Chapter 8:
Data Abstractions

Computer Science: An Overview
Tenth Edition

by
J. Glenn Brookshear

Chapter 8: Data Abstractions

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

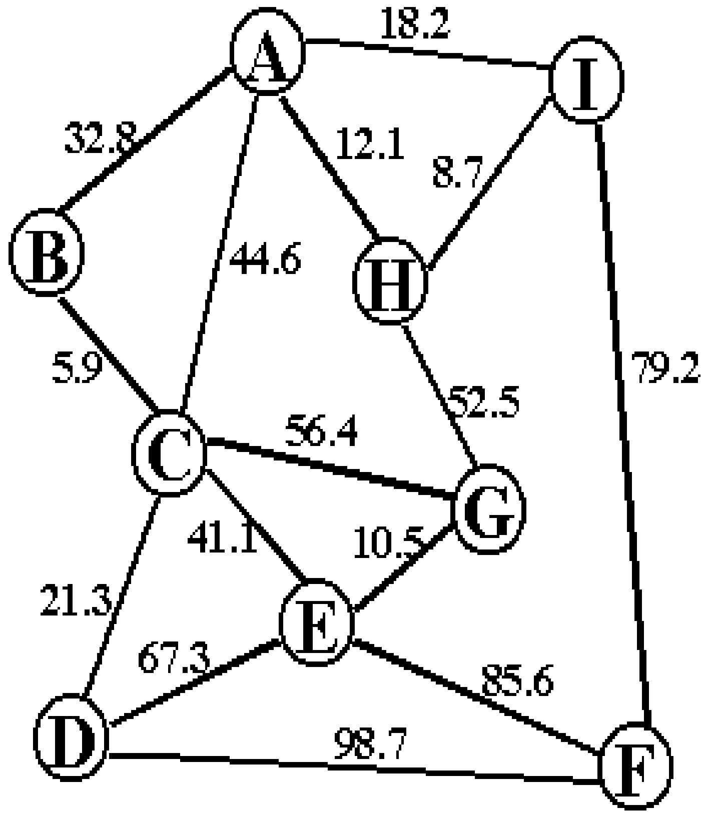
课程编号	课程名称	先修课程
C_1	计算机导论	无
C_2	数据结构	C_1，C_4
C_3	汇编语言	C_1
C_4	C程序设计语言	C_1
C_5	计算机图形学	C_2，C_3，C_4
C_6	接口技术	C_3
C_7	数据库原理	C_2，C_9
C_8	编译原理	C_4
C_9	操作系统	C_2

（a）计算机专业的课程设置

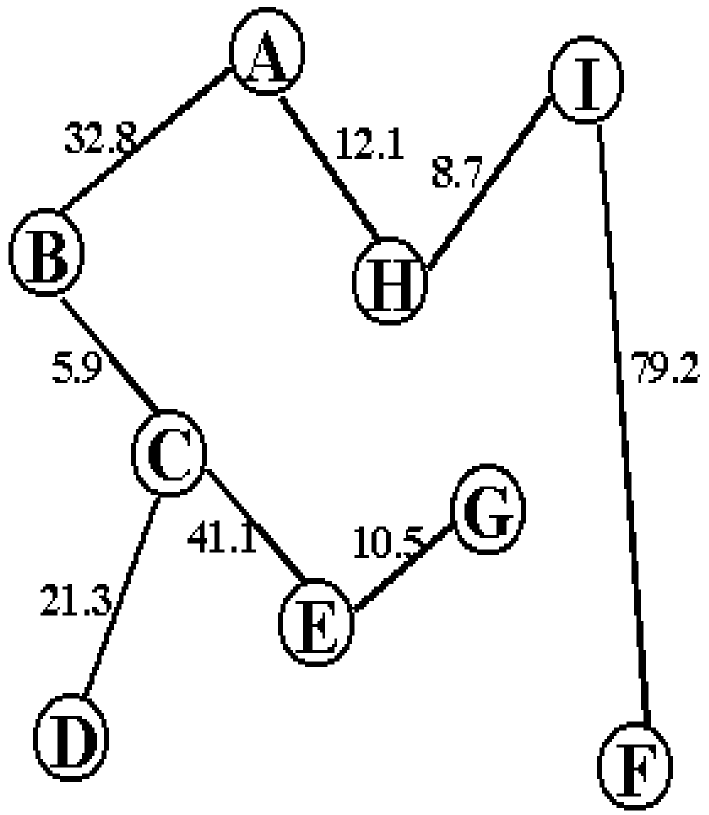
$$\begin{tikzcd}
 & C_3 \arrow[r] \arrow[drr] & C_6 & \\
C_1 \arrow[ur] \arrow[drr] \arrow[ddr] & & & C_5 \\
 & & C_2 \arrow[ur] \arrow[r] \arrow[ddr] & C_9 \arrow[dd] \\
 & C_4 \arrow[uurr] \arrow[ur] \arrow[dr] & & \\
 & & C_8 & C_7
\end{tikzcd}$$

(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

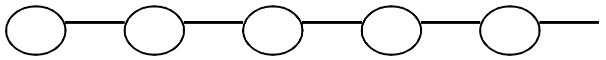
数据结构由某一数据对象及该对象中所有数据成员之间的关系组成。记为：

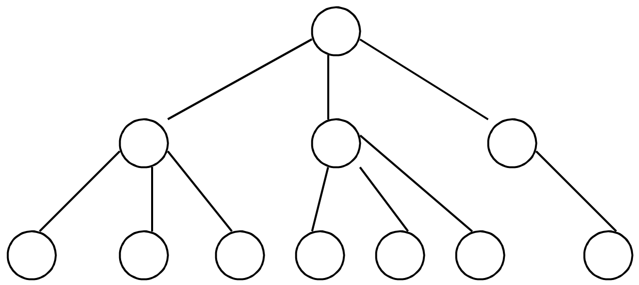
Data_Structure = {D, R}

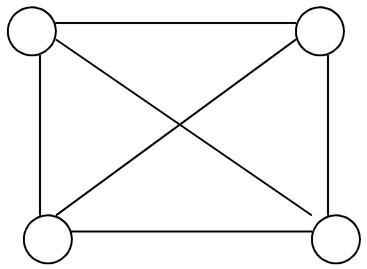
其中，**D**是某一数据对象，**R**是该对象中所有数据成员之间的关系的有限集合。

