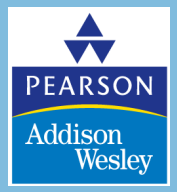
Chapter 8:
Data Abstractions

Computer Science: An Overview
Tenth Edition

by
J. Glenn Brookshear



Chapter 8: Data Abstractions

- 8.1 Data Structure Fundamentals
- 8.2 Implementing Data Structures
- 8.3 A Short Case Study
- 8.4 Customized Data Types
- 8.5 Classes and Objects

数据类型

早期：数值计算 ——

运算对象是简单的整型、实型或布尔类型数据

中后期：非数值计算 ——

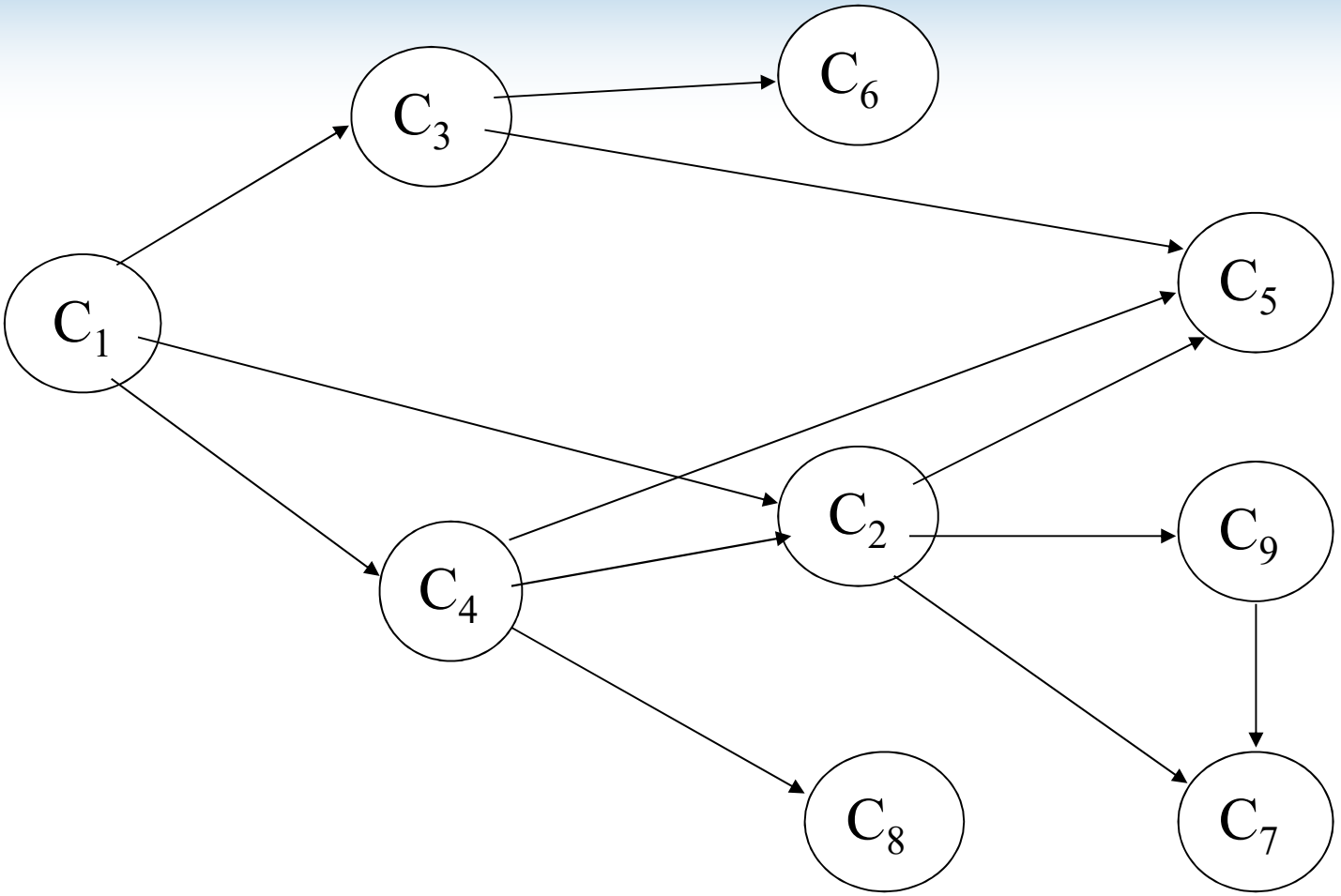
处理对象是类型复杂的数据，数据元素之间的相互关系一般无法用数学方程式加以描述

	学　号	姓　名	性别	籍　贯	出生年月
1	98131	刘激扬	男	北　京	1979.12
2	98164	衣春生	男	青　岛	1979.07
3	98165	卢声凯	男	天　津	1981.02
4	98182	袁秋慧	女	广　州	1980.10
5	98203	林德康	男	上　海	1980.05
6	98224	洪　伟	男	太　原	1981.01
7	98236	熊南燕	女	苏　州	1980.03
8	98297	宫　力	男	北　京	1981.01
9	98310	蔡晓莉	女	昆　明	1981.02
10	98318	陈　健	男	杭　州	1979.12

教学计划编排问题

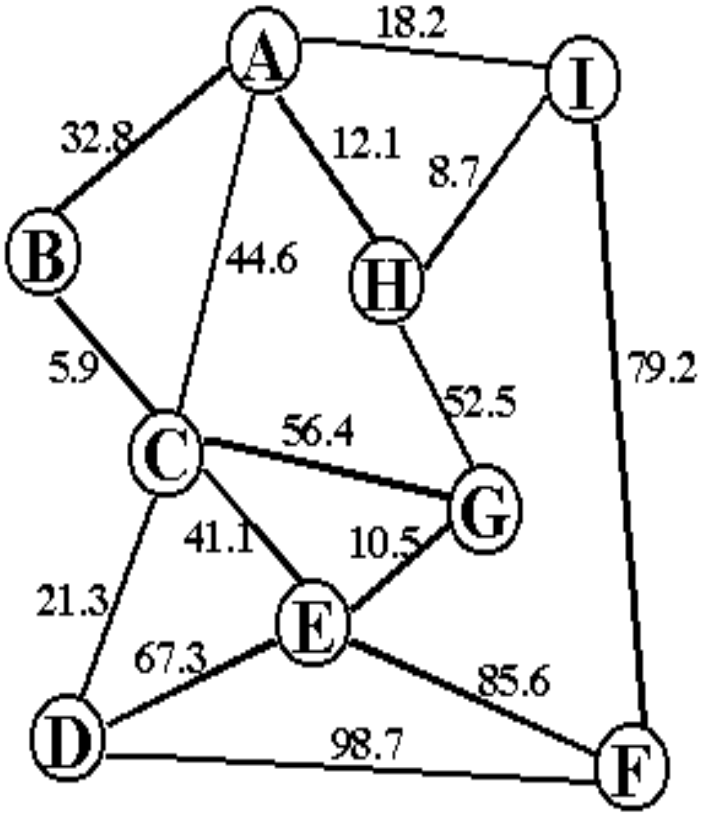
课程编号	课程名称	先修课程
C_1	计算机导论	无
C_2	数据结构	C_1，C_4
C_3	汇编语言	C_1
C_4	C程序设计语言	C_1
C_5	计算机图形学	C_2，C_3，C_4
C_6	接口技术	C_3
C_7	数据库原理	C_2，C_9
C_8	编译原理	C_4
C_9	操作系统	C_2

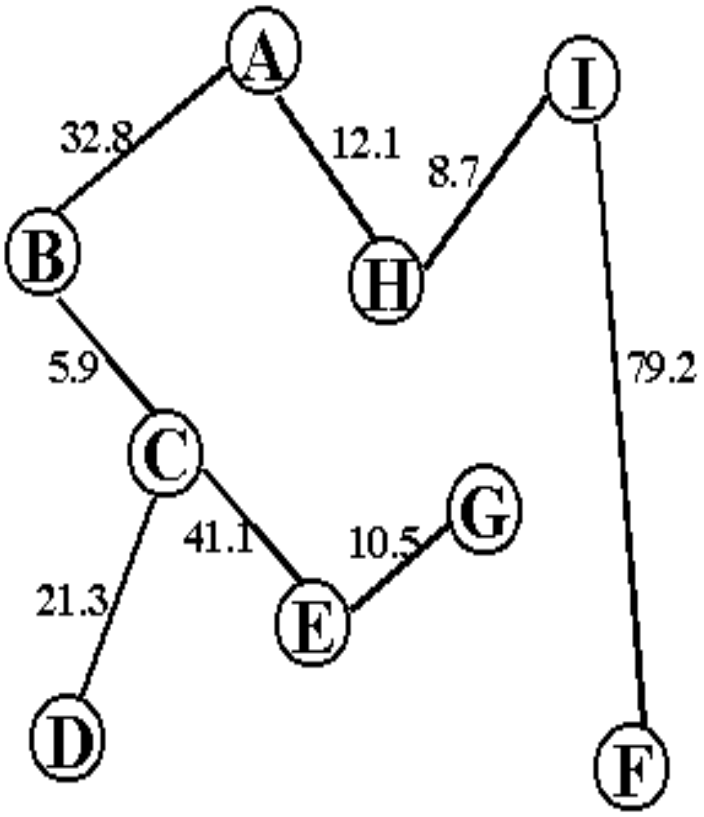
（a）计算机专业的课程设置



（b）表示课程之间优先关系的有向图

城市的煤气管道问题





（a）结点间管道的代价　　（b）最经济的管道铺设

- 描述这类非数值计算问题的数学模型不再是数学方程，而是诸如表、树、图之类的数据结构。
- 数据结构是一门研究（非数值计算的）程序设计问题中所出现的计算机操作对象以及它们之间的关系和操作的学科。

What is a Data Structure?

