Arrays

2.2 The Array as an Abstract Data Type

Array:

A set of pairs: <index, value> (correspondence or mapping)

Two operations: retrieve, store

Now we will use the C++ class to define an ADT.

> **Note:** This is not the usual perspective, since many programmers view an array only as a consecutive of memory locations.

GeneralArray

class *GeneralArray* **{**

// 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.

public:

GeneralArray(**int** *j*, *RangeList list*, **float** *initValue* = defaultValue)**;**

// The constructor GeneralArray creates a ***j*** dimensional array of floats;

// Range of the *k*th dimension is given by the *k*th 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;

// otherwise throw an exception.

void *Store*(*index i*, **float** *x*)**;**

// If *i* is in the index set of the array, replace the old value associated with *i* by *x*;

// otherwise throw an exception.

}; //end of GeneralArray

Note:

Not necessarily implemented using consecutive memory

Index can be coded any way

***GeneralArray* is more general than C++ array as it is more flexible about the composition of the index set**

To be simple, we will hereafter use the C++ array

C++ array:
The index set is a set of consecutive integers starting at 0.
The ith element can be accessed in two ways: *floatArray*[i] and *(*floatArray* + i).

Array can be used to implement other abstract data types. The simplest one might be:

Ordered or linear list.

Example:

(Sun, Mon, Tue, Wed, Thu, Fri, Sat)

(2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A)

( )   // empty list

More generally, An ordered list is either empty or $(a_0, a_1, .., a_{n-1})$. // index important

Main operations:

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

(2) Read the list from left to right (or right to left)

(3) Retrieve the ith element, $0 \leq i < n$.

(4) Store a new value into the ith position, $0 \leq i < n$.

(5) Insert a new element at position i, $0 \le i < n$, causing elements numbered i, i+1, …, n-1 to become numbered i+1, i+2, …, n.

(6) Delete the element at position i, $0 \le i < n$, causing elements numbered i+1, i+2, …, n-1 to become numbered i, i+1, …, n-2.

How to represent ordered list efficiently?

Sequential mapping

Use array: a_i ←→index i

Complexity

Random access any element in

O(1)

Operations (5) and (6)?

Data movement

O(n)

Now let us look at a problem requiring ordered list.

Problem:

Build an ADT for the representation and manipulation of symbolic polynomials.

$$A(x) = 3x^2 + 2x + 4$$

$$B(x) = x^4 + 10x^3 + 3x^2 + 1$$

Degree: the largest exponent

ADT Polynomial

class Polynomial {

// $p(x) = a_0 x^{e_0} + \cdots + a_n x^{e_n}$

// a set of ordered pairs of $<e_i, a_i>$

// a_i is a nonzero **float** coefficient

// e_i is a non-negative **integer** exponent

public:

Polynomial ( )**;**

// Construct 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 polynomial ***this** at *f* and return the result

}

Polynomial Representation

Let a polynomial be $A(x)=a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$

Representation 1

private:

int *degree*; // *degree* $\leq$ *MaxDegree*

float *coef*[*MaxDegree*+1];

$a.degree = ?$

$n;$

MaxDegree?

$a.coef[i] = ?$

$a_{n-i}, 0 \leq i \leq n$

Simple algorithms for many operations.

Representation 2

When a.degree << MaxDegree, representation 1 is quite wasteful in its use of computer memory. To address this issue, the variable-sized data member is defined.

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

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

Representation 2 is still not desirable.

For instance, $x^{1000} + 1$

makes 999 entries of the coef be zero.

So, we store only the nonezero terms:

Representation 3

$$A(x) = b_m x^{e_m} + b_{m-1} x^{e_{m-1}} + \cdots + b_0 x^{e_0}$$

where $b_i \neq 0, \quad e_m > e_{m-1} > \cdots > e_0 \geq 0$

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

class Polynomial {
public:
            …
private:
   Term *termArray;
   int capacity; // size of termArray
   int terms; // number of nonzero terms
}
```

For $A(x) = 2x^{1000} + 1$

A.termArray looks like:

coef	2	1		
exp	1000	0		

Many zero --- good

Few zero --- ?

not very good

may use twice as much space as in presentation 2.

Polynomial Addition

Use presentation 3 to obtain C = A + B.

Idea:

Because the exponents are in descending order, we can adds A(x) and B(x) term by term to produce C(x).

The terms of C are entered into its termArray by calling function NewTerm.

If the space in termArray is not enough, its capacity is doubled.