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

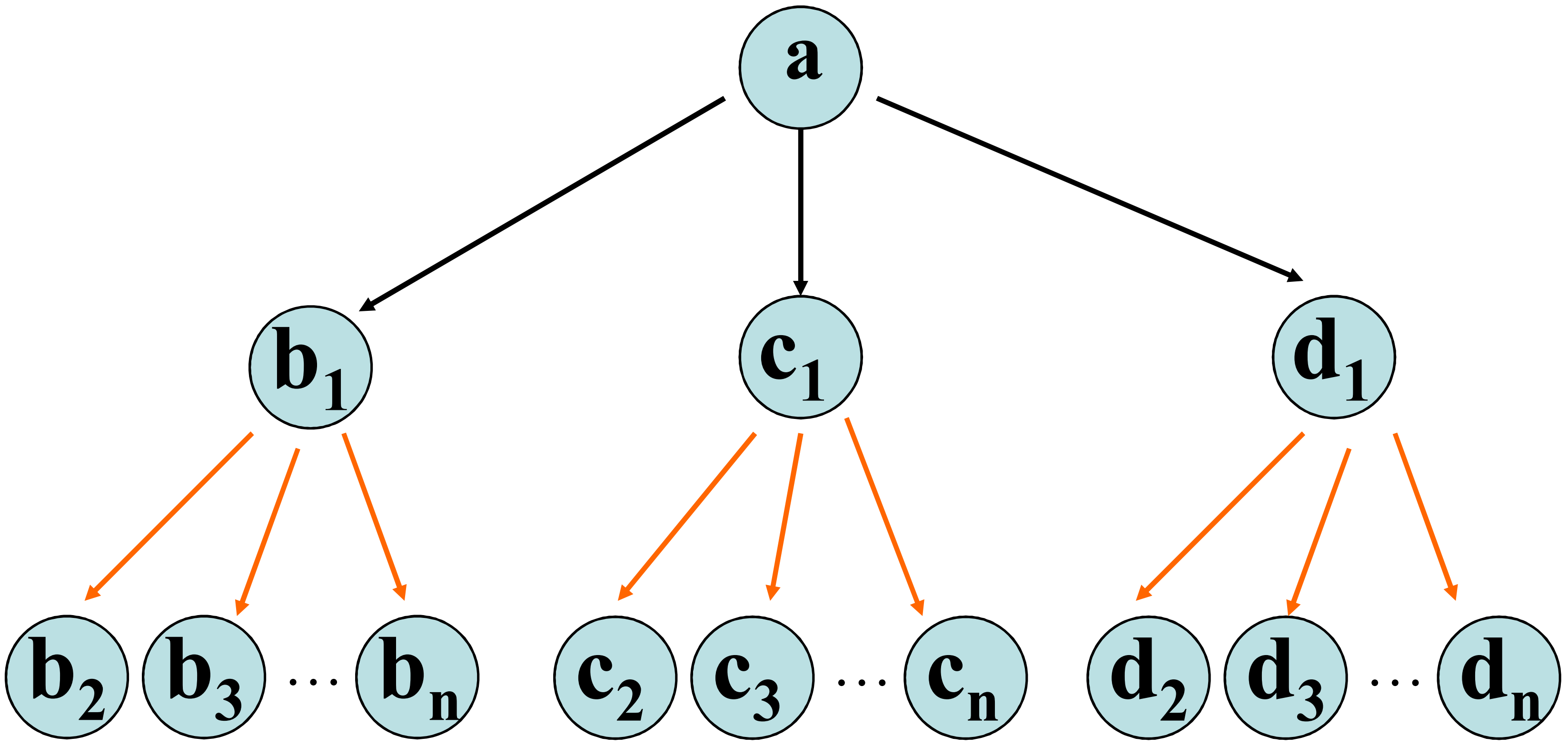
关系集合：$S = \{ R_1, R_2 \}$

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

//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



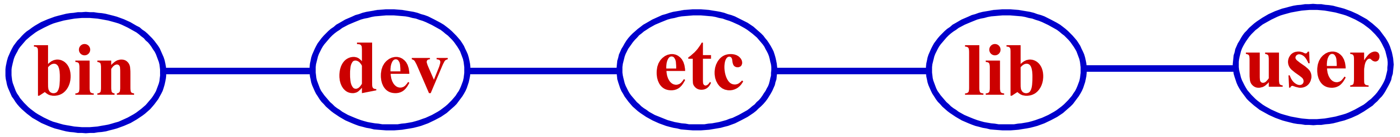
班级Class的逻辑结构的图示

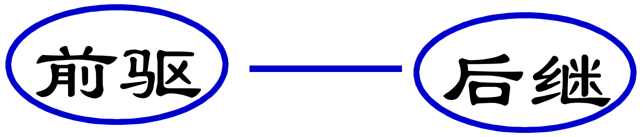
数据结构的分类

- 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

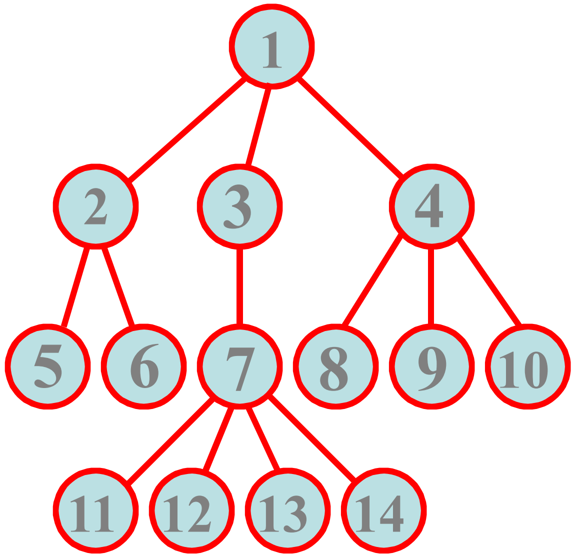
线性结构



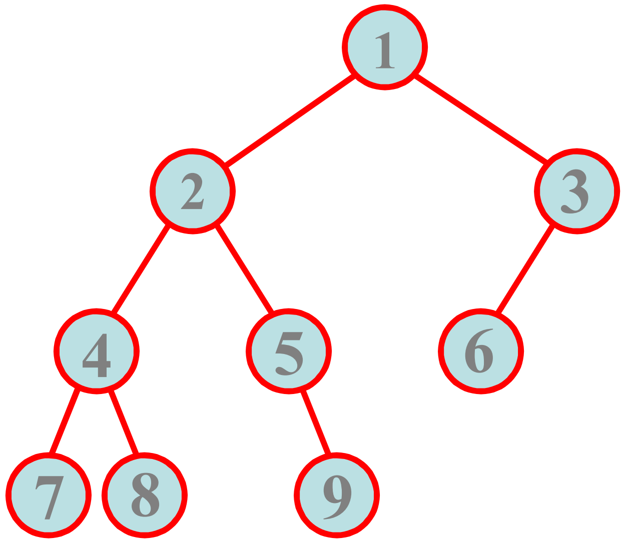


非线性结构——层次结构

树