- A data structure is a collection of data organized in some fashion

- A data structure not only stores data, but also supports the operations for manipulating data in the structure

Why to learn?

- You will understand
  - what the tools are for storing and processing common data types
  - which tools are appropriate for which need
- So that you will be able to
  - make good design choices as a developer, project manager, or system customer

Data Structures: Why?

- Program design depends crucially on how data is structured for use by the program
  - Implementation of some operations may become easier or harder
  - Speed of program may dramatically decrease or increase
  - Memory used may increase or decrease
  - Debugging may be become easier or harder

- 基本概念：
  - 数据
  - 数据元素（数据成员）
  - 数据对象

- 数据：数据是信息的载体，是描述客观事物的数、字符、以及所有能输入到计算机中，被计算机程序识别和处理的符号（数值、字符等）的集合。
- 数据元素（数据成员）：是数据的基本单位。在不同的条件下，数据元素又可称为元素、结点、顶点、记录等。

- 数据对象：具有相同性质的数据元素（数据成员）的集合。
  - 整数数据对象 $N = \{ 0, \pm1, \pm2, \ldots \}$
  - 学生数据对象

数据结构的形式定义

数据结构由某一数据对象及该对象中所有数据成员之间的关系组成。记为：

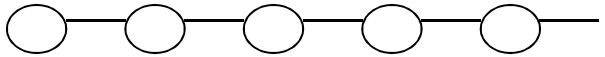
Data_Structure = {D, R}

其中，**D**是某一数据对象，**R**是该对象中所有数据成员之间的关系的有限集合。

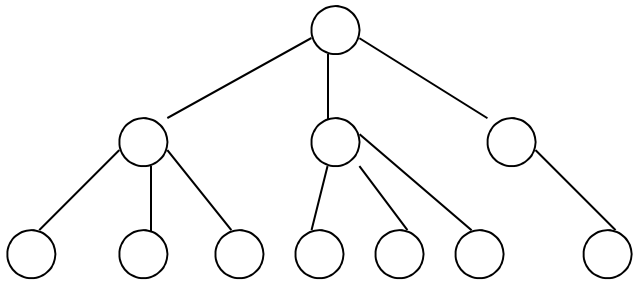
What is Data Structure

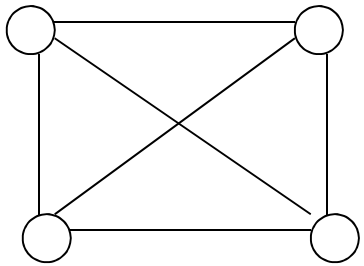
Data structure

- Linear structure



- Non-linear structure





数据结构涉及三个方面：

1.数据的逻辑结构-----从用户视图看，是面向问题的。

2. 数据的物理结构-----从具体实现视图看，是面向计算机的。

3. 相关的操作及其实现。

Example:

学生表：逻辑结构-----线性表

物理结构-----数组

操作-----插入，删除，查找

数据结构包括“逻辑结构” 和“物理结构”两个方面(层次):

- 逻辑结构 是对数据成员之间的逻辑关系的描述，它可以用一个数据成员的集合和定义在此集合上的若干关系来表示;
- 物理结构 是逻辑结构在计算机中的表示和实现，故又称“存储结构” 。

逻辑结构和物理结构的关系

- 数据的*逻辑结构*是从逻辑关系（某种顺序）上观察数据，它是独立于计算机的；可以在理论上、形式上进行研究、推理、运算等各种操作。
- 数据的*存储结构*是逻辑结构在计算机中的实现，是依赖于计算机的；是数据的最终组织形式。
- 任何一个*算法的设计*取决于选定的逻辑结构；而*算法的最终实现*依赖于采用的存储结构。

根据问题来建立逻辑结构

例如：Class = (D, S)

