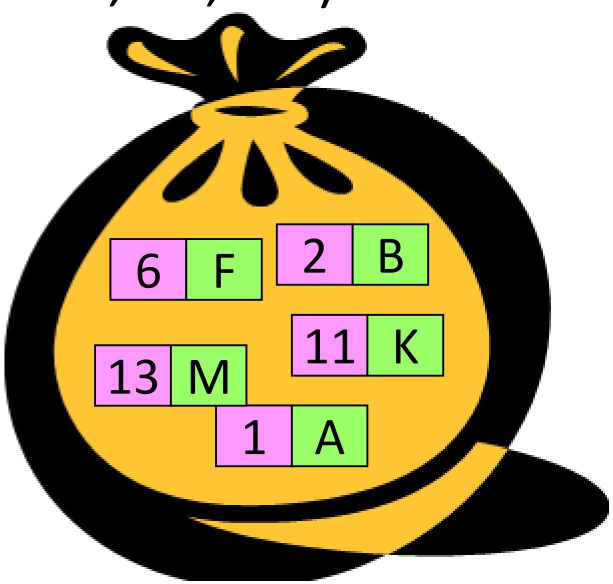
Outline

- C++ class
- From general arrays to ordered list
- Polynomial as an example
  - Space optimization in data structure
- Sparse matrices as another example
  - Time optimization in associated operations

From General Arrays to Ordered Lists

- A general array only specifies that there is a set of <index, value> pairs → no special order imposed
- An ordered (linear) list is a special case of general arrays, in which the items are ordered linearly
  - Days of Week: (Sun, Mon, Tue, Wed, Thu, Fri, Sat)
  - Months: (Jan, Feb, Mar, …, Nov, Dec)
  - Poker: (2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace)





Operations on Ordered Lists

- Find the length, n, of the list
- Read the items from left to right (or right to left)
- Retrieve the ith item, $0 \le i < n$
- Store a new value into ith position, $0 \le i < n$
- Insert/delete the item at position i, $0 \le i < n$

- It is not necessary to include all operations
- Depending on applications, different representations support different subsets of operations efficiently

Outline

- C++ class
- From general arrays to ordered list
- Polynomial as an example
  - Space optimization in data structure
  - Representation, addition, time complexity analysis
- Sparse matrices as another example
  - Time optimization in associated operations

Polynomials

- $p(x) = a_0x^{e0} + a_1x^{e1} +, \ldots, a_nx^{en} = \sum a_ix^{ei}$
  - Each a_ix^{ei} is a term with coefficient a_i and exponent e_i
  - **Degree** of p(x) is largest exponent of the non-zero term
  - Ex.: $p(x)=x^5+4x^3+2x^2+1$
    has 4 terms with coefficients 1, 4 ,2, 1, and a degree of 5
- Intuitive representation in a C-type array
  - Store (a_i, e_i) by assigning a_i to A[n-i], where n is the degree

x^5		$4x^3$	$2x^2$		x^0
1	0	4	2	0	1
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

exponent → index
coefficient → value

Operations for Polynomials

Let $a(x) = \sum a_i x^i$ and $b(x) = \sum b_i x^i$

- Polynomial addition
  - $a(x) + b(x) = \sum (a_i+b_i)x^i$
  - Ex.: $a(x)=x^5+4x^3+2x^2+1$ (degree = 5)
    $b(x)=3x^6+4x^3+x$ (degree = 6)
    $a(x) + b(x) = 3x^6+x^5+8x^3+2x^2+x+1$ (degree = 6)
- Polynomial multiplication
  - $a(x) \times b(x) = \sum (a_i x^i \times \sum (b_j x^j))$
- How about insertion/deletion?

