Chapter 1 : Basic Concepts

Data Structures Lecture Note
Prof. Jihoon Yang
Data Mining Research Laboratory

Goals

- To provide the tools and techniques necessary to design and implement large-scale computer systems.
  - solid foundation in data abstraction, algorithm specification and performance analysis and measurement provides the necessary methodology.

1.1 SYSTEM LIFE CYCLE

- **Requirement**
  - a set of specifications that define the purpose of the project.
  - input/output
- **Analysis**
  - break the problems down into manageable pieces.
  - bottom-up / top-down
- **Design**
  - creation of abstract data types
  - specification of algorithms and consideration of algorithm design strategies.

    (* language independent *)

- **Refinement and Coding**
  - choose representations for data objects and write algorithms for each operation on them.
  - data object's representation can determine the efficiency of the algorithms related to it.

- **Verification**
  - Developing correctness proof for the program
  - Testing the program with a variety of input data
  - Error removal
  - Performance analysis
    - running time
    - amount of memory used

1.2 ALGORITHM SPECIFICATION

1.2.1 Introduction

- ***Definition:***

  An algorithm is a finite set of instructions that, if followed, accomplishes particular task.

  All algorithms must satisfy the following criteria:

  (1) Input
  (2) Output
  (3) Definiteness
  (4) Finiteness
  (5) Effectiveness

  algorithm / program (procedure)

- **How to describe an algorithm**

  natural language
  flowchart
  programming language

- **<u>Example 1.1</u> [Selection Sort]**

  Sorting a set of n ≥ 1 integers

  From those integers that are currently unsorted,
  find the smallest and place it next in the sorted list.

- **[Program 1.1 Selection sort algorithm]**

```
for (i=0; i<n; i++) {
    Examine list[i] to list[n-1]
     and suppose that the smallest integer is at list[min];
    Interchange list[i] and list[min];
}
```

- **first task : finding the smallest integer;**
- **second task : exchange;**

  either a function or a macro

- **[Program 1.2 swap function]**

```
void swap(int *x, int *y)
/* both parameters are pointers to ints */
{
        int temp = *x; /* declare temp as an int and assign to it
                          the contents of what x points to */
        *x = *y;       /* stores what y points to into the location
                          where x points  */
        *y = temp;     /* place the contents of temp in the location
                          pointed to by y */
}
```

Call -- swap(&a, &b)

- **macro version of swap -**

```
#define SWAP(x,y,t)    ((t) = (x), (x) = (y), (y) = (t))
```

- **[Program 1.3 Selection sort]**

```
void sort (int list[], int n)
{
        int i, j, min, temp;
        for (i=0; i<n-1; i++)  {
                min = i;
                for (j = i+1; j < n; j++)
                if (list[j] < list[min])
                        min = j;
                SWAP(list[i], list[min], temp);
        }
}
```

- **Theorem 1.1:**

  Function *sort*(*list,n*) correctly sorts a set of n≥1 integers. The result remains in list[0], . . ., list[n-1] such that list[0]≤list[1]≤ . . . ≤list[n-1].

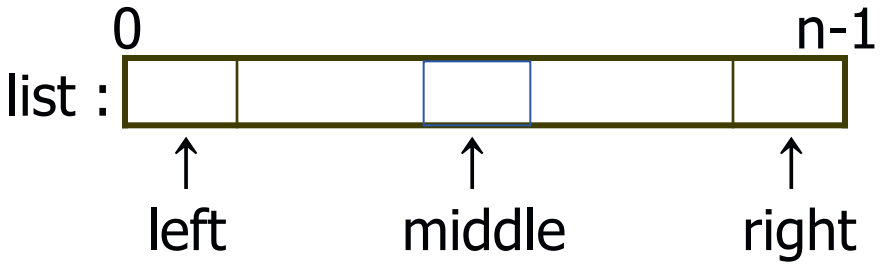
  proof : consider *loop invariant.*

- **Example 1.2 [Binary Search]**

  Find out if an integer *searchnum* is in a list.

  If so, return i such that list[i] = *searchnum,*
  Otherwise, return -1.

  For a sorted list (in ascending order)



middle = (left + right) / 2

- **Compare list[*middle*] with *searchnum***
  - ***searchnum* < list[*middle*]**
    if *searchnum* is present, it must be in the position
    between *left* and *middle-1.*
    set *right* to *middle-1.*
  - ***searchnum* = list[*middle*]**
    return *middle.*
  - ***searchnum* > list[*middle*]**
    if *searchnum* is present, it must be in the position
    between *middle+1* and *right.*
    set *left* to *middle+1*

- **Implementing this search strategy :**

```
while (there are more integers to check) {
        middle = (left + right) / 2;
        if (searchnum < list[middle])
                right = middle - 1;
        else if (searchnum == list[middle])
                return middle;
        else  left = middle + 1;
}
```

- **Handling the comparisons:**

<	returns	-1
=		0
>		1

- ***function* -**

```
int compare (int x, int y)
{
        /* compare x and y, return -1 for less than,
        0 for equal, 1 for greater */
        if (x < y)  return -1;
        else  if  (x == y)  return 0;
        else  return 1;
}
```

- ***macro* -**

```
# define COMPARE (x,y) ((x) < (y)) ? -1: ((x) == (y)) ? 0: 1)
```

- **[Program 1.6]**

```
int binsearch(int list[], int searchnum, int left, int right)
{
  /* search list[0] <= list[1] <= . . . <= list[n-1] for    searchnum.
     Return its position if found.  Otherwise return -1 */
  int middle;
  while (left <= right)  {
        middle = (left + right)/2;
        switch (COMPARE(list[middle], searchnum)) {
                case -1 : left = middle + 1;
                          break;
                case 0 : return middle;
                case 1 : right = middle - 1;
        }
   }
   return -1;
}
```

1.2.2 Recursive Algorithms

Direct recursion

Indirect recursion

- Recursion is a general control scheme.
- Often recursive function is easier to understand than its iterative counterpart.
- Many problems can be defined recursively in natural way.

- **[Binomial Coefficients]**

$$\begin{bmatrix} n \\ m \end{bmatrix} = \frac{n!}{m!(n-m)!}$$

can be recursively computed by the formula:

$$\begin{bmatrix} n \\ m \end{bmatrix} = \begin{bmatrix} n-1 \\ m \end{bmatrix} + \begin{bmatrix} n-1 \\ m-1 \end{bmatrix}$$

- **Examples :**
  - **[factorial]**

$$n! = \begin{cases} n * (n-1)! & \text{if } n>1 \\ 1 & \text{if } n=1 \end{cases}$$

  - **[Binary search]**

$$\text{Bsrch}[key, left, right] = \begin{cases} \text{Bsrch}[key, left, middle-1] & \text{if key} < \text{list}[middle] \\ \text{list}[middle] & \text{if key} = \text{list}[middle] \\ \text{Bsrch}[key, middle+1, right] & \text{if key} > \text{list}[middle] \end{cases}$$

- **[Fibonacci numbers]**

$$f_n = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ f_{n-1} + f_{n-2} & \text{if } n>1 \end{cases}$$

- **[Permutations]**

We can construct the set of permutations by printing:

(1) *a* followed by all permutations of (*b, c, d*)
(2) *b* followed by all permutations of (*a, c, d*)
(3) *c* followed by all permutations of (*a, b, d*)
(4) *d* followed by all permutations of (*a, b, c*)

Iterative function

```
int fibo(int n)
{
   int g, h, f, i;
   if (n>1) {
       g = 0;
       h = 1;
       for (i = 2; i<= n; i++) {
            f = g+h;
            g = h;
            h = f;
       }
   }
   else   f = n;
   return  f;
}
```

Recursive function

```
int rfibo (int n)
{
   if (n > 1)
       return rfibo(n-1) + rfibo(n-2);
   else
       return n;
}
```

- **[Program 1.7]**

```
int binsearch(int list[], int searchnum, int left, int right)
{
 /* search list[0] <= list[1] <= . . . <= list[n-1] for searchnum.
     Return its position if found.  Otherwise return -1 */

 int middle;
 if (left <= right)  {
    middle = (left + right)/2;
    switch (COMPARE(list[middle], searchnum)) {
         case -1 : return  binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return  binsearch(list, searchnum, left, middle - 1);
    }
  }
  return -1;
}
```

- **[Program 1.8]**

```
void perm(char *list, int i, int n)
{
   /* generate all the permutations of list[i] to list[n] */
   int j, temp;
   if (i == n)  {
       for (j=0; j<=n; j++) printf("%c", list[j]);
       printf("     ");
   }
   else  {
   /* list[i] to list[n] has more than one permutation,
       generate these recursively  */
         for (j=i; j<=n; j++)  {
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
         }
   }
}
```

1.3 DATA ABSTRACTION

- **basic data types of C :**
  char, int, float, double, . . .
  short, long, unsigned

- **mechanisms for grouping data together :**
  Arrays and Structs

```
int  list[5];

struct student {
        char last_name[10];
        int  student_id;
        char grade;
};
```

- **pointer data type :**

for every basic data type
there is a corresponding pointer data type, such as
pointer-to-an-int,
pointer-to-a-real,
pointer-to-a-char,
and pointer-to-a-float.

```
int   i,  *pi;
```

predefined data types / user-defined data types

"What is a data type?"

- **Definition :**

  A *data type* is a collection of *objects* and a set of *operations* that act on those objects.

  - **specification of objects**

    e.g., type *int*,

    {0, +1, -1, +2, -2, . . ., INT_MAX, INT_MIN}

    specification of operations

  - **representation of objects**

    implementation of operations

- **Definition :**

  An *abstract data type* (ADT) is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operations.

- ***an abstract data type is implementation independent.***

  Specification of operations consists of the names of operations, the type of its arguments, and the type of its result. Also a description what the function does without appealing to internal representation details.

  ***package* in Ada**

  ***class* in C++**

- **Categories to classify the operations of a data types:**
  - Creator/constructor
  - Transformers
  - Observers/reporters

Example 1.5 [Abstract data type *Natural_Number*]

ADT *Natural_Number* is

object: an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT_MAX*) on the computer

functions:

for all x, y IN *Nat_Number*, *TRUE, FALSE* IN *Boolean*

and where +, -, <, and == are usual integer operations

Nat_No Zero()	::=	0
Boolean Is_Zero(*x*)	::=	if (*x*) return *FALSE* else return *TRUE*
Nat_No Add(*x, y*)	::=	if ((*x+y*)<= *INT_MAX*) return *x+y* else return *INT_MAX*
Boolean Equal(*x, y*)	::=	if (*x==y*) return *TRUE* else return *FALSE*
Nat_No Successor(*x*)	::=	if (*x* == *INT_MAX*) return *x* else return *x+1*
Nat_No Subtract(*x,y*)	::=	if (*x<y*) return 0 else return *x-y*

end *Natural_Number*

1.4 PERFORMANCE ANALYSIS

- ***Criteria of judging a program:***
  1. Does the program meet the original specification of the task?
  2. Does it work correctly?
  3. Is the program well documented?
  4. Does the program effectively use functions to create logical units?
  5. Is the program's code readable?

  [Performance Evaluation]
  6. Does the program efficiently use primary and secondary storage?
  7. Is the program's running time acceptable for the task?

- ***Performance Analysis*** **:**
  estimates of time and space that are machine independent.

- ***Performance Measurement*** **:**
  obtaining machine-dependent running times.
  used to identify inefficient code segments.

- **Definition :**
  The *space complexity* of a program is the amount of memory that it needs to run to completion.
  The *time complexity* of a program is the amount of computer time that it needs to run to completion.

1.4.1 Space Complexity

- ***Fixed space requirements*** **:**

  independent from the number and size of the program's inputs and outputs, e.g., the instruction space, space for simple variables, fixed-size structured variables, and constants.

- ***Variable space requirements*** **:**

  space needed by structured variables whose size depends on the particular instance, I, of the problem being solved.

$$S(P) = c + S_P(I)$$

- **Example 1.6 : [simple arithmetic function]**

$S_{abc}(I) = 0.$

[Program 1.9]

```
float abc (float a, float b, float c)
{
    return a+b+b*c + (a+b-c)/(a+b) + 4.00;
}
```

- **Example 1.7 : [*adding a list of numbers iteratively*]**

[Program 1.10]

```
float sum(float list[], int n)
{
        float  tempsum = 0;
        int i;
        for (i=0; i<n; i++)
                tempsum += list[i];
        return tempsum;
}
```

$S_{sum}(n) = n$ if parameters are passed by value.
$S_{sum}(n) = 0$ if parameters are passed by reference

- **Example 1.8 : [*adding a list of numbers recursively*]**

  [Program 1.11]

```
float rsum(float list[], int n)
{
        if (n) return rsum(list, n-1) + list[n-1];
        return 0;
}
```

$S_{rsum}(n) = 12*n$

Type	Name	Number of bytes
parameter: array pointer parameter: integer return address: (used internally)	list[] n	4 4 4
TOTAL per recursive call		12

Figure 1.1 : Space needed for one recursive call of program 1.11

1.4.2 Time Complexity

(1) *Compile Time*

(2) *Execution (Running) Time*

We are really concerned only with the program's execution time.

- ***Determining the execution time*** **:**
  - the times needed to perform each operation.
  - the number of each operation performed for the given instance (dependent on the compiler).

$$T_P(n) = c_a ADD(n) + c_s SUB(n) + c_l LDA(n) + c_{st} STA(n)$$

- Obtaining such a detailed estimate of running time is rarely worth the effort.

- Counting the number of operations the program performs gives us a machine-independent estimate.

- **Definition :**

  A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

  Determining the number of steps that a program or a function needs to solve a particular problem instance by creating a global variable, *count*, and inserting statements that increment count

- **[Example 1.9] [*Iterative summing of a list of numbers*]**

[Program 1.12]

```
float sum(float list[], int n)
{
    float  tempsum = 0;  count++;    /*for assignment*/
    int i;
    for (i=0; i<n; i++)  {
        count++;   /*for the for loop */
        tempsum += list[i];  count++;  /*for assignment*/
    }
    count++;  /* last execution of for */
    count++;  /* for return */  return tempsum;
}
```

[Program 1.13]

```
float sum(float list[], int n)
{
        float  tempsum = 0;
        int i;
        for (i=0; i<n; i++)
            count += 2;   /*for the for loop */
        count += 3;
        return 0;
}
```

If the initial value of count is 0, its final value will be $2n+3$.

- **[Example 1.10] [*Recursive summing of a list of numbers*]**

[Program 1.14]

```
float rsum(float list[], int n)
{
    count++;    /* for if conditional */
    if (n) {
        count++;            /* for return and rsum invocation */
        return rsum(list, n-1) + list[n-1];
    }
    count++;
    return list[0];
}
```

the step count is $2n+2$.

- **[Example 1.11] : [*Matrix addition*]**

[Program 1.15]

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
                int c[][MAX_SIZE], int rows, int cols)
{
      int i, j;
      for (i=0; i<rows; i++)
         for (j=0; j<cols; j++)
       c[i][j] = a[i][j] + b[i][j];
}
```

[Program 1.16]

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
                int c[][MAX_SIZE], int rows, int cols)
{
        int i, j;
        for (i=0; i<rows; i++)   {
                count++;    /* for i for loop */
                for (j=0; j<cols; j++)  {
                        count++;     /* for j for loop */
                        c[i][j] = a[i][j] + b[i][j];
                        count++;     /* for assignment statement */
                }
                count++;  /* last time of j for loop */
        }
        count++;  /* last time of i for loop */
}
```

[Program 1.17]

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
                    int c[][MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i=0; i<rows; i++)   {
      for (j=0; j<cols; j++)
            count += 2;
      count += 2;
   }
   count++;
}
```

The step count will be 2 *rows* *cols* + 2*rows* +1

- **Tabular method: *steps/execution***

[Figure 1.2]

Statement	s/e	Frequency	Total steps
float sum(float list[], int n)	0	0	0
{	0	0	0
float tempsum=0;	1	1	1
int i;	0	0	0
for (i=0; i<n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

- **[Example 1.13]**

[Figure 1.3]

Statement	s/e	Frequency	Total steps

float rsum(float list[], int n)
{
  if (n)
  return rsum(list, n-1)+list[n-1];
  return list[0];
}
	0
	0
	1
	1
	1
	0
		0
		0
		n+1
		n
		1
		0
			0
			0
			n+1
			n
			1
			0

Total			2n+2

- **[Example 1.14]**

[Figure 1.4]

Statement	s/e	Frequency	Total Steps

void add(int a[][MAX_SIZE])
{
  int i, j;
  for (i=0; i<rows; i++)
  for (j=0; j<cols; j++)
  c[i][j]=a[i][j]+b[i][j];
}
	0
	0
	0
	1
	1
	1
	0
		0
		0
		0
		rows + 1
		rows·(cols+1)
		rows·cols
		0
			0
			0
			0
			rows + 1
			rows·cols + rows
			rows·cols
			0

Total			2rows·cols + 2rows + 1

Summary

- Time complexity of a program is given by the number of steps taken by the program to compute the function it was written for.
- The number of steps is itself a function of the instance characteristics.

  e.g., the number of inputs, the number of outputs,

  the magnitudes of the inputs and outputs, etc.
- Before the step count of a program can be determined, we need to know exactly which characteristics of the problem are to be used.

- For many programs, the time complexity is not dependent solely on the characteristics specified.

- The step count varies for different inputs of the same size.
  - Best case
  - **Worst case**
  - **Average**

Examples :

Binary Search

Insertion Sort

1.4.3 Asymptotic Notation (O, Ω, Θ)

- **Our motivation to determine step counts:**

  to compare the time complexities of two programs for the same function, and

  to predict the growth in run time as the instance characteristics change.

- Determining the exact step count (either worst case or average) of a program can prove to be an exceedingly difficult task.

- Expending immense effort to determine the step count exactly isn't a worthwhile endeavor as the notion of a step is itself inexact.

  (e.g., x = y and x = y+z+(x/y)+(x*y*z-x/t) count as one step)

- Because of the inexactness of what a step stands for, the exact step count isn't very useful for comparative purposes.

- For most situations, step counts can be represented as a function of instance characteristics, such as $c_1 n^2 \leq T_P(n) \leq c_2 n^2$ or $T_Q(n, m) = c_1 n + c_2 m$.

  What if the difference of two step counts are large?
  e.g., 3n+3 versus 100n+10.

  What if two step counts are of different orders?
  e.g., $c_1 n^2 + c_2 n$ versus $c_3 n$.

- ***break even point*** :
  The exact break even point cannot be determined analytically.
  The programs have to be run on a computer
  in order to determine the break even point.

Some terminology :

- **Definition : [Big "*oh*"]**

  f(n) = O(g(n))

  iff there exist positive constants c and n_0 such that $f(n) \leq c\,g(n)$ for all n, $n \geq n_0$.

$3n + 2 = O(n)$

$100n + 6 = O(n)$

$1000n^2 + 100n - 6 = O(n^2)$

$3n + 3 = O(n)$

$10n^2 + 4n + 2 = O(n^2)$

$6*2^n + n^2 = O(2^n)$

$3n + 3 = O(n^2)$

$3n + 2 \neq O(1)$

$10n^2 + 4n + 2 = O(n^4)$

$10n^2 + 4n + 2 \neq O(n)$

$O(1)$	a constant	$O(n^2)$	quadratic
$O(\log n)$	logarithm	$O(n^3)$	cubic
$O(n)$	linear	$O(2^n)$	exponential

In order for the statement $f(n) = O(g(n))$ to be informative, $g(n)$ should be as small a function of n as one can come up with for which $f(n) = O(g(n))$.

- **Theorem 1.2 :**
  If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ **then** $f(n) = O(n^m)$.

- **Proof :**

$$f(n) \le \sum_{i=0}^{m} |a_i| n^i$$

$$\le n^m \sum_{i=0}^{m} |a_i| n^{i-m}$$

$$\le n^m \sum_{i=0}^{m} |a_i|, \quad \text{for } n \ge 1.$$

SO, $f(n) = O(n^m)$

- **Definition : [Omega]**

$$f(n) = \Omega(g(n))$$

iff there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all n, $n \geq n_0$.

- **Example 1.16 :**

$3n + 2 = \Omega(n)$

$3n + 3 = \Omega(n)$

$100n + 6 = \Omega(n)$

$10n^2 + 4n + 2 = \Omega(n^2)$

$6*2^n + n^2 = \Omega(2^n)$

$10n^2 + 4n + 2 = \Omega(n)$

$10n^2 + 4n + 2 = \Omega(1)$

$6*2^n + n^2 = \Omega(n^{100})$

$6*2^n + n^2 = \Omega(n^2)$

$6*2^n + n^2 = \Omega(n)$

$6*2^n + n^2 = \Omega(1)$

In order for the statement $f(n) = \Omega(g(n))$ to be informative, $g(n)$ should be as large a function of n as possible for which $f(n) = \Omega(g(n))$ is true.

- **Theorem 1.3 :**

  If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ **and** $a_m > 0$ **,**
  then $f(n) = \Omega(n^m)$ **.**

- **Definition : [Theta]**

$$f(n) = \Theta(g(n))$$

iff there exist positive constants c_1, c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$.

- **Example 1.17 :**

$3n + 2 = \Theta(n)$

$3n + 3 = \Theta(n)$

$10n^2 + 4n + 2 = \Theta(n^2)$

$6*2^n + n^2 = \Theta(2^n)$

$10*\log n + 4 = \Theta(\log n)$

$3n + 2 \neq \Theta(1)$

$3n + 2 \neq \Theta(n^2)$

$10n^2 + 4n + 2 \neq \Theta(n)$

$10n^2 + 4n + 2 \neq \Theta(1)$

$6*2^n + n^2 \neq \Theta(n^{100})$

$6*2^n + n^2 \neq \Theta(n^2)$

$6*2^n + n^2 \neq \Theta(1)$

- **Theorem 1.4 :**
  If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ **and** $a_m > 0$ **,**
  then $f(n) = O(n^m)$ **.**

- **Example 1.18*: [Complexity of matrix addition]***

Statement	Asymptotic complexity
void add(int a[][MAX_SIZE] ...)	0
{	0
int i, j;	0
for (i=0; i<rows; i++)	Θ(rows)
for (j=0; j<cols; j++)	Θ(rows · cols)
c[i][j] = a[i][j] + b[i][j];	Θ(rows · cols)
}	0
Total	Θ(rows · cols)

- **Example 1.19 : *[Binary Search]***

  [Program 1.6]

  The instance characteristic -- number of elements in the list.

  Each iteration of *while* loop takes Θ(1) time.

  The *while* loop is iterated at most $\lceil \log_2(n+1) \rceil$ times.

  Worst case - the loop is iterated **Θ(log n)** times

  Best case - **Θ(1)**.

- **Example 1.21 : *[Magic square]***

  The magic square is an $n \times n$ matrix of integers from 1 to n^2 such that the sum of each row and column and two major diagonals is the same.

  When n=5 : the common sum is 65.

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

- **Coxeter's rule :**

  Put a one in the middle of the top row. Go up and left assigning numbers in increasing order to empty boxes. If your move cause you to jump off the square (that is, you go beyond the square's boundaries), figure out where you would be if you landed on a box on the opposite side of the square. Continue with this box. If a box is occupied, go down instead of up and continue.

[Program 1.22]

```
#include <stdio.h>
#define MAX_SIZE  15  /* maximum size of square */

void main(void)
/* construct a magic square, iteratively */
{
  static int square[MAX_SIZE] [MAX_SIZE];
  int i, j, row, column;        /* indices */
  int count;                    /* counter */
  int size;                     /* Square size */
```