数据集合：$D = \{ a, b_1, \ldots, b_n, c_1, \ldots c_n, d_1, \ldots d_n \}$

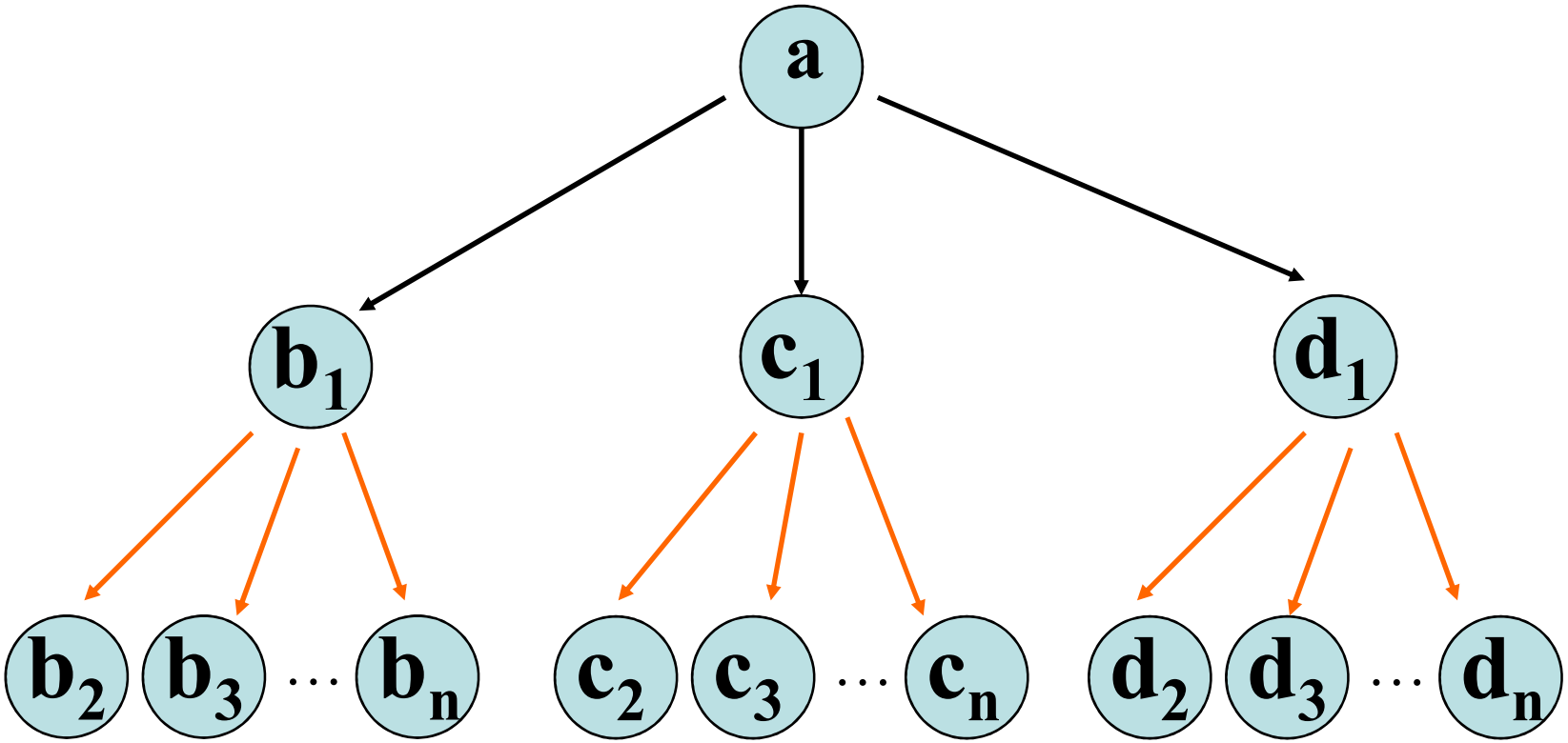
关系集合：$S = \{ R_1, R_2 \}$

$R1 = \{ <a, b_1>, <a, c_1>, <a, d_1> \}$

//班长-组长

$R2 = \{ <b_1, b_j>, <c_1, c_j>, <d_1, d_j> \mid j = 2, 3, \ldots, n \}$

//组长-组员



班级Class的逻辑结构的图示

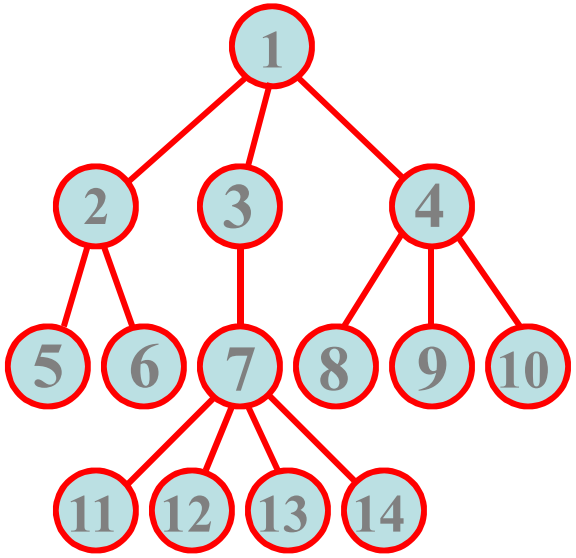
数据结构的分类

- 线性结构：表、栈、队列
- 非线性结构
  - 层次结构： 树，二叉树，堆
  - 网状结构： 图
  - 其它：集合

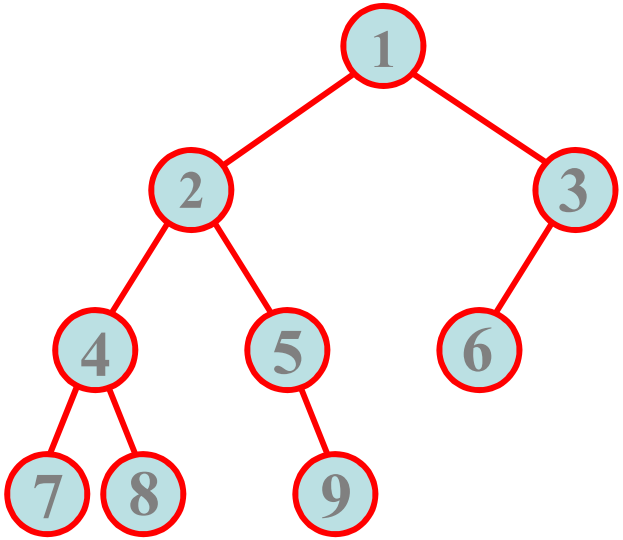
线性结构

非线性结构——层次结构

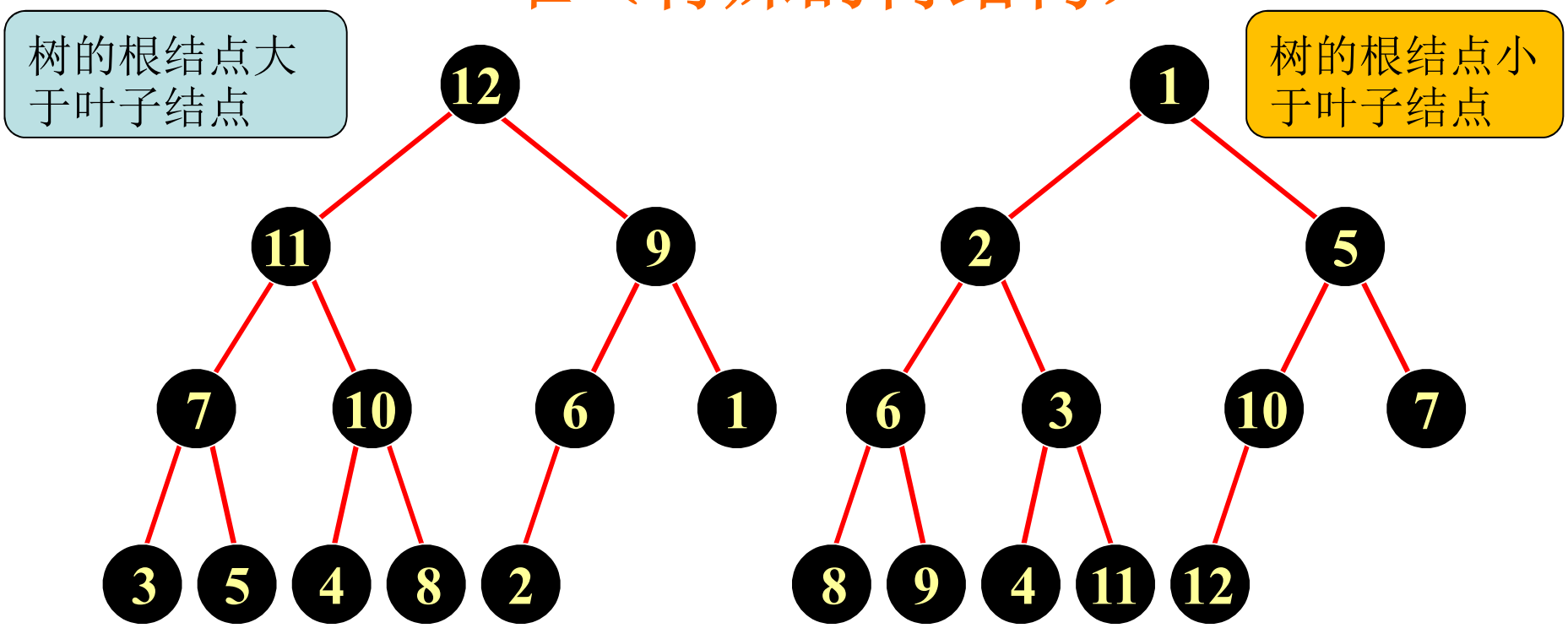
树



二叉树

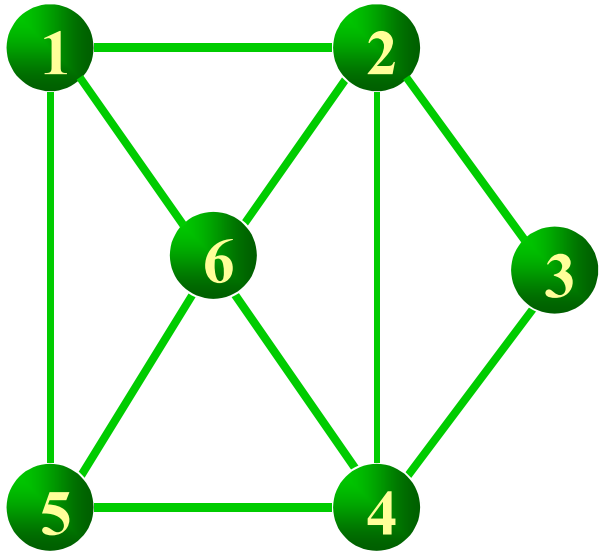


堆（特殊的树结构）

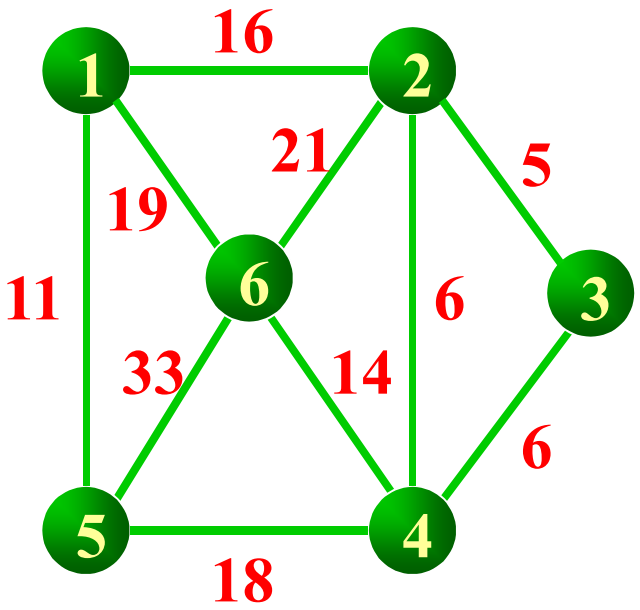


“最大”堆　　“最小”堆

非线性结构——群结构



图结构



网络结构

数据结构的抽象形式

- **C语言中的数据类型**

  char　int　float　double　void

  字符型 整型 浮点型 双精度型 无值

- 数据类型

定义：一组性质相同的值的集合，以及定义于这个值集合上的一组操作的总称。

抽象数据类型 (*ADTs: Abstract Data Types*)

- 由用户定义，用以表示应用问题的数据模型
- 由基本的数据类型组成, 并包括一组相关的服务（或称操作）
- 支持了逻辑设计和物理实现的分离，支持封装和信息隐蔽

抽象：抽取反映问题本质的东西，忽略非本质的细节

抽象数据类型的两种视图：

- 设计者的角度：

　　根据问题来定义抽象数据类型所包含的信息，给出其相关功能的实现，并提供公共界面的接口。

- 用户的角度：

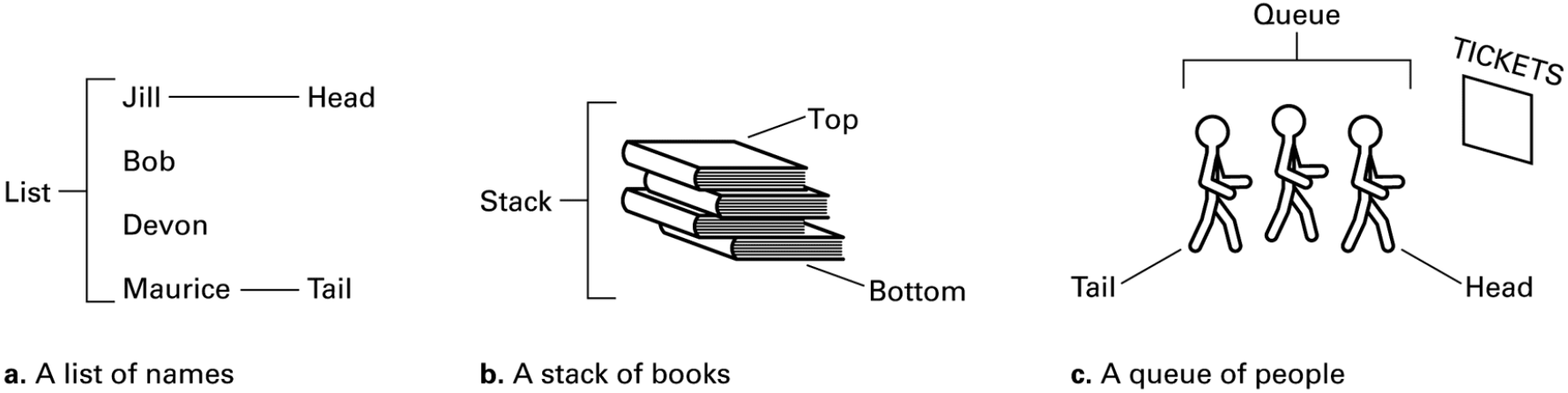
　　使用公共界面的接口对抽象数据类型进行操作，不需要考虑其物理实现。对于外部用户来说，抽象数据类型应该是一个黑盒子。

Basic Data Structures

- Homogeneous array（同构数组）
- Heterogeneous array（异构数组）
- List （列表）
- Stack（栈）
- Queue（队列）
- Tree（数）