```
Polynomial  Polynomial::Add (Polynomial b)
{   // return the sum of the polynomials *this and 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 + 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++;
     }
```

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

```
void Polynomial::NewTerm(const float theCoeff, const int theExp)
{ // add a new term to the end of termArray.
   if (terms == capacity)
   { // double capacity of termArray
      capacity *= 2;
      Term *temp = new Term[capacity]; // new array
      copy(termArray, termArray + terms, temp);
      delete [ ] termArray;  // deallocate old memory
      termArray = temp;
   }
  termArray[terms].coef = theCoeff;
  termArray[terms++].exp = theExp;
}
```

Analysis of Add:

Let m, n be the number of nonzero terms in a and b respectively.

- line 3 and 4---O(1)
- In each iteration of the while loop, *aPos* or *bPos* or both increase by 1, thus the number of iterations of this loop ≤ $m + n - 1$.
- If the time spent on array doubling is ignored, each iteration takes O(1).
- line 20--- O(m), line 23--- O(n)

Asymptotic computing time for Add: O($m + n$)

Analysis of array doubling:

- the time for doubling is linear in the size of new array
- initially, *c.capacity* is 1
- suppose when Add terminates, *c.capacity* is 2^k
- the total time spent over all array doubling is

$$O\left(\sum_{i=1}^{k} 2^i \right) = O(2^{k+1}) = O(2^k)$$

- since *c.terms* $> 2^{k-1}$ and $m + n \geq$ *c.terms*, the total time for array doubling is

$$O(c.terms) = O(m + n)$$

• so, **even consider array doubling, the total run time of Add is O(m + n).**

• experiments show that array doubling is responsible for very small fraction of the total run time of Add.

Sparse Matrices

Introduction

A general matrix consists of m rows and n columns ($m \times n$) of numbers, as:

$$
\begin{array}{c|ccc|}
 & 0 & 1 & 2 \\
\hline
0 & -27 & 3 & 4 \\
1 & 6 & 82 & -2 \\
2 & 109 & -64 & 11 \\
3 & 12 & 8 & 9 \\
4 & 48 & 27 & 47 \\
\hline
\end{array}
$$

Fig.2.2(a) 5×3

$$
\begin{array}{c c}
 & \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} &
\left[\begin{array}{cccccc}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{array}\right]
\end{array}
$$

Fig. 2.2(b) 6×6

A matrix of $m \times m$ is called a square.

A matrix with many zero entries is called sparse.

Representation:

- **A natural way ---**
  - **a[m][n]**
  - **access element by a[i][j], easy operations. But**
  - **for sparse matrix, wasteful of both memory and time.**
- **Alternative way ---**
  - **store nonzero elements explicitly. 0 as default.**

SparseMatrix