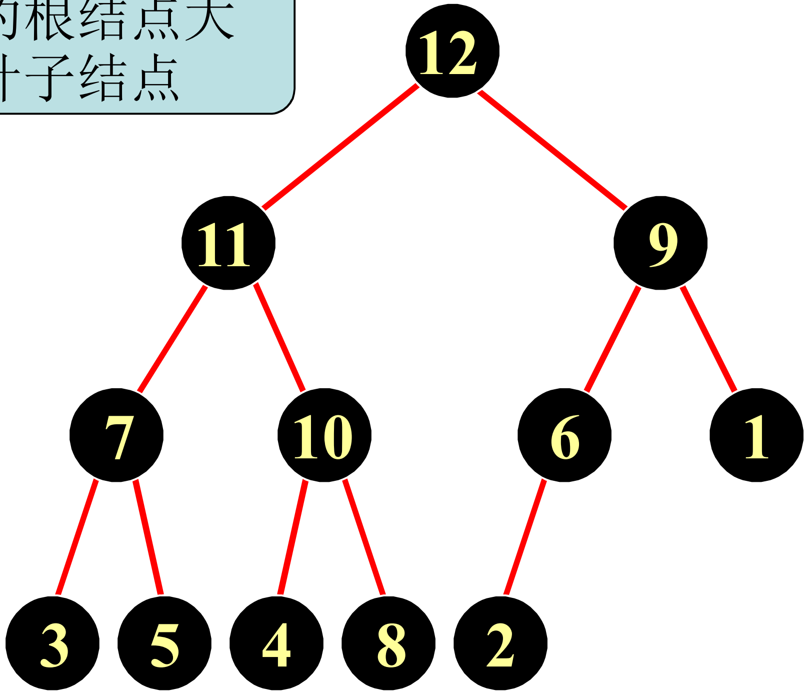
二叉树

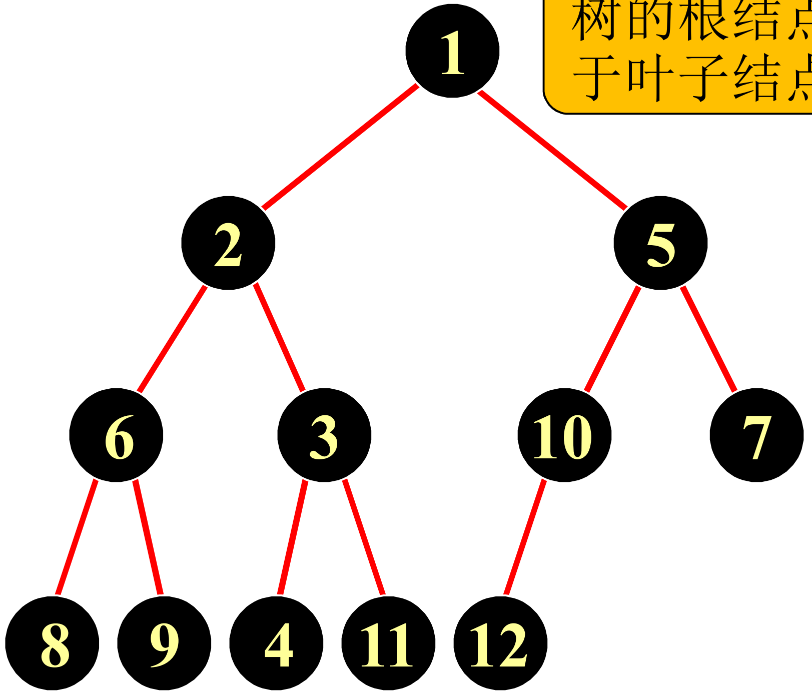


堆（特殊的树结构）

树的根结点大于叶子结点

树的根结点小于叶子结点

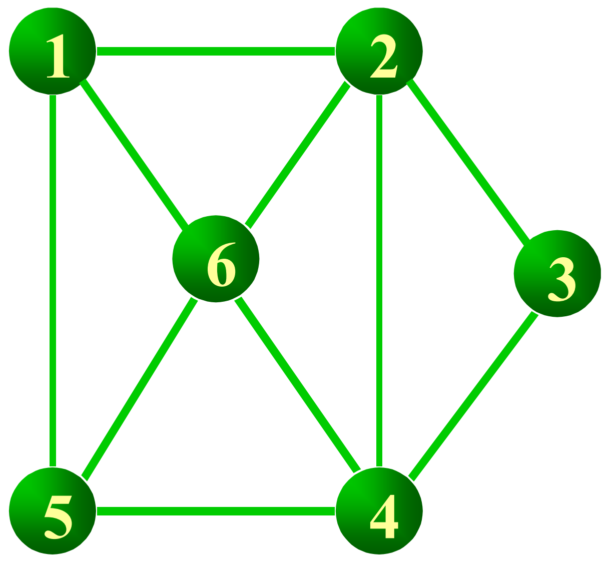


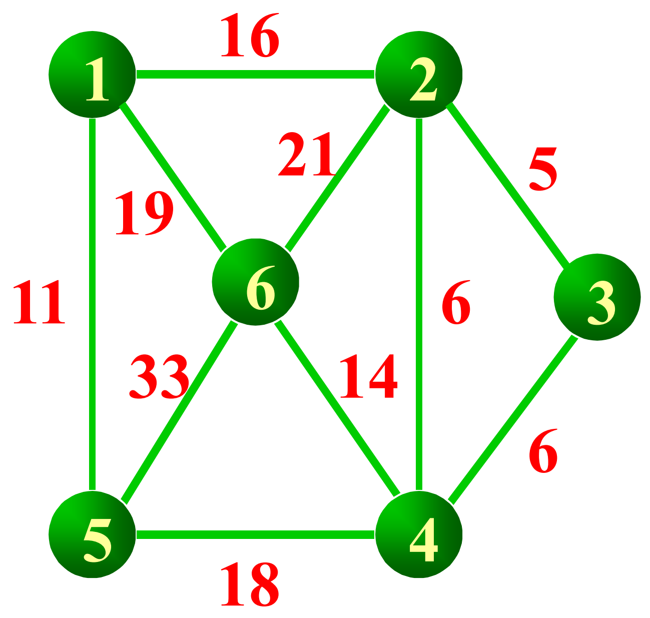


“最大”堆

“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：

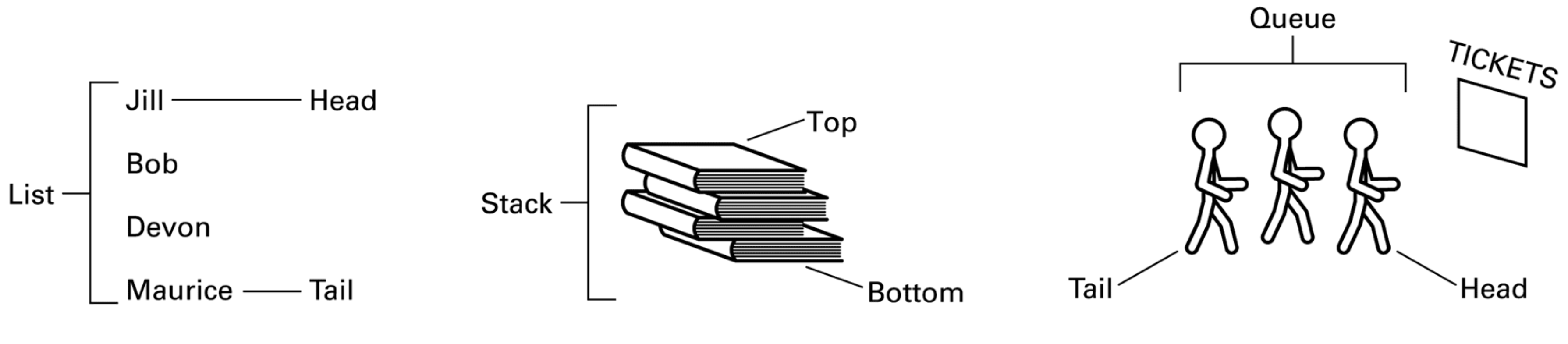
    使用公共界面的接口对抽象数据类型进行操作，不需要考虑其物理实现。对于外部用户来说，抽象数据类型应该是一个黑盒子。

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）。



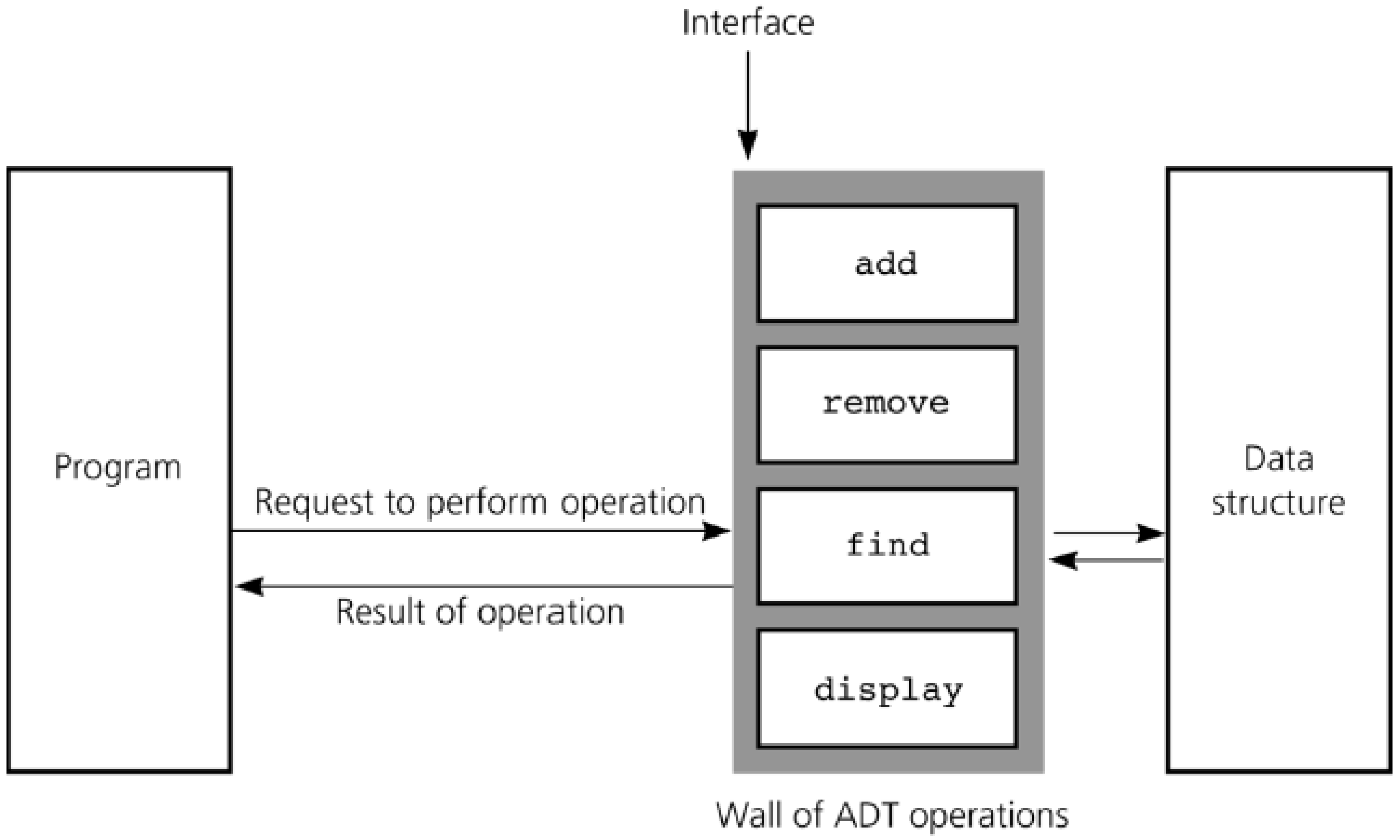
a. A list of names **b.** A stack of books **c.** A queue of people