Figure 8.1 Lists, stacks, and queues

List列表:一组数据，其表项按顺序排列，表开头为表头（head）,表尾端为表尾（tail）。



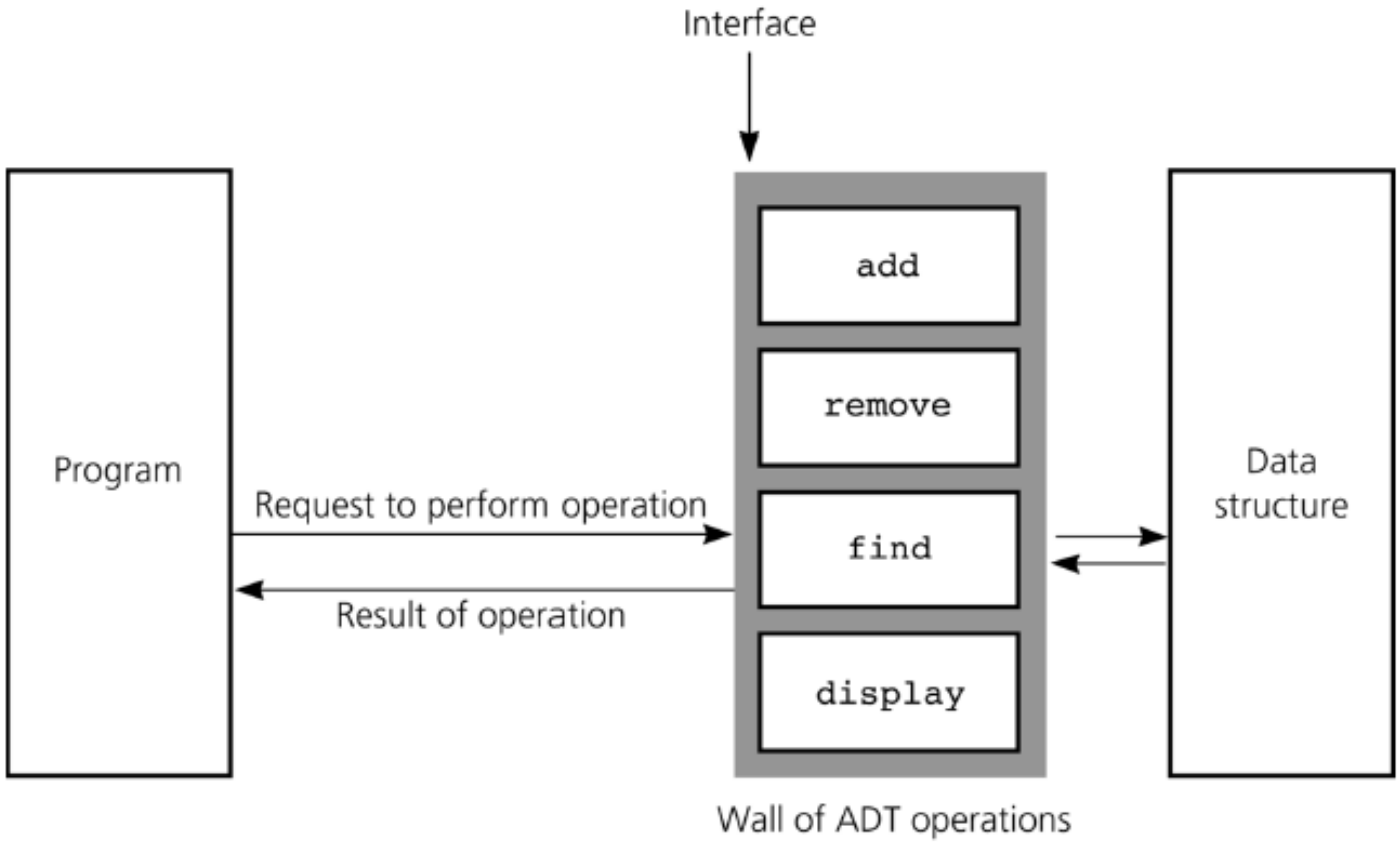
a. A list of names **b.** A stack of books **c.** A queue of people

通过严格限制列表中项的访问方式，可获得两种特殊类型的表：栈和队列。

栈：后进先出

队列：表头删除，表尾插入

Illustration



Additional Concepts

- Static Data Structures: Size and shape of data structure does not change
- Dynamic Data Structures: Size and shape of data structure can change
- Pointers: Used to locate data

Storing Arrays（存储数组）

- Homogeneous arrays
  - **Row-major order**（行主序） versus **column major order**（列主序）
  - Address polynomial
- Heterogeneous arrays
  - Components can be stored one after the other in a contiguous block
  - Components can be stored in separate locations identified by pointers

Figure 8.5 The array of temperature readings stored in memory starting at address x

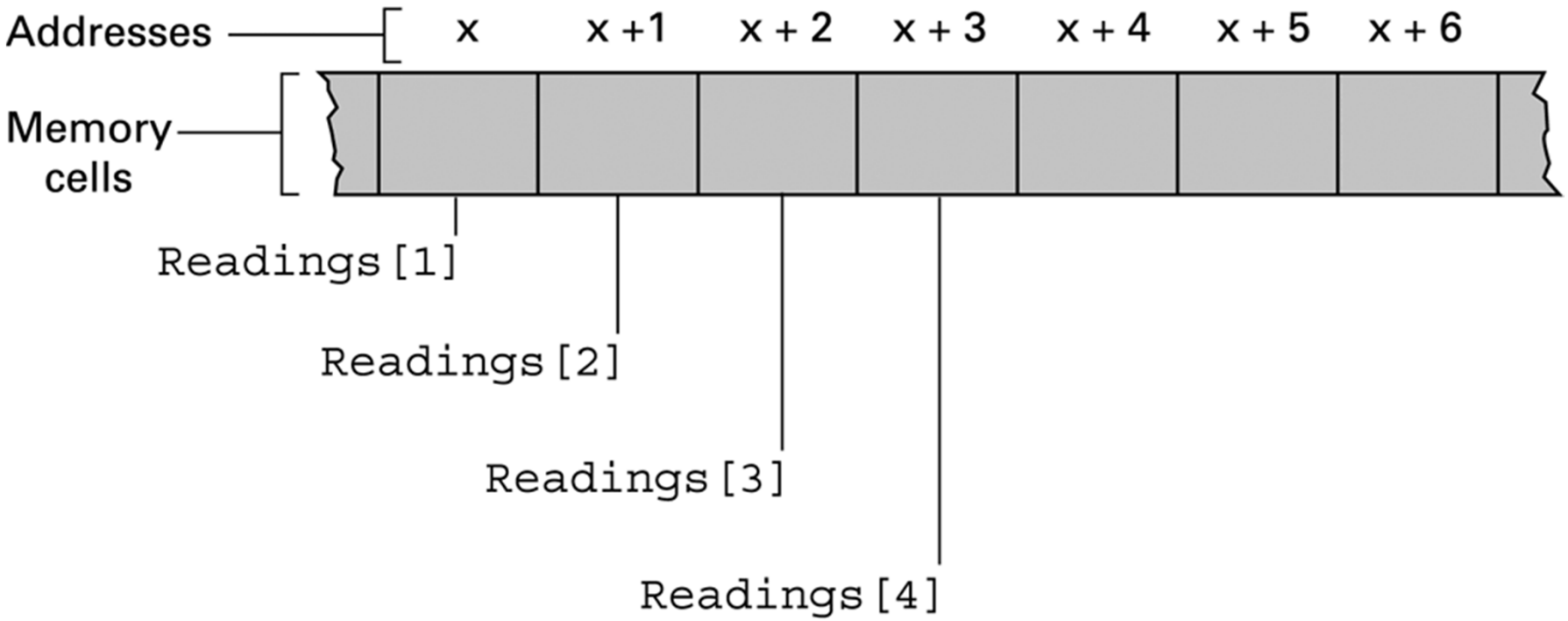
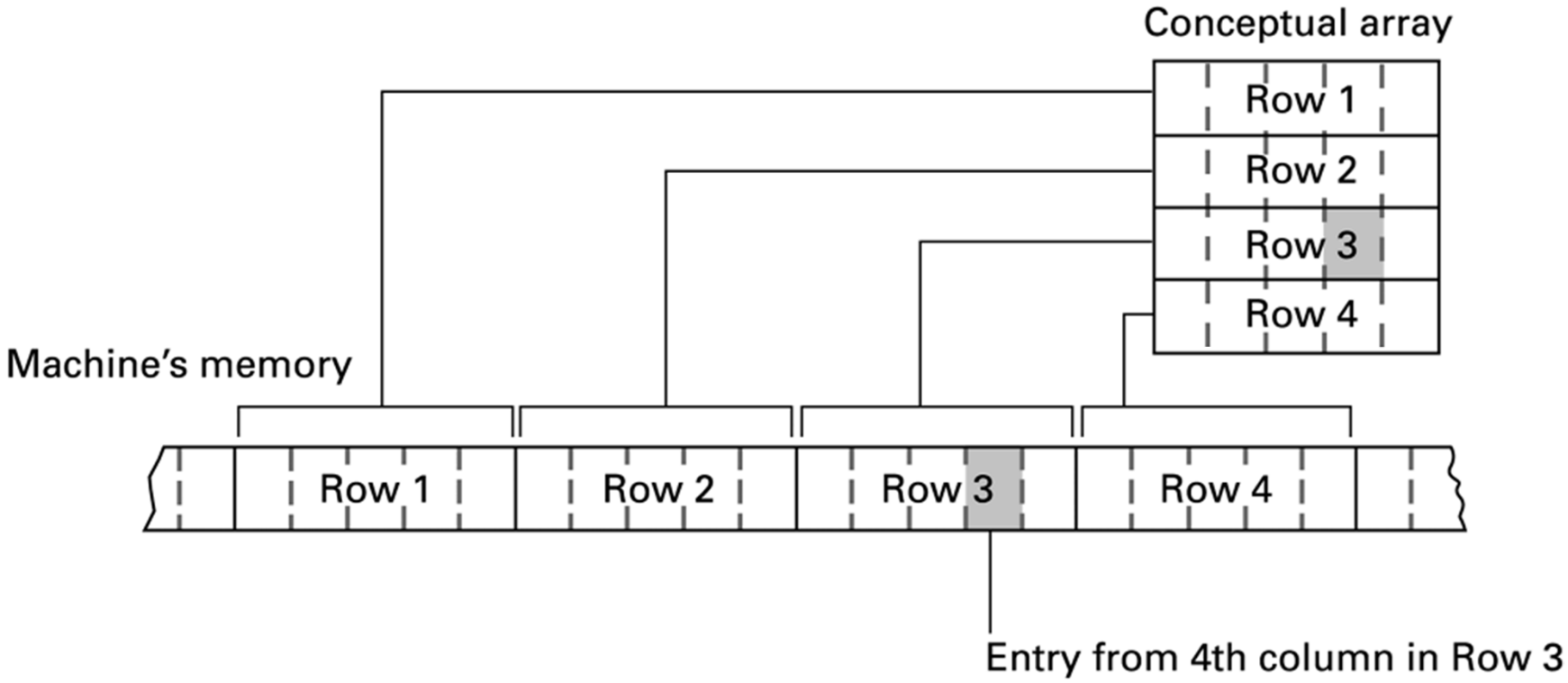