```
class SparseMatrix
 { // a set of triples, <row, column, value>, where row, column are
  // non-negative integers and form a unique combination;
  //  value is also an integer.
 public:
    SparseMatrix ( int r, int c, int t);
    // create a r×c SparseMatrix with a capacity of t nonzero terms
    SparseMatrix Transpose ( );
    // return the SparseMatrix obtained by transposing *this
    SparseMatrix  Add ( SparseMatrix b);
    SparseMatrix  Multiply ( SparseMatrix b);
 };
```

Sparse Matrix Representation

Use triple <*row*, *col*, *value*>, sorted in ascending order by <*row*, *col*>.

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

We need also

the number of rows

the number of columns

the number of nonzero elements

And in class *SparseMatrix*:

private:

int *rows*, *cols*, *terms*, *capacity***;**

MatrixTerm **smArray***;**

Now we can store the matrix of Fig. 2.2 (b) as Fig. 2.3 (a).

	row	col	value
smArray[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

Fig. 2.3 (a)

Transposing a Matrix

Transpose:

If an element is at position[*i*][*j*] in the original matrix, then it is at position[*j*][*i*] in the transposed matrix.

for (int $j = 0$; $j < cols$; $j{+}{+}$)

for (int $i = 0$; $i < rows$; $i{+}{+}$)

B[j][i] = A[i][j];

T = O(*cols*×*rows*)

6×7 matrixs,
8 nonzore elements

	6 rows	7 cols	8 terms
	i	j	v
[0]	0	1	12
[1]	0	2	9
[2]	2	0	-3
[4]	2	5	14
[4]	3	2	24
[5]	4	1	18
[6]	5	0	15
[7]	5	3	-7

A

7×6 transposed matrixs,
8 nonzore elements

7 rows	6 cols	8 terms	
i	j	v	
0	2	-3	[0]
0	5	15	[1]
1	0	12	[2]
1	4	18	[4]
2	0	9	[4]
2	3	24	[5]
3	5	-7	[6]
5	2	14	[7]

B

6×7 matrixs,
8 nonzore elements

	6 rows	7 cols	8 terms
	i	j	v
[0]	0	1	12
[1]	0	2	9
[2]	2	0	-3
[4]	2	5	14
[4]	3	2	24
[5]	4	1	18
[6]	5	0	15
[7]	5	3	-7

A

?

7×6 transposed matrixs,
8 nonzore elements

7 rows	6 cols	8 terms	
i	j	v	
0	2	-3	[0]
0	5	15	[1]
1	0	12	[2]
1	4	18	[4]
2	0	9	[4]
2	3	24	[5]
3	5	-7	[6]
5	2	14	[7]

B

First try:

For (each row *i*)

✓take element (*i*, *j*, *value*)

✓store it in (*j*, *i*, *value*) of the transpose;

Difficulty:

NOT knowing where to put (*j*, *i*, *value*) until all other elements preceding it have been processed.

smArray	*row*	*col*	*value*
[0]	0	2	-3
[1]	0	5	15
[2]	1	0	12
[3]	1	4	18
[4]	2	0	9
[5]	2	3	24
[6]	3	5	-7
[7]	5	2	14

Improvement:

For (all elements in col *j*)

✓store (*i*, *j*, *value*) of the original matrix as (*j*, *i*, *value*) of the transpose;

Since the rows are in order, we will locate elements in the correct column order.

smArray	*row*	*col*	*value*
[0]	0	2	-3
[1]	0	5	15
[2]	1	0	12
[3]	1	4	18
[4]	2	0	9
[5]	2	3	24
[6]	3	5	-7
[7]	5	2	14

col = 0

	6 rows	7 cols	8 terms
	i	j	v
[0]	0	1	12
[1]	0	2	9
[2]	2	**0**	-3
[4]	2	5	14
[4]	3	2	24
[5]	4	1	18
[6]	5	**0**	15
[7]	5	3	-7

7 rows	6 cols	8 terms	
i	j	v	
0	2	-3	[0]
0	5	15	[1]
			[2]
			[4]
			[4]
			[5]
			[6]
			[7]

col = 1

	rows	cols	terms
	6	7	8
	i	j	v
[0]	0	1	12
[1]	0	2	9
[2]	2	0	-3
[4]	2	5	14
[4]	3	2	24
[5]	4	1	18
[6]	5	0	15
[7]	5	3	-7

rows	cols	terms	
7	6	8	
i	j	v	
0	2	-3	[0]
0	5	15	[1]
1	0	12	[2]
1	4	18	[4]
			[4]
			[5]
			[6]
			[7]

col = 2, col = 3, …, col = 5, col = 6

	6 rows	7 cols	8 terms
	i	j	v
[0]	0	1	12
[1]	0	2	9
[2]	2	0	-3
[4]	2	5	14
[4]	3	2	24
[5]	4	1	18
[6]	5	0	15
[7]	5	3	-7

7 rows	6 cols	8 terms	
i	j	v	
0	2	-3	[0]
0	5	15	[1]
1	0	12	[2]
1	4	18	[4]
2	0	9	[4]
2	3	24	[5]
3	5	-7	[6]
5	2	14	[7]

```
SparseMatrix SparseMatrix::Transpose ( )
{ // return the transpose of *this
   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 and move terms 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;
        }
  } // end of if (terms > 0)
  return b;
}
```

Time complexity of Transpose:

- line 7-15 loop --- *cols* times
- line 10 condition --- *terms* times
- other line --- O(1)

Total time: O(*cols* * *terms*)

Additional space: O(1)

Think:

O(*cols* * *terms*) is not good. If *terms* = O(*cols* * *rows*) then it becomes O($cols^2$ * *rows*)---too bad!

Since with 2-dimensional representation, we can get an easy O(*cols * rows*) algorithm as:

```
for (int j=0; j < cols; j++)
    for (int i=0; i < rows; i++)
        B[j][i] = A[i][j];
```

Further improvement:

If we use some more space to store *some knowledge* about the matrix, we can do much better: doing it in O(cols + terms).

6×7 matrixs,
8 nonzore elements

7×6 transposed matrixs,
8 nonzore elements

	6 rows	7 cols	8 terms
	i	j	v
[0]	0	1	12
[1]	0	2	9
[2]	2	0	-3
[4]	2	5	14
[4]	3	2	24
[5]	4	1	18
[6]	5	0	15
[7]	5	3	-7

A

?

7 rows	6 cols	8 terms	
i	j	v	
0	2	-3	[0]
0	5	15	[1]
1	0	12	[2]
1	4	18	[4]
2	0	9	[4]
2	3	24	[5]
3	5	-7	[6]
5	2	14	[7]

B

- **get the number of elements in each column of**
  ***this = the number of elements in each row of *B*;**
- **obtain the starting point in *B* of each of its rows;**
- **move the elements of *this one by one into their right position in *B*.**

Now the algorithm *FastTranspose.*

Initiate *rowStart*

	6 rows	7 cols	8 terms
	i	j	v
[0]	0	1	12
[1]	0	2	9
[2]	2	0	-3
[4]	2	5	14
[4]	3	2	24
[5]	4	1	18
[6]	5	0	15
[7]	5	3	-7

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
rowSize	2	2	2	1	0	1	0

$$rowStart[i] = rowStart[i-1] + rowSize[\mathrm{i}-1]$$

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
rowStart	0	2	4	6	7	7	8

7 rows	6 cols	8 terms
i	j	v
0	1	12
0	2	9
2	0	-3
2	5	14
3	2	24
4	1	18
5	0	15
5	3	-7

rowStart

[0]	[1]	[2]	[3]	[4]	[5]	[6]
0	**2**$_{+1}$	4	6	7	7	8
0	3	**4**$_{+1}$	6	7	7	8
0$_{+1}$	3	5	6	7	7	8
1	3	5	6	7	**7**$_{+1}$	8
1	3	**5**$_{+1}$	6	7	8	8
1	**3**$_{+1}$	6	6	7	8	8
1$_{+1}$	4	6	6	7	8	8
2	4	6	**6**$_{+1}$	7	8	8

	6 rows	7 cols	8 terms
	i	j	v
[0]	0	2	-3
[1]	0	5	15
[2]	1	0	12
[3]	1	4	18
[4]	2	0	9
[5]	2	3	24
[6]	3	5	-7
[7]	5	2	14

```
SparseMatrix SparseMatrix::FastTranspose ( )
{ // return the transpose of *this in O(terms+cols) time.
  SparseMatrix b(cols, rows, terms);
  if (terms > 0)
  { // nonzero matrix
    int *rowSize = new int[cols];
    int *rowStart = new int[cols];
    // compute rowSize[i] = number of terms in row i of b
    fill(rowSize, rowSize + cols, 0);  // initialze
    for (int i=0; i<terms; i++ )  rowSize[smArray[i].col]++;
```

```
// rowStart[i] = starting position of row i in b
rowStart[0] = 0;
for (int i=1;i<cols;i++) rowStart[i]=rowStart[i-1]+rowSize[i-1];
for (int i=0; i<terms; i++)
{     // copy from *this to b
      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]++;
}   // end of for
```