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**

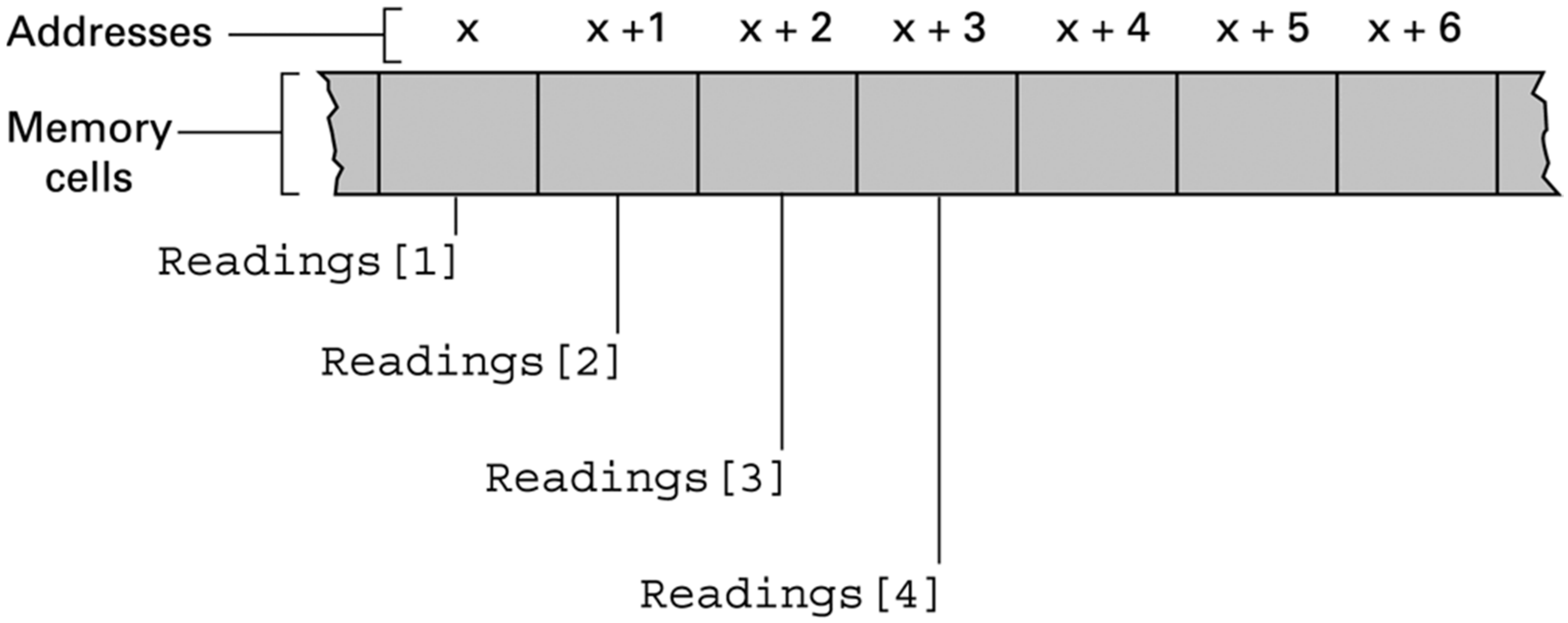
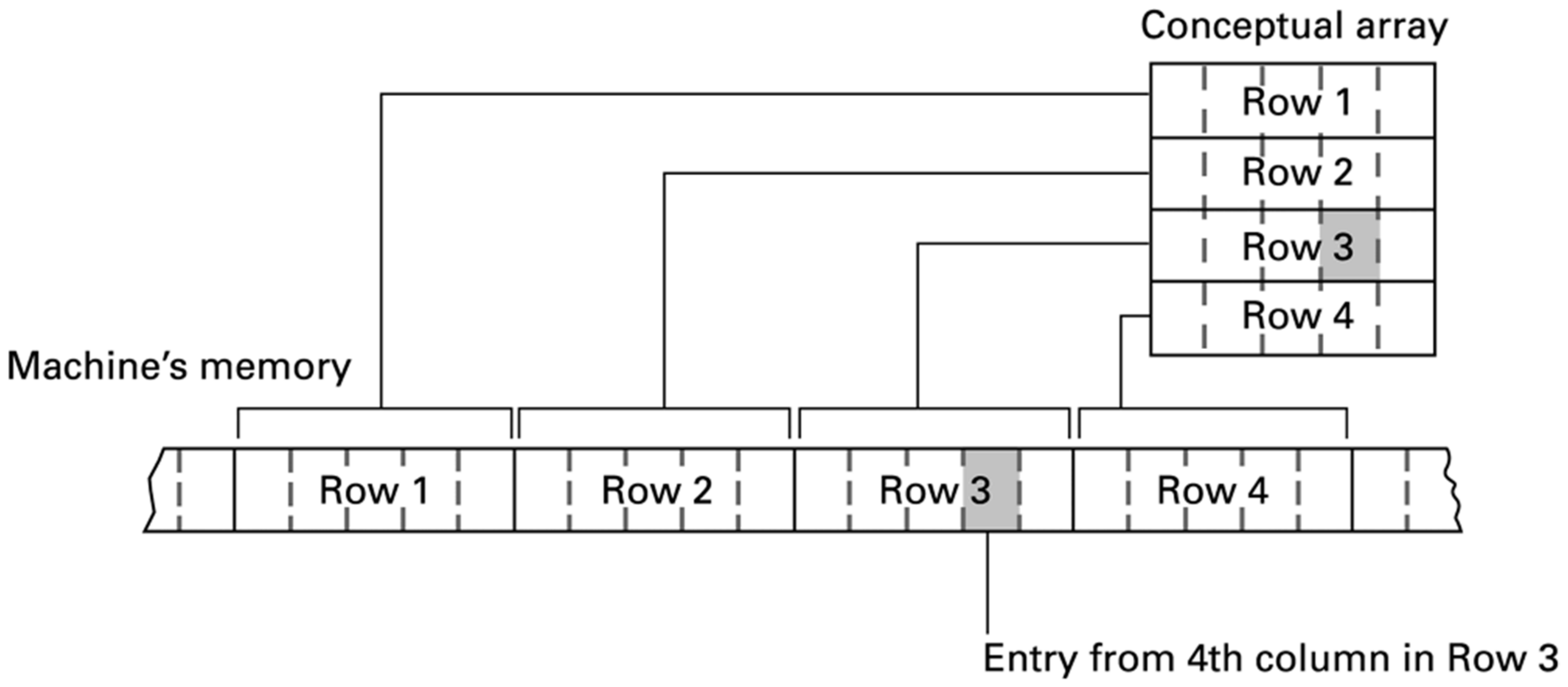


Figure 8.6 **A two-dimensional array with four rows and five columns stored in row major order**



Homogeneous array

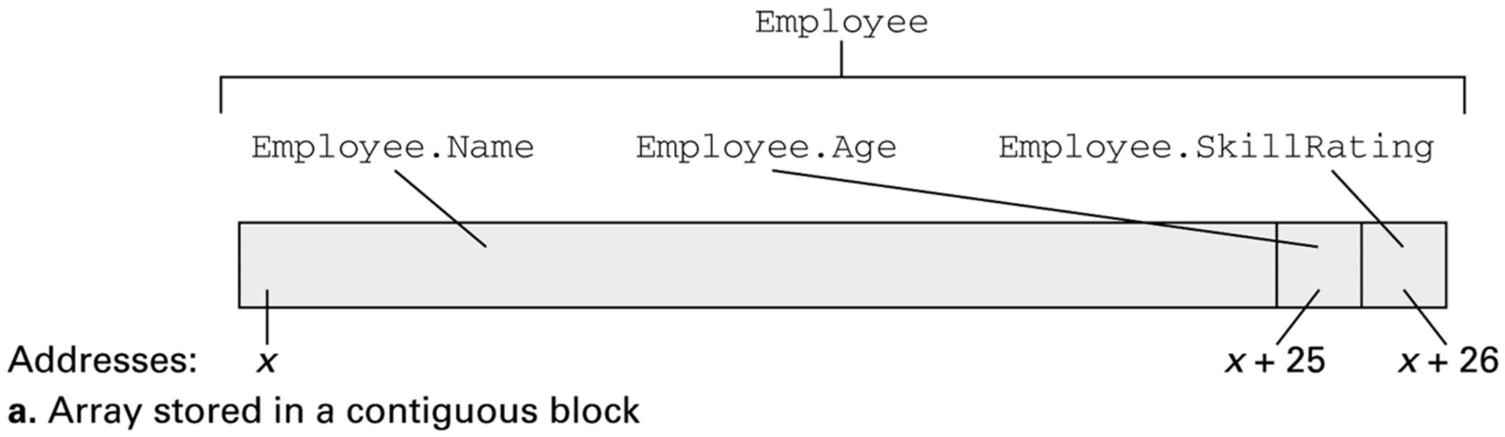
```
/* 程序 1：求 10 个数之和 */
#include <stdio.h>
main()
{
 int i,s=0;
 int a[10]={66,55,75,42,86,77,96, 89,78,56};
 for(i=0;i<10;i++)
   s=s+a[i];
 printf(“%d”,s);
}
```

```
/* 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);
```