Figure 8.6 A two-dimensional array with four rows and five columns stored in row major order



Homogeneous array

```
/* 程序 1：求 10 个数之和 */
#include <stdio.h>
main()
{
 int i,s=0;
 int a[10]={66,55,75,42,86,77,96, 89,78,56};
 for(i=0;i<10;i++)
    s=s+a[i];
 printf("%d",s);
}
```

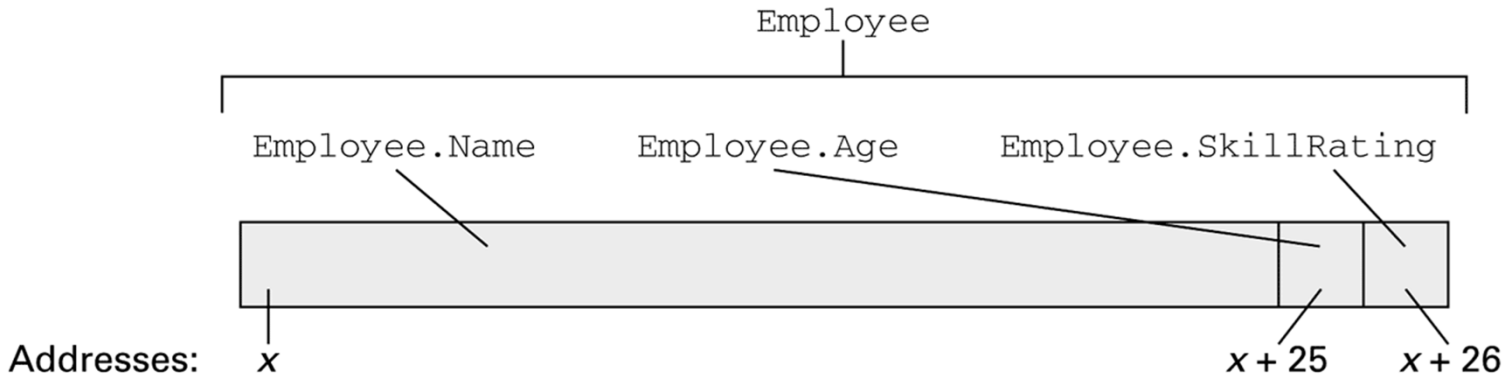
```
/* 程序 2：求 10 个数中的最大值 */
#include <stdio.h>
main()
{
 int i,s;
 int a[10]={66,55,75,42,86,77,96, 89,78,56};
 s=a[0];
 for(i=1;i<10;i++)
    if (s<a[i]) s=a[i];
 printf("%d",s);
```

Limitations of arrays

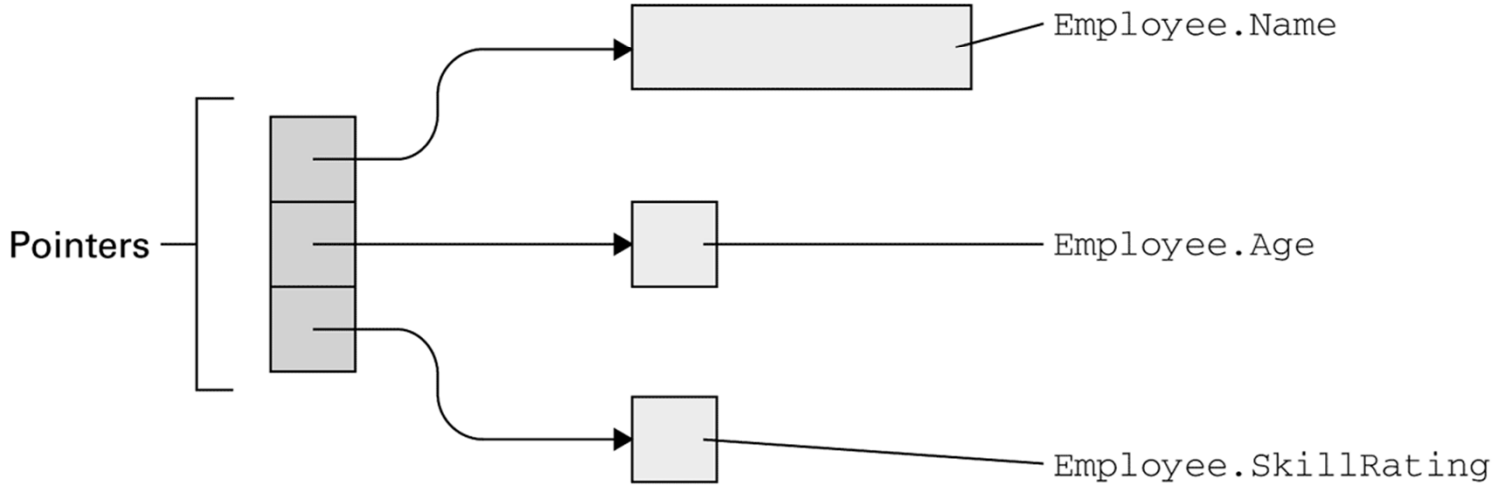
- Once an array is created, its size cannot be altered.

- Array provides inadequate support for inserting, deleting, sorting, and searching operations.

Figure 8.7 **Storing the heterogeneous array Employee**



a. Array stored in a contiguous block



b. Array components stored in separate locations

Heterogeneous array

```c
#include <stdio.h>

/* Define a type point to be a struct with integer members x, y */
typedef struct {
    int     x;
    int     y;
} point;

int main(void) {

/* Define a variable p of type point, and initialize all its members inline! */
     point p = {1,3};

/* Define a variable q of type point. Members are uninitialized. */
     point q;

/* Assign the value of p to q, copies the member values from p into q. */
     q = p;

/* Change the member x of q to have the value of 3 */
     q.x = 3;

/* Demonstrate we have a copy and that they are now different. */
     if (p.x != q.x) printf("The members are not equal! %d != %d", p.x, q.x);

     return 0;
}
```

List

Lists

- List: a finite sequence of data items

  a1, a2, a3, …, an
- Lists are pervasive in computing
  - e.g. class list, list of chars, list of events
- Typical operations:
  - Creation
  - Insert / remove an element
  - Test for emptiness
  - Find an item/element
  - Current element / next / previous
  - Find k-th element
  - Print the entire list

Terminology for Lists

- **List:** A collection of data whose entries are arranged sequentially
- **Head:** The beginning of the list
- **Tail:** The end of the list

Storing Lists

- **Contiguous list**（邻接表）: List stored in a homogeneous array
- **Linked list**（链表）: List in which each entries are linked by pointers
  - **Head pointer:** Pointer to first entry in list
  - **NIL pointer**（空指针）: A “non-pointer” value used to indicate end of list