```
printf (“Enter the size of the square: ”);
scanf(“%d’, &size);
 /* check for input errors */
 if (size<1 || size>MAX_SIZE+1) {
            fprintf(stderr, “Error! Size is out of range□n”);
            exit(1);
    }
 if (!(size % 2))   {
            fprintf(stderr, “Error! Size is even”);
            exit(1);
    }
 for (i=0; i<size; i++)
    for (j=0; j<size; j++)
            square[i][j] = 0;
 square[0][(size-1)/2] = 1;  /* middle of first row */
```

```
/* i and j are current position  */
   i = 0;
   j = (size-1) / 2;
   for (count = 2; count <= size * size; count++)  {
          row = (i-1 < 0) ? (size-1) : (i-1);    /* up */
          column = (j-1 < 0) ? (size-1) : (j-1);  /* left */
          if (square[row][column])           /* down  */
            i = (++i) % size;
          else  {                    /* square is unoccupied */
            i = row;
            j = column;
          }
          square[i][j] = count;
     }
```

```
/* output the magic square */
   printf("Magic Square of the size %d : □n□n", size);
   for (i = 0; i < size; i++)   {
     for (j = 0; j < size; j++)
            printf ("%5d", square[i][j];
     printf("□n");
    }
  printf("□n □ n");
}
```

instance characteristic -- *n* denoting the size of the magic square.

the nested for loops -- $O(n^2)$

next for loop -- $O(n^2)$

Others --- $O(1)$

Total asymptotic complexity is $O(n^2)$.

1.4.4 Practical Complexities

- The time complexity of a program is generally some function of the instance characteristics.

- This complexity function:
  - is very useful in determining how the time requirements vary as the instance characteristics changes, and
  - may also be used to compare two programs P and Q that perform the same task.

Assume that program P has complexity $\Theta(n)$ and
program Q has complexity $\Theta(n^2)$.

We can assert that
P is faster than program Q for *sufficiently large* n.

How the various functions grow with n?

Instance characteristic *n*							
Time	Name	1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
log n	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
nlog n	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
n!	Factorial	1	2	24	40326	20922789888000	26313×10^{33}

Figure 1.7 Function values

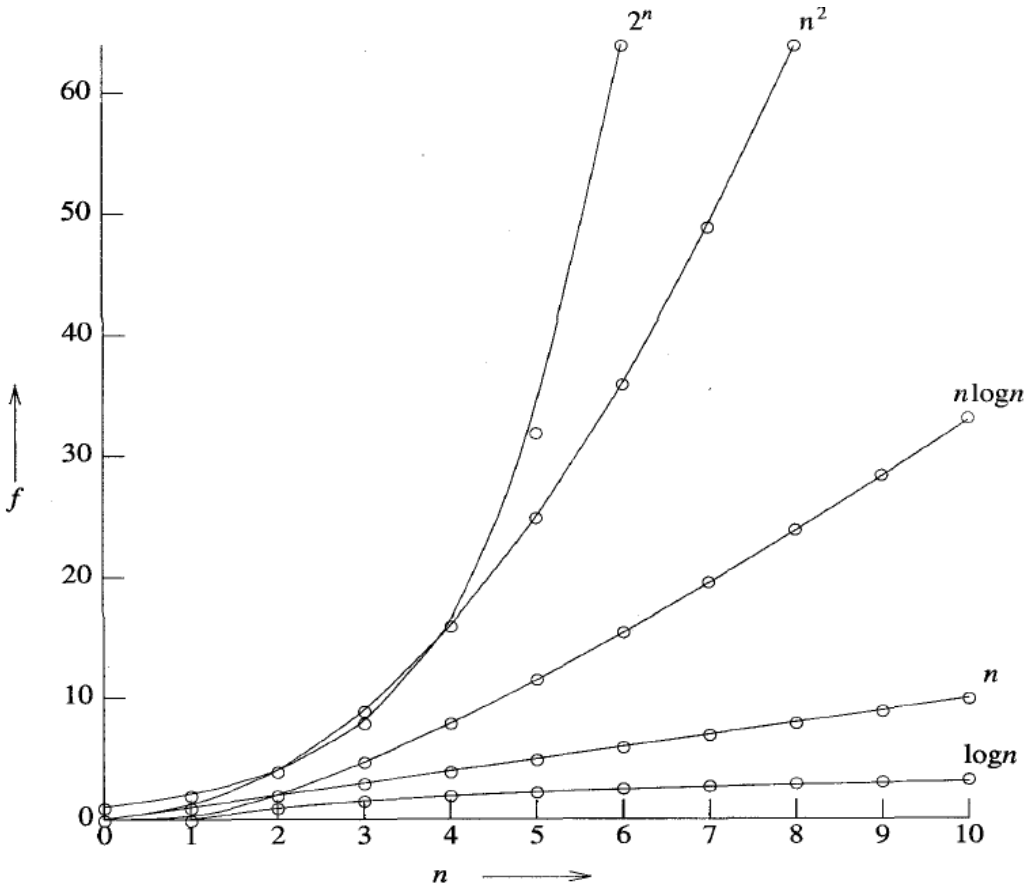


Figure 1.8 Plot of function values

	$f(n)$						
n	n	$n\log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 μs	.03 μs	.1 μs	1 μs	10 μs	10 s	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84 h	1 ms
30	.03 μ	.15 μ	.9 μ	27 μ	810 μ	6.83 d	1 s
40	.04 μs	.21 μs	1.6 μs	64 μs	2.56 ms	121 d	18 m
50	.05 μs	.28 μs	2.5 μs	125 μs	6.25 ms	3.1 y	13 d
100	.10 μs	.66 μs	10 μs	1 ms	100 ms	3171 y	$4*10^{13}$ y
10^3	1 μs	9.96 μs	1 ms	1 s	16.67 m	$3.17*10^{13}$ y	$32*10^{283}$ y
10^4	10 μs	130 μs	100 ms	16.67 m	115.7 d	$3.17*10^{23}$ y	
10^5	100 μs	1.66 ms	10 s	11.57 d	3171 y	$3.17*10^{33}$ y	
10^6	1 ms	19.92 ms	16.67 m	31.71 y	$3.17*10^7$ y	$3.17*10^{43}$ y	

μs = microsecond = 10^{-6} seconds; ms = milliseconds = 10^{-3} seconds
s = seconds; m = minutes; h = hours; d = days; y = years

Figure 1.9 Times on a 1 billion instruction per second computer

1.5 PERFORMANCE MEASUREMENT

- **How to measure real execution time.**
  - Use of C's standard library.
    Functions are accessed through the statement:
    #include <time.h>.
  - Inaccurate results can be produced for small data

(e.g. if the value of CLK_TCK is 18 on our computer, the number of clock ticks for n < 500 is less than 10)

	Method 1	Method 2
Start timing	Start=clock();	Start=time(NULL);
Stop timing	Stop=clock();	Stop=time(NULL);
Type returned	Clock_t	Time_t
Result in seconds	Duration= ((double)(stop-start))/ CLOCKS_PER_SEC;	Duration= (double) difftime(stop, start);

Figure 1.10: Event timing in C

```
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
   int i, n, step = 10;
   int a[MAX_SIZE];
   double duration;
   clock_t start;

   /* times for n = 0, 10, ..., 100, 200, ..., 1000 */
   printf("    n     time\n");
   for (n = 0; n <= 1000; n += step)
   {/* get time for size n */

      /* initialize with worst-case data */
      for (i = 0; i < n; i++)
         a[i] = n - i;

      start = clock( );
      sort(a, n);
      duration = ((double) (clock() - start))
                              / CLOCKS_PER_SEC;
      printf("%6d   %f\n", n, duration);
      if (n == 100) step = 100;
   }
}
```

Program 1.24: First timing program for selection sort

```c
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
   int i, n, step = 10;
   int a[MAX_SIZE];
   double duration;

   /* times for n = 0, 10, ..., 100, 200, ..., 1000 */
   printf("    n    repetitions     time\n");
   for (n = 0; n <= 1000; n += step)
   {
      /* get time for size n */
      long repetitions = 0;
      clock_t start = clock( );
      do
      {
         repetitions++;

         /* initialize with worst-case data */
         for (i = 0; i < n; i++)
            a[i] = n - i;

         sort(a, n);
      } while (clock( ) - start < 1000);
           /* repeat until enough time has elapsed */

      duration = ((double) (clock() - start))
                              / CLOCKS_PER_SEC;
      duration /= repetitions;
      printf("%6d  %9d   %f\n", n, repetitions, duration);
      if (n == 100) step = 100;
   }
}
```

Program 1.25: More accurate timing program for selection sort

n	repetitions	time
0	8690714	0.000000
10	2370915	0.000000
20	604948	0.000002
30	329505	0.000003
40	205605	0.000005
50	145353	0.000007
60	110206	0.000009
70	85037	0.000012
80	65751	0.000015
90	54012	0.000019
100	44058	0.000023
200	12582	0.000079
300	5780	0.000173
400	3344	0.000299
500	2096	0.000477
600	1516	0.000660
700	1106	0.000904
800	852	0.001174
900	681	0.001468
1000	550	0.001818

Figure 1.11: Worst case performance of selection sort (in seconds)

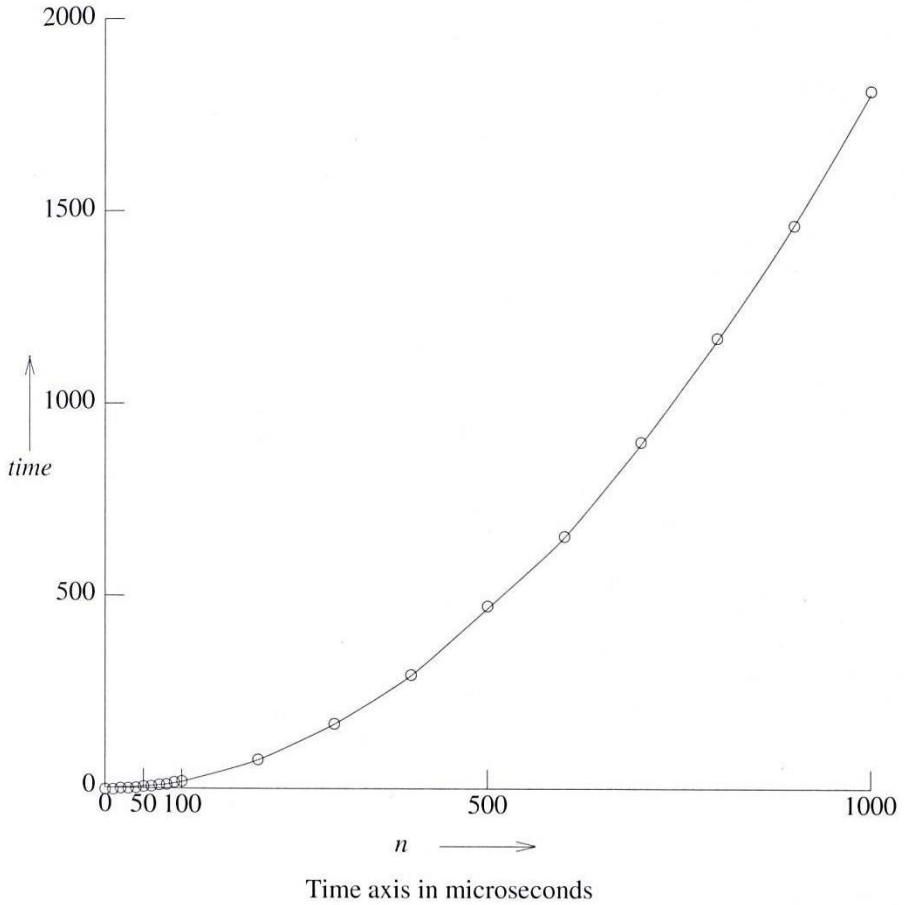


Figure 1.12: Graph of worst case performance of selection sort

Generating Test Data

- Generating a data set that results in the worst case performance of a program isn't always easy.
- We may generate a suitably large number of random test data.
- Obtaining average case data is usually much harder.
- It is desirable to analyze the algorithm being tested to determine classes of data that should be generated for the experiment - algorithm specific task.

Chapter 2 : ARRAYS AND STRUCTURES

Data Structure Lecture Note
Prof. Jihoon Yang
Data Mining Research Laboratory

2.1 ARRAYS

2.1.1 The Abstract Data Type

- An *array* is usually viewed as "a consecutive set of memory locations" which is a usual implementation.
- An *array* as an ADT is a set of pairs, <*index, value*>, such that each index that is defined has a value associated with it.
- Aside from creating a new array, most languages provide only two standard operations for arrays,

  (1) *retrieving a value*
  (2) *storing a value*

<Abstract Data Type *Array*>

ADT *Array* is

objects : A set of pairs <*index*, *value*> where for each value of *index* there is a value from the set *item*. *Index* is a finite 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.

functions :

for all A ∈ *Array*, *i* ∈ *index*, *x* ∈ *item*, *j*, *size* ∈ integer

Array Create(*j*, *list*) ::= **return** an array of *j* dimensions where *list* is a *j*-tuple whose *i*th element is the size of the *i*th dimension. *Items* are undefined.

Item Retrieve(A, *i*) ::= **if** (*i* ∈ *index*) **return** the item associated with index value *i* in array A
else return error.

Array Store(A, *i*, *x*) ::= **if** (*i* ∈ *index*) **return** an array that is identical to array A except the new pair <*i*, *x*> has been inserted
else return error.

end *Array*

2.1.2 Arrays in C

- Declaration of one-dimensional arrays in C :

```
int   list[5], *plist[5];
```

- Memory allocation of arrays :

Variable	Memory address
list[0]	base address = α
list[1]	α + sizeof(int)
list[2]	α + 2·sizeof(int)
list[3]	α + 3·sizeof(int)
list[4]	α + 4·sizeof(int)

C interprets list[i] as a pointer to an integer.

- Observe the difference between a declaration such as

```
int  *list1;
```

and

```
int  list2[5];
```

Variables *list1* and *list2* are both pointers to an integer type object. *list2* is a pointer to *list2*[0] and *list2*+*i* is a pointer to *list2*[i].

Thus, (*list2*+*i*) equals *&list2*[i].
So, *(*list2*+*i*) equals *list2*[i].

[Program 2.1]

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
int i;
void main(void)
{
        for (i=0; i<MAX_SIZE; i++)
                input[i] = i;
        answer = sum(input, MAX_SIZE);
        printf("The sum is: %f\n", answer);
}

float sum(float list[], int n)
{
        int i;
        float tempsum = 0;
        for (i=0; i<n; i++)
                tempsum += list[i];
        return tempsum;
}
```

- When sum is invoked, *input* = &*input*[0] is copied into a temporary location and associated with the formal parameter *list*.
- When *list*[i] occurs on the right-hand side of '=' in an assignment statement, a dereference takes place and the value pointed at by (*list+i*) is returned.
- If *list*[i] appears on the left-hand side of '=', then the value produced on the right-hand side is stored in the location (*list+i*).

- **Example 2.1 [One-dimensional array addressing]**

  int one[]={0, 1, 2, 3, 4};

  A function that prints out both the address of the *i*th element of this and the value found at this address.

- **[Program 2.2]**

```
void print1(int *ptr, int rows)
{
 /* print out a one-dimensional array using a pointer */
  int i;
  printf("Address Contents\n");
  for (i=0; i<rows; i++)
          printf("%8u%5d\n", ptr + i, *(ptr + i));
  printf("\n");
}
```

- **[Figure 2.1] One-dimensional array addressing**

Address	Contents
12244868	0
12344872	1
12344876	2
12344880	3
12344884	4

2.2 DYNAMICALLY ALLOCATED ARRAYS

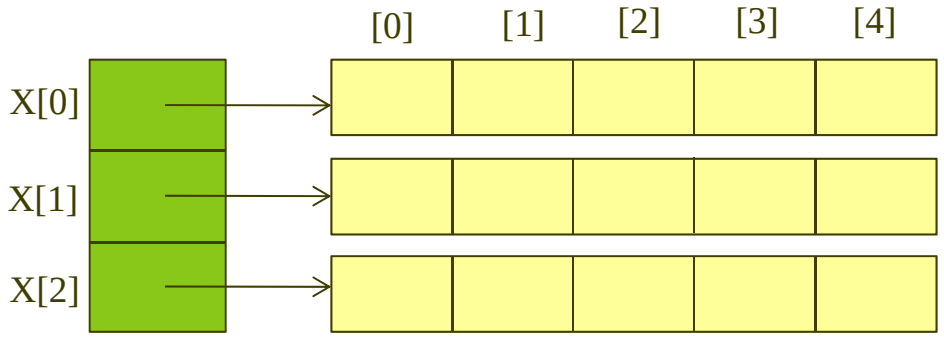
2.2.1 ONE-DIMENSIONAL ARRAYS

- If the user wishes to change array size, we have to change the definition of *MAX_SIZE* and recompile the program.
- A good solution to this problem is to defer this decision to run time and allocate the array when we have a good estimate of the required array size.

```
int i, n, *list;
printf("Enter the number of numbers to generate: ");
scanf("%d", &n);
if ( n < 1 ) {
        fprintf(stderr, "Improper value of n \n");
        exit(EXIT_FAILURE);
}
MALLOC(list, n * sizeof(int));
```

2.2.2 TWO-DIMENSIONAL ARRAYS

- A 2-D array is represented as a 1-D array in which each element is itself a 1-D array
- (e.g.) int x[3][5];



- A 3-D array is represented as a 1-D array in which each element is itself a 2-D array

[Program 2.3]

```
int** make2dArray(int rows, int cols)
{ /* create a two dimensional rows * cols array */
        int **x, i;

        /* get memory for row pointers */
        MALLOC (x, rows * sizeof (*x));;

                                                    =>    int x[rows][cols]
        /* get memory for each row */
        for (i=0; i < rows; i++)
                MALLOC (x[i], cols * sizeof (**x));
        return x;
}
```

cf. x = (int **)malloc(rows*sizeof(*x));

- void* calloc(elt_count, elt_size)

  → allocates a region of memory large enough to hold an array of *elt_count* elements, each of size *elt_size*, and the region of memory is set to zero

- void* realloc(p, s)

  → changes the size of memory block pointed at by p to s

2.3 STRUCTURES AND UNIONS

2.3.1 Structures

- A *structure* (called a *record* in many other programming language) is a collection of data items, where each item is identified as to its *type* and *name*.
- For example, the following declaration creates a variable whose name is *person* with three fields.

```
struct {
   char  name[10];
   int  age;
   float salary;
} person;
```

- The structure member operator · is used to select a particular member of the structure.

```
strcpy (person.name, "james");
person.age = 10;
person.salary = 35000;
```

- Creating new structure data types by using the *typedef* statement :

```
typedef struct humanBeing {
        char  name[10];
        int  age;
        float  salary;
        } ;
```

or

```
typedef struct {
        char name[10];
        int  age;
        float  salary;
        } humanBeing;
```

```
humanBeing  person1,  person2;
```

```
if (strcmp(person1.name, person2.name))
    printf("The two people do not have the same name");
else
    printf("The two people have the same name");
```

* Entire structure operation ?

<person1 = person2>

```
strcpy(person1.name, person2.name);
person1.age = person2.age;
person1.salary = person2.salary;
```

<person1 == person2>

```
#define  FALSE  0
#define  TRUE  1
```

[Program 2.4]

```
int humans_equal(human_being person1, human_being person2)
{
   /* return TRUE if person1 and person2 are the same human  being
   otherwise  return FALSE */
   if (strcmp(person1.name, person2.name))
         return FALSE;
   if (person1.age != person2.age)
         return FALSE;
   if (person1.salary != person2.salary)
           return FALSE;
   return TRUE;
}

if (humans_equal(person1, person2))
   printf("The two human beings are the same");
else
   printf("The two human beings are not the same");
```

- A structure within a structure

```
typedef  struct  {
            int  month;
            int  day;
            int  year;
           }  date;

typedef struct human_being {
            char  name[10];
            int  age;
            float  salary;
            date  dob;
            };

person1.dob.month = 2;
person1.dob.day = 11;
person1.dob.year = 1944;
```

2.3.2 Unions

- Fields share their memory space → only one field of union is active at any given time

```
typedef  struct sex-type  {
        enum  tag_field  {female, male}  sex;
        union  {
                int  children;
                int  beard;
              }  u;
        };

typedef struct human_being {
        char  name[10];
        int  age;
        float  salary;
        date  dob;
        sex_type  sex_info;
        };
```

```
human_being  person1, person2;

person1.sex_info.sex = male;
person1.sex_info.u.beard = FALSE;

person2.sex_info.sex = female;
person2.sex_info.u.children = 4;
```

2.3.3 Internal Implementation of Structures

- In most cases we need not be concerned with exactly how the C compiler will store the fields of structure in memory.
- Generally, the values will be stored in the same way using increasing address location in the order specified in the structure definition.

2.3.4 Self-Referential Structures

- A *self-referential structure* is one in which one or more of its components is a pointer to itself.
- Self-referential structure usually require dynamic storage management routine (*malloc* and *free*) to explicitly obtain and release memory.

```
typedef struct list {
        char  data;
        list  *link;
        };

list  item1, item2, item3;

item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;

item1.link = &item2;
item2.link = &item3;
```

2.4 POLYNOMIALS

2.4.1 The Abstract Data Type

- Arrays are not only data structures in their own right, we can also use them to implement other abstract data types.

- One of the simplest and most commonly found data structures:

  ordered list or *linear list*.

  $$(item_0 , item_1 , \ldots , item_{n-1})$$

- Examples :
  - Days of the week : (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)
  - Values in a deck of cards :
    (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)
  - Floors of the building :
    (basement, lobby, mezzanine, first, second)

  etc.

- Possible operations on the ordered lists :
  - Finding the length, n, of a list.
  - Reading the items in a list from left to right (or right to left).
  - Retrieving the ith item from a list, $0 \le i < n$.
  - Replacing the item in the ith position of a list, $0 \le i < n$.
  - Inserting a new item in the ith position of a list, $0 \le i < n$.
    The items previously numbered $i, i+1, \ldots, n-1$ become items numbered $i+1, i+2, \ldots, n$.
  - Deleting an item from the ith position of a list, $0 \le i < n$.
    The items previously numbered $i+1, \ldots, n$ become items numbered $i, i+1, \ldots, n-1$.

- Implementations (ways to represent an ordered list) :
  - Sequential mapping
  - Nonsequential mapping

- A *polynomial* (viewed from a mathematical perspective) is a sum of terms, where each term has a form ax^e,
  where *x* is a variable, *a* is the coefficient, and *e* is the exponent.

  For example :

  A(X) = $3X^2 + 2X^5 + 4$

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

  Standard mathematical definitions for sum and product of polynomials.

  For A(x) = $\sum a_i x^i$ and B(x) = $\sum b_i x^i$

  A(x) + B(x) = $\sum (a_i + b_i) x^i$

  A(x)·B(x) = $\sum (a_i x^i \bullet (\sum b_i x^i))$

[ADT 2.2] Abstract Data Type *Polynomial*

ADT *Polynomial* is

Objects : $p(x) = a_1 x^{e_1} + \cdots + a_n x^{e_n}$; a set of ordered pairs of $<a_i, e_i>$
where a_i in *Coefficients* and e_i in *Exponents*, are integers >=0.