ADT of Polynomials

How to represent to be most efficient in space?

```
class Polynomial {
// a set of ordered pairs of <a_i, e_i>, where a_i is
// nonzero float, e_i is non-negative integer
public:
    Polynomial(void); //
    ~Polynomial(void); //
    Polynomial Add(Polynomial poly);
    // Return the sum of *this and poly
    Polynomial Mult(Polynomial poly);
    // Return multiplication of *this and poly
    float Eval(float x);
    // Evaluate *this at x and return result
};
```

We will ignore destructor hereafter. It is programmer’s responsibility to treat her memory well ☺

1st Representation of Data Members

- Use C-type arrays with fixed space

```
private:
    // degree ≤ MaxDegree
    int degree;
    // coefficient array
    float coef[MaxDegree+1];
```

```
Usage:
    Polynomial a;
    a.degree = n;
    a.coef[i] = an-i
```

x^5		$4x^3$	$2x^2$		x^0
1	0	4	2	0	1
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

  – Must know *MaxDegree*, may allocate too much space, waste memory in a **sparse** polynomial, e.g., $x^{1000}+1$

2nd Representation of Data Members

- Use C-type arrays with dynamically allocated space:

```
private:
    int degree;
    float *coef;
```

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

  - No need to know *MaxDegree* in advance, allocate exact space as needed
  - Disadvantage: waste memory in a sparse polynomial

3rd Representation of Data Members

- Store only nonzero terms:

```
class Polynomial;
// forward decl.
class Term {
  friend Polynomial;
  float coef;
  int exp;
};
```

```
private:
  // array of nonzero terms
  Term* termArray;
  // termArray size
  int capacity;
  // # nonzero terms
  int terms;
```

  - Coefficients are stored in order of decreasing exponents
  - Better if polynomial is sparse, but if polynomial is full, it requires double the space of 2nd representation
    → considerations for space optimization

Polynomial Addition (1/3)

```
Polynomial Polynomial::Add(Polynomial b) {
 Polynomial c;
 int aPos = 0, bPos = 0;
 while((aPos < terms) && (bPos < b.terms))
  if(termArray[aPos].exp == b.termArray[bPos].exp){
   float t = termArray[aPos].coef
             + b.termArray[bPos].coef;
   if (t) c.NewTerm(t, termArray[aPos].exp);
   aPos++; bPos++;
  } else
  if(termArray[aPos].exp < b.termArray[bPos].exp){
   c.NewTerm(b.termArray[bPos].coef,
                     b.termArray[bPos].exp);
   bPos++;
  }
```

Append to the end of Polynomial c

Polynomial Addition (2/3)

```
  else{
   c.NewTerm(termArray[aPos].coef,termArray[aPos].exp);
   aPos++;
  }
  // add in remaining terms of *this
  for(; aPos < terms; aPos++)
    c.NewTerm(termArray[aPos].coef,
                termArray[aPos].exp);
  // add in remaining terms of b
  for(; bPos < b.terms; bPos++)
    c.NewTerm(b.termArray[bPos].coef,
                b.termArray[bPos].exp);
  return c;
}
```

Polynomial Addition (3/3)

```
void Polynomial::NewTerm(const float c, const int e)
{ //Add a new term to the end of termArray
  if (terms == capacity)
  {  // double capacity of termArray
      capacity *= 2;
      term *temp = new term[capacity];
      copy(termArray, termArray + terms, temp);
      delete[] termArray;
      termArray = temp;
  }
  termArray[terms].coef = c;
  termArray[terms].exp = e;
}
```

A Running Example

$$a(x) = x^5 + 9x^4 + 7x^3 + 2x$$

aPos aPos aPos aPos aPos

$$b(x) = x^6 + 3x^5 + 6x + 3$$

bPos bPos bPos bPos

$$c(x) = x^6 + (1+3)x^5 + 9x^4 + 7x^3 + (2+6)x + 3$$

$$= x^6 + 4x^5 + 9x^4 + 7x^3 + 8x + 3$$

Time Complexity Analysis

- Inside the while loop, every statement has O(1) time
- How many times the "while loop" is executed in the worst case?
  - Let a(x) have m terms and b(x) have n terms
  - Each iteration accesses next element in a(x), b(x), or both
  - Worst case: $m + n – 1$
    e.g., $a(x) = 7x^5 + x^3 + x$; $b(x) = x^6 + 2x^4 + 6x^2 +3$
  - Access remaining terms in a(x): O(m), and b(x): O(n)
- Hence, total running time = $O(m + n)$

Outline

- C++ class
- From general arrays to ordered list
- Polynomial as an example
  - Space optimization in data structure
- Sparse matrices as another example
  - Time optimization in associated operations
  - Representation, transpose, multiplication, time complexity analysis