Figure 8.4 Novels arranged by title but linked according to authorship

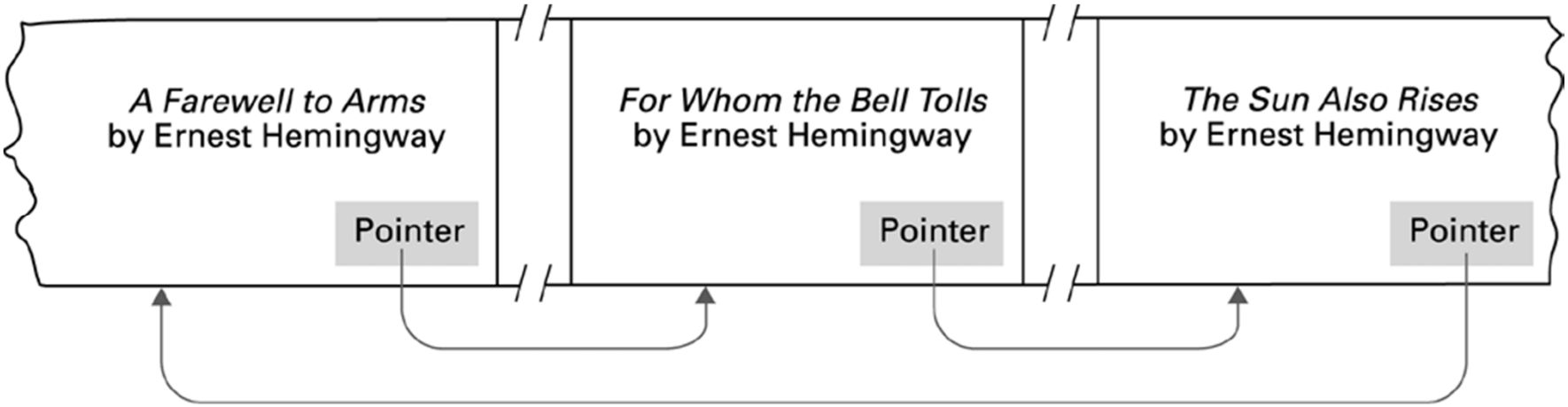


Figure 8.8 Names stored in memory as a contiguous list

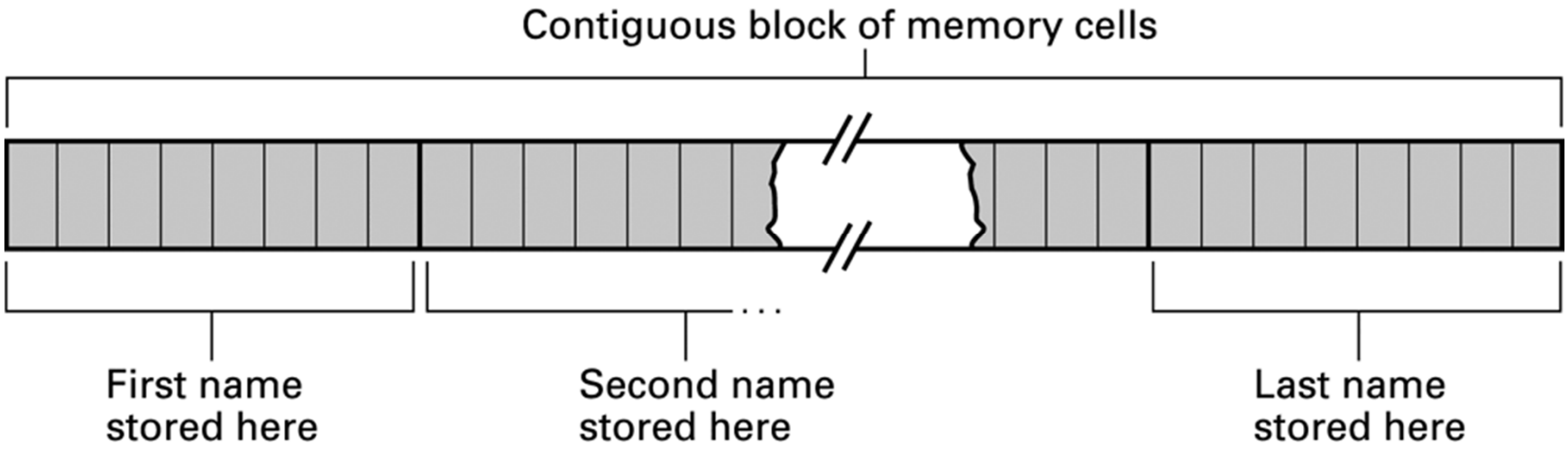


Figure 8.9 The structure of a linked list

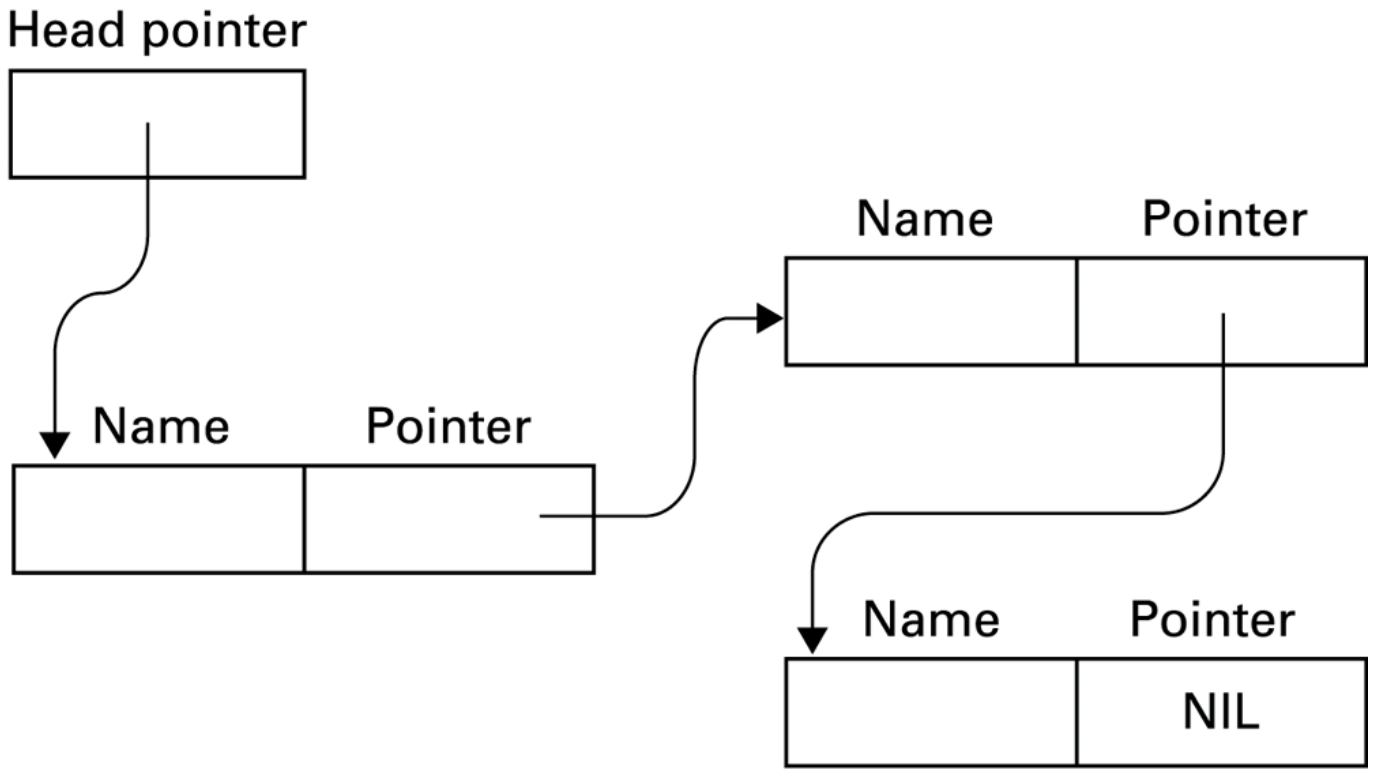


Figure 8.10 Deleting an entry from a linked list

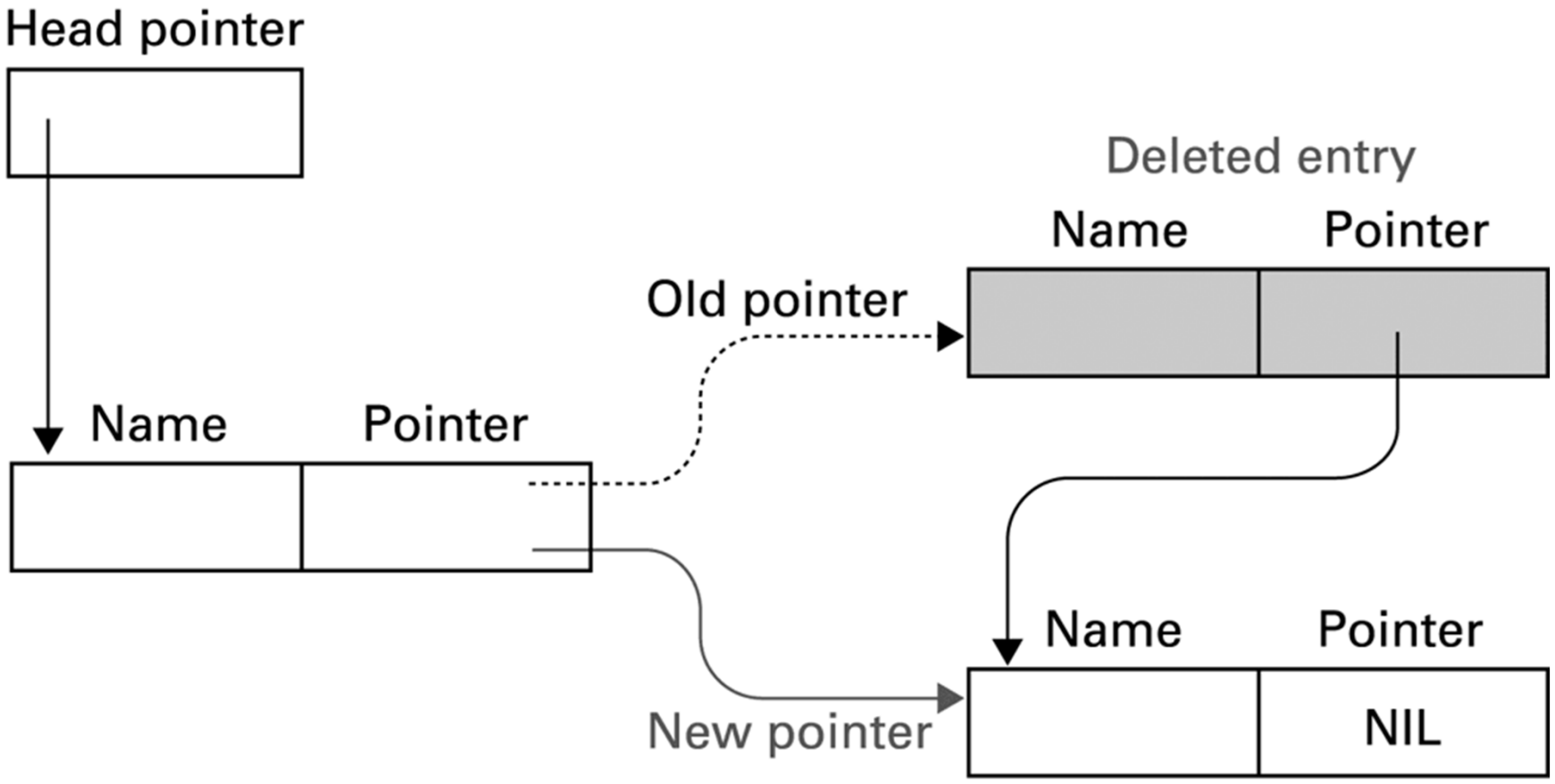
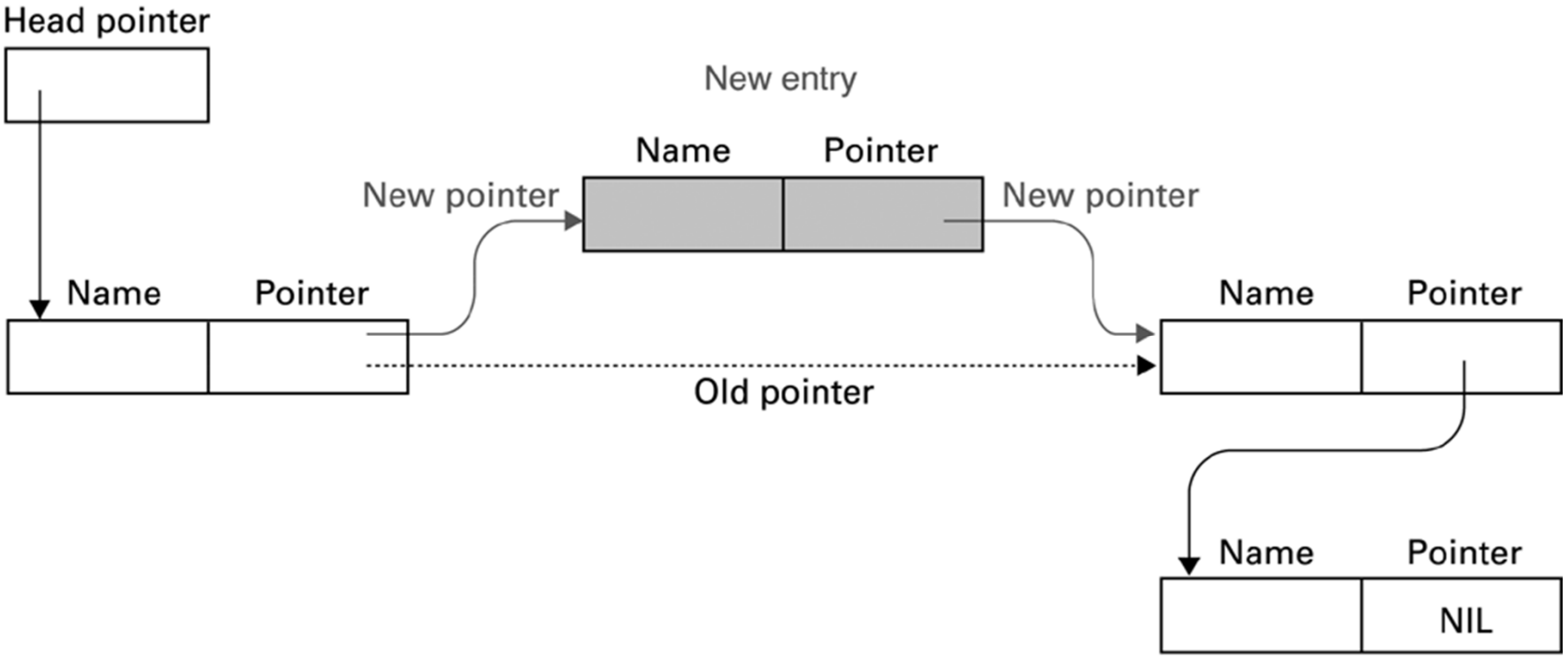
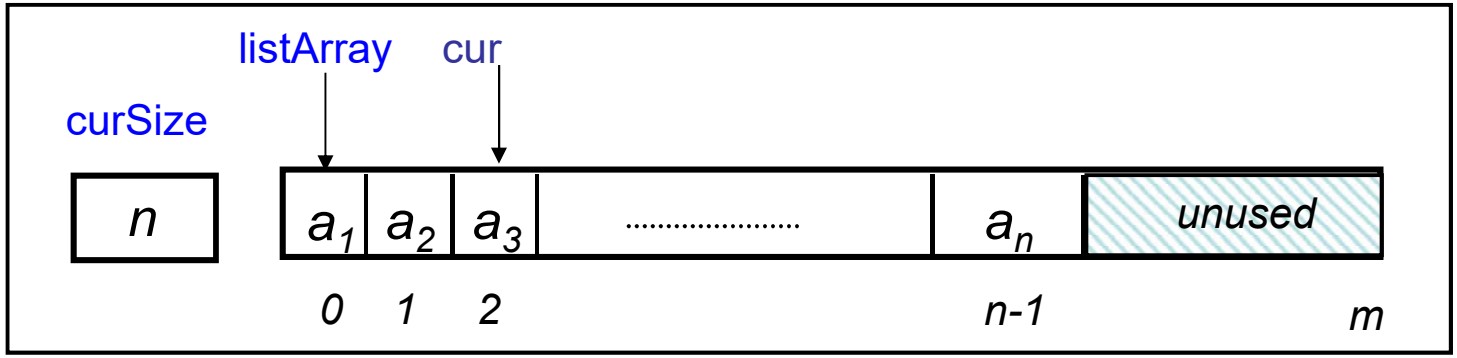


Figure 8.11 Inserting an entry into a linked list