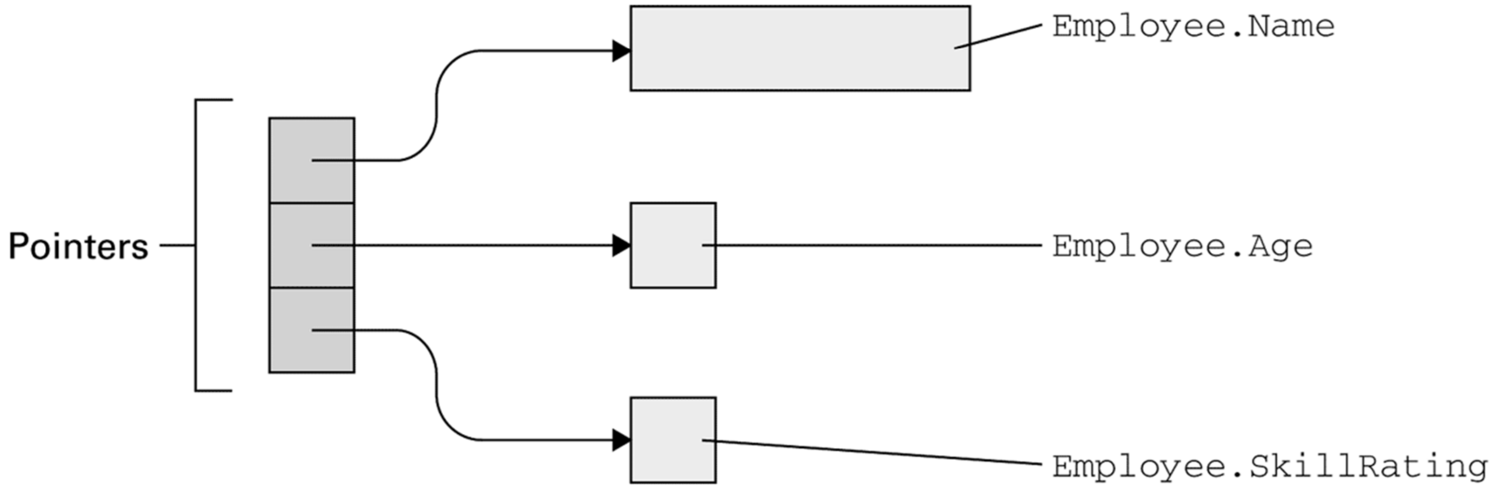
Limitations of arrays

- 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

```
#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**

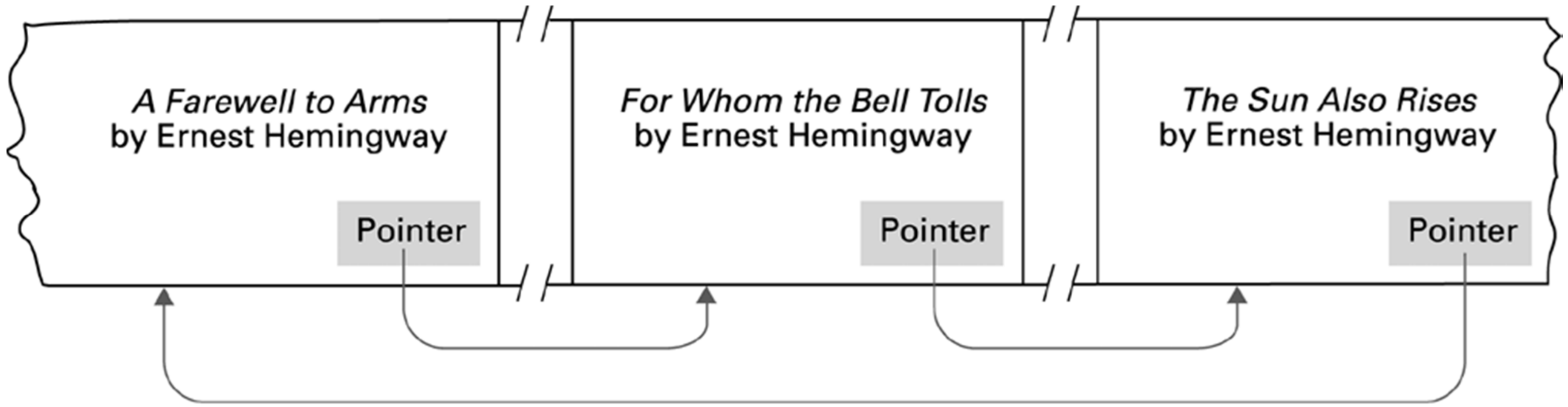


Figure 8.8 **Names stored in memory as a contiguous list**

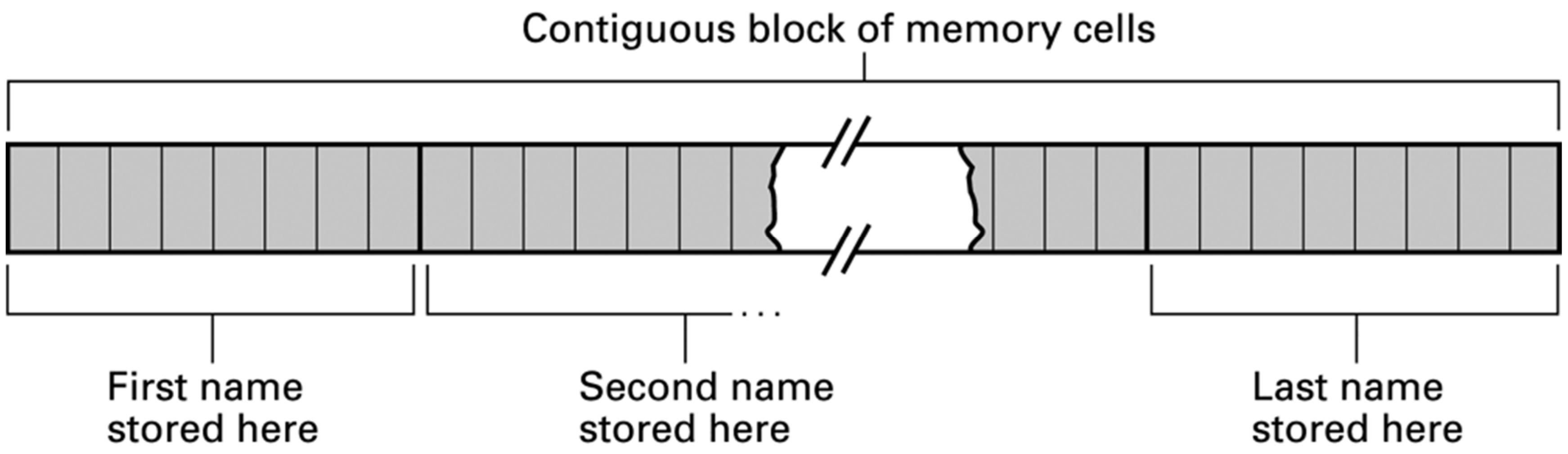


Figure 8.9 The structure of a linked list

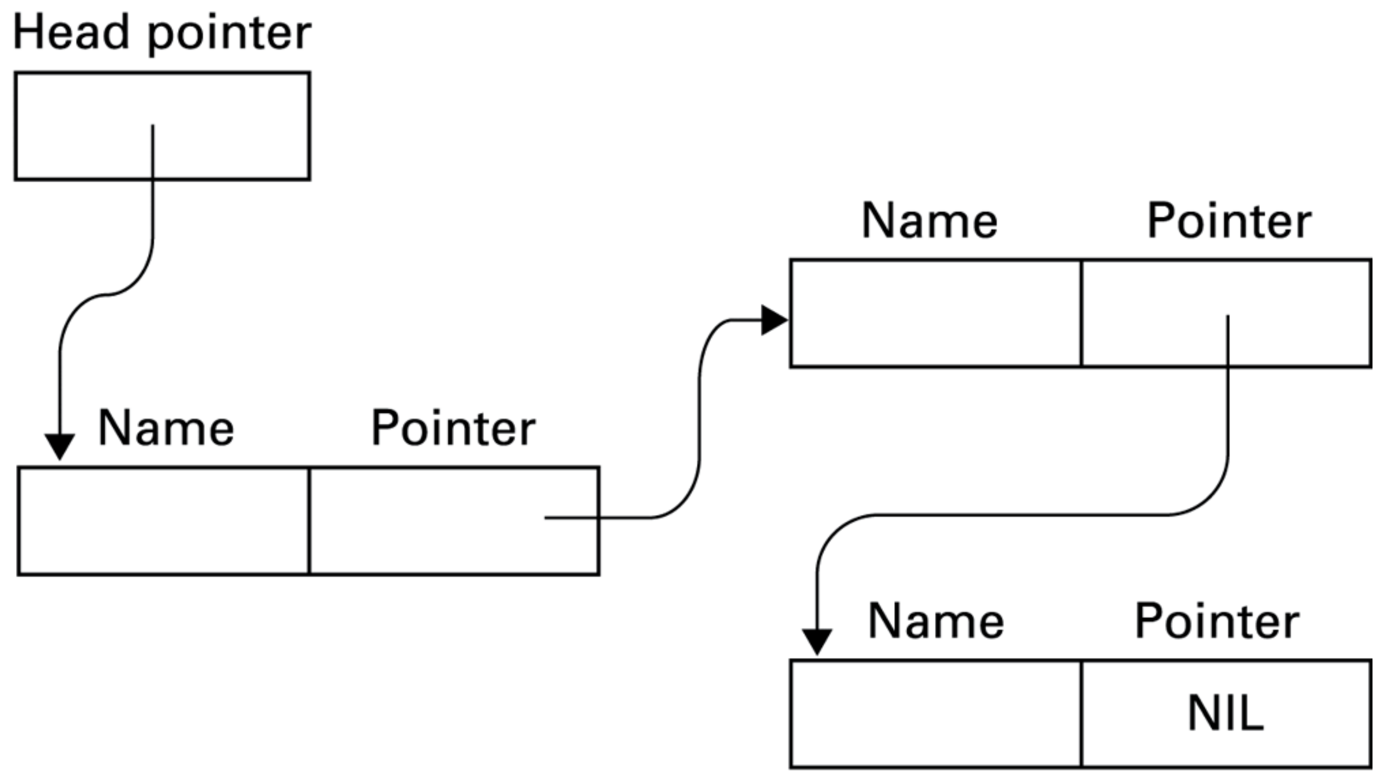