Functions :

for all *poly*, *poly*1, *poly*2 $\in$ *Polynomial*, *coef* $\in$ *Coefficients*, *expon* $\in$ *Exponents*

Polynomial Zero()	::= **return** the polynomial $p(x)=0$
Boolean IsZero(*poly*)	::= **if** (*poly*) **return** FALSE **else return** TRUE
Coefficients Coef(*poly*,*expon*)	::= **if** (*expon* $\in$*poly*) **return** its coefficient **else return** zero
Exponent LeadExp(*poly*)	::= **return** the largest exponen in *poly*.
Polynomial Attach(*poly*,*coef*,*expon*)	::= **if** (*expon* $\in$*poly*) **return** error **else return** the polynomial *poly* with the term <*coef*,*expon*> inserted

Polynomial Remove(*poly*,*expon*) ::= **if** (*expon* $\in$ *poly*)
return the polynomial *poly* with the term whose exponent is *expon* deleted
else return error

Polynomial SingleMult(*poly*,*coef*,*expon*) ::= **return** the polynomial $poly \cdot coef \cdot x^{expon}$

Polynomial Add(*poly1*,*poly2*) ::= **return** the polynomial *poly1* + *poly2*

Polynomial Mult(*poly1*,*poly2*) ::= **return** the polynomial *poly1* · *poly2*

end *Polynomial*

2.4.2 Polynomial Representation

- **[Program 2.5] Initial version of padd function**

```
/* d = a + b, where a, b, and d are polynomials */
d = Zero();
While(!IsZero(a) && ! IsZero(b)) do {
    switch COMPARE(Lead_Exp(a), Lead_Exp(b)) {
      case  -1 :    d = Attach(d, Coef(b, Lead_Exp(b)), Lead_Exp(b));
                    b = Remove(b, Lead_Exp(b));
                    break;
      case   0 :    sum = Coef(a, Lead_Exp(a)) + Coef(b, Lead_Exp(b));
                    if (sum) {
                        Attach(d, sum, Lead_Exp(a));
                        a = Remove(a, Lead_Exp(a));
                        b = Remove(b, Lead_Exp(b));
                    }
                    break;
      case   1 :    d = Attach(d, Coef(a, Lead_Exp(a)), Lead_Exp(a));
                    a = Remove(a, Lead_Exp(a));
    }
}
insert  any remaining terms of a or b into d
```

- Representation
  Exponents are uniquely arranged in decreasing order.

<Dense Representation>

Include all the terms in a polynomial :

A(x) = $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$, where $a_n \neq 0$.

```
#define MAX_DEGREE 101
typedef  struct  {
    int  degree;
    float  coef[MAX_DEGREE];
    }  polynomial;
```

Let *a* be a variable of type polynomial.

We can represent the polynomial A(x) = $\sum a_i x^i$ in *a*,

by setting a.degree = *n* and a.coef[i] = a_{n-i} , 0≤*i*≤*n*.

- Although this representation leads to very simple algorithms for most of the operations, it wastes a lot of space.

- For instance, if a.degree << MAX_DEGREE
  or if the polynomial is sparse.

Examples: $A(x) = 2x^{1000} + 1$ and

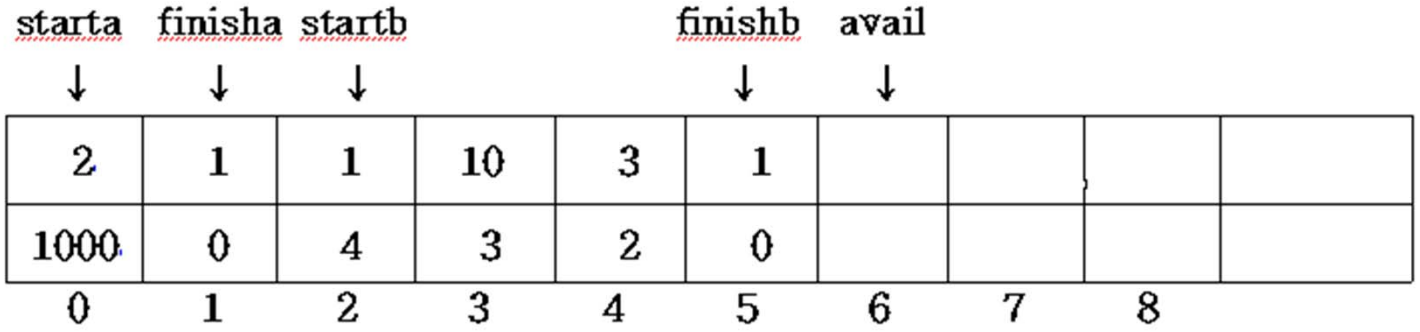
$$B(x) = x^4 + 10x^3 + 3x^2 + 1$$

- **<Sparse Representation>**

  To preserve space, we use only one global array to store all our polynomials.

```
#define  MAX_TERMS  100
typedef  struct {
        float  coef;
        int  expon;
        } polynomial;
polynomial  terms[MAX_TERMS];
int  avail = 0;
```

- **[Figure 2.2] : Array representation of two polynomials**



Examples : $A(x) = 2x^{1000} + 1, B(x) = x^4 + 10x^3 + 3x^2 + 1$

To represent a zero polynomial *c*, set *startc* > *finishc*.

2.4.3 Polynomial Addition

- **[Program 2.6] : Function to add two polynomials**

```
void padd(int starta, int finisha, int startb, int finishb, int *startd, int *finishd)
{
    /* add A(x) and B(x) to obtain D(x) */
    float coefficient;
    *startd = avail;
    while (starta <= finisha && startb <= finishb)
        switch (COMPARE(terms[starta].expon, terms[startb].expon)) {

            case -1 :  /* a expon < b expon */
                attach(terms[startb].coef, terms[startb].expon);
                startb++;
                break;
```

```
            case  0 :  /* equal exponents */
                        coefficient = terms[starta].coef + terms[startb].coef;
                        if (coefficient)
                                    attach(coefficient, terms[starta].expon);
                        starta++;     startb++;
                        break;
            case  1 :   /* a expon > b expon */
                        attach(terms[starta].coef, terms[starta].expon);
                        starta++;
      }
   /* add in remaining terms of A(x) */
   for (; starta <= finisha; starta++)
            attach(terms[starta].coef, terms[starta].expon);
   /* add in remaining terms of B(x) */
   for (; startb <= finishb; startb++)
            attach(terms[startb].coef, terms[startb].expon);
   *finishd =avail-1;
}
```

[Program 2.7] : Function to add a new term

```
void attach(float coefficient, int exponent)
{
    /*  add a new term to the polynomial  */
    if (avail >= MAX_TERMS)  {
            fprintf(stderr, "Too many terms in the polynomial");
            exit(1);
    }
    terms[avail].coef = coefficient;
    terms[avail++].expon = exponent;
}
```

- Analysis of *padd* :

  Time complexity is $O(n+m)$, where *m* and *n* are the number of terms in A and B, respectively.

  When *avail* > MAX_TERMS, must we quit?

2.5 THE SPARSE MATRIX

2.5.1 The Abstract Data Type

- A *sparse matrix* is a matrix which contains many zero entries.
- If a two-dimensional array is used to represent a sparse matrix, a lot of space is used to store the same value 0 and this implementation does not work when the matrices are large since most compilers impose limits on array sizes.

[Figure 2.3]

	col 0	col 1	col 2
row 0	-27	3	4
row 1	6	82	-2
row 2	109	-64	11
row 3	12	8	9
row 4	48	27	47

(a)

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

(b)

[ADT 2.3] ADT *Sparse Matrix*

ADT *Sparse_Matrix* is

objects : a set of triples, <*row*, *column*, *value*>, where *row* and *column* are integers and from a unique combination, and value comes from the set *item*.

functions :

for all $a, b \in Sparse\_Matrix, x \in item, i, j, max\_col, max\_row \in index$

Sparse_Matrix Create(*max_row*, *max_col*) ::=

return a *Sparse_Matrix* that can hold up to *max_items* = *max_row* × *max_col* and whose maximum row size is *max_row* and whose maximum column size is *max_col*.

Sparse_Matrix Transpose(*a*) ::=

return the matrix produced by interchanging the row and column value of every triple.

Sparse_Matrix Add(*a*, *b*) ::=

if the dimension of *a* and *b* are the same

return the matrix produced by adding corresponding items, namely those with identical *row* and *column* values.

else return error.

Sparse_Matrix Multiply(*a*, *b*) ::=

if number of columns in *a* equals number of rows in *b*

return the matrix d produced by multiplying *a* by *b* according to the formula : $d(i, j) = \sum a(i, k) \cdot b(k, j)$, where d(i,j) is the (i,j)th element

else return error.

end *Sparse_matrix*

2.5.2 Sparse Matrix Representation

- We can characterize uniquely any element within a matrix
  by a triple <*row, col, value*>.
  Thus we can use an array of triples.

- We organize the triples so that row indices are in ascending order and among those with the same row indices are ordered in ascending order of column indices.

- To insure that the operations terminate,
  we must know the number of rows and columns,
  and the number of nonzero elements in the matrix.

Sparse_Matrix Create(*max_row*, *max_col*) ::=

```
#define  Max_TERMS 101  /* maximum number of terms +1*/
typedef  struct  {
            int  col;
            int  row;
            int  value;
            } term;
term  a[MAX_TERMS];
```

[Figure 2.5] For example,

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

(a)

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

(b)

2.5.3 Transposing A Matrix

- < A simple algorithm >

```
for each row i
        take element <i, j, value> and store it
        as element <j, i, value> of the transpose;
```

We will not know exactly where to place element <j, i, value> in the transpose until we have processed all the elements that precede it.

For instance,

(0, 0, 15) becomes (0, 0, 15)
(0, 3, 22) becomes (3, 0, 22)
(0, 5,-15) becomes (5, 0,-15)

Consecutive insertions are required.
We must move elements to maintain the correct order.

- We can avoid this data movement by using the column indices to determine the placement of elements in the transpose matrix.

  for all elements in column j
  　　place element <i, j, value> in element <j, i, value>;

- **[Program 2.8]**

```
void transpose (term a[], term b[])
/*  b is set to the transpose of a  */
{
     int n, i, j, currentb;
     n = a[0].value;              /*  total number of elements  */
     b[0].row = a[0].col;         /*  rows in b = columns in a  */
     b[0].col = a[0].row;         /*  columns in b = rows in a  */
     b[0].value = n;
```

```
if (n > 0)  {              /*  nonzero matrix  */
            currentb = 1;
            for (i=0; i<a[0].col; i++)  /*  transpose by columns in a  */
                for (j=1; j<=n; j++)  /* find elements from the current column */
                        if (a[j].col == i)  {  /*element is in current column, add it to b*/
                                b[currentb].row = a[j].col;
                                b[currentb].col = a[j].row;
                                b[currentb].value = a[j].value;
                                currentb++;
                        }
            }
}
```

Time complexity : O(*columns*·*elements*).

If *elements* = O(*rows*·*columns*), then

O(*columns*·*elements*) becomes O(*rows*·*columns*^{2}).

- A transpose algorithm using dense representation :

```
for (j = 0; j < columns; j++)
        for (i = 0; i < rows; i++)
                b[j][i] = a[i][j];
```

Time complexity : O(*rows*·*columns*).

- <A much better algorithm by using a little more storage>

This algorithm, *fast_transpose*, proceeds by first determining the number of elements in each column of the original matrix.

This number gives the number of elements in each row of the transpose matrix.

[Program 2.9]

```
void fast_transpose(term a[], term b[])
{    /*  the transpose of a is placed in b  */
      int  row_terms[MAX_COL],  starting_pos[MAX_COL];
      int  i, j, num_cols = a[0].col,  num_terms = a[0].value;
      b[0].row = num_cols;  b[0].col = a[0].row;
      b[0].value = num_terms;
      if (num_terms > 0) {      /* nonzero matrix  */
                for (i = 0; i < num_cols; i++)        row_terms[i] = 0;
                for (i = 1; i <= num_terms; i++)   row_terms[a[i].col]++;
                starting_pos[0] = 1;
                for (i = 1; i < num_cols; i++)
                          starting_pos[i] =  starting_pos[i-1] + row_terms[i-1];
                for (i = 1; i <= num_terms; i++) {
                           j = starting_pos[a[i].col]++;
                          b[j].row = a[i].col;    b[j].col = a[i].row;
                          b[j].value = a[i].value;
                }
      }
}
```

- Time complexity : O(*columns + elements*).
  If *elements* = O(*rows·columns*), then
  O(*columns +elements*) becomes O(*rows·columns*).

- Additional arrays, *row_terms* and *starting_pos*, are used.

- We can reduce this space to one array
  if we put the starting positions into the space used by *row_terms*.

2.5.4 Matrix Multiplication

- **Definition** :

  Given two matrices A and B where A is $m \times n$ and B is $n \times p$,
  the product matrix D has dimension $m \times p$. Its $<i, j>$ element is :

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

  for $0 \le i < m$ and $0 \le j < p$.

- <Matrix Multiplication Algorithm using dense representation>

```
for (i = 0; i < rows_a; i++) {
    for (j = 0; j < cols_b; j++) {
        sum = 0;
        for (k = 0; k < cols_a; k++)
            sum += a[i][k]*b[k][j];
        d[i][j] = sum;
    }
}
```

Time Complexity : O(*rows_a*·*cols_a*·*cols_b*)

Note that the product of two sparse matrices may no longer be sparse.

For example : $\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

- <Multiplying two sparse matrices represented as an ordered list>

Need to compute the elements of D by rows so that we can store them in their proper place without moving previously computed elements.

A

rows_a	cols_a	totala
rows_a		

B

rows_b	cols_b	totalb

B^T

cols_b	rows_b	totalb
cols_b	-1	

D

rows_a	cols_b	totald

- **[Program 2.10]**

Matrices A, B, and D are stored in the arrays *a, b*, and *d*, respectively.
Transpose of B is stored in *new_b*.

Variables used :

row - the row of A that we are currently multiplying with the columns in B.

row_begin - the position in *a* of the first element of the current row.

column - the column of B that we are currently multiplying with a row in A.

totald - the current number of elements in the product matrix D.

i, j - pointers which are used to examine successively elements from a row of A and a column B.

```
void mmult(term a[], term b[], term d[])
/*  multiply two sparse matrices  */
{
   int  i, j, column, totalb = b[0].value, totald = 0;
   int  rows_a = a[0].row, cols_a = a[0].col, totala = a[0].value;
   int  cols_b = b[0].col;
   int  row_begin = 1, row = a[1].row,  sum = 0;
   term  new_b[MAX_TERMS];
   if (col_a != b[0].row)  {
      fprintf(stderr, “Incompatible matrices\n”);
      exit(1);
   }
   fast_transpose(b, new_b);
   /* set boundary condition  */
   a[totala+1].row = rows_a;
   new_b[totalb+1].row = cols_b;     new_b[totalb+1].col = -1;
```

```
for (i = 1; i <= totala;  )   {
      column = new_b[1].row;
      for (j = 1; j <= totalb+1;  )   {
      /* multiply row of a by column of b */
            if (a[i].row != row)  {
                  storesum(d, &totald, row, column, &sum);
                  i = row_begin;
                  for (  ; new_b[j].row == column; j++)
                                    ;
                  column = new_b[j].row;
            }
            else  if (new_b[j].row != column)  {
                  storesum(d, &totald, row, column, &sum);
                  i = row_begin;
                  column = new_b[j].row;
            }
```

```
            else  switch  (COMPARE(a[i].col, new_b[j].col))   {
                        case –1 : /* go to next term in a */
                                    i++;  break;
                        case  0 :  /* add terms, go to next term in a and b */
                                    sum  += (a[i++].value * new_b[j++].value);
                                    break;
                        case  1 : /* go to next term in b */
                                    j++;
                    }
            }   /*  end of for j <= totalb+1  */
            for  (  ; a[i].row == row; i++)
                        ;
            row_begin = i; row = a[i].row;
      } /*  end of for i <= totala  */
      d[0].row = row_a;  d[0].col = cols_b;
      d[0].value = totald;
}
```

- Notice that we have introduced an additional term into both *a* and *new_b* :

```
a[totala+1].row = rows_a;
new_b[totalb+1].row = cols_b;
new_b[totalb+1].col = -1;
```

- Time complexity :

lines before the *for* loop:

fast transpose - O(*cols_b + totalb*) time.

the outer *for* loop is iterated *rows_a* times:

at each iteration - one row of the product matrix D is computed

by the inner *for* loop in which at each iteration

either *i* or *j* or both increase by 1,

or *i* is reset to *row_begin*.

The maximum total increment in *j* is *totalb*+1.

Let r_k be the number of terms in row *k*.

Then when row *k* is processed, *i* can increase at most r_k times

and *i* is reset to *row_begin* at most *cols_b* times.

Thus the maximum total increment in *i* is $cols\_b \cdot r_k$.

The inner *for* loop requires $O(cols\_b \cdot r_k + totalb)$ time.

column is reset.

Therefore the outer *for* loop requires

$$\sum_{k=0}^{rows\_a-1} O(cols\_b \cdot r_k + totalb)$$

$$= O(cols\_b \cdot \sum_{k=0}^{rows\_a-1} r_k + rows\_a \cdot totalb)$$

$$= O(cols\_b \cdot totala + rows\_a \cdot totalb).$$

Note that if *totala* = O(*rows_a*·*cols_a*) and *totalb*=O(*rows_b*·*cols_b*),

its complexity becomes O(*rows_a*·*cols_a*·*cols_b*).

2.6 REPRESENTATION OF MULTIDIMENSIONAL ARRAYS

- If an array is declared a[upper$_0$][upper$_1$]$\cdots$[upper$_{n-1}$],

  the number of elements in the array is $\prod_{i=0}^{n-1} upper_i$

- Two common ways to represent multidimensional arrays :
  row major order
  column major order

- <row major order>

  We interpret the two-dimensional array a[$upper_0$][$upper_1$]

  as $upper_0$ rows, row_0, row_1, . . ., $row_{upper0\ -1}$

  each row containing $upper_1$ elements.

  If we assume that α is the address of a[0][0],

  then the address of a[i][j] is $\alpha + i \cdot upper_1 + j$.

  To represent a three-dimensional array a[$upper_0$][$upper_1$][$upper_2$],

  we interpret the array as $upper_0$ two-dimensional arrays

  of dimension $upper_1 \times upper_2$.

  Then the address of a[i][j][k] is

  $\alpha + i \cdot upper_1 \cdot upper_2 + j \cdot upper_2 + k$.

- Generalizing on the preceding discussion, we can obtain the addressing formula for any element $a[i_0][i_1]\cdots[i_{n-1}]$ in an array declared as $a[upper_0][upper_1]\cdots[upper_{n-1}]$.

  If α is the address of $a[0][0]\ldots[0]$, the address of $a[i_0][i_1]\cdots[i_{n-1}]$ is :

$$\begin{aligned}
&\alpha + i_0 \cdot upper_1 \cdot upper_2 \cdots upper_{n-1} \\
&\quad + i_1 \cdot upper_2 \cdot upper_3 \cdots upper_{n-1} \\
&\quad + i_2 \cdot upper_3 \cdot upper_4 \cdots upper_{n-1} \\
&\qquad\quad . \\
&\qquad\quad . \\
&\qquad\quad . \\
&\quad + i_{n-2} \cdot upper_{n-1} \\
&\quad + i_{n-1}
\end{aligned}$$

$$= \alpha + \sum_{j=0}^{n-1} i_j a_j \quad \text{where} \quad a_j = \prod_{k=j+1}^{n-1} upper_k, \quad 0 \le j < n-1, \; a_{n-1} = 1$$

2.7 STRINGS

2.7.1 The Abstract Data Type

A *string* is a finite sequence of zero or more characters,

$S = s_0, \ldots, s_{n-1}$, where s_i are characters.

- **[ADT2.4] Abstract data type *String* :**

ADT *String* is

objects : a finite sequence of zero or more characters.

functions : for all $s, t \in$ *String*, $i, j, m \in$ non-negative integers

String Null(m) ::= **return** a string whose maximum length is m characters,
but is initially set to *NULL*. We write *NULL* as "".

Integer Compare(*s*, *t*) ::= **if** *s* equals *t* **return** 0
else if *s* precedes *t* **return** –1
else return +1.

Boolean IsNull(*s*) ::= **if** (Compare(*s*, *NULL*)) **return** *FALSE*
else return *TRUE*.

Integer Length(*s*) ::= **if** (Compare(*s*, *NULL*))
return the number of characters in *s* **else** return 0.

String Concat(*s*, *t*) ::= **if** (Compare(*t*, *NULL*))
return a string whose elements are
those of *s* followed by those of *t*
else return *s*.

String Substr(*s*,*i*,*j*) ::= **if** ((*j*>0) && (*i*+*j*-*1*)<Length(*s*))
return the string containing the
characters of *s* at positions *i*, *i*+1,…. *i*+*j*-1.
else return *NULL*.

C provides many string operations in its library : see Fig. 2.8 (string.h)

2.7.2 Strings in C

- <Representation>

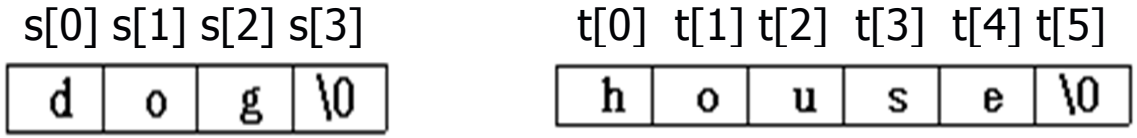
In C, we represent strings as character arrays terminated with the null character .

For instance,

```
#define MAX_SIZE 100
char s[MAX_SIZE] = "dog";
char t[MAX_SIZE] = "house";
```

Internal representation in C :

[Figure 2.8]



- Alternative declaration :

```
char s[] = "dog";
char t[] = "house";
```

Concatenating these two strings by calling *strcat*(s,t) which stores the result in *s*. This produces the new string, "doghouse".

Although *s* has increased in length by five,
we have no additional space in *s* to store the extra five characters.