Sparse Matrix

- A matrix has many zero elements

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

- 2D array representation is inefficient
  - Waste both memory and running time to store and compute those zero elements

Example of Sparse Matrices

- Web page matrix
  - Web pages are numbered 1 through n
  - web(i,j) = number of links from page i to page j
- Space analysis
  - n = 2 billion = 2×10^9 pages
  - If use n x n array of ints → $4 \times 10^{18} \times 4$ bytes
  - Each page links to 10 (say) other pages on average, i.e. 10 nonzero entries per row
  - If use general array → $2 \times 10^9 \times 10 \times 8$ bytes

Example of Sparse Matrices

- Social network
  - People are numbered 1 through n
  - friend(i,j) = 1, if i and j are friends; 0, otherwise
  - What does it mean by (friend matrix)^2?
  - n = 100M (say), each person has 100 friends in average
  - If use n x n array → $10^{16} \times 4$ bytes
  - If use general array → $10^{8} \times 100 \times 8$ bytes

Sparse Matrix Representation

- Use an array, smArray[], of triple <row, col, value> to store nonzero elements (2D index space)
- Triples are stored in row-major order → ordered list

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

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

How about insertion/deletion?

ADT of Sparse Matrix

```
class SparseMatrix{
public:
   SparseMatrix(int r, int c, int t);
   // t is the capacity of nonzero terms
   SparseMatrix Transpose(void);
   SparseMatrix Add(SparseMatrix b);
   SparseMatrix Multiply(SparseMatrix b);
private:
    int rows, cols;
    int terms, capacity;
    MatrixTerm *smArray;
};
```

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

Approximate Memory Requirements

- 500 x 500 matrix with 1994 nonzero elements, 4 bytes per element

2D array	500 x 500 x 4 = 1million bytes
Class SparseMatrix	3 x 1994 x 4 + 4 x 4
	= 23,944 bytes

Matrix Transpose

$$A = \begin{bmatrix} \boxed{15 \quad 0 \quad 0 \quad 22 \quad 0 \quad -15} \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 15 & 0 & 0 & 0 & 91 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 28 \\ 22 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -15 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Transpose of Matrix

- Intuitive idea: check columns sequentially and collect terms with same column together

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

A^T	row	col	value
smArray[0]	0	0	15
smArray[1]	0	4	91
smArray[2]	1	1	11
smArray[3]	2	1	3
smArray[4]	2	5	28
smArray[5]	3	0	22
smArray[6]	3	2	-6
smArray[7]	5	0	-15

1st Transpose Algorithm

```
SparseMatrix SparseMatrix::Transpose() {
 SparseMatrix b(cols, rows, terms);
 if (terms > 0) { // has nonzero terms
  int currentB = 0;
   for(int c=0; c<cols; c++)      // O(cols)
    for(int i=0; i<terms; i++)   // O(terms)
     if(smArray[i].col == c){
      b.smArray[currentB].row = c;
      b.smArray[currentB].col = smArray[i].row;
      b.smArray[currentB++].value=smArray[i].value;
     }
  }
  return b;
}
```

Time complexity: $O(cols \times terms)$
$\sim O(cols \times cols \times rows)$

2nd Transpose Algorithm: Fast Transpose

- Cause of inefficiency for 1^{st} transpose algorithm:
  - Do not know locations of different columns
  - This information can be calculated beforehand
- Use additional space to calculate and store
  - rowSize[i]: # of nonzero terms in the i^{th} row of A^T
  - rowStart[i]: location of nonzero term of the i^{th} row of A^T in smArray
  - For i>0, rowStart[i]=rowStart[i-1]+rowSize[i-1]
- Then copy elements from A to A^T one by one
- Time complexity: O(terms + cols)!