Figure 8.10 **Deleting an entry from a linked list**

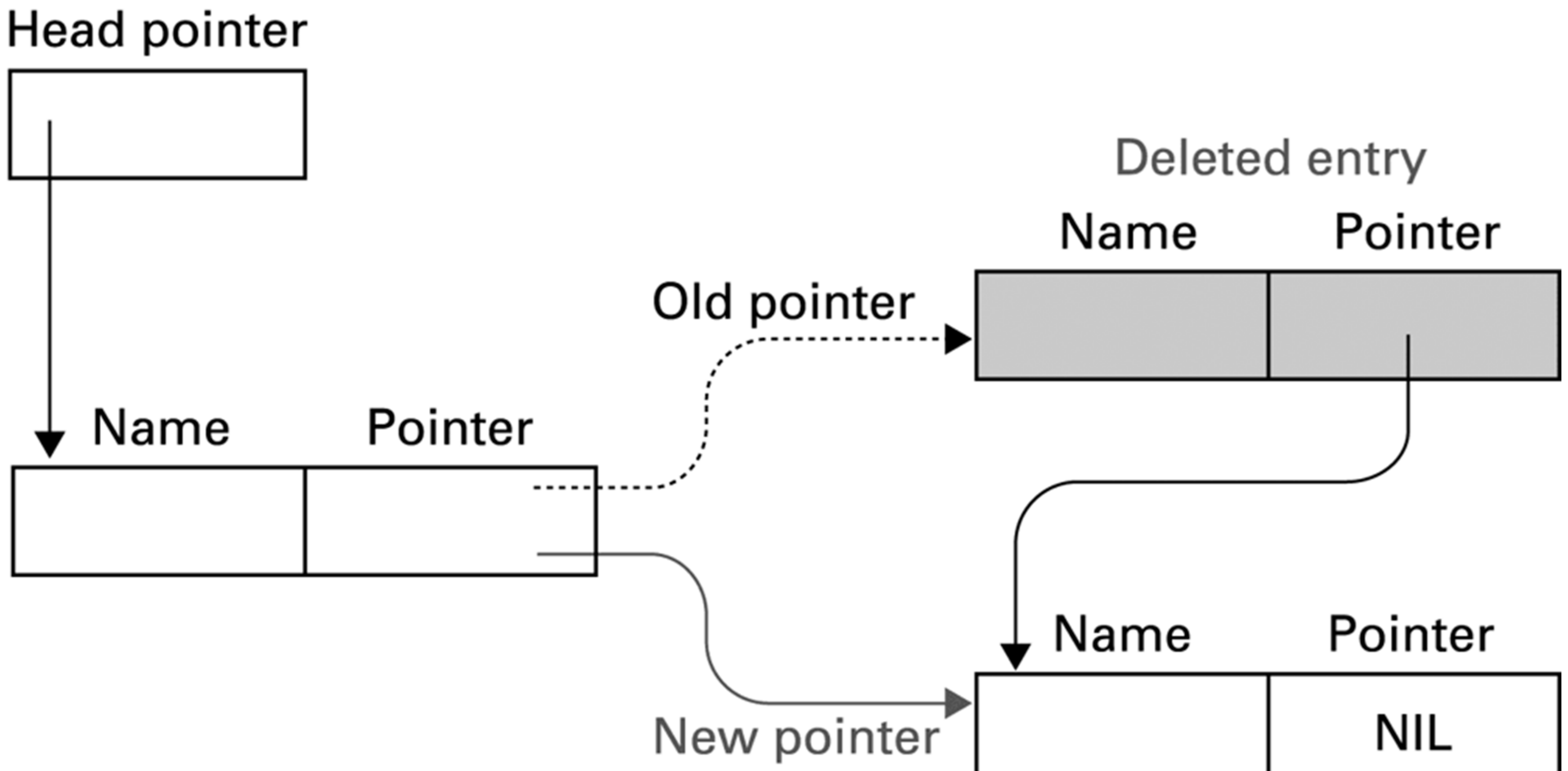
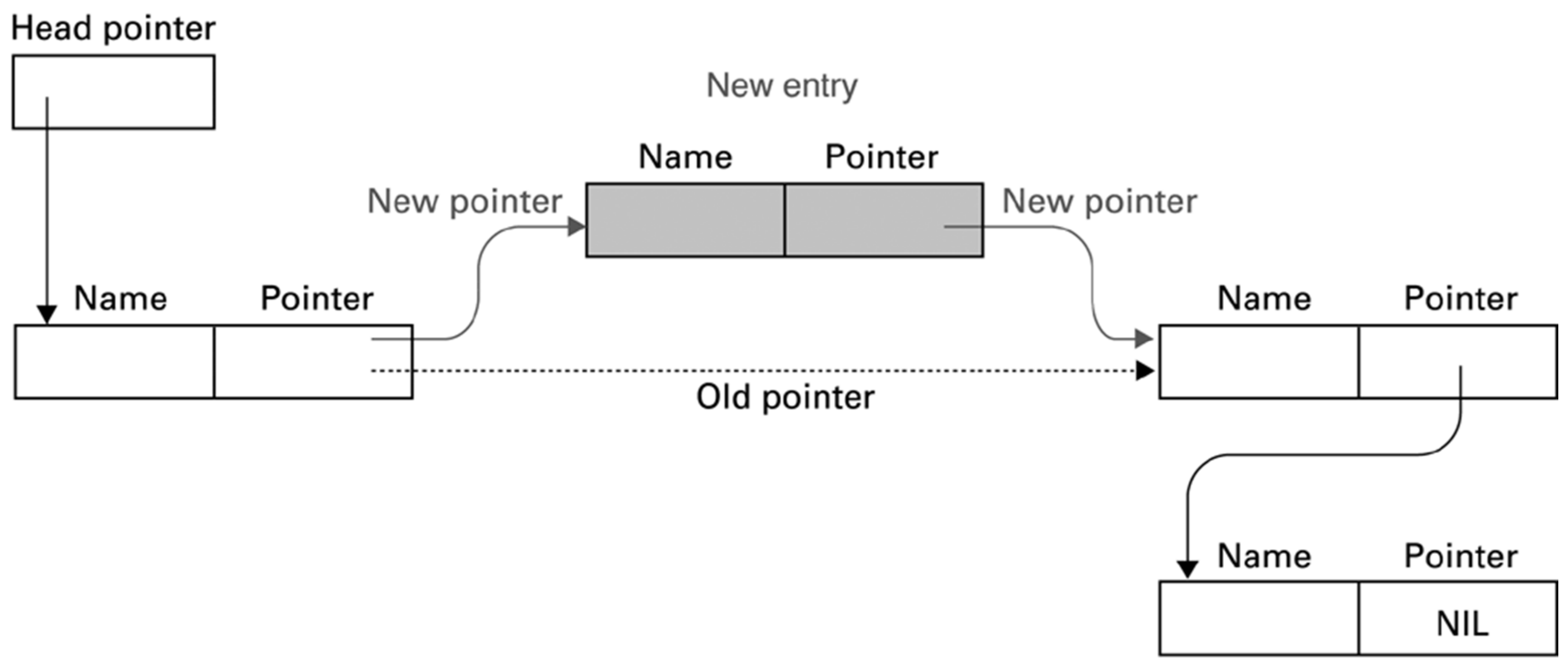
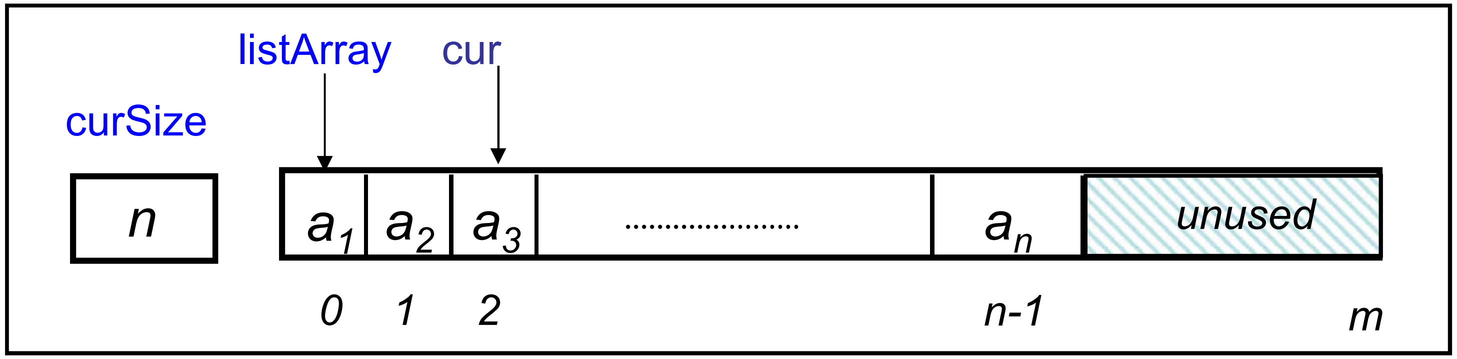


Figure 8.11 Inserting an entry into a linked list



Array-Based List Implementation

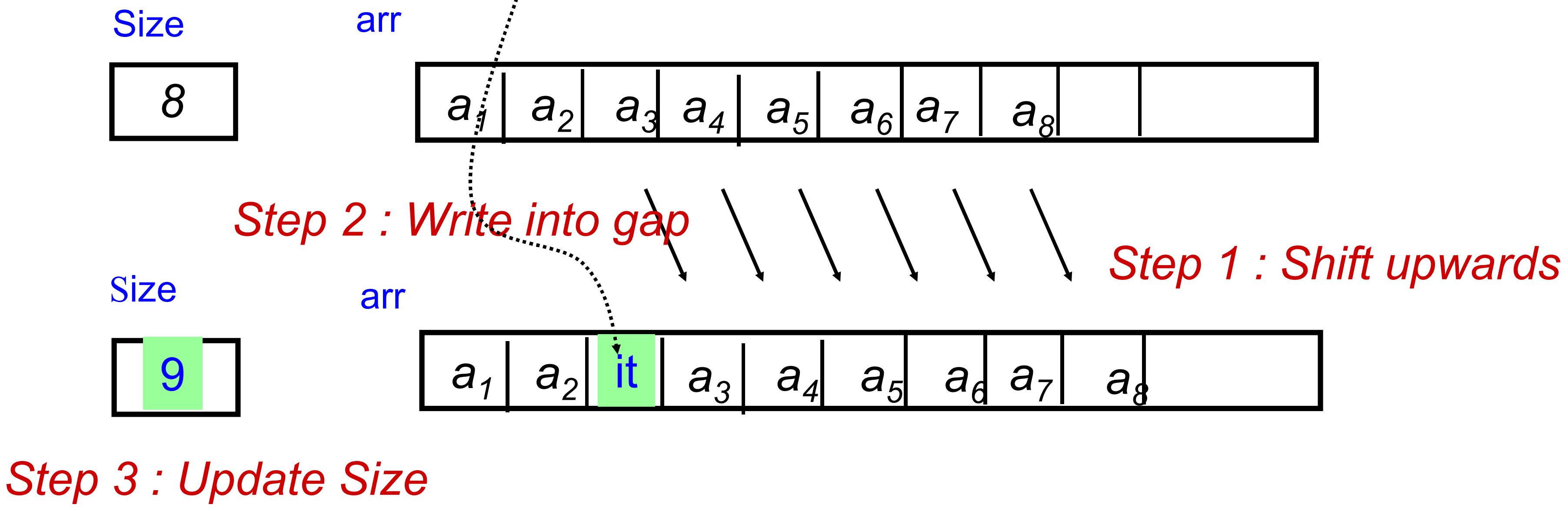
- One simple implementation is to use arrays
  - A sequence of n-elements
- Maximum size is anticipated a priori.