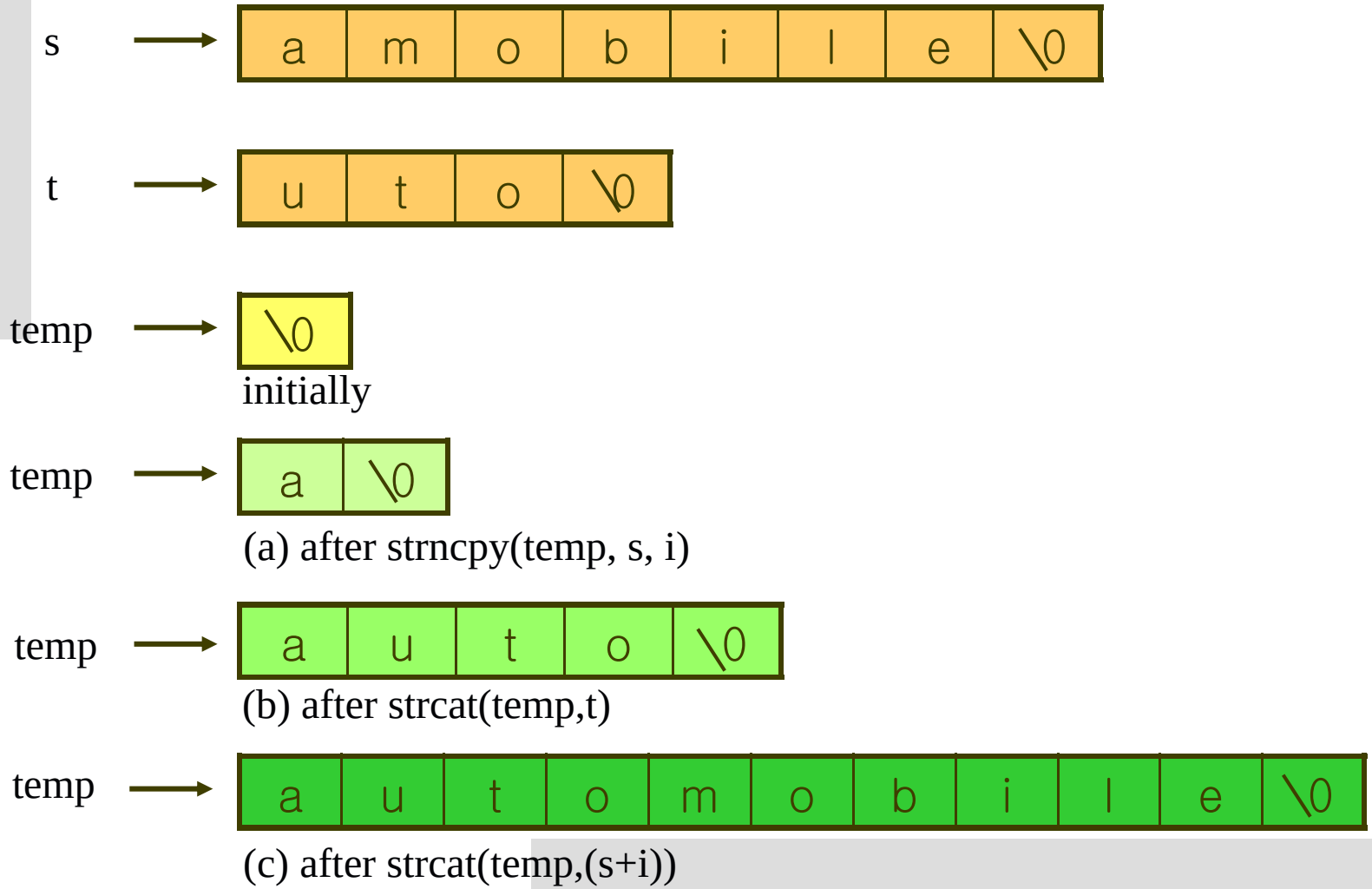
Most of *C* compilers simply *overwrite* the memory
to fit in the extra five characters.

- C provides built-in other string functions which we access through the statement

  #include <string.h>

- **Example 2.2 [String insertion]**

```
# include <string.h>
# define MAX_SIZE 100
char string1 [MAX_SIZE], *s = string1;
char string2 [MAX_SIZE], *t = string2;
```


s
a m o b i l e \0
t
u t o \0
temp
\0
initially
temp
a \0
(a) after strncpy(temp, s, i)
temp
a u t o \0
(b) after strcat(temp,t)
temp
a u t o m o b i l e \0
(c) after strcat(temp,(s+i))

[Program 2.12]

```
void strnins(char *s, char *t, int i)
{ /*  insert string t into string s at position i  */
      char string[MAX_SIZE],  *temp = string;

      if (i<0 && i>strlen(s))  {
         fprint(stderr, "Position is out of bounds ");
         exit(1);
      }
      if (!strlen(s))
          strcpy(s, t);
      else if (strlen(t))  {
          strncpy(temp, s, i);
          strcat(temp, t);
          strcat(temp, (s+i));
          strcpy(s, temp);
      }
}
```

2.7.3 Pattern Matching

```
char pat[MAX_SIZE], string[MAX_SIZE], *t;
```

To determine if *pat* is in *string* :

```
if (t = strstr(string, pat))
    printf("The string from strstr is: %s", t);
else
    printf("The pattern was not found with strstr");
```

The call (*t = strstr(string, pat)*) returns

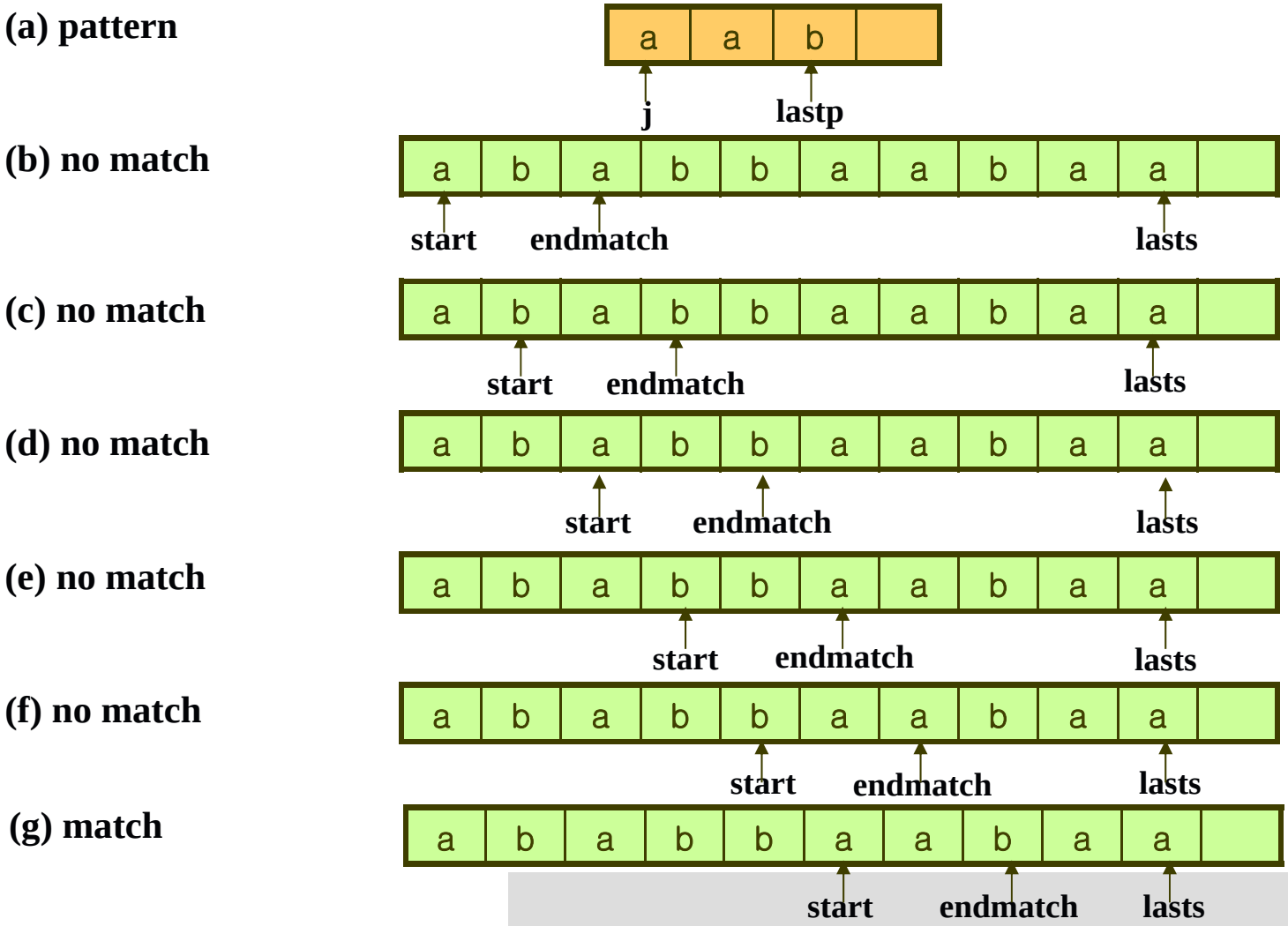
- a null pointer if *pat* is not in *string*.
- a pointer to the start of *pat* in *string* if *pat* is in *string*.

- Reasons of developing our own pattern matching function:
  (1) The function *strstr* may not be available with the compiler we are using.
  (2) There are several different methods
  for implementing a pattern matching function.

- A simple matching algorithm :
  At each position *i* of *string*, check if *pat* == *string*[*i*+*strlen*(*pat*)-1].

- If *pat* is not in *string*, this algorithm has a computing time of O(*nm*),
  where *n* is the length of *pat* and *m* is the length of *string*.

- Improvements :
  1. Quitting when *strlen*(*pat*) is greater than the number of remaining characters in the string.
  2. Checking the first and last characters of pat before we checking the remaining characters.


(a) pattern
a a b
j
lastp
(b) no match
a b a b b a a b a a
start
endmatch
lasts
(c) no match
a b a b b a a b a a
start
endmatch
lasts
(d) no match
a b a b b a a b a a
start
endmatch
lasts
(e) no match
a b a b b a a b a a
start
endmatch
lasts
(f) no match
a b a b b a a b a a
start
endmatch
lasts
(g) match
a b a b b a a b a a
start
endmatch
lasts

[Program 2.13]

```
int nfind(chaqr * string, char *pat)
{
/*  match the last character of the pattern first,  and then match from the beginning   */
     int i, j, start = 0;
     int  lasts = strlen(string)-1;
     int  lastp = strlen(pat)-1;
     int endmatch = lastp;
     for (i = 0; endmatch <= lasts; endmatch++, start++)  {
               if  (string[endmatch] == pat [lastp])
                         for (j = 0, i = start;  j < lastp && string[i] == pat[j]; i++, j++)
                                                   ;
               if  (j == lastp)    return  start;   /*  successful  */
      }
      return  -1;
 }
```

- Analysis of *nfind* :

  For *string* = "*aa…a*" and *pat* = "*aa…ab*", the computing time is O(*m*).

  Bur for *string* = "*aa…a*" and *pat* = "*aa…aba*",

  the computing time is still O(*nm*).

- **<KMP Algorithm>**
  - When a mismatch occurs, use our knowledge of the characters in the pattern and the position in the pattern where the mismatch occurred to determine where we should continue the search.
  - We want to search the string for the pattern without moving backwards in the string

pat = 'a b c a b c a c a b'

p_0 p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9

if $s_i \neq p_0$, ?

if $s_i = p_0$ and $s_{i+1} \neq p_1$, ?

if $s_i = p_0$, $s_{i+1} = p_1$, and $s_{i+2} \neq p_2$, ?

if $s_i = p_0$, $s_{i+1} = p_1$, $s_{i+2} = p_2$, $s_{i+3} = p_3$, and $s_{i+4} \neq p_4$, ?

- **Definition** :

  If $p = p_0 p_1 p_2 \ldots p_{n-1}$ is a pattern, then its *failure function*, f, is defined as :

$$f(j) = \begin{cases} \text{largest } i<j \text{ such that } p_0 p_1 \ldots p_i = p_{j-i} p_{j-i+1} \ldots p_j & \text{if such an } i \geq 0 \text{ exists} \\ -1 & \text{otherwise} \end{cases}$$

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

- A 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.

- Assumed declarations:

```
#include <stdio.h>
#include <string.h>
#define max_string_size 100
#define max_pattern_size 100
int pmatch();
void fail();
int failure[max_pattern_size];
char string[max_string_size];
char pat[max_pattern_size];
```

[Program 2.14]

```
int pmatch(char *string, char *pat)
{
/*  Knuth, Morris, Pratt string matching algorithm  */
    int i = 0, j = 0;
    int lens = strlen(string);
    int lenp = strlen(pat);
    while (i < lens && j < lenp)  {
        if  (string[i] == pat[j])   {
                i++; j++;   }
        else  if  (j == 0)  i++;
        else  j = failure[j-1] + 1;
     }
     return  ( (j == lenp)  ?  (i - lenp)  :  -1);
}
```

- Analysis of *pmatch* :

  The *while* loop is iterated until the end of either the string or the pattern is reached.

  In each iteration, one of the following three actions occurs :

  1) increment i.
  2) increment both i and j.
  3) reset j to failure[j-1]+1

     -- this cannot be done more than j is incremented by the statement j++ as otherwise, j falls off the pattern.

  Note that j cannot be incremented more than *m* = *strlen*(*string*) times.

  Hence the complexity of *pmatch* is O(*m*).

- **Another definition of the failure function:**

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

[program 2.15]

```
void fail(char *pat)
{
/*  compute the pattern's failure function  */
    int i, n = strlen(pat);
    failure[0] = -1;
    for (j = 1; j < n; j++)  {
          i = failure[j-1];
          while ((pat[j] != pat[i+1]) && (i >=  0))
                    i = failure[i];
          if  (pat[j] == pat[i+1])
                    failure[j] = i+1;
          else  failure[j] = -1;
    }
}
```

- Analysis of *fail* :

  In each iteration of the *while* loop, the value of *i* decreases (by the definition of *f*).

  The variable *i* is reset at the beginning of each iteration of the *for* loop.

  However, it is either reset to –1

  or it is reset to a value 1 greater than

  its terminal value on the previous iteration.

  Since the *for* loop is iterated only *n*-1 times,

  the value of *i* has a total increment of at most *n*-1.

  Hence it cannot be decremented more than *n*-1 times.

  Consequently, the *while* loop is iterated at most *n*-1 times over the whole algorithm

  Hence the complexity of *fail* is O(*n*) = O(*strlen*(*pat*))

Chapter 3 : STACKS AND QUEUES

Data Structures Lecture Note
Prof. Jihoon Yang
Data Mining Research Laboratory

3.1 STACKS

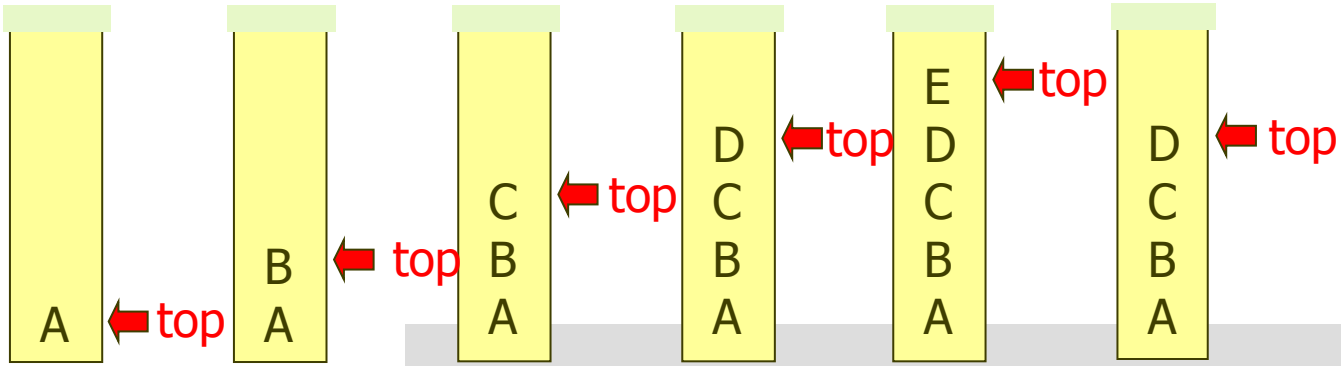
- The *stacks* and *queues* are special cases of the more general data type, *ordered list*.

- A *stack* is an ordered list in which insertions and deletions are made at one end called the *top*.

$$S = (a_0, a_1, \ldots, a_{n-1})$$

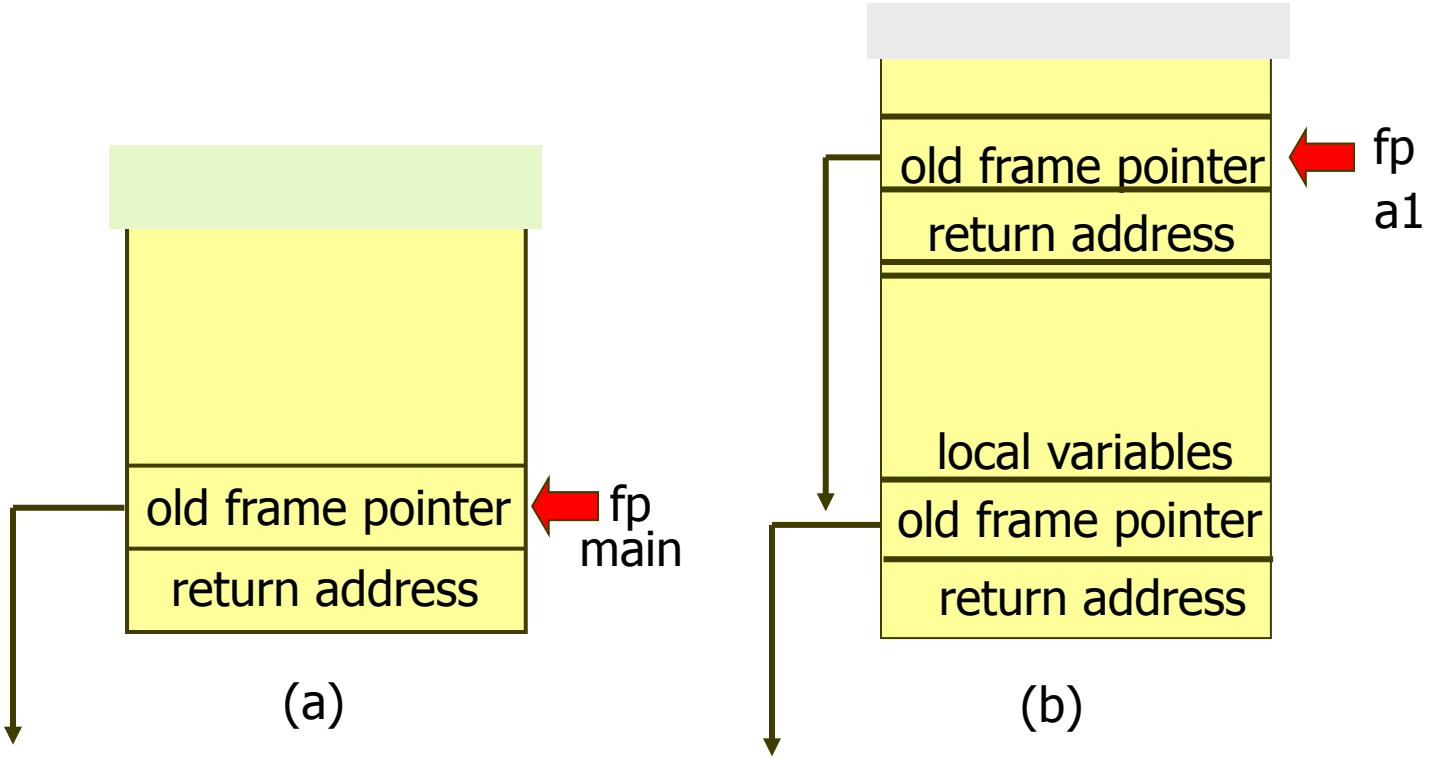
bottom **top**

***Last-In-First-Out (LIFO)* list.**



- **Example 3.1 [System stack] :**

  To process function calls, whenever a function is invoked, the program creates a structure, referred to as *activation record* or *stack frame*



[ADT 3.1]: Abstract data type Stack

ADT *Stack* is

objects : a finite ordered list with zero or more elements.

functions : for all *stack* ∈ *Stack*, *item* ∈ *element*, *maxStackSize* ∈ positive integer

Stack CreateS(*maxStackSize*) ::=

create an empty stack whose maximum size is *maxStackSize*

Boolean IsFull(*stack, maxStackSize*) ::=

If (number of elements in *stack*==*maxStackSize*) **return** *TRUE*

else return *FALSE*

Stack Push(*stack, item*) ::=

if (IsFull(*stack*)) *stackFull*

else insert *item* into top of *stack* and **return**

Boolean IsEmpty(*stack*) ::=

if (*stack*==Create(*maxStackSize*)) **return** *TRUE*

else return *FALSE*

Element Pop(*stack*) ::=

if (IsEmpty(*stack*)) **return**

else remove and return the *item* on the top of the stack.

<Implementation of *Stack*>

Using a one-dimensional array.

Stack CreateS(*maxStackSize*) ::=

```
#define  MAX_STACK_SIZE 100   /* maximum stack size */
typedef  struct  {
        int  key;
        /*  other fields  */
} element;
element  stack[MAX_STACK_SIZE];
int  top = -1;
```

Boolean IsEmpty(*stack*) ::= top < 0;

Boolean IsFull(*stack*) ::= top >= MAX_STACK_SIZE-1;

- **[Program 3.1] (Add to a stack)**

```
void push(element item)
{
/* add an item to the global stack */
     if (top >= MAX_STACK_SIZE-1)
          stackFull();
     stack[++top] = item;
}
```

- **[Program 3.2] (Delete from a stack)**

```
element pop()
{
/* return the top element from the stack  */
     if (top == -1)
          return stackEmpty();      /* return an error key */
     return stack[top--];
}
```

3.2 STACKS USING DYNAMIC ARRAYS

```
Stack CreateS() ::=
    typedef  struct  {
        int  key;
        /*  other fields  */
    } element;
    element  *stack;
    MALLOC(stack, sizeof(*stack));
    int capacity = 1;
    int top = -1;

Boolean IsEmpty(stack) ::= top < 0;
Boolean IsFull(stack) ::= top >= capacity-1;


void stackFull()
{
    REALLOC(stack, 2*capacity*sizeof(*stack));
    capacity *= 2;
}
```

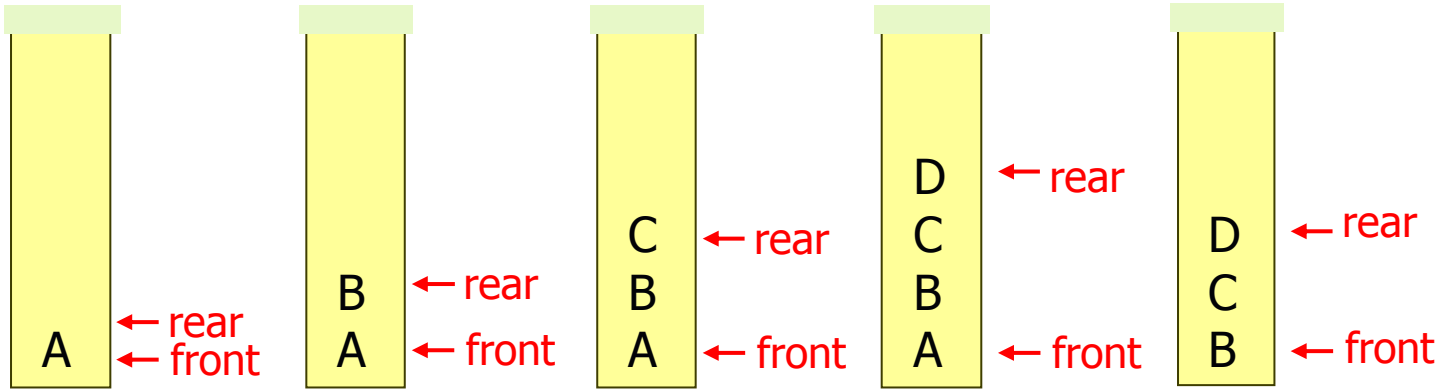
3.3 QUEUES

- A *queue* is an ordered list in which all insertions take place at one end called the *rear* and all deletions take place at the opposite end called the *front*.

$$Q = (a_0, a_1, \ldots, a_{n-1})$$

front rear

First-In-First-Out (FIFO) list.



[ADT 3.2]: Abstract data type Queue

ADT *Queue* is

objects : a finite ordered list with zero or more elements.

functions : for all *queue* ∈ *Queue*, *item* ∈ *element*, *maxQueueSize* ∈ positive integer

Queue CreateQ(*maxQueueSize*) ::=

create an empty queue whose maximum size is *maxQueueSize*

Boolean IsFullQ(*queue*, *maxQueueSize*) ::=

if (number of elements in *queue*==*maxQueueSize*) **return** *TRUE*

else return *FALSE*

queue AddQ(*queue*, *item*) ::=

if (IsFullQ(*queue*)) *queueFull*

else insert *item* at rear of *queue* and **return**

Boolean IsEmptyQ(*queue*) ::=

if (*queue* == CreateQ(*maxQueueSize*)) **return** *TRUE*

else return *FALSE*

Element DeleteQ(*queue*) ::=

if (IsEmptyQ(*queue*)) **return**

else remove and return the *item* at front of queue.