```
22     delete [ ] rowSize;
23     delete [ ] rowStart;
24  }  // end of if
25  return b;
26 }
```

Try sparse matrix of Fig. 2.3(a), after line 13, we get :

	[0]	[1]	[2]	[3]	[4]	[5]
RowSize=	**2**	**1**	**2**	**2**	**0**	**1**
RowStart=	**0**	**2**	**3**	**5**	**7**	**7**

Note the error in P101 of the text book!

Analysis:

3 loops:

- line 10--- O(*terms*)
- line 13--- O(*cols*)
- line 14 – 21--- O(*terms*)

and line 9--- O(*cols*), other lines--- O(1)

Total: O(*cols*+*terms*)

- This is a typical example for trading space for time.
- The space required by *FastTranspose* can be reduced by utilizing the same space to represent the two arrays *rowSize* and *rowStart*.

The String Abstract data Type

A string $S = s_0, s_1, \ldots, s_{n-1}$,

where s_i **is a character,** $0 \le i < n$, n **is the length.**

ADT 2.5 String

class *String*

{ // a finite set of zero or more characters**;**

public:

String (**char** **init*, **int** *n*)**;**

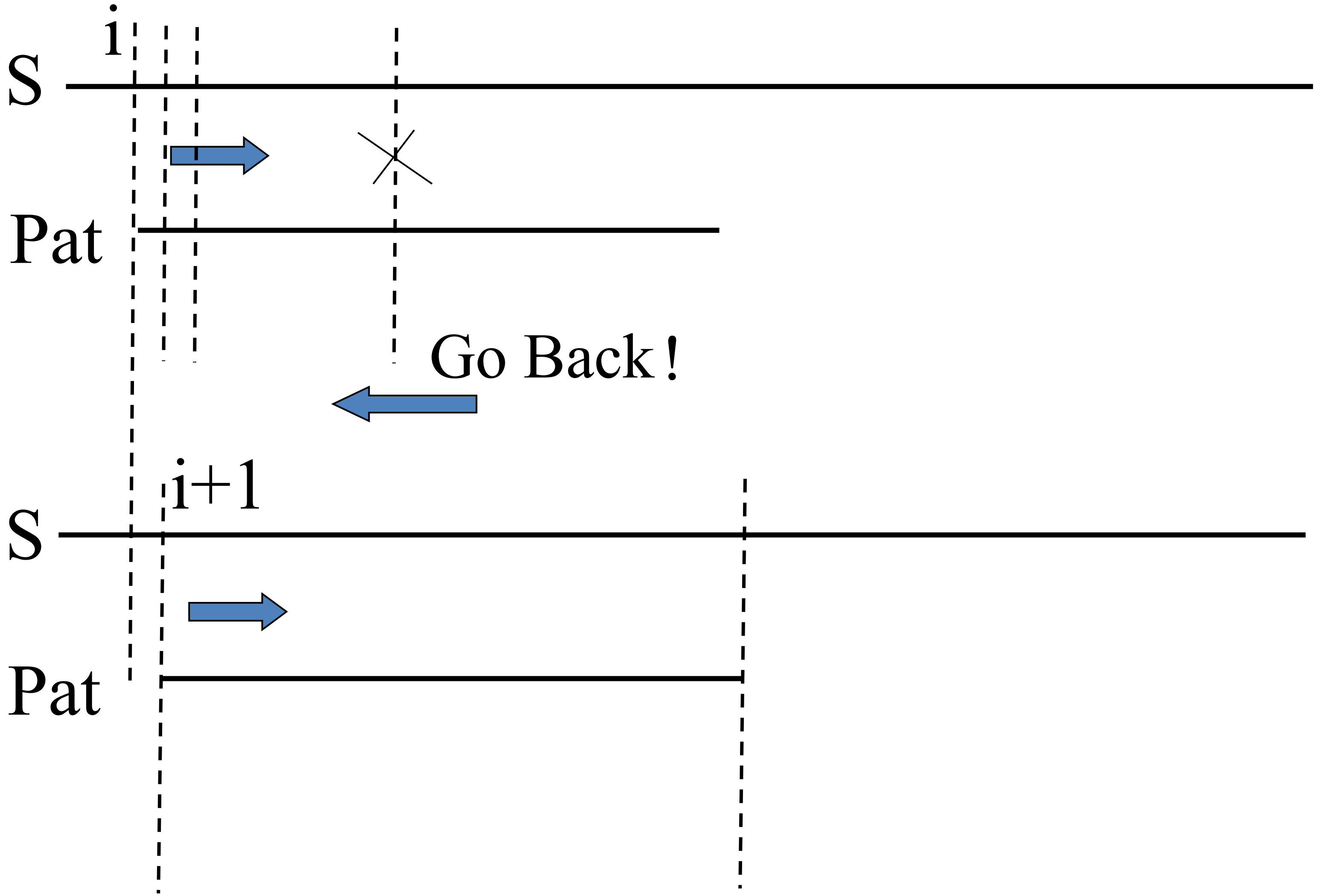
// initialize ***this** to string init of length *n*

```
bool operator == (String t );
// if *this equals t, return true else false.
bool operator ! (  );
// if *this is empty return true else false.
int Length ( );
 // return the number of chars in *this
 String Concat (String t);
 String Substr (int  i, int  j);
 int Find (String pat);
// return i such that pat matches the substring of *this that
// begins at position i. Return –1 if pat is either empty or not
// a substring of *this.
};
```

Assume the String class is represented by:

```
private:
    char* str;
```

String Pattern Matching: A Simple Algorithm



```
int String::Find ( String pat )
{ // Return –1 if pat does not occur in *this; otherwise
  // return the first position in *this, where pat begins.
    if (pat.Length( ) == 0) return -1; // pat is empty
    for (int start=0; start<=Length( ) - pat.Length( ); start++)
    { //  check  for match beginning at str[start]
          int j;
          for (j=0;  j<pat.Length( )&&str[start+j]==pat.str[j]; j++);
          if (j == pat.Length( )) return start; // match found
          // no match at position start
     }
     return -1;     // pat does not occur in s
}
```

The complexity of it is O(*LengthP* * *LengthS*).

Problem:

rescanning.

Even if we check the last character of pat first, the time complexity can’t be improved!

String Pattern Matching: The Knuth-Morris-Pratt Algorithm

Can we get an algorithm which avoid rescanning the strings and works in O(*LengthP* + *LengthS*)?

This is optimal for this problem, as in the worst it is necessary to look at characters in the pattern and string at least once.

Basic Ideas:

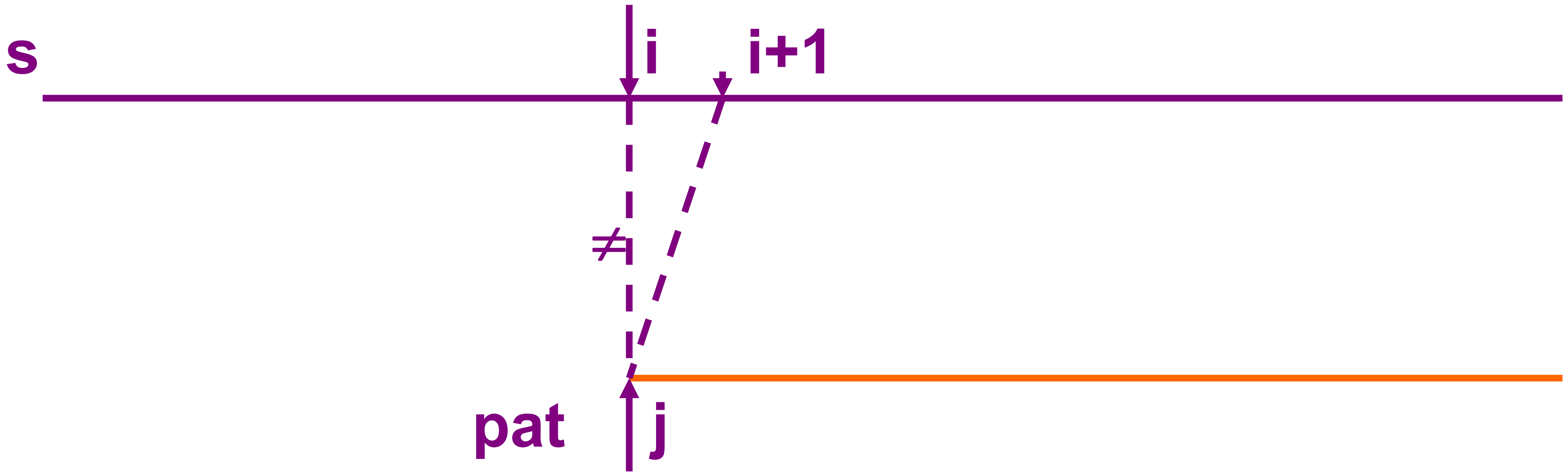
Rescanning to avoid missing the target ---

too conservative

If we can go without rescanning, it is likely to do the job in O(*LengthP* + *LengthS*).