- Internal variables:
  - Maximum size *maxSize* (m)
  - Current size *curSize* (n)
  - Current index *cur*
  - Array of elements listArray



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)



Size
arr
8
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
Step 2 : Write into gap
Step 1 : Shift upwards
Size
arr
9
a_1 a_2 it a_3 a_4 a_5 a_6 a_7 a_8
Step 3 : Update Size

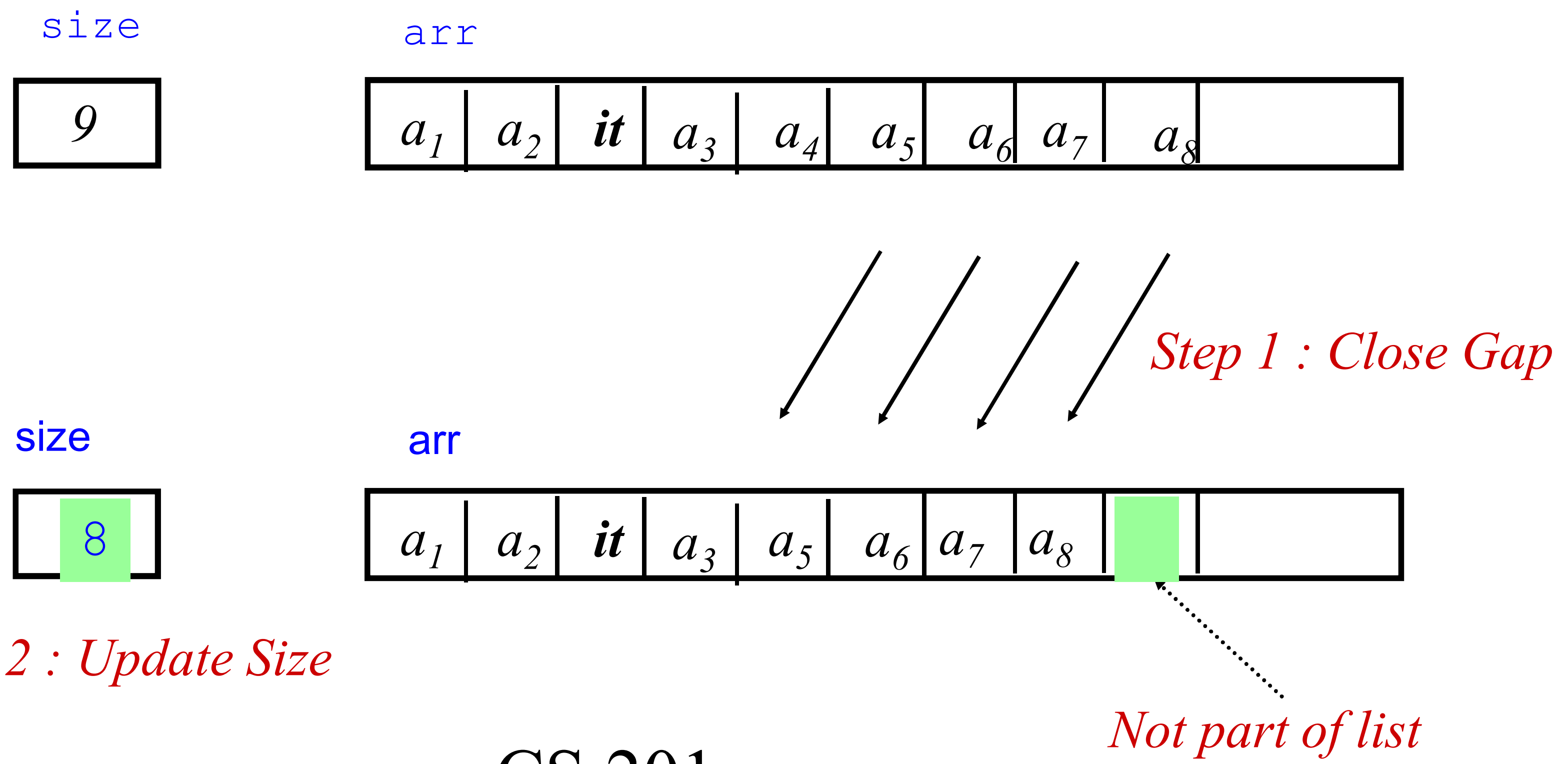
Coding