Fast Transpose

Count the # of nonzero terms in each row of A^T

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

A^T	rowSize	rowStart
[0]	2	
[1]	1	
[2]	2	
[3]	2	
[4]	0	
[5]	1	

A^T	row	col
smArray[0]		
smArray[1]		
smArray[2]		
smArray[3]		
smArray[4]		
smArray[5]		
smArray[6]		
smArray[7]		

Fast Transpose

Calculate location of 1st nonzero term of ith row of A^T in smArray

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

A^T	rowSize	rowStart
[0]	2	0
[1]	1	2
[2]	2	3
[3]	2	5
[4]	0	7
[5]	1	7

A^T	row	col
smArray[0]		
smArray[1]		
smArray[2]		
smArray[3]		
smArray[4]		
smArray[5]		
smArray[6]		
smArray[7]		

Fast Transpose

Copy elements from A to A^T one by one

	row	col	value
[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

A^T	rowSize	rowStart
[0]	2	0
[1]	1	2
[2]	2	3
[3]	2	5
[4]	0	7
[5]	1	7

A^T	row	col	value
smArray[0]	0	0	15
smArray[1]			
smArray[2]			
smArray[3]			
smArray[4]			
smArray[5]			
smArray[6]			
smArray[7]			

Fast Transpose

Copy elements from A to A^T one by one

	row	col	value
[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

A^T	rowSize	rowStart
[0]	2	1
[1]	1	2
[2]	2	3
[3]	2	5
[4]	0	7
[5]	1	7

A^T	row	col	value
smArray[0]	0	0	15
smArray[1]			
smArray[2]			
smArray[3]			
smArray[4]			
smArray[5]	3	0	22
smArray[6]			
smArray[7]			

Fast Transpose

Copy elements from A to A^T one by one

	row	col	value
[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

A^T	rowSize	rowStart
[0]	2	1
[1]	1	3
[2]	2	4
[3]	2	7
[4]	0	7
[5]	1	8

A^T	row	col	value
smArray[0]	0	0	15
smArray[1]	0	4	91
smArray[2]	1	1	11
smArray[3]	2	1	3
smArray[4]			
smArray[5]	3	0	22
smArray[6]	3	2	-6
smArray[7]	5	0	-15

Fast Transpose

Copy elements from A to A^T one by one

	row	col	value
[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

A^T	rowSize	rowStart
[0]	2	2
[1]	1	3
[2]	2	5
[3]	2	7
[4]	0	7
[5]	1	8

A^T	row	col	value
smArray[0]	0	0	15
smArray[1]	0	4	91
smArray[2]	1	1	11
smArray[3]	2	1	3
smArray[4]	2	5	28
smArray[5]	3	0	22
smArray[6]	3	2	-6
smArray[7]	5	0	-15

Fast Transpose (1/2)

```
SparseMatrix SparseMatrix::FastTranspose( )
{ SparseMatrix b(cols, rows, terms);
  if (terms > 0) {
    int *rowSize = new int[cols];
    int *rowStart = new int[cols];
    // compute rowSize[i]=# of terms in row i of b
    fill(rowSize, rowSize+cols, 0);
    for(int i=0; i<terms; i++)
       rowSize[smArray[i].col]++;
    // rowStart[i] = starting pos. of row i in b
    rowStart[0] = 0;
    for(int i=1; i<cols; i++)
      rowStart[i]=rowStart[i-1]+rowSize[i-1];
```

Fast Transpose (2/2)

```
  // copy terms from *this to b
  for(int i=0; i<terms; i++){
      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]++;
      // Increase the start pos by 1
    }
    delete [] rowSize;
    delete [] rowStart;
  }
  return b;
}
```

Running Time Comparison

1st Transpose Algorithm	2nd Transpose Algorithm
O(cols × terms)	**O(cols + terms)**

- For a dense matrix (terms = rows × cols)
  - 2nd algorithm is faster: O(rows × cols)
  - 1st algorithm is slower: O(rows × cols^2)
- For a sparse matrix (terms << rows × cols)
  - 2nd algorithm is much faster
- Considerations for time optimization