Preprocess the pattern, to get some knowledge of the characters in it and the position in it, so that if a mismatch occurs we can determine where to continue the search and avoid moving backwards in the string.

Now we show details about the idea.



case: $j = 0$

An concrete example:

i

↓

s = . . . a b d a b ? . . . .

≠

pat = a b d a b c a c a b

↑

j=5

An concrete example:

i

↓

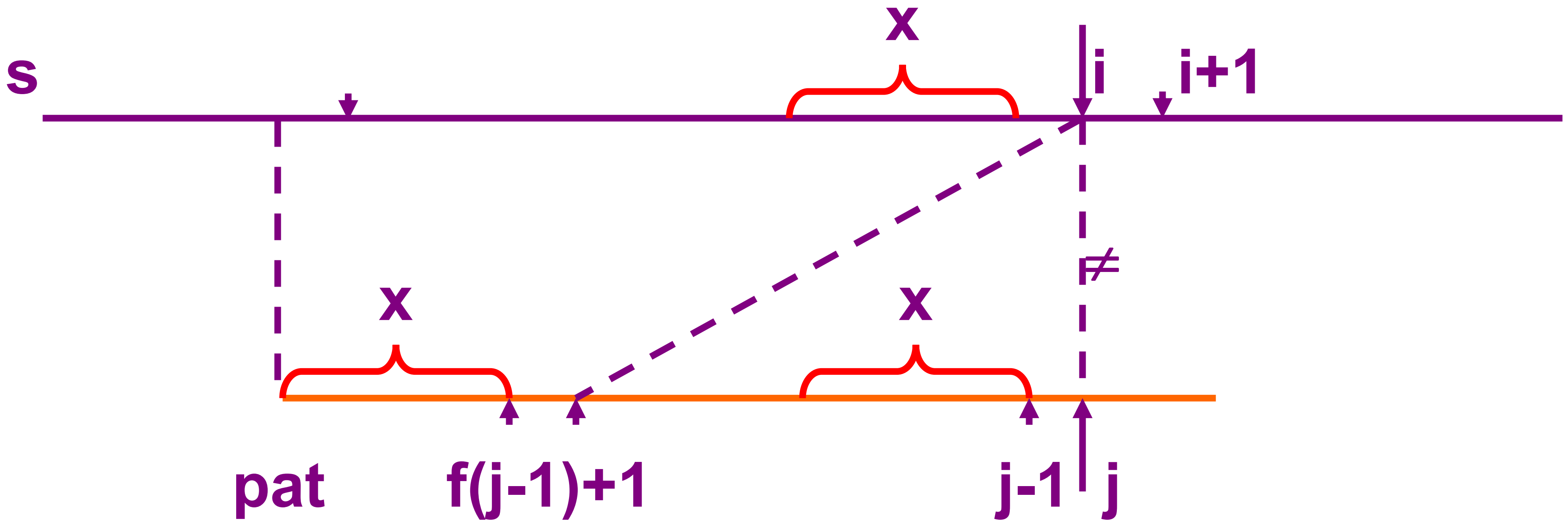
s = . . . a b d **a b** ? . . . .

$\neq$

pat = **a b** d **a b** c a c a b

↑

j=5



case: $j \neq 0$

To formalize the above idea:

Definition: if $p=p_0p_1\ldots p_{n-1}$ is a pattern, then its failure function f, is defined as:

$$f(j) = \begin{cases} \text{largest } k < j, \text{ such that } p_0p_1\ldots p_k = p_{j-k}p_{j-k+1}\ldots p_j & \text{if such } k \geq 0 \text{ exists} \\ -1 & \text{otherwise} \end{cases}$$

For example, pat = a b c a b c a c a b, we have

j	0	1	2	3	4	5	6	7	8	9
pat	a	b	c	a	b	c	a	c	a	b
f	-1	-1	-1	0	1	2	3	-1	0	1

Note:

- largest : no match be missed
- k < j : avoid dead loop

From the definition of *f*, we have the following rule for pattern matching:

If a partial match is found such that $s_{i-j} \ldots s_{i-1} = p_0 p_1 \ldots p_{j-1}$ and $s_i \neq p_j$ then matching may be resumed by comparing s_i and $p_{f(j-1)+1}$ if $j \neq 0$.

If j=0, then we may continue by comparing s_{i+1} and p_0.

The failure function is represented by an array of integers f, which is a private data member of String.

Now the algorithm *FastFind*.