- **<Implementation of *Queue*>**

  The simplest scheme employs a one-dimensional array and two variables, *front* and *rear*.

  The *front* points the position in front of the first element and the *rear* points the position of the last element.

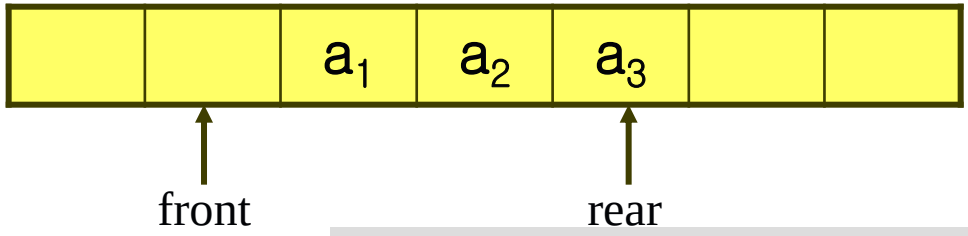
  Thus, we can use a simple condition front == rear to check if the queue is empty.

Queue CreateQ(*maxQueueSize*) ::=

```
#define  MAX_QUEUE_SIZE 100   /* maximum queue size */
typedef  struct  {
        int  key;
        /*  other fields  */
        } element;
element  queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = -1;
```

Boolean IsEmptyQ(*queue*) ::= front == rear;

Boolean IsFullQ(*queue*) ::= rear == MAX_QUEUE_SIZE-1;



- **[Program 3.5] Add to a queue**

```
void addq(element item)
{
/* add an item to the queue */
      if (rear == MAX_QUEUE_SIZE-1)
            queueFull();
      queue[++rear] = item;
}
```

- **[Program 3.6] Delete from a queue**

```
element deleteq()
{
/* remove element at the front of the queue  */
      if (front == rear)
            return queueEmpty();  /* return an error key */
      return queue[++front];
}
```

- **Example 3.2 [Job scheduling] :**

A job queue by an operating system.

If the operating system does not use priorities, then the jobs are processed in the order they enter the system.

- **[Figure 3.5] : Insertion and deletion from a sequential queue**

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job1 is added
-1	1	J1	J2			Job2 is added
-1	2	J1	J2	J3		Job3 is added
0	2		J2	J3		Job1 is deleted
1	2			J3		Job2 is deleted

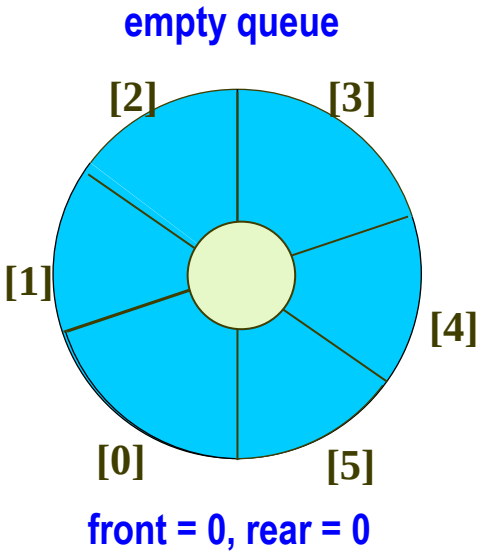
- **Handling of *queueFull* :**
  - As jobs enter and leave the system, the queue gradually shift to the right.
  - Eventually the rear index equals MAX_QUEUE_SIZE - 1, suggesting that the queue is full.
  - In this case, *queueFull* should move the entire queue to the left so that the first element is again at *queue*[0] and *front* is at -1.
  - The *rear* is also recalculated.
  - Shifting an array is very time-consuming.

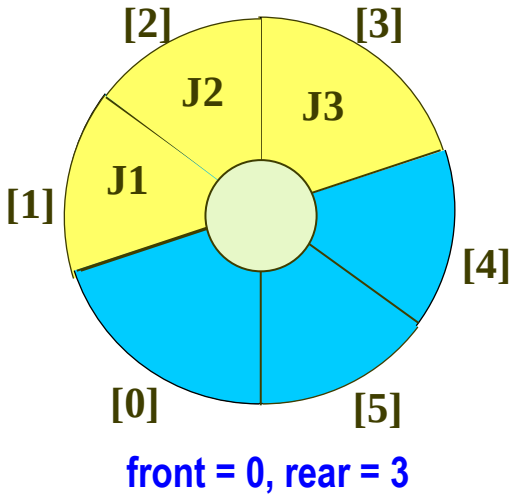
- **A more efficient implementation :**

  By regarding the array *queue*[MAX_QUEUE_SIZE] as **circular**.

  The *front* index always points one position counterclockwise from the first element in the queue.

  The *rear* index points to the current end of the queue.

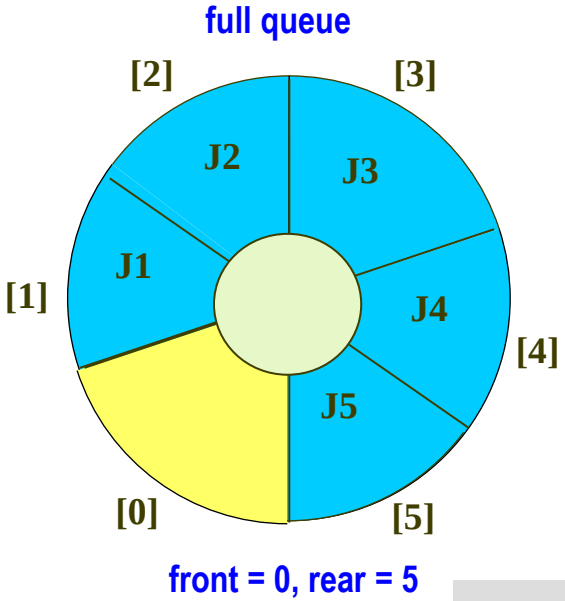


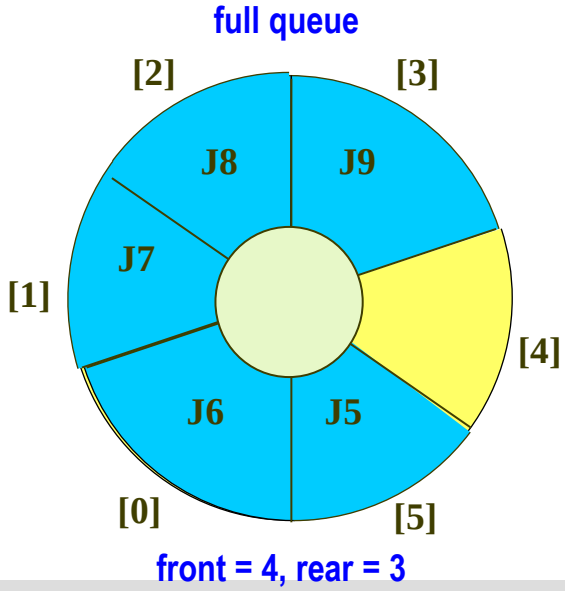


- The queue is empty iff *front == rear*.

- **When is the queue full?**

To distinguish between an empty and a full queue, we adapt the convention that a circular queue of size MAX_QUEUE_SIZE will be permitted to hold at most MAX_QUEUE_SIZE-1 elements.





- To implement *addq* and *deleteq* for a circular queue, we must assure that a circular rotation occurs.

- Using the modulus operator :

  rear = (rear + 1) % MAX_QUEUE_SIZE;

  front = (front + 1) % MAX_QUEUE_SIZE;

- **[Program 3.7] : Add to a circular queue**

```
void addq(element item)
{    /* add an item to the queue */
     rear = (rear + 1) % MAX_QUEUE_SIZE;
     if (front == rear)
          queueFull();  /* print error and exit */
     queue[rear] = item;
}
```

- **[Program 3.8] : Delete from a circular queue**

```
element deleteq()
{   /* remove element at the front of the queue  */
    if (front == rear)
        return queueEmpty();  /* return an error key */
    front = (front + 1) % MAX_QUEUE_SIZE;
    return queue[front];
}
```

3.4 CIRCULAR QUEUES USING DYNAMICALLY ALLOCATED ARRAYS

- To add an element to a full queue, we must first increase the size of this array using a function such as *realloc*.
- As with dynamically allocated stacks, we use array doubling.

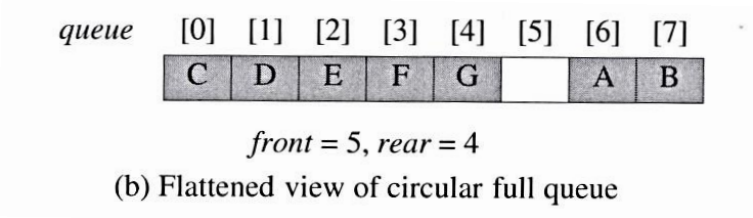
- **< Doubling queue >**

Let *capacity* be the number of positions in the array *queue* .

(1) Create a new array *newQueue* of twice the *capacity*.

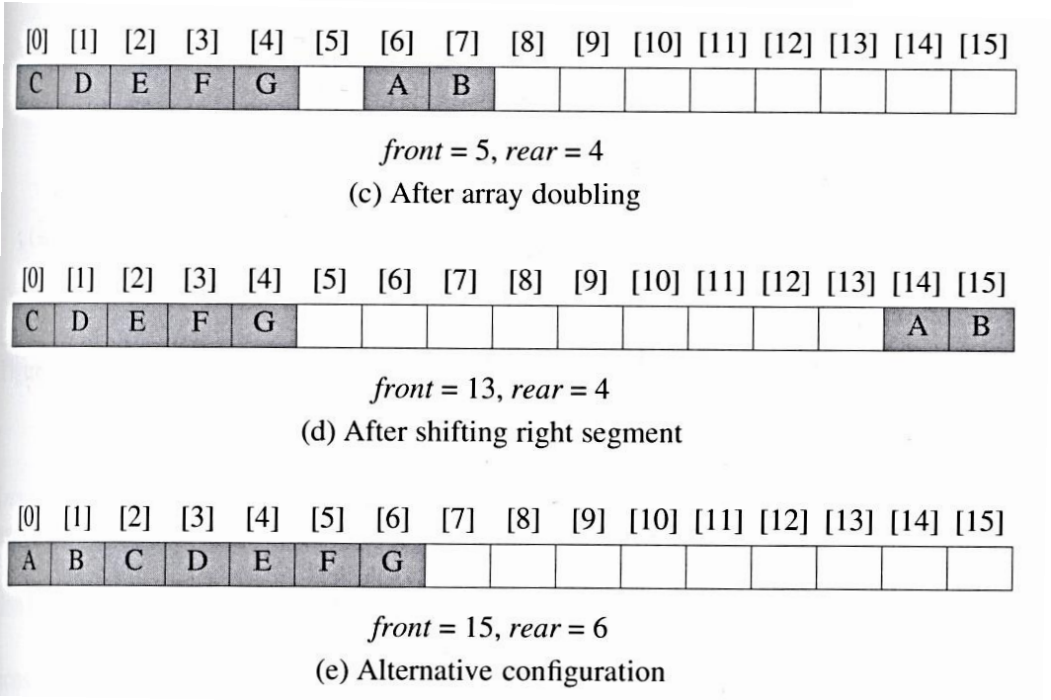
(2) Copy the second segment to positions in *newQueue* beginning at 0.

(3) Copy the first segment to positions in *newQueue* beginning at *capacity-front-1*.



front = 5, *rear* = 4

(b) Flattened view of circular full queue



front = 5, *rear* = 4

(c) After array doubling

front = 13, *rear* = 4

(d) After shifting right segment

front = 15, *rear* = 6

(e) Alternative configuration

[Program 3.9] : Add to a circular queue

```
void addq(element item)
{ /* add an item to the queue */
     rear = (rear + 1) % capacity;
     if (front == rear)
          queueFull(); /* double capacity */
     queue[rear] = item;
}
```

[Program 3.10] : Doubling queue capacity

```
void queueFull()
{
    /* allcoate an array with twice the capacity */
    element* newQueue;
    MALLOC(newQueue, 2 * capacity * sizeof(*queue));

    /* copy from queue to newQueue */
    int start = (front +1) % capacity;
    if (start < 2)
        /* no wrap around */
    else
    { /* queue wraps around */
        copy(queue+start, queue+capacity, newQueue);
        copy(queue, queue+rear+1, newQueue+capacity-start);
    }

    /* switch to newQueue */
    front = 2 * capacity -1;
    rear = capacity – 2;
    capacity *= 2;
    free(queue)
    queue = newQueue;
}
```

copy(*a*,*b*,*c*): copy elements from locations *a* thru *b*-1 to locations beginning at *c*

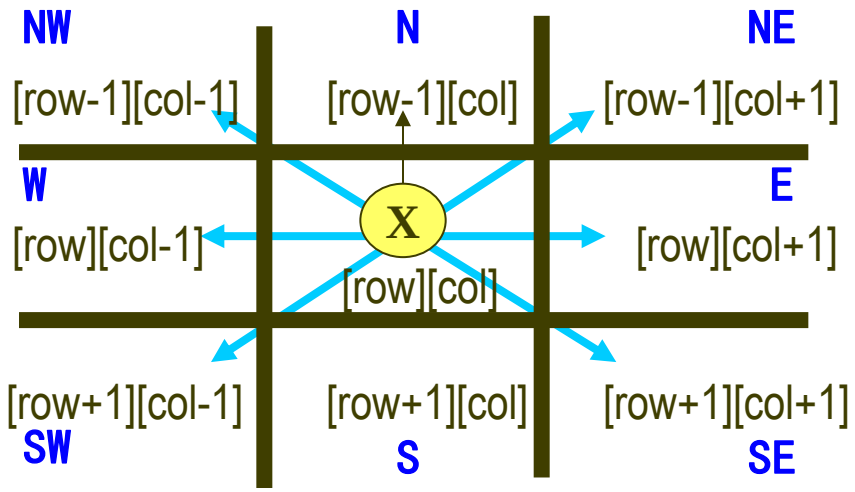
3.5 A MAZING PROBLEM

Representation of a maze :

- Two-dimensional array in which 0's represent the open paths and 1's the barriers.

- To avoid checking for the border conditions we surround the maze by a border of 1's. Thus an m×p maze will require an (m+2)×(p+2) array.
- The entrance is at position [1][1] and the exit at [m][p].

- Predefine the possible directions to move in an array, *move*.




```
typedef struct {
    short int vert;
    short int horiz;
    }  offsets;

offsets  move[8];    /*array of moves for each direction*/
```

- **[Figure 3.10]: Table of moves**

Name	Dir	move[dir].vert	move[dir].horiz
N	0	-1	0
NE	1	-1	1
E	2	0	1
SE	3	1	1
S	4	1	0
SW	5	1	-1
W	6	0	-1
NW	7	-1	-1

- If we are at position, *maze*[*row*][*col*], we can find the position of the next move, *maze*[*nextRow*][*nextCol*], by setting

  nextRow = row + move[dir].vert;

  nextCol = col + move[dir].horiz;

- To record the maze positions already checked, we maintain a second two-dimensional array, *mark*.

- A stack is used to store the positions on the path from the entrance to the current position.

[Program 3.11] : Initial maze program

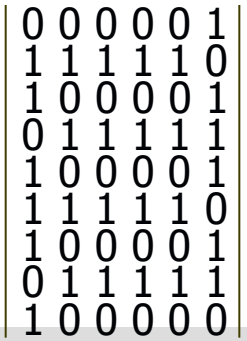
```
initialize a stack to the maze's entrance coordinates and direction to north;
while (stack is not empty)  {
        /* move to position at the top of stack  */
        <row,col,dir> = delete from top of stack;
        while (there are more moves from current position)  {
                <nextRow, nextCol> = coordinates of next move;
                dir = direction of move;
                if ((nextRow==EXIT_ROW) && (nextCol==EXIT_COL))
                        success;
                if ((maze[nextRow][nextCol]==0) &&(mark[nextRow][nextCol]==0)) {
                        /* legal move and haven't been there */
                        mark[nextRow][nextCol] = 1;
                        /* save current position and direction */
                        add <row,col,dir> to the top of the stack;
                        row = nextRow;  col = nextCol;  dir = north;
                }
        }
}
printf("No path found");
```

- **Defining a stack:**

```
#define MAX_STACK_SIZE 100
typedef struct  {
        short int row;
        short int col;
        short int dir;
        }  element;
element stack[MAX_STACK_SIZE];
```

- Need to determine a reasonable bound for the stack size.

[Figure 3.11] Simple maze with a long path



- **[Program 3.12] : Maze search function**

we assumed that arrays, *maze, mark, move, stack*, and constants *EXIT_ROW, EXIT_COL, TRUE, FALSE*, and variable, *top*, are declared as global.

```
void path(void)
{
 /*  output a path through the maze if such a path exists  */
      int i, row, col, nextRow, nextCol, dir, found = FALSE;
      element position;
      mark[1][1] = 1;  top = 0;
      stack[0].row = 1; stack[0].col = 1; stack[0].dir = 1;
      while  (top > -1 && !found)  {
            position = pop();
            row = position.row;  col = position.col, dir = position.dir;
            while  (dir < 8 && !found)  {
                  /*  move in direction dir  */
                  nextRow = row + move[dir].vert;
                  nextCol = col + move[dir].horiz;
                  if  (nextRow == EXIT_ROW && nextCol == EXIT_COL)
                        found = TRUE;
```

```
            else  if  (!maze[nextRow][nextCol] && !mark [nextRow][nextCol])  {
                    mark [nextRow][nextCol] = 1;
                    position.row = row;  position.col = col;
                    position.dir = ++dir;
                    push(position);
                    row = nextRow;  col = nextCol;  dir = 0;
            }
            else  ++dir;
          }
      }
      if  (found)  {
            printf("The path is : \n");
            printf("row  col \n");
            for  (i = 0; i <= top; i++)
                  printf("%2d%5d", stack[i].row, stack[i].col);
            printf("%2d%5d\n", row, col);
            printf("%2d%5d\n", EXIT_ROW, EXIT_COL);
      }
      else  printf("The maze does not have a path \n");
}
```

- Analysis of *path* :

The size of the maze determines the computing time of *path*.

Since each position within the maze is visited no more than once, the worst case time complexity of the algorithm is O(*mp*) where *m* and *p* are, respectively, the number of rows and columns of the maze.

3.6 EVALUATION OF EXPRESSIONS

3.6.1 Expressions

- The representation and evaluation of expressions is of great interest to computer scientists.
- For examples :

((*rear*+1 == *front*) || ((*rear* == MAX_QUEUE_SIZE-1) && ! *front*))

x = a/b-c+d*e-a*c

Tokens in expressions :

operators,
operands, and
parentheses

- For x = a/b-c+d*e-a*c, when a=4, b=c=2, d=e=3, what is the value of x?

  ((4 / 2) - 2) + (3 * 3) - (4 * 2)
  = 0 + 9 - 8
  = 1

  or

  (4 / (2 - 2 + 3)) * (3 - 4) * 2
  = 4 / 3 * (-1) * 2
  = -2.666…

- If we wanted the second answer, we would have written it as follows,
  x = (a / (b - c + d)) * (e - a) * c.

- Within any programming language, there is a precedence hierarchy that determines the order in which we evaluate operators.
- Parentheses are used to override precedence, and the expressions are always evaluated from the innermost parenthesized expression first.
- See Figure 3.12 for precedence hierarchy for C

cf. associativity

- **Ways of writing expressions :**

Infix notation - a * b
Prefix notation - * a b
Postfix notation - a b *

〉 parenthesis-free notations

Although *infix* notation is the most common ways of writing expressions, it is not the one used by compilers to evaluate expressions.

Compilers typically use a parentheses-free notation referred to as *postfix*

Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	1 2+7*
a*b/c	ab*c/
((a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*+ac*-

[Figure 3.13] Infix and postfix notation

3.6.2 Evaluating Postfix Expressions

- To evaluate a postfix expression, we make a single left-to-right scan of it.

  1) Place the operands on a stack until we find an operator.
  2) Remove, from the stack, the correct number of operands for the operator.
  3) Perform the operation
  4) Place the result back on the stack.

- To simplify our task, we assume that the expression contains only the binary operators

  +, -, *, /, and %

  and that the operands in the expression are single digit integers.

- **The complete declarations :**

```
#define MAX_STACK_SIZE 100 /*maximum stack size*/
#define MAX_EXPR_SIZE 100    /*max size of expression*/
typedef enum {lparen, rparen, plus, minus, times, divide, mod, eos, operand} precedence;
int stack[MAX_STACK_SIZE];    /* global stack */
char expr[MAX_EXPR_SIZE];     /* input string */
```

Token	Stack			Top
	[0]	[1]	[2]	
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
−	6/2−3			0
4	6/2−3	4		1
2	6/2−3	4	2	2
*	6/2−3	4*2		1
+	6/2−3+4*2			0

[Figure 3.14] Postfix evaluation

- **[Program 3.13] : Function to evaluate a postfix expression**

```
int eval(void)
{
/*  evaluate a postfix expression, expr, maintained as a global  variable.  '\0' is the end of the
      expression. The stack and  top of the stack are global variables.  getToken is used to
      return the tokentype and the character symbol.  Operands are assumed to be single
      character digits  */
      precedence  token;
      char  symbol;
      int  op1, op2;
      int  n = 0;  /*  counter for the expression string  */
      int  top = -1;
      token = getToken(&symbol, &n);
```

```
    while (token != eos)     {
        if (token == operand)
            push(symbol-'0');    /*  stack insert  */
        else    {
        /* remove two operands, perform operation, and return result to the stack   */
            op2 = pop();    /*  stack delete  */
            op1 = pop();
            switch (token)   {
                case plus : push(op1+op2);      break;
                case minus : push(op1-op2);     break;
                case times : push(op1*op2);     break;
                case divide: push(op1/op2);     break;
                case mod : push(op1%op2);
            }
        }
        token = getToken(&symbol, &n);
    }
    return  pop(&top);     /*  return result  */
}
```

- **[Program 3.14] : Function to get a token from the input string**

```
precedence getToken(char *symbol, int *n)
{
/*  get the next token, symbol is the character representation, which is returned, the
token is represented by its enumerated value, which is returned in the function name  */
     *symbol = expr[(*n)++];
     switch  (*symbol)   {
          case '(' : return  lparen;
          case ')' : return  rparen;
          case '+' : return  plus;
          case '-' : return  minus;
          case '/' : return  divide;
          case '*' : return  times;
          case '%' : return  mod;
          case ' ' : return eos;
          default : return  operand;  /* no error checking, default is operand */
      }
}
```

3.6.3 Infix To Postfix

- An algorithm for producing a postfix expression from an infix one :
  1) Fully parenthesize the expression.
  2) Move all binary operators so that
     they replace their corresponding right parenthesis.
  3) Delete all parentheses.

- For example, a/b-c+d*e-a*c
  After step 1,
  ((((a/b)-c)+(d*e))-(a*c))
  Performing step 2 and 3,
  ab/c-de*+ac*-

- Although this algorithm works well when done by hand,
  it is inefficient on a computer because it requires two passes.

- Note that the order of operands is the same in infix and postfix.
- Thus we can form the postfix equivalent by scanning the infix expression left-to-right.