Array-Based List Implementation

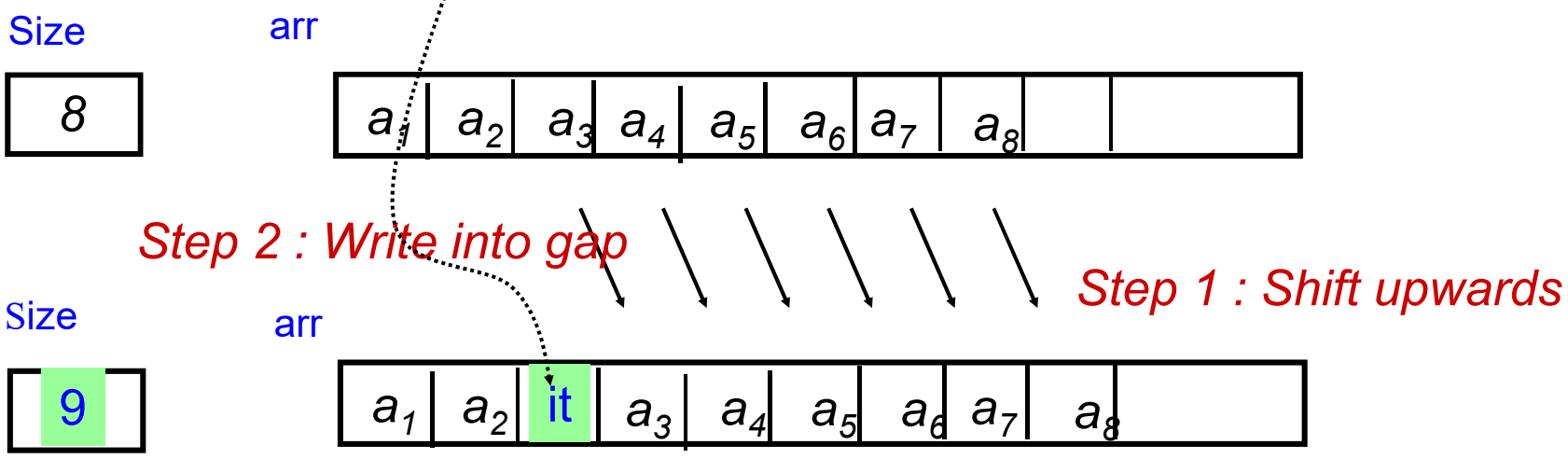
- One simple implementation is to use arrays
  - A sequence of n-elements
- Maximum size is anticipated a priori.
- Internal variables:
  - Maximum size *maxSize* (m)
  - Current size *curSize* (n)
  - Current index *cur*
  - Array of elements listArray



Inserting Into an Array

- While retrieval is very fast, insertion and deletion are very slow
  - Insert has to shift upwards to create gap

Example : insert(2, it, arr)