Sparse Matrix Multiplication

- Compute the transpose of b

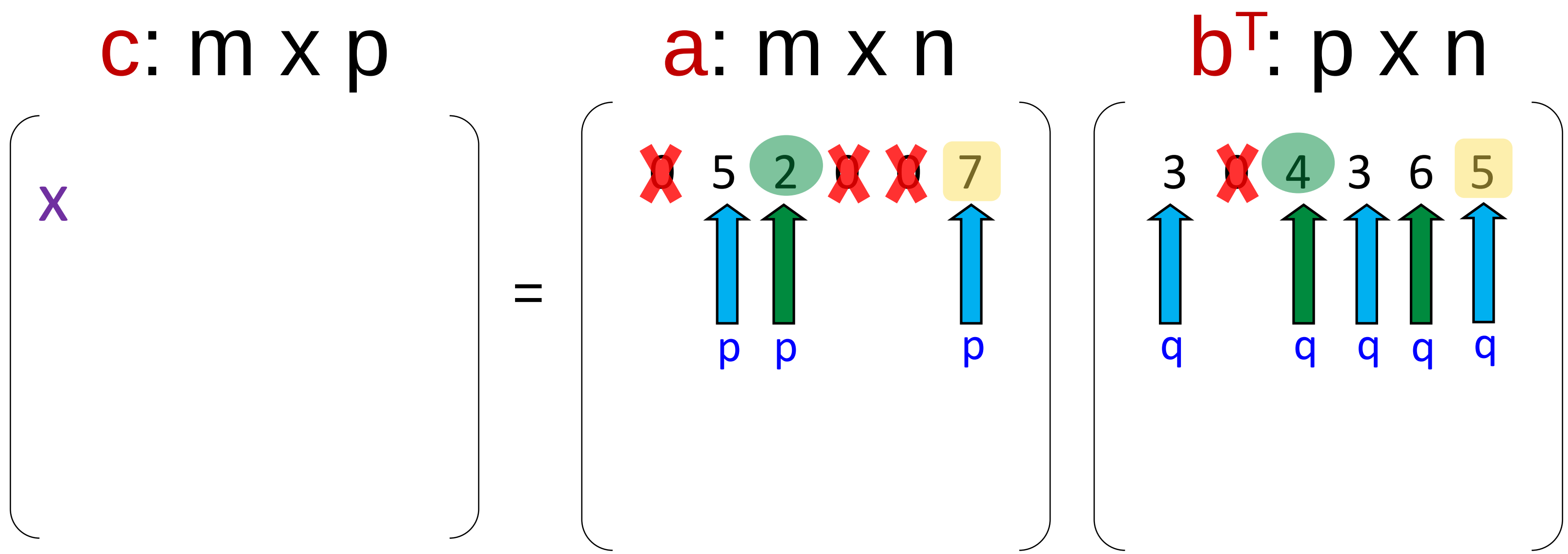
$$\underset{c:\ m \times p}{\begin{pmatrix} x & & \\ & & \\ & & \\ & & \end{pmatrix}} = \underset{a:\ m \times n}{\begin{pmatrix} 0 & 5 & 2 & 0 & 0 & 7 \\ & & & & & \\ & & & & & \end{pmatrix}} \underset{b:\ n \times p}{\begin{pmatrix} 3 & & \\ 0 & & \\ 4 & & \\ 3 & & \\ 6 & & \\ 5 & & \end{pmatrix}}$$

$$X = 0\times3 + 5\times0 + 2\times4 + 0\times3 + 0\times6 + 7\times5 = 43$$

$$c(i,j) = \Sigma\ a(i,k) \times b(k,j)$$

Sparse Matrix Multiplication

- Use approach similar to Polynomial Addition to compute the X



Please refer to the textbook for code

$$x = (2)(4) + (7)(5) = 43$$

Time Complexity

```
SparseMatrix SparseMatrix::Multiply(SparseMatrix b) {
 SparseMatrix bT = b.FastTranspose(); //O(b.terms+b.cols)
 for ith row in smArray                // O(rows)
  for jth row in bT.smArray            // O(b.cols)
   Perform "Polynomal Addition"        // O(Terms[i]+b.Terms[j])
}
```

- Complexity:
  - $O(\text{rows} \times \text{b.cols} \times (\text{Term[i]} + \text{b.Terms[j]}))$
  - $\text{rows} \times \text{Term[i]} = \text{a.terms}$
    $\text{b.cols} \times \text{b.Terms[j]} = \text{b.terms}$
  - $O(\text{rows} \times \text{b.terms} + \text{b.cols} \times \text{a.terms})$

Summary

- General arrays as ADT with easy C-type array ext.
- C++ class
- Polynomial as an example of ordered list (linear index space)
  - 3 versions of presentations for space optimization
- Sparse matrix as another example of ordered list (2 dimensional index space)
  - 2 transpose algorithms for time optimization
- What if we want to support insertion/deletion efficiently?