  However, the order in which the operators are output depends on their precedence.
- Since we must output the higher precedence operators first, we save operators until we know their correct placement.

  A stack is one way of doing this, but removing operators correctly is problematic.

- Method:

  Operators with higher precedence must be output *before* those with lower precedence. Therefore, we stack operators as long as the precedence of the operator at the top of the stack is *less than* the precedence of the incoming operator.

- Parenthesized expression:

  Stack "(" and operators till we reach ")". At this point we unstack till we reach the corresponding "(". Then delete ")".

- **[Figure 3.15] Translation of a+b*c to postfix**

Token	Stack [0]	[1]	[2]	Top	Output
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*+

- **[Figure 3.16] Translation of a*(b+c)/d to postfix**

Token	Stack [0]	[1]	[2]	Top	Output
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
c	*	(	+	2	abc
)	*			0	abc+
/	/			0	abc+*
d	/			0	abc+*d
eos	/			0	abc+*d/

- The analysis of the two examples suggests a precedence-based scheme for stacking and unstacking operators.

- We need two types of precedence, an *in-stack precedence* (*isp*) and an *incoming precedence* (*icp*).
  → Remove an operator from the stack only if its *isp >= icp* of the new operator.

```
precedence stack[MAX_STACK_SIZE];
 /* isp and icp arrays -- index is value of precedence
 lparen, rparen, plus, minus, times, divide, mod, eos  */
static int isp[] = {0, 19, 12, 12, 13, 13, 13, 0};
static int icp[] = {20, 19, 12, 12, 13, 13, 13, 0};
```

[program 3.15] : Function to convert from infix to postfix

```
void postfix(void)
{
/*  output the postfix of the expression. The expression string, stack, and the top are global  */
     char symbol;
     int n = 0;
     int top = 0;  /*  place eos on stack  */
     stack[0] = eos;
     for  (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n))  {
          if (token == operand)
               printf("%c", symbol);
          else  (token == rparan)  {
               /*  unstack tokens until left paranthesis  */
               while (stack[top] != lparen)
                    printToken(pop(&top));
               pop();  /* discard the left paranthesis  */
          }
```

```
        else  {
            /*  remove and print symbols whose isp is greater
                than or equal to the current token's icp  */
            while (isp[stack[top]] >= icp[token])
                printToken(pop());
            push(token);
        }
    }
    while ( (token = pop()) != eos)
        printToken(token);
    printf("\n");
}
```

- **Analysis of *postfix* :**

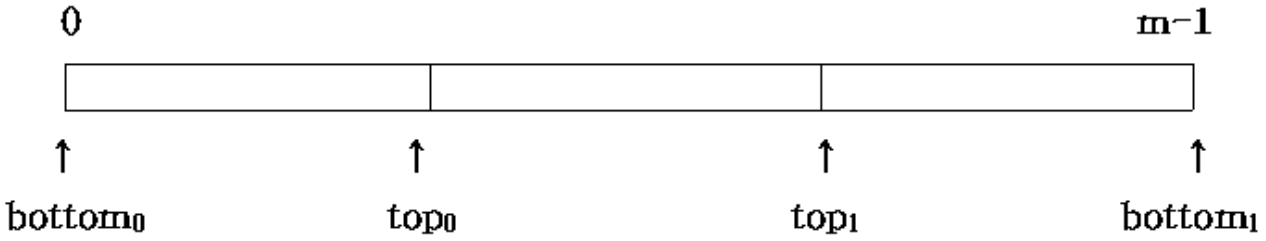
  Let *n* be the number of tokens in the expression.

  Θ(n) time is spent extracting tokens and outputting them.
  Θ(n) time is spent in two *while* loops
  as the number of tokens that get stacked and unstacked is linear in *n*.

  So, the time complexity of function *postfix* is Θ(n).

3.7 MULTIPLE STACKS AND QUEUES

- Implementing multiple stacks (queues) which are mapped sequentially into an array, *memory*[MEMORY_SIZE].
- If we have only two stacks to represent, the solution is simple.



IsEmpty (stacki)
IsFull (stacki)
Push (stacki, item)
Pop (stacki)

- **More than two stacks :**

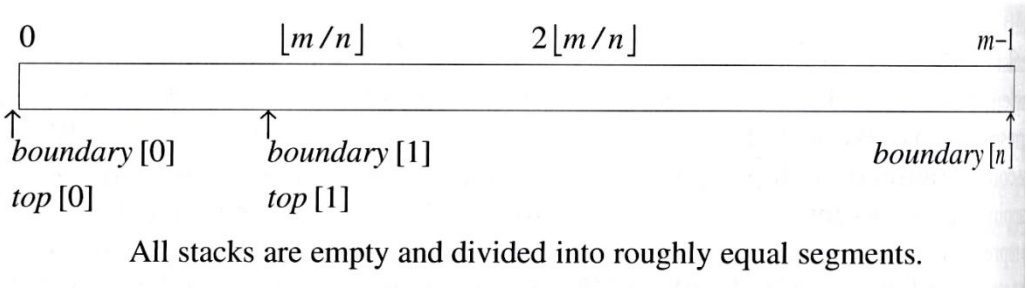
  Assuming *n* stacks, initially we divide the available memory into *n* segments.

  Let *stack_no* refers to the stack number of one of *n* stacks.

  The bottom element, *boundary*[*stack_no*], 0≤*stack_no*<MAX_STACKS, always points to the position immediately to the left of the bottom element, while *top*[*stack_no*], 0≤*stack_no*<MAX_STACKS, always points to the top element.

```
#define  MEMORY_SIZE 100       /* size of memory */
#define  MAX_STACKS 10     /* max number of stacks plus 1 */
/*  global memory declaration */
element memory[MEMORY_SIZE];
int  top[MAX_STACKS];
int  boundary[MAX_STACKS];
int  n;     /* number of stacks entered by the user  */

top[0] = boundary[0]  = -1;
for (i =1; i < n; i++)
      top[i] = boundary[i] = (MEMORY_SIZE/n)*i;
boundary[n] = MEMORY_SIZE-1;
```

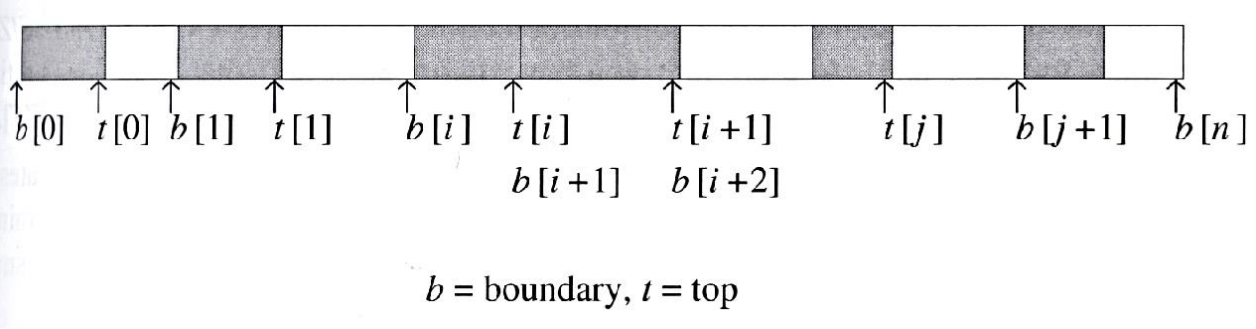


All stacks are empty and divided into roughly equal segments.

- What do we do if a stack is full?

  We create an error recovery function, *stackFull*, which determines if there is any free space in memory.

  If there is space available, it should shift the stacks so that space is allocated to the full stack



b = boundary, t = top

Chapter 4 : LISTS

Data Structure Lecture Note
Prof. Jihoon Yang
Data Mining Research Laboratory

4.1 POINTERS

- **Sequential representation**
  - storing successive elements of the data object a fixed distance apart.
  - adequate for many operations.

  But difficulties occurs when
  - insertion and deletion of an arbitrary element (time-consuming)
  - storing several lists of varying sizes in different arrays of maximum size (waste of storage)
  - maintaining the lists in a single array (frequent data movements)

- **Linked representation**
  - A node, associated with an element in the list, contains a *data component* and a *pointer* to the next item in the list. The pointers are often called *links.*

- C provides extensive support for pointers
  - actual value of a pointer type is an address of memory.
  - operators
    - & : the address operator
    - * : the dereferencing (or indirection) operator.

```
int  i, *pi;
pi = &i;
```

- To assign a value to i,
  i = 10; or *pi = 10;

- C allows us to perform arithmetic operations and relational operations on pointers. Also we can convert pointers explicitly to integers.

- The null pointer points to no object or function.
- Typically the null pointer is represented by the integer 0.
- There is a macro called NULL which is defined to be this constant.

- The macro is defined either

  in *stddef.h* for ANSI C or in *stdio.h* for K&R C.

- To test for the null pointer on C

  ```
  if (pi == NULL) or if (!pi)
  ```

4.1.1 Pointers Can Be Dangerous

- By using pointers we can attain a high degree of flexibility and efficiency.
- But pointer can be dangerous: accessing unexpected memory locations
  - Set all pointers to NULL when they are not actually pointing to an object.
  - Explicit *type casts* when converting between pointer types.

```
pi = malloc(sizeof(int));     /* assign to pi a pointer to int */
pf = (float *) pi;      /* casts an int pointer to a float pointer */
```

  - Define explicit return types for functions.

4.1.2 Using Dynamically Allocated Storage

- **malloc**
- **free**

[Program 4.1]

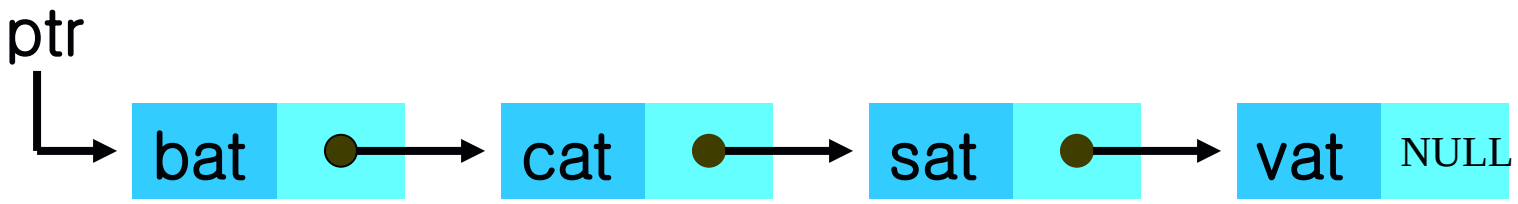
```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc(sizeof(float));
*pi = 1024;
*pf = 3.14;
printf("an integer = %d, a float = %f□n", *pi, *pf);
free(pi);
free(pf);
```

inserting pf = (float *) malloc(sizeof(float));

Creates *Garbage, Dangling reference*

4.2 SINGLY LINKED LISTS

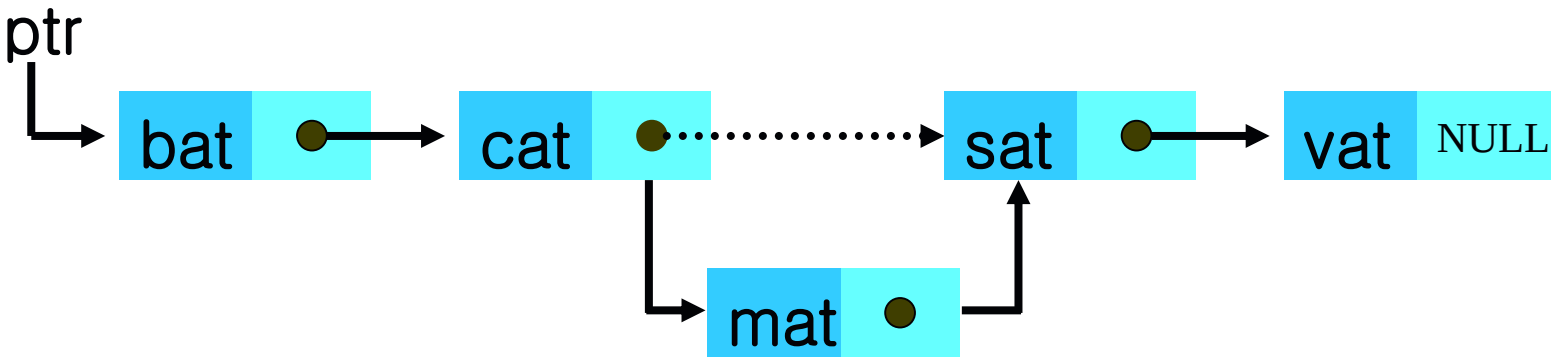
[Figure 4.1]



- The name of the pointer to the first node in the list is the name of the list.
- **Note that**
  (1) the nodes do not reside in sequential locations
  (2) the locations of the nodes may change on the different runs.
- When we write a program that works with lists, we almost never look for a specific address except when we test for the end of the list.

- **To insert the word *mat* between *cat* and *sat*, we must :**
  (1) Get a node that is currently unused; let its address be *paddr*.
  (2) Set the data field of this node to *mat*.
  (3) Set *paddr*'s link field to point to the address found in the link field of the node containing *cat*.
  (4) Set the link field of the node containing *cat* to pointer to *paddr*.

[Figure 4.2]

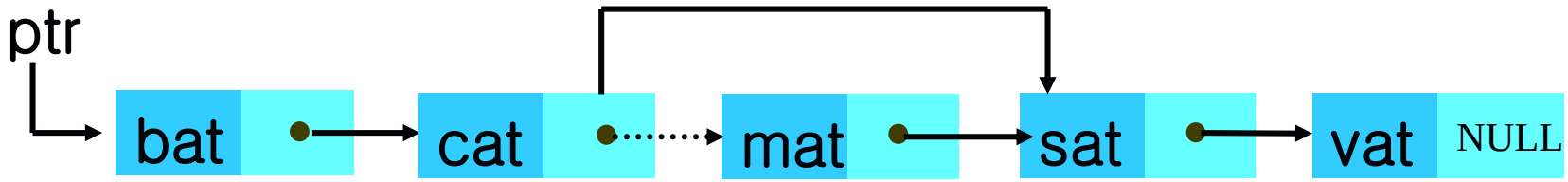


- **To delete *mat* from the list.**

  (1) Find the element (node) that immediately precedes *mat*, which is *cat*.

  (2) Set *cat*'s link field to point to *mat*'s link field.

[Figure 4.3]



- **Necessary capabilities to make linked list possible :**

  (1) A mechanism for defining a node's structure,
  self-referential structures.

  (2) A way to create new nodes when we need them, *malloc*.

  (3) A way to remove nodes that we no longer need, *free.*

- **Example 4.1 [List of words ending in *at*]**

Necessary declarations are :

```
typedef struct  list_node  *list_pointer;
typedef struct  list_node {
      char  data[4];
      list_pointer  link;
};
list_pointer  ptr = NULL;      /* creating a new empty list */
```

- **A macro to test for an empty list :**

  #define IS_EMPTY(ptr) (!(ptr))

- **Creating new nodes :**

  use the *malloc* function provided in <*stdio.h*>.

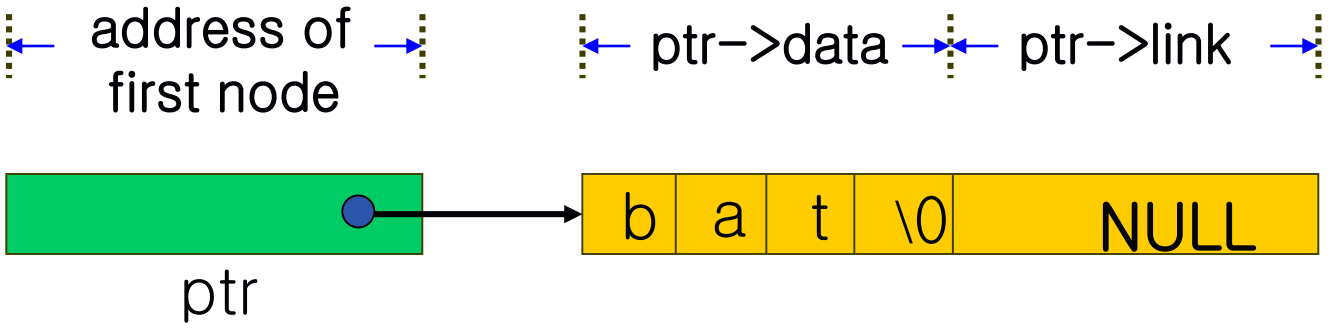
  ptr = (list_pointer) malloc (sizeof(list_node));

- **Assigning the values to the fields of the node:**
  - If *e* is a pointer to a structure that contains the field *name*, *e->name* is a shorthand way of writing the expression (**e*).*name*.

- **To place the word bat into the list :**

  strcpy (ptr->data, "bat");

  ptr->link = NULL;



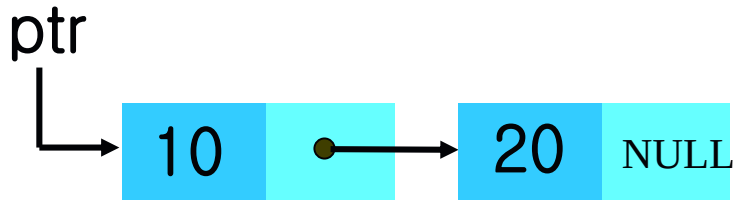
- **Example 4.2 [Two-node linked list] :**

```
typedef struct  list_node  *list_pointer;
typedef struct  list_node {
        int  data;
        list_pointer  link;
};
list_pointer  ptr = NULL;
```

- [Program 4.2]

```
list_pointer create2()
{
/*  create a linked list with two nodes  */
list_pointer first, second;
first = (list_pointer) malloc(sizeof(list_node));
second = (list_pointer) malloc(sizeof(list_node));
second->link = NULL;
second->data = 20;
first->data = 10;
first->link = second;
return first;
}
```

[Figure 4.5]

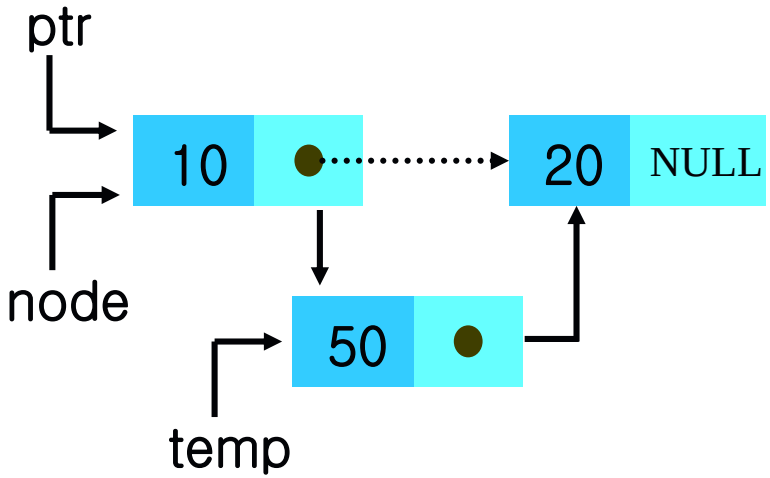


- **Example 4.3 [List insertion] :**
  - To insert a node with data field of 50 after some arbitrary node. Note that we use the parameter declaration *list_pointer *ptr*.
  - We use a new macro, IS_FULL, that allows us to determine if we have used all available memory.

```
#define IS_FULL (ptr)  (!(ptr))
```

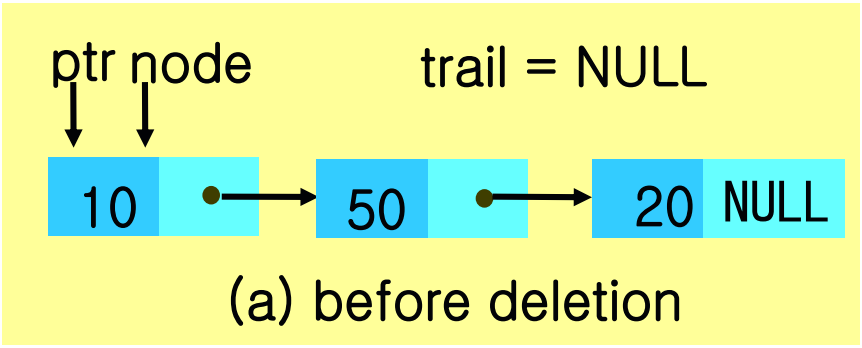
- **[Program 4.3]**

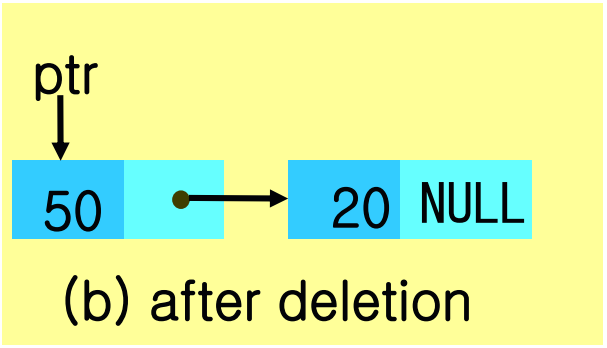
```
void insert(list_pointer *ptr, list_pointer node)
{
    /* insert a new node with data=50 into the list ptr after node */
    list_pointer temp;
    temp = (list_pointer) malloc(sizeof(list_node));
    if (IS_FULL(temp))  {
        fprintf(stderr, "The memory is full□n");
        exit(1);
     }
    temp->data = 50;
    if (*ptr)  {
        temp->link = node->link;
        node->link = temp;
     }
    else  {
        temp->link = NULL;
        *ptr = temp;
    }
}
```



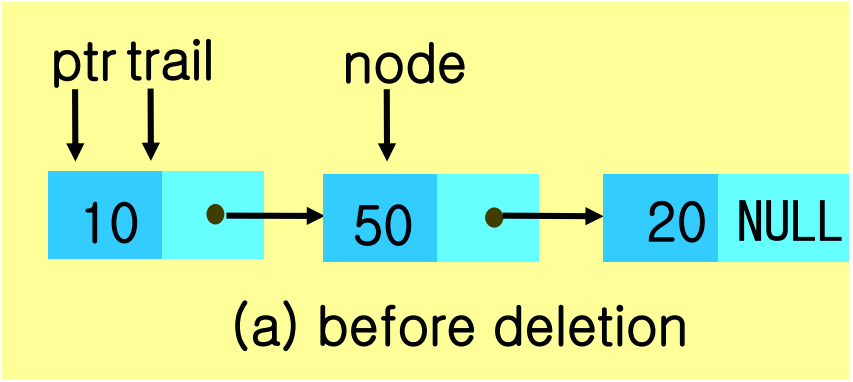
- **Example 4.4 [List deletion] :**
  - Deletion depends on the location of the node to be deleted.
  - Assume three pointers :
    ptr points to the start of the list.
    node points to the node that we wish to delete.
    trail points to the node that precedes the node to be deleted.
- **[Program 4.4]**

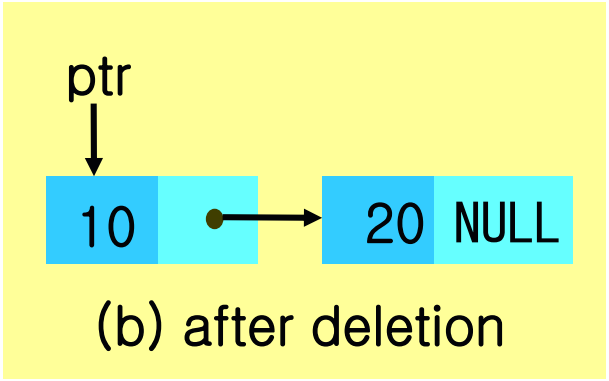
```
void delete(list_pointer *ptr, list_pointer trail,  list_pointer node)
{
    /* delete node from the list, trail is the preceding node
    ptr is the head of the list  */
    if (trail)
        trail->link = node->link;
    else
        *ptr = (*ptr)->link;
    free(node);
}
```





[Figure 4.7] ***delete(&ptr, NULL, ptr);***





[Figure 4.8] ***delete(&ptr, ptr, ptr->link);***

- **Example 4.5 [Printing out a list] :**
- **[Program 4.5]**

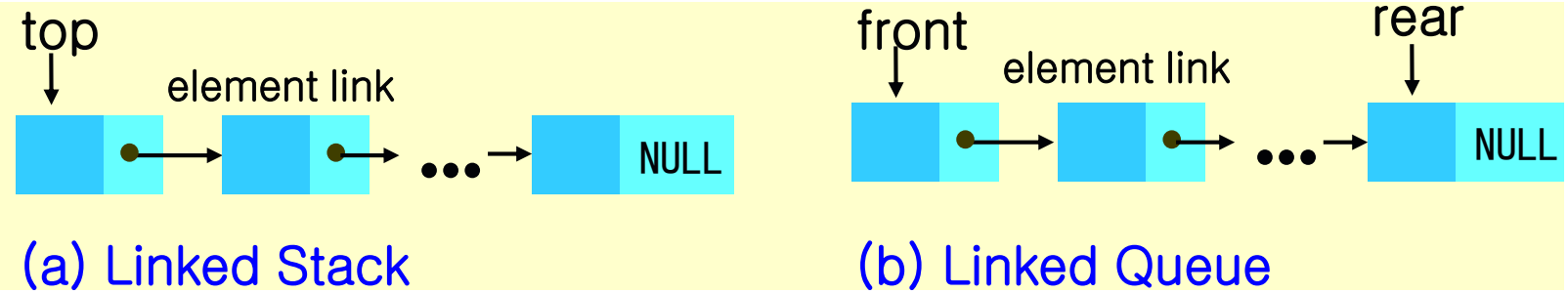
```
void print_list(list_pointer ptr)
{
     printf("The list contains: ");
     for ( ; ptr; ptr = ptr->link)
          printf("%4d", ptr->data);
     printf("□n");
}

list_pointer  search (list_pointer ptr, int num)
{
     for ( ; ptr; ptr = ptr->link)
          if (ptr->data == num)   return  ptr;
return  ptr;
}
```

```
void  merge (list_pointer x,  list_pointer y,  list_pointer *z)
{
      list_pointer  last;
      last = (list_pointer) malloc(sizeof(list_node));
      *z = last;
      while (x && y)  {
            if (x->data <= y->data)  {
                  last->link = x;
                  last = x;
                  x = x->link;
            }
            else  {
                  last->link = y;
                  last = y;
                  y = y->link;
            }
      }
      if (x)   last->link = x;
      if (y)   last->link = y;
      last = *z;  *z = last->link;  free(last);
 }
```

4.3 DYNAMICALLY LINKED STACKS AND QUEUES

- Sequential representation is proved efficient if we had only one stack or one queue.
- When several stacks and queues coexisted, there was no efficient way to represent them sequentially.
- Linked stacks and linked queues.