```
int String::FastFind (String pat)
{ // Determine if pat is a substring of s
      int PosP = 0,  PosS = 0;
      int LengthP = pat.Length( ), LengthS = Length( );
      while ((PosP  < LengthP) && (PosS  < LengthS))
          if ( pat.str[PosP] == str[PosS] ) { // characters match
              PosP ++; PosS ++;
          }
          else
              if (PosP==0)
                  PosS++;
              else PosP= pat.f [PosP-1] + 1;
        if ((PosP < LengthP) || (LengthP==0)) return –1;
        else return PosS  - LengthP ;
}
```

Analysis of FastFind:

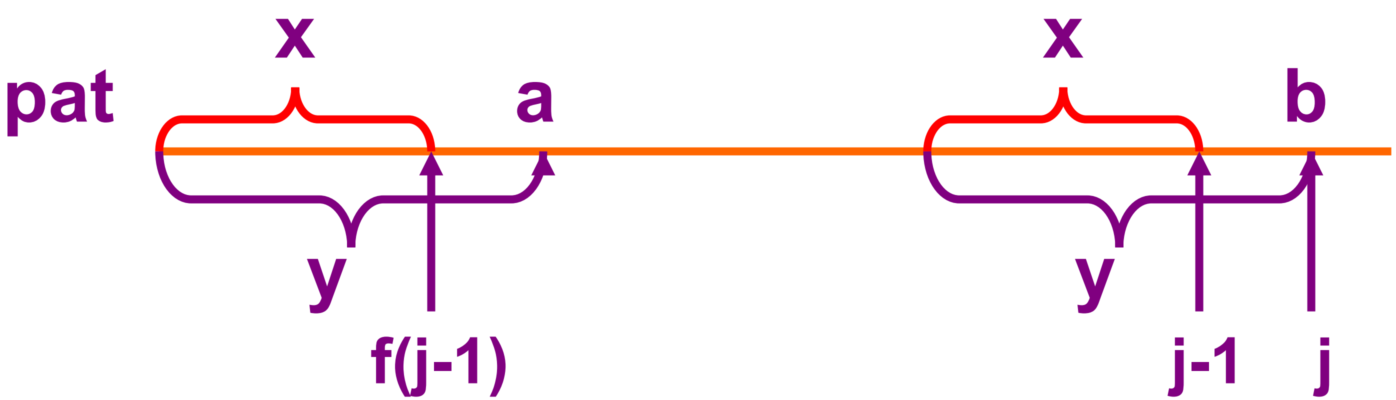
Line 7 and 11 --- at most *LengthS* times, since *PosS* is increased but never decreased. So *PosP* can move right on pat at most *LengthS* times (line 7).

Line 12 moves *PosP* left, it can be done at most *LengthS* times. Note that $f(j\text{-}1)+1< j$.

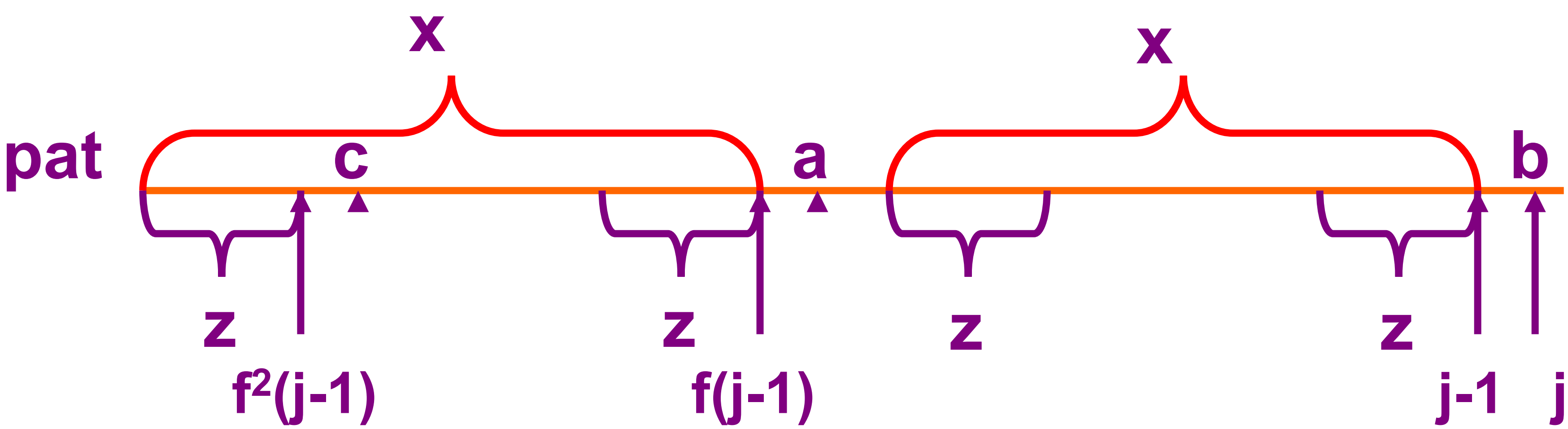
Consequently, the computing time is O(*LengthS*).

How about the computing of the *f* for the pattern? By similar idea, we can do it in O(*LengthP*).

f(0)=-1, now if we have f(j-1), we can compute f(j) from it by the following observation:



If a=b, then f(j)=f(j-1)+1 else



If c=b, $f(j)=f(f(j-1))+1=f^2(j-1)+1$ else ……

In general, we have the following restatement of the failure function:

$$
f(j) = \begin{cases} -1 & \text{if } j=0 \\ f^m(j-1)+1 & \text{where } m \text{ is the least } k \text{ for which } p_{f^k(j-1)+1} = p_j \\ -1 & \text{if there is no } k \text{ satisfying the above} \end{cases}
$$

Now we get the algorithm to compute f.

```
void String::Failurefunction( )
{ // compute the failure function of the pattern *this.
  int LengthP= Length( );
  f [0] = -1;
  for (int j = 1; j < LengthP; j++) // compute  f[j]
    {
     int i = f [j-1];
     while ((*(str+j) != *(str+i+1)) && (i >= 0)) i = f [i]; // try for m
      if ( *(str+j) == *(str+i+1))
         f [j] = i+1;
     else f [j]= -1;
   }
}
```

Analysis of fail:

In each iteration of the while, i decreases (line 8, and $f(j)<j$)

i is reset (line 7) to –1 (when the previous iteration went through line 11), or to a value 1 greater than its value on the previous iteration (when through line 10).

There are only *LengthP* –1 executions of line 7, the value of i has a total increment of at most *LengthP* –1.

i cannot be decremented more than *LengthP* –1 times, the while is iterated at most *LengthP* –1 times over the whole algorithm.

Consequently, the computing time is O(*LengthP*).

Now we can see, when the failure function is not known in advance, pattern matching can be carried out in **O(*LengthP* + *LengthS*)** by first computing the failure function and then using the *FastFind*.