Step 3 : Update Size

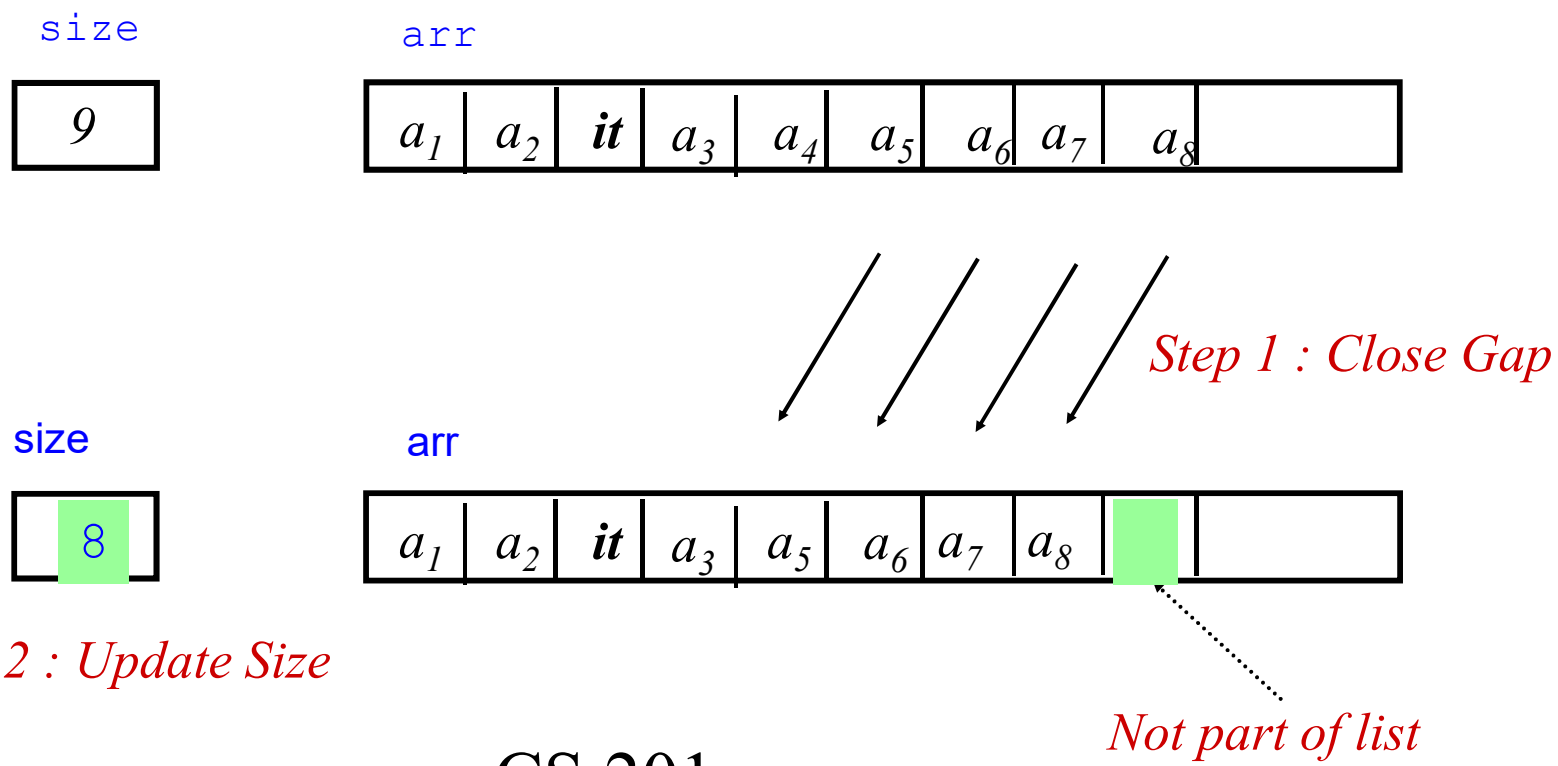
Coding

```
typedef struct {
    int arr[MAX];
    int max;
    int size;
} LIST


void insert(int j, int it, LIST *pl)
  { // pre : 1<=j<=size+1

      int i;

       for (i=pl->size; i>=j; i=i-1)
                                  // Step 1: Create gap
         { pl->arr[i+1]= pl->arr[i]; };

       pl->arr[j]= it;          // Step 2: Write to gap

       pl->size = pl->size + 1; // Step 3: Update size
  }
```

Deleting from an Array

- Delete has to shift downwards to close gap of deleted item

Example: `deleteItem(4, arr)`

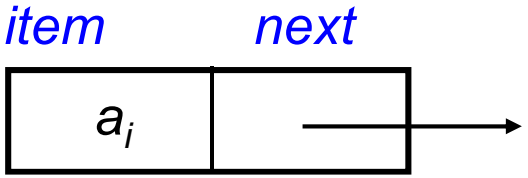


Coding

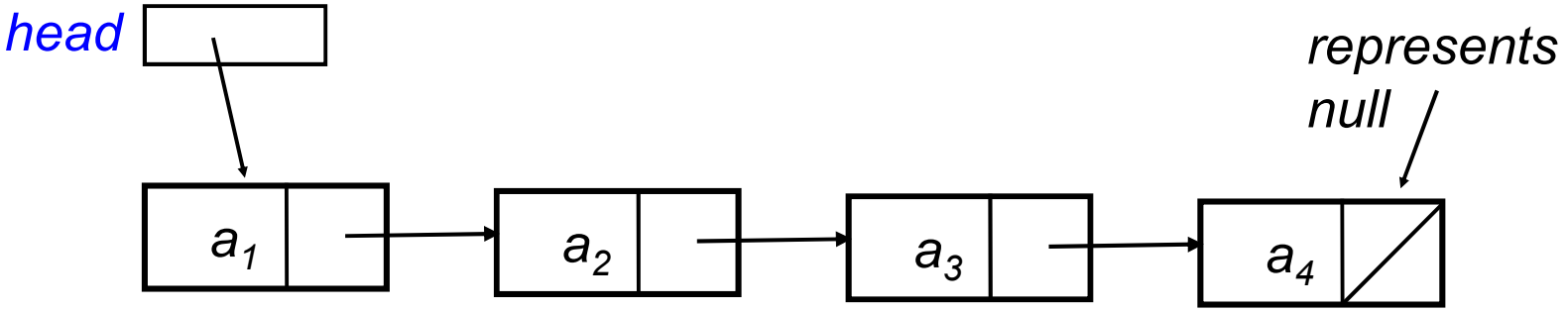
```
void delete(int j, LIST *pl)
{                                  // pre : 1<=j<=size
   for (i=j+1; i<=pl->size; i=i+1)
                                   // Step1: Close gap
     { pl->arr[i-i]=pl->arr[i]; };
                                   // Step 2: Update size
     pl->size = pl->size - 1;
   }
```

Linked List Approach

- Main problem of array is the slow deletion/insertion since it has to shift items in its *contiguous* memory
- **Solution**: linked list where items need *not be contiguous* with nodes of the form



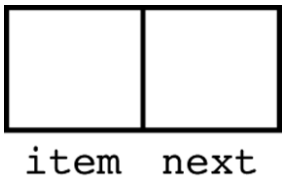
- Sequence (list) of four items < a_1, a_2, a_3, a_4 > can be represented by:



Pointer-Based Linked Lists

- A node in a linked list is usually a `struct`

```
struct Node
{  int item
   Node *next;
}; //end struct
```



A node

- A node is dynamically allocated

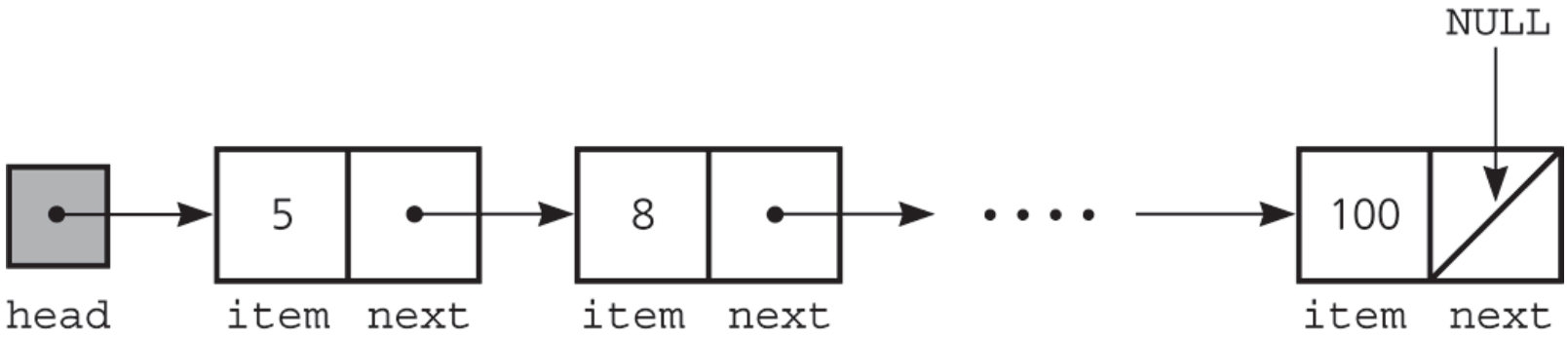
```
Node *p;
p = malloc(sizeof(Node));//申请分配
```

node 这个结构体占用的内存空间，首地址放入指针变量p

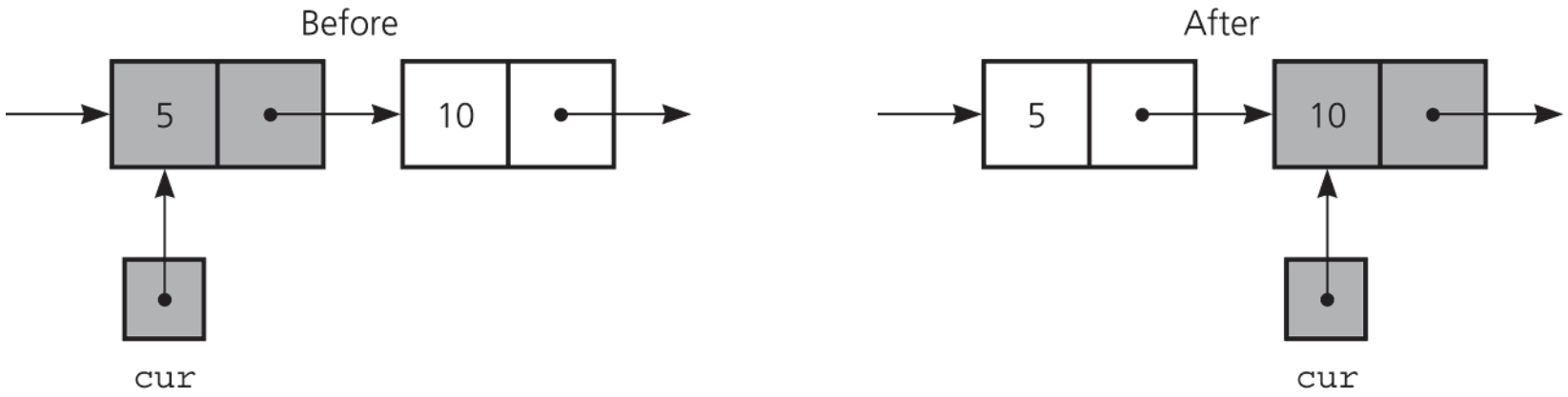
Pointer-Based Linked Lists

- The head pointer points to the first node in a linked list
- If head is *NULL*, the linked list is empty
  - `head=NULL`
- `head=malloc(sizeof(Node))`

A Sample Linked List



Traverse遍历 a Linked List



The effect of the assignment `cur = cur->next`

Traverse遍历 a Linked List

- Reference a node member with the -> operator

```
p->item;
```

- A traverse operation visits each node in the linked list
  - A pointer variable cur keeps track of the current node

```
for (Node *cur = head;
     cur != NULL; cur = cur->next)
    x = cur->item;
```