(a) Linked Stack (b) Linked Queue

Notice that the direction of links for both the stack and the queue facilitate easy insertion and deletion of nodes.

```
#define MAX_STACKS 10  /* maximum number of stacks */
typedef struct  {
      int key;
      /* other fields */
} element;
typedef struct stack *stack_pointer;
typedef struct stack  {
      element  item;
      stack_pointer  link;
};
stack_pointer  top[MAX_STACKS];
```

- **initialize empty stacks :**

  top[*i*] = NULL, 0<=i<MAX_STACKS

- **the boundary conditions :**

  top[*i*] == NULL *iff* the *i* th stack is empty

  and

  IS_FULL(temp) *iff* the memory is full

- **[Program 4.6]**

```
void add(stack_pointer *top, element item)
{
    /*  add an element to the top of the stack  */
    stack_pointer  temp = (stack_pointer) malloc(sizeof(stack));
    if (IS_FULL(temp))  {
        fprintf(stderr, "The memory is full□n");
        exit(1);
    }
    temp->item = item;
    temp->link = *top;
    *top = temp;
}
```

call : *add*(&*top*[*stack_no*], *item*);

[Program 4.7]

```
element delete(stack_pointer *top)
{
        /*  delete an element from the stack */
        stack_pointer  temp = *top;
        element  item;
        if (IS_EMPTY(temp))  {
                fprintf(stderr, "The stack is empty□n");
                exit(1);
        }
        item = temp->item;
        *top = temp->link;
        free(temp);
        return  item;
}
```

call : ***item*** **=** ***delete*****(&*****top*****[*****stack_no*****]);**

```
#define MAX_QUEUES 10  /* maximum number of queues */
typedef struct  {
     int key;
     /* other fields */
} element;
typedef struct queue *queue_pointer;
typedef struct queue  {
     element  item;
     queue_pointer  link;
};
queue_pointer  front[MAX_QUEUES], rear[MAX_QUEUES];
```

- **initialize empty queues :**

  front[*i*] = NULL, 0<=i<MAX_QUEUES

- **the boundary conditions :**

  front[*i*] == NULL *iff* the *i* th queue is empty

  and

  IS_FULL(temp) *iff* the memory is full

- **[Program 4.8] call : *addq*(&*front[queue_no]*, &*rear[queue_no], item*);**

```
void addq(queue_pointer *front, queue_pointer *rear, element item)
{
    /*  add an element to the rear of the queue  */
    queue_pointer  temp =  (queue_pointer) malloc(sizeof(queue));
    if (IS_FULL(temp))  {
        fprintf(stderr, "The memory is full□n");
        exit(1);
    }
    temp->item = item;
    temp->link = NULL;
    if (*front)  (*rear)->link = temp;
    else  *front = temp;
    *rear = temp;
}
```

- **[Program 4.9] call : *item* = *deleteq*(&*front[queue_no]*);**

```
element deleteq(queue_pointer *front)
{
    /*  delete an element from the queue */
    queue_pointer  temp = *front;
    element  item;
    if (IS_EMPTY(*front))  {
        fprintf(stderr, "The queue is empty□n");
        exit(1);
    }
    item = temp->item;
    *front = temp->link;
    free(temp);
    return  item;
}
```

4.4 POLYNOMIALS

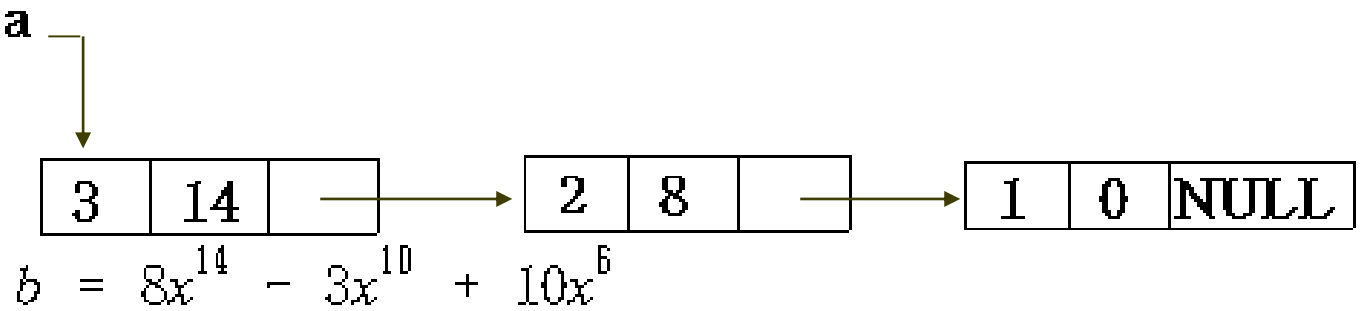
4.4.1 Representing Polynomials As Singly Linked Lists

- We want $A(x) = a_{m-1}\, x^{e_{m-1}} + \cdots + a_0\, x^{e_0}$
  - where a_i's are nonzero coefficients and e_i's are nonnegative integer exponents such that $e_{m-1} > e_{m-2} > \ldots > e_1 > e_0 \geq 0$.

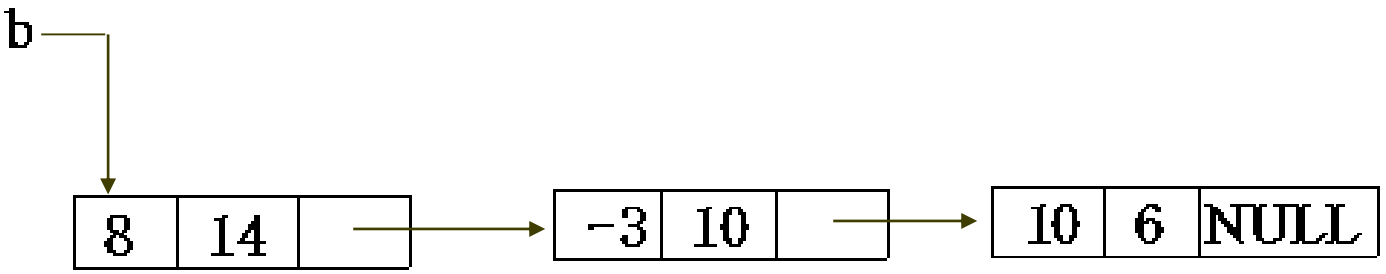
```
typedef struct poly_node *poly_pointer;
typedef struct poly_node {
    float coef;
    int expon;
    poly_pointer link;
};
poly_pointer a, b, d;
```

- **[Figure 4.11]**

$a = 3x^{14} + 2x^{8} + 1$



$b = 8x^{14} - 3x^{10} + 10x^{6}$



4.4.2 Adding Polynomials

- Compare Program 4.10 and Program 4.11 with Program 2.5 and Program 2.6.

- **[Program 4.10]**

```
poly_pointer padd(poly_pointer a, poly_pointer b)
{
    /*  return a polynomial which is the sum of a and b  */
    poly_pointer  front, rear, temp;
    float sum;
    rear = (poly_pointer) malloc(sizeof(poly_node));
    if  (IS_FULL(rear))  {
        fprintf(stderr, "The memory is full□n");
        exit(1);
    }
    front = rear;
```

```
    while (a && b)
        switch (COMPARE(a->expon, b->expon)){
        case –1 :  /* a->expon < b->expon */
              attach (b->coef, b->expon, &rear);
              b = b->link;    break;
        case  0 : /* a->expon = b->expon */
              sum =  a->coef + b->coef;
              if (sum)  attach(sum, a->expon, &rear);
              a = a->link;    b = b->link;     break;
        case  1 :  /* a->expon > b->expon */
              attach (a->coef, a->expon, &rear);
              a = a->link;
    }
    /*  copy rest of list a then list b  */
    for ( ; a; a = a->link) attach (a->coef, a->expon, &rear);
    for ( ; b; b = b->link) attach (b->coef, b->expon, &rear);
    rear->link = NULL;
    /*  delete extra initial node  */
    temp = front;  front = front->link;  free(temp);
    return  front;
}
```

- **[Program 4.11]**

```
void attach(float coefficient, int exponent,    poly_pointer *ptr)
{
     /* create a new node with coef = coefficient and
     expon = exponent,  attach it to the node pointed to
     by ptr.  ptr is updated to point to this new node  */
     poly_pointer  temp;
     temp = (poly_pointer)malloc(sizeof(poly_node));
     if  (IS_FULL(temp))  {
          fprintf(stderr, "The memory is full□n");
          exit(1);
     }
     temp->coef = coefficient;
     temp->expon = exponent;
     (*ptr)->link = temp;
     *ptr = temp;
}
```

- **Analysis of *padd* :**
  - Similar to the analysis of Program 2.5.

    Three cost measures :

    (1) coefficient additions

    (2) exponent comparisons

    (3) creation of new nodes for d

- Clearly, ≤ 0 number of coefficient additions $\leq \min[m, n]$, number of exponent comparisons and creation of new nodes is at most $m+n$.

- Therefore, its time complexity is $O(m + n)$.

4.4.3 Erasing Polynomials

- Let's assume that we are writing a collection of functions for input, addition, subtraction, and multiplication of polynomials using linked lists as the means of representation.
- Suppose we wish to compute e(x) = a(x) * b(x) + d(x) :

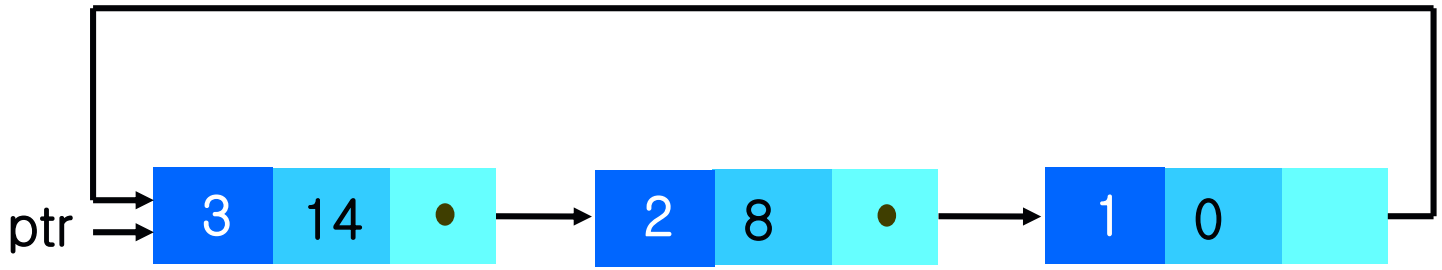
```
poly_pointer  a, b, d, e;
⋮
a = read_poly();
b = read_poly();
d = read_poly();
temp = pmult(a, b);
e = padd(temp, d);
print_poly(e);
```

- Note that we create polynomial *temp(x)* only to hold a partial result for d(x).
- By returning the nodes of *temp(x)*, we may use them to hold other polynomials.
- **[Program 4.12]**

```
void  erase(poly_pointer  *ptr)
{
     /* erase the polynomial pointed by ptr */
     poly_pointer  temp;
     while (*ptr) {
          temp = *ptr;
          *ptr = (*ptr) -> link;
          free(temp);
     }
}
```

4.4.4 Representing Polynomials As Circularly Linked Lists

- To free all the nodes of a polynomial more efficiently, we modify our list structure so that the link field of the last node points to the first node in the list.



- We call this a *circular list*.
- A *chain* : a singly linked list in which the last node has a null link.

- We want to free nodes that are no longer in use so that we may reuse these nodes later.
- We can obtain an efficient erase algorithm for circular lists, by maintaining our own list (as a chain) of nodes that have been "freed".
- When we need a new node, we examine this list.
  If the list is not empty, then we may use one of its nodes.
  Only when the list is empty, use *malloc* to create a new node.
- Let *avail* be a variable of type *poly_pointer* that points to the first node in the list of freed nodes.

- **[Program 4.13]**

```
poly_pointer get_node(void) {
    /* provide a node for use */
    poly_pointer node;
    if (avail)  {
        node = avail;
        avail = avail->link;
    }
    else  {
        node = (poly_pointer) malloc(sizeof(poly_node));
        if (IS_FULL(node))  {
            fprintf(stderr, "The memory is full□n");
            exit(1);
        }
     }
    return node;
}
```

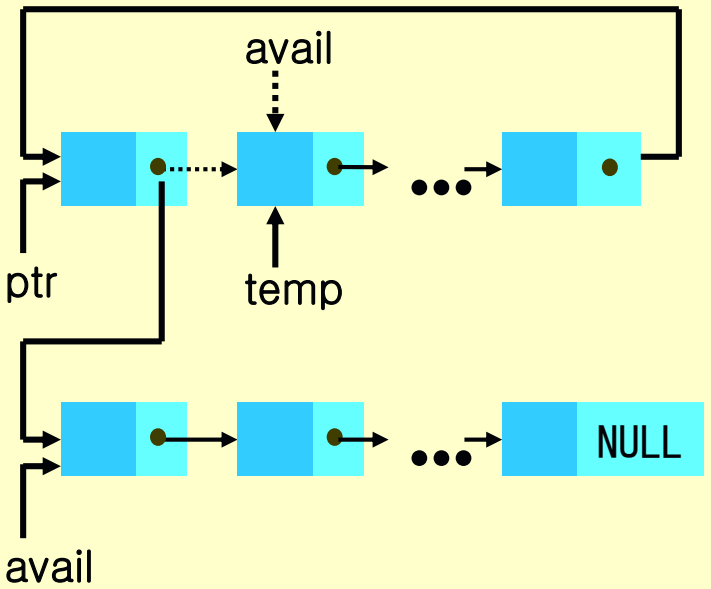
- **[Program 4.14]**

```
void ret_node(poly_pointer ptr)  {
     /*  return a node to the available list */
     ptr->link = avail;
     avail = ptr;
}
```

- **[Program 4.15]**

```
void cerase(poly_pointer *ptr)  {
     /* erase the circular list ptr */
     poly_pointer  temp;
     if (*ptr)  {
          temp = (*ptr)->link;
          (*ptr)->link = avail;
          avail = temp;
          *ptr = NULL;
     }
}
```

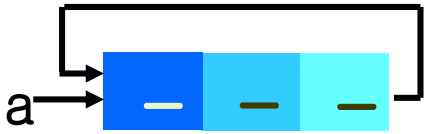
[Figure 4.14]



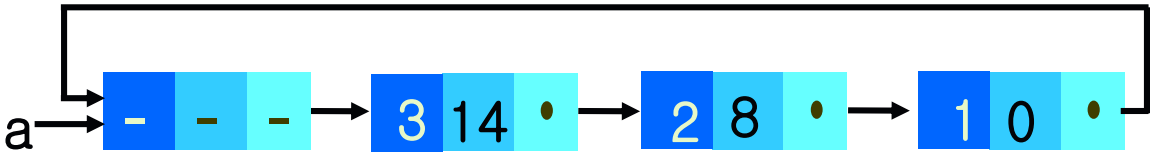
- A direct changeover to the structure of Figure 4.13 creates problems when we implement the other polynomial operations since we must handle the zero polynomial as a special case.

- We introduce a *head node* into each polynomial.

  [Figure 4.15]



(a) zero polynomial



(b) $3x^{14} + 2x^{8} + 1$

- For the circular list with head node representation, we may remove the test for *(*ptr)* from *cerase*.
- The only changes that we need to make to *padd* are :

(1) Add two variables, *starta = a* and *startb =* b.
(2) Prior to the *while* loop, assign *a = a->link* and *b = b->link*.
(3) Change the *while* loop to *while* (*a* != *starta* && *b* != *startb*).
(4) Change the first *for* loop to *for* (; *a* != *starta*; *a = a->link*).
(5) Change the second *for* loop to *for* (; *b* != *startb*; *b = b->link*).
(6) Delete the lines :

```
rear -> link = NULL;
/* delete extra initial node */
```

(7) Change the lines :

```
temp = front;
front = front -> link;
free(temp);
```

to

```
rear -> link = front;
```

- We may further simplify the addition algorithm
  if we set the *expon* field of the head node to -1.

- **[Program 4.16]**

```
poly_pointer cpadd(poly_pointer a, poly_pointer b)
{
/* polynomials a and b are singly linked circular lists with  a head
  node.  Return a polynomial which is the sum of  a and b */
    poly_pointer starta, d, lastd;
    int sum,  done = FALSE;
    starta = a;              /*  record start of a  */
    a = a->link;             /*  skip head node for a and b  */
    b = b->link;
    d = get_node();          /(  get a head node for sum  */
    d->expon = -1;           lastd = d;
```

```
    do  {
        switch (COMPARE(a->expon, b->expon)){
        case –1 :  /* a->expon < b->expon */
            attach (b->coef, b->expon, &lastd);
            b = b->link;      break;
        case  0 : /* a->expon = b->expon */
            if (starta == a)  done = TURE;
            else  {
                    sum =  a->coef + b->coef;
                    if (sum)   attach(sum, a->expon, &lastd);
                    a = a->link;   b = b->link;
            }
            break;
         case  1 :  /* a->expon > b->expon */
            attach (a->coef, a->expon, &lastd);
            a = a->link;
         }
    } while (!done)
            lastd->link = d;
    return d;
}
```

4.5 ADDITIONAL LIST OPERATIONS

4.5.1 Operations For Chains

- It is often necessary, and desirable to build a variety of functions for manipulating singly linked lists. We have seen *get_node* and *ret_node*.

- We use the following declarations :

```
typedef struct list_node *list_pointer;
typedef struct list_node {
        char data;
        list_pointer link;
};
```

- ***Inverting a chain*** **:**
  - we can do it "in place" if we use three pointers.
- **[Program 4.17]**

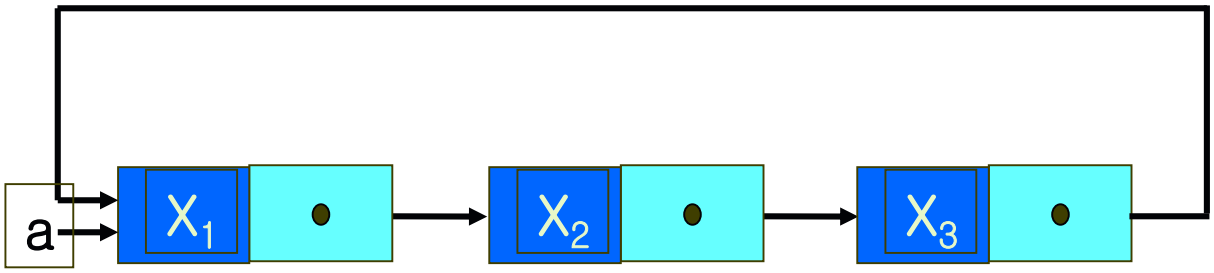
```
list_pointer invert(list_pointer lead)
{
    /* invert the list pointed to by lead  */
    list_pointer  middle, trail;
    middle = NULL;
    while (lead)  {
        trail = middle;
        middle = lead;
        lead = lead->link;
        middle->link = trail;
    }
    return  middle;
}
```

- ***Concatenating two chains*** **:**
- **[Program 4.18]**

```
list_pointer concatenate(list_pointer ptr1, list_pointer ptr2)
{
    /* produce a new list that contains the list ptr1 followed
    by the list ptr2. The list pointed to by ptr1 is changed
    permanently */
    list_pointer  temp;
    if (IS_EMPTY(ptr1))  return ptr2;
    else  {
        if (!IS_EMPTY(ptr2))  {
            for (temp = ptr1; temp->link; temp = temp->link)
                ;
            temp->link = ptr2;
        }
        return  ptr1;
    }
}
```

4.5.2 Operations For Circularly Linked Lists

- ***Inserting a new node at the front of a circular list*** **:**
  - Since we have to change the link field of the last node, we must move down the list until we find the last node.
  - It is more convenient if the name of the circular list points to the last node rather than the first.



- **[Program 4.19]**

```
void insert_front(list_pointer *ptr, list_pointer node)
      /* insert node at the front of the circular list ptr,
      where ptr is the last node in the list.    */
{
      if (IS_EMPTY(*ptr))  {
            /* list is empty, change ptr to point to new entry */
            *ptr = node;
            node->link = node;
      }
      else  {
            /* list is not empty, add new entry at front */
            node->link = (*ptr)->link;
            (*ptr)->link = node;
      }
}
```

- ***Inserting a new node at the rear of a circular list*** **:**

We only need to add the additional statement **ptr = node* to the *else* clause of *insert_front*.

- **[Program 4.20]**

```
int length(list_pointer ptr)
{
    /* find the length of the circular list ptr */
    list_pointer temp;
    int count = 0;
    if (ptr)  {
        temp = ptr;
        do  {
            count++;
            temp = temp->link;
        } while (temp != ptr);
    }
    return count;
}
```

4.6 EQUIVALENCE RELATIONS

- R is a *binary relation* on a set S if $R \subseteq S \times S$.
  If $(a, b) \in R$ then we may write aRb.

  - R is *reflexive* if aRa for all $a \in S$.
  - R is *symmetric* if aRb implies bRa.
  - R is *transitive* if aRb and bRc implies aRc.

  - R is an *equivalence relation* over S
    if R is reflexive, symmetric and transitive over S.

- **[Example]**
  - One of the steps in the manufacture of a VLSI circuit involves exposing a silicon wafer using a series of masks. Each mask consists of several polygons. Polygons that overlap electrically are equivalent and electrical equivalence specifies an equivalence relation ≡ over the set of mask polygons.

(1) For any polygon x, x≡x, that is, x is electrically equivalent to itself. Thus, ≡ is reflexive.

(2) For any two polygons, x and y, if x≡y then y≡x. Thus, the relation ≡ is symmetric.

(3) For any three polygons, x, y, and z, if x≡y and y≡z then x≡z. For example, if x and y are electrically equivalent and y and z are also equivalent, then x and z are also electrically equivalent.

Thus the relation ≡ is transitive.

- Any equivalence relation R over S can partition the set S into disjoint subsets called *equivalence classes*.

- An *equivalence class* E is a subset of S such that if x is in E then E contains every element which is related to x by R. That is, for any $x \in S$, $[x] = \{y \mid y \in S \text{ and } x \equiv y\}$.

- For any x and y in S, either $[x] = [y]$ or $[x] \cap [y] = \emptyset$.

- **Example :**
  - If we have 12 polygons numbered 0 through 11
    and the following pairs overlap :
    0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡0
  - as a result of the reflexivity, symmetry, and transitivity of the relation ≡, we can obtain the following equivalence classes :
    {0, 2, 4, 7, 11}; {1, 3, 5}; {6, 8, 9, 10}

- **The algorithm to determine equivalence works in two phases :**
  - *First phase* : read in and store the equivalence pairs.
  - *Second phase* : determining equivalence class as follows

    we begin at 0 find all pairs of the form <0, j>.

    By transitivity, find all pairs of the form <j, k>.

    /* <0, j> and <j, k> ⇒ <0, k> i.e, 0≡j and j≡k ⇒ 0≡k */

    We continue in this way until we have found, marked, and printed the entire equivalence class containing 0.

    Then we continue on.

- **Our first design attempt :**
- **[Program 4.21]**

```
void equivalence()
{
    initialize;
    while (there are more pairs)  {
        read the next pair <i, j>;
        process this pair;
    }
    initialize the output;
    do
    output a new equivalence class;
    while  (not done);
}
```

- Let *m* and *n* represent the number of related pairs and the number of objects, respectively.
- We must first figure out which data structure we should use to hold these pairs.
- The pair <i, j> is essentially two random integers in the range 0 to n-1.
- Use an array, *pairs*[n][m], for easy random access.
  this could waste a lot of space and require considerable time
  or use more storage to insert a new pair.

- These considerations lead us to a linked representation for each row.
- Since we still need random access to the *i* -th row, we use a one-dimensional array, *seq*[n], to hold the head nodes of the *n* lists.
- In the second phase of the algorithm, we need to check whether or not the object, *i*, has been printed.
- We use the array *out*[n].

- **[Program 4.22]**

```
void equivalence()
{
    initialize seq to NULL and out to TRUE;;
    while (there are more pairs)  {
        read the next pair <i, j>;
        put j on the seq[i] list;
        put i on the seq[j] list;
    }
    for (i=0; i<n; i++)
        if (out[i])  {
            out[i] = FALSE;
            output this equivalence class;
        }
}
```

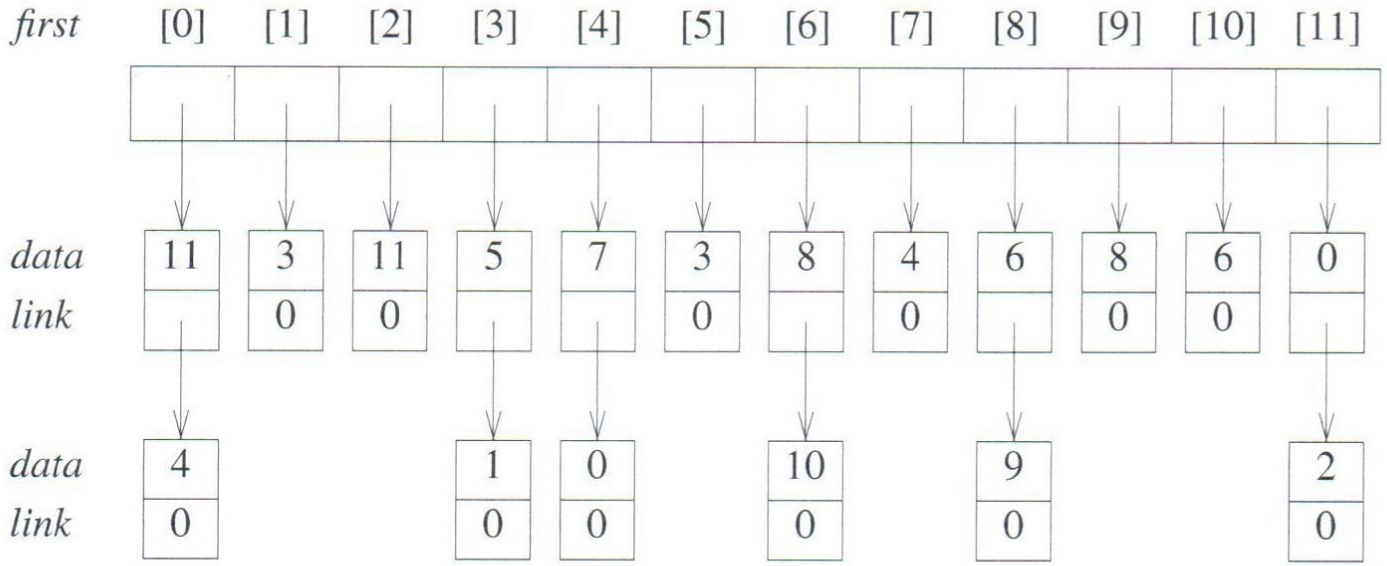


Figure 4.16: Lists after pairs have been input

- In phase two :
  - We scan the *seq* array for the first *i*, $0 \le i < n$, such that *out*[i] = TRUE.
  - Each element in the list *seq*[i] is printed.
- To process the remaining lists which, by transitivity, belong in the same class as *i*, we create a stack of their nodes.
- For the complete equivalence algorithm, see the following declaration and Program 4.22.

```
#include <stdio.h>
#include <alloc.h>
#define MAX_SIZE  24
#define IS_FULL (ptr)  (!(ptr))
#define FALSE 0
#define TRUE 1

typedef  struct node *node_pointer;
typedef  struct node {
     int data;
     node_pointer link;
};
```

```
void main(void)
{
     short  int out[MAX_SIZE];
     node_pointer  seq[MAX_SIZE];
     node_pointer x, y, top;
     int  i, j, n;

     printf("Enter the size (<= %d) ", MAX_SIZE);
     scanf("%d", &n);
     for (i = 0; i < n; i++)  {
          /*  initialize seq and out  */
          out[i] = TRUE;        seq[i] = NULL;
     }
```

```
/*  Phase 1: Input the equivalence pairs :  */
printf("Enter a pair of numbers (-1 –1 to quit): ");
scanf("%d%d", &i, &j);
while (i >=0)  {
      x = (node_pointer)malloc(sizeof(node));
      if (IS_FULL(x))  {
            fprintf(stderr, "The memory is full□n");
            exit(1);
      }
      x->data = j;  x->link = seq[i];  seq[i] = x;
      x = (node_pointer)malloc(sizeof(node));
      if (IS_FULL(x))  {
            fprintf(stderr, "The memory is full□n");
            exit(1);
      }
      x->data = i;  x->link = seq[j];  seq[j] = x;
      printf("Enter a pair of numbers (-1 –1 to quit): ");
      scanf("%d%d", &i, &j);
}
```

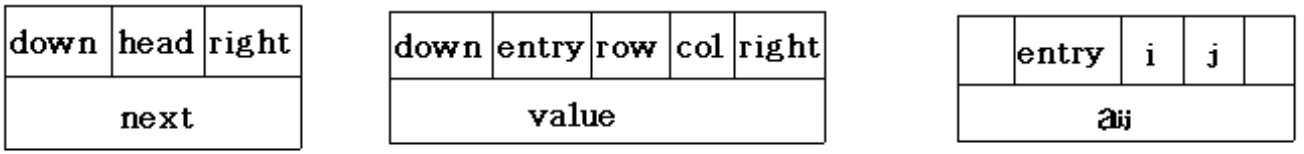
```
    /* Phase 2 : output the equivalence classes  */
    for (i = 0; i < n; i++)  {
        if  (out[i])  {
            printf("□nNew Class : %5d", i);
            out[i] =FALSE;   /*   set class to false  */
            x = seq[i];  top = NULL;  /*  initialize stack  */
            for ( ; ; )   {    /*  find rest of class  */
                while (x)   {    /*  process list  */
                    j = x->data;
                    if  (out[j])   {
                        printf("%5d", j);   out[j] = FALSE;
                        y = x->link;  x->link = top;  top = x;  x = y;
                    }
                    else  x = x->link;
                }
                if  (!top)   break;
                x = seq[top->data];  top = top->link;   /* unstack */
            }
        }
    }
}
```

- Analysis of the equivalence program :
  - Initialization of *seq* and *out* takes O(*n*) time.
  - Each of Phase 1 and 2 takes O(*m + n*) time.
  - Time complexity is O(*m+n*) and space complexity is also O(*m+n*).
  - In Chapter 5, we will look at an alternate solution that requires only O(*n*) space.

4.7 SPARSE MATRIX

- In Chapter 2, we considered a sequential representation of sparse matrices and implemented matrix operations.
- However we found that the sequential representation of sparse matrices suffered from the same inadequacies as the similar representation of polynomials.
- As we have seen previously, linked lists allow us to efficiently represent structures that vary in size, a benefit that also applies to sparse matrices.
- In our data representation, we represent each column of a sparse matrix as a circularly linked list with a head node. We use a similar representation for each row of a sparse matrix.

[Figure 4.19]

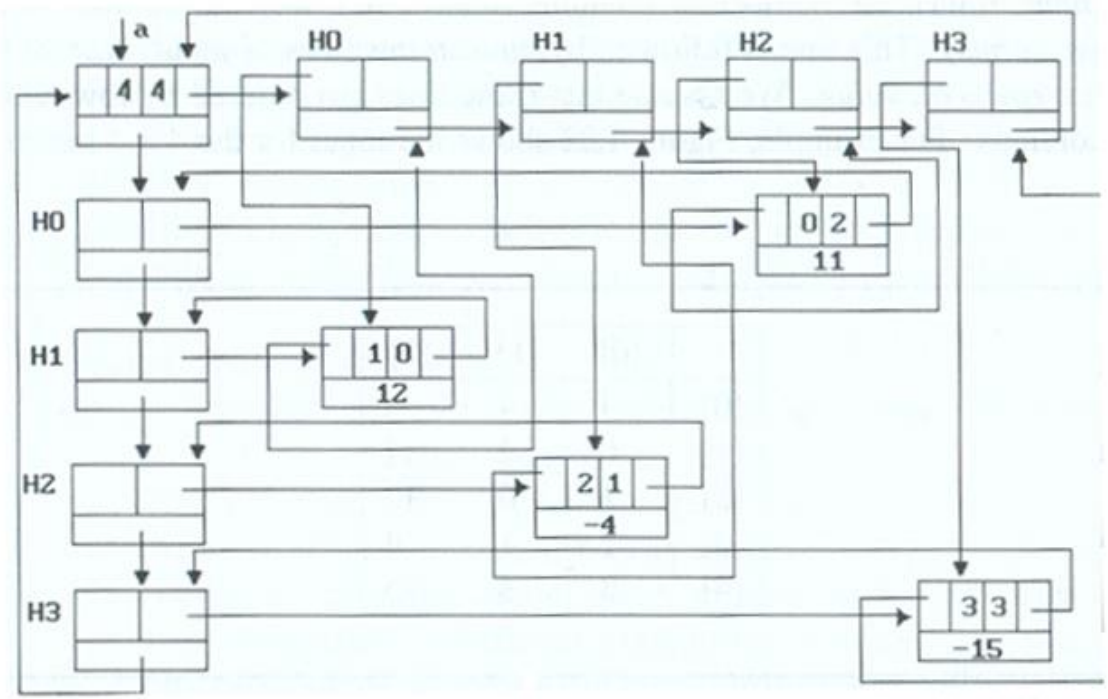


(a) head node (b) entry node (c) set up for a_{ij}

[Figure 4.20]

$$\begin{bmatrix} 0 & 0 & 11 & 0 \\ 12 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & -15 \end{bmatrix}$$

- **[Figure 4.21]**
  - Each head node is in three lists:
    a list of rows, a list of columns, and a list of head nodes.
    The list of head nodes also has a head node
    that has the same structure as an entry node.



NOTE: The tag field of a node is not shown; its value for each node should be clear from the node structure.

Figure 4.21: Linked representation of the sparse matrix *a*

```
#define MAX_SIZE 50

typedef enum {head, entry} tagfield;
typedef  struct  matrix_node  *matrix_pointer;
typedef  struct  entry_node  {
     int row;
     int col;
     int value;
};
typedef struct matrix_node {
     matrix_pointer down;
     matrix_pointer right;
     tagfield tag;
     union {
          matrix_pointer next;
          entry_node entry;
     } u;
     matrix_pointer hdnode[MAX_SIZE];
```

```
matrix_pointer mread()
{
    int num_rows, num_cols,num_terms, num_heads,i;
    int row,  col, value, current_row;
    matrix_pointer temp, last, node;

    scanf(&num_rows, &num_cols, &num_terms);
    num_heads = (num_cols > num_rows) ? num_cols : num_rows;
    node = new_node(); node_tag = entry;
    node->u.entry.row = num_rows;
    node->u.entry.col = num_cols;
```

```
if (!num_heads) node->right = node;
else {
    for (i=0; i<num_heads; i++) {
        temp = new_node();
        hdnode[i] = temp; hdnode[i]->tag = head;
        hdnode[i]->right = temp; hdnode[i]->u.next=temp;
    }
    current_row = 0;  last = hdnode[0];
    for (i=0; i<num_terms; i++) {
        scanf(&row, &col, &value);
        if (row > current_row) {
            last->right = hdnode[current_row];
            current_row = row;  last = hdnode[row];
        }
        temp = new_node();  temp->tag = entry;
        temp->u.entry.row = row;  temp->u.entry.col = col;
        temp->u.entry.value = value;  last->right = temp;  last = temp;
        hdnode[col]->u.next->down = temp;
        hdnode[col]->u.next = temp;
    }
```

```
    // close last row
    last->right = hdnode[current_row];
    // close all column lists
    for (i=0; i<num_cols; i++)
        hdnode[i]->u.next->down = hdnode[i];
    // link all head nodes together
    for (i=0; i<num_heads-1; i++)
        hdnode[i]->u.next = hdnode[i+1];
    hdnode[num_heads-1]->u.next = node;
    node->right = hdnode[0];
  }
  return node;
}
```

```
// print out the matrix in row major form
void mwrite(matrix_pointer node)
{
    int i;
    matrix_pointer temp, head = node->right;

    for (i=0; i<node->u.entry.row; i++) {
        for (temp = head->right; temp != head;
            temp = temp->right)
            printf(temp->u.entry.row, temp->u.entry.col,
            temp->u.entry.value);
        head = head->u.next;
    }
}
```

```
void merase(matrix_pointer *node)
{
     int i, num_heads;
     matrix_pointer x,y, head = (*node)->right;

     for (i=0; i<(*node)->u.entry.row; i++) {
          y = head->right;
          while (y != head) {
               x = y;  y = y->right;  free(x);
          }
          x = head;  head = head->u.next;  free(x);
     }
     // free remaining head nodes
     y = head;
     while (y != *node) {
          x = y;  y = y->u.next;  free(x);
     }
     free(*node);  *node = NULL;
}
```

- **Analysis of *mread* : [Program 4.24]**
  O(max{*num_rows*, *num_cols*} + *num_terms*)
  = O(*num_rows* + *num_cols* + *num_terms*).

- **Analysis of *mwrite* : [Program 4.26]**
  O(*num_rows* + *num_terms*).

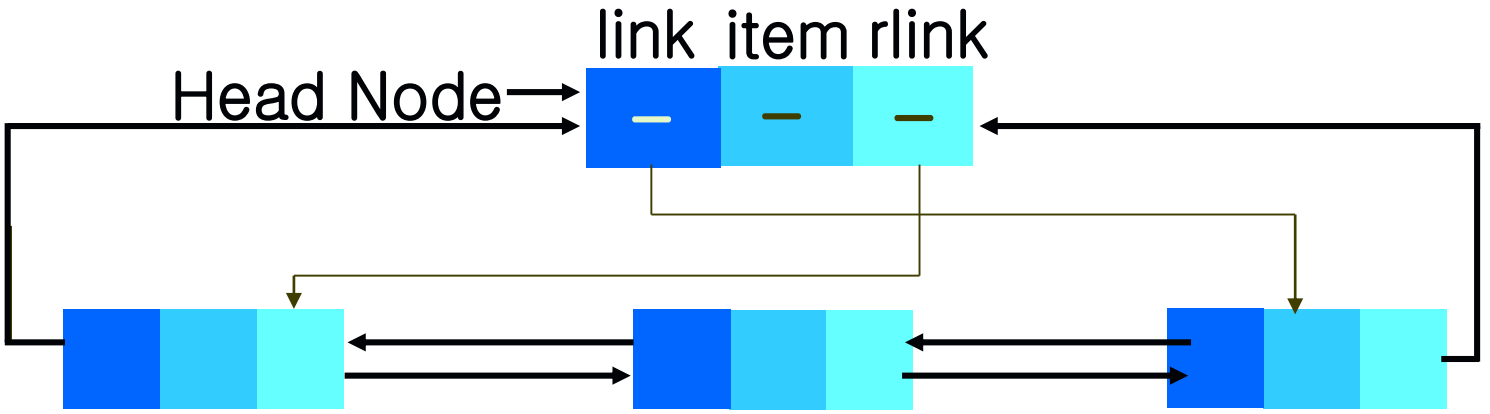
- **Analysis of *merase* : [Program 4.27]**
  O(*num_rows* + *num_cols* + *num_terms*).

4.8 DOUBLY LINKED LISTS

- Singly linked lists pose problems because we can move easily only in the direction of the links.
- Whenever we have a problem that requires us to move in either direction, it is useful to have doubly linked lists.
- The necessary declarations are :

```
typedef struct node *node_pointer;
typedef struct node {
    node_pointer llink;
    element item;
    node_pointer rlink;
};
```

- A doubly linked list may or may not be circular.
- **[Figure 4.23] Doubly linked circular list with head node**



- **[Figure 4.24] Empty doubly linked circular list with head node**

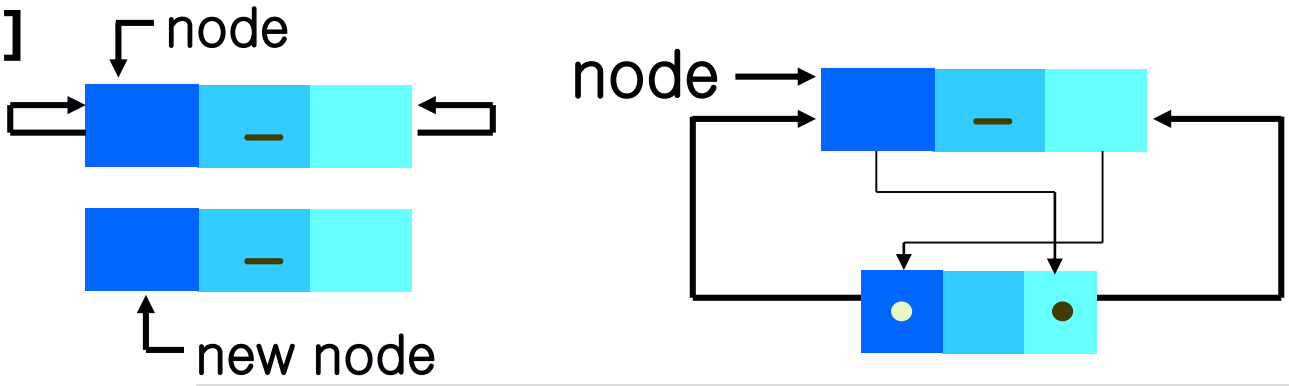
- Now suppose that *ptr* points to any node in a doubly linked list. Then :

  ptr == ptr->llink->rlink == ptr->rlink->llink

- **Insertion into a doubly linked circular list :**
- **[Program 4.28]**

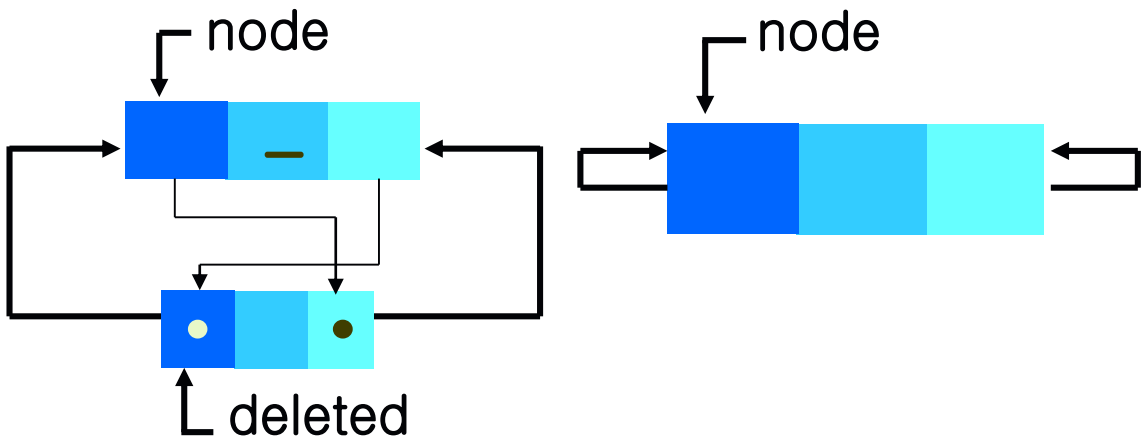
```
void dinsert(node_pointer node, node_pointer newnode)
{
    /* insert newnode to the right of node */
    newnode->llink = node;
    newnode->rlink = node->rlink;
    node->rlink->llink = newnode;
    node->rlink = newnode;
}
```

[Figure 4.25]



- **Deletion from a doubly linked circular list :**
- **[Program 4.29]**

```
void ddelete(node_pointer node,  node_pointer deleted)  {
        /* delete from the doubly linked list  */
        if (node == deleted)
                printf("Deletion of head node not permitted.□n");
        else  {
                deleted->llink->rlink = deleted->rlink;
                deleted->rlink->llink = deleted->llink;
                free(deleted);
        }
}
```



[Figure 4.26]