```
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)`



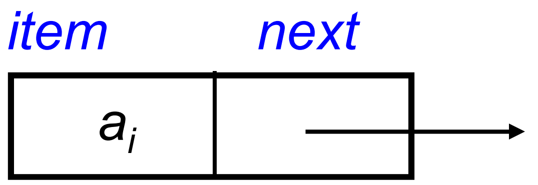
size
arr
9
a_1 a_2 it a_3 a_4 a_5 a_6 a_7 a_8
Step 1 : Close Gap
size
arr
8
a_1 a_2 it a_3 a_5 a_6 a_7 a_8
Step 2 : Update Size
Not part of list

Coding

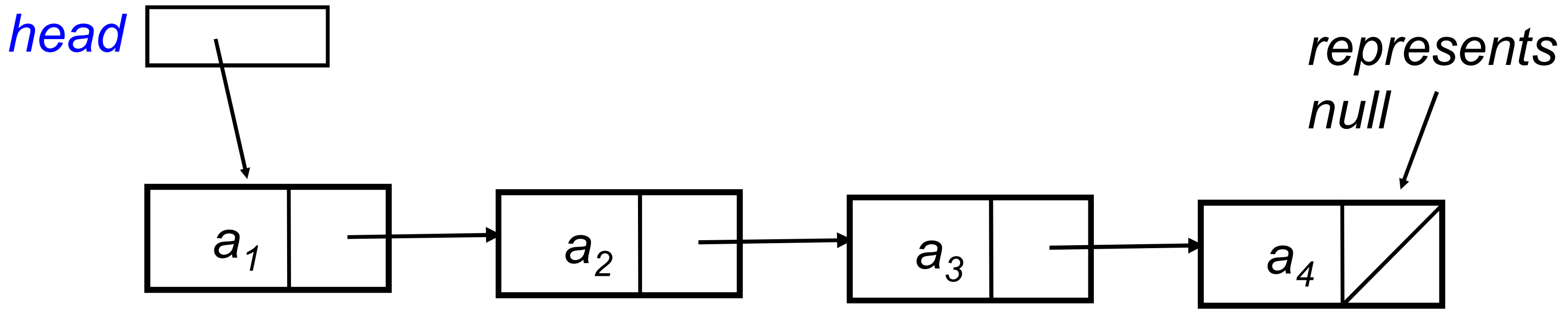
```
void delete(int j, LIST *pl)
{                                  // pre : 1<=j<=size
   for (i=j+1; i<=pl->size; i=i+1)
                                   // Step1: Close gap
      { pl->arr[i-i]=pl->arr[i]; };
                                   // Step 2: Update size
     pl->size = pl->size - 1;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

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



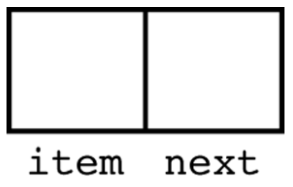
- Sequence (list) of four items < a_1, a_2, a_3, a_4 > can be represented by:



Pointer-Based Linked Lists

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

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



A node

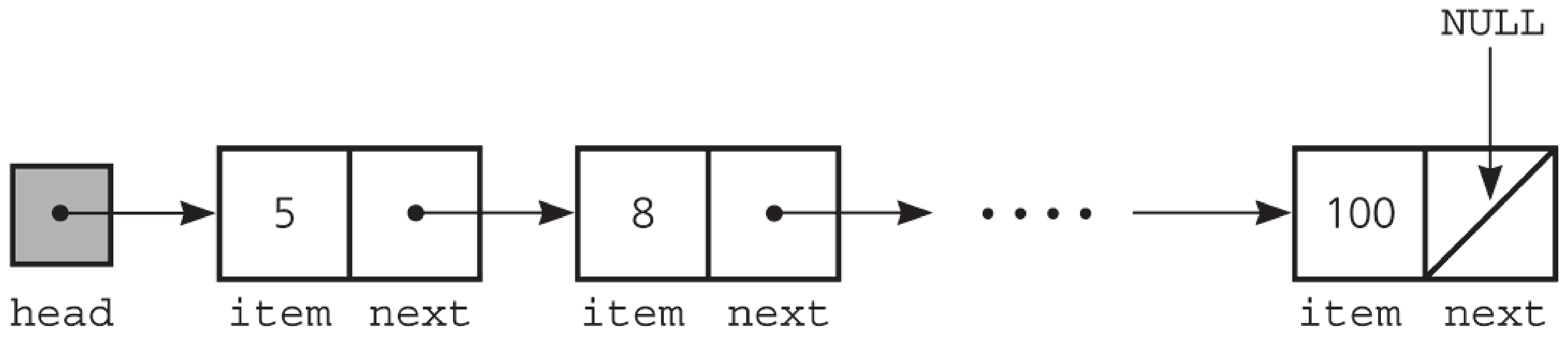
- A node is dynamically allocated

```
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

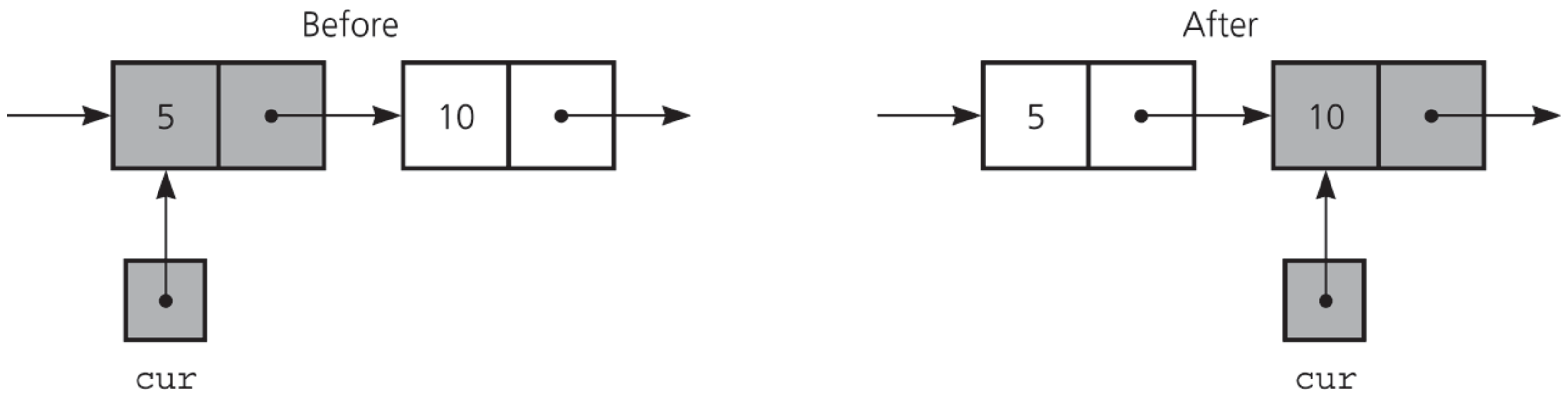
Pointer-Based Linked Lists

- 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

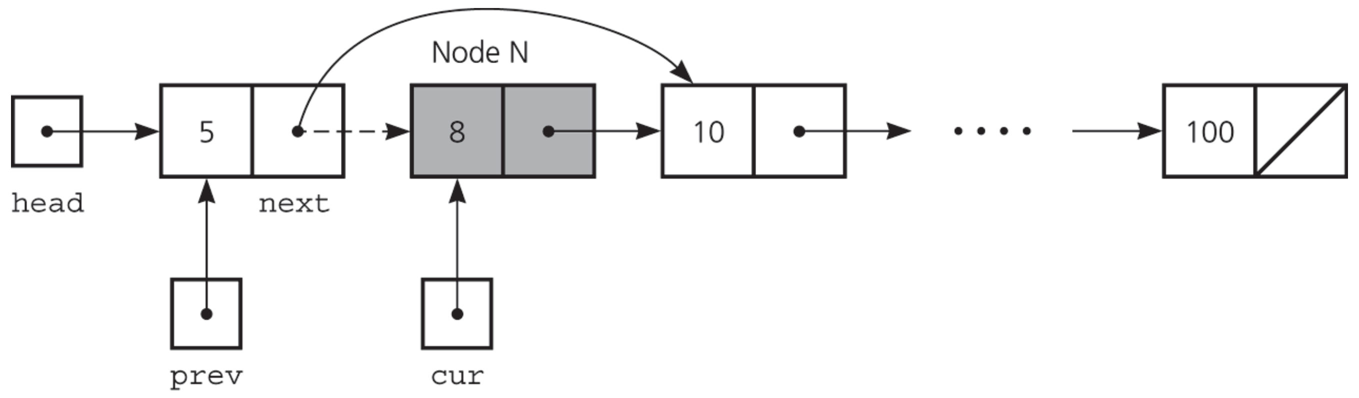
- Reference a node member with the -> operator

```
p->item;
```

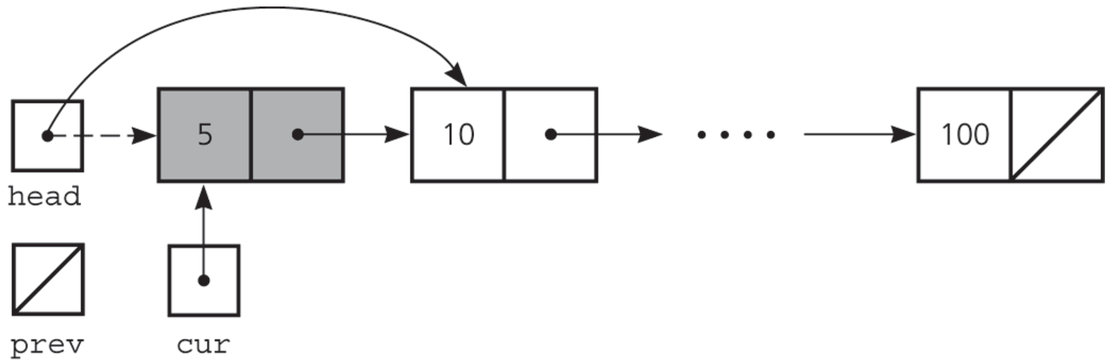
- A traverse operation visits each node in the linked list
  - A pointer variable cur keeps track of the current node

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

Delete a Node from a Linked List



Deleting a node from a linked list



Deleting the first node

Delete a Node from a Linked List

- Deleting an interior/last node

```
prev->next=cur->next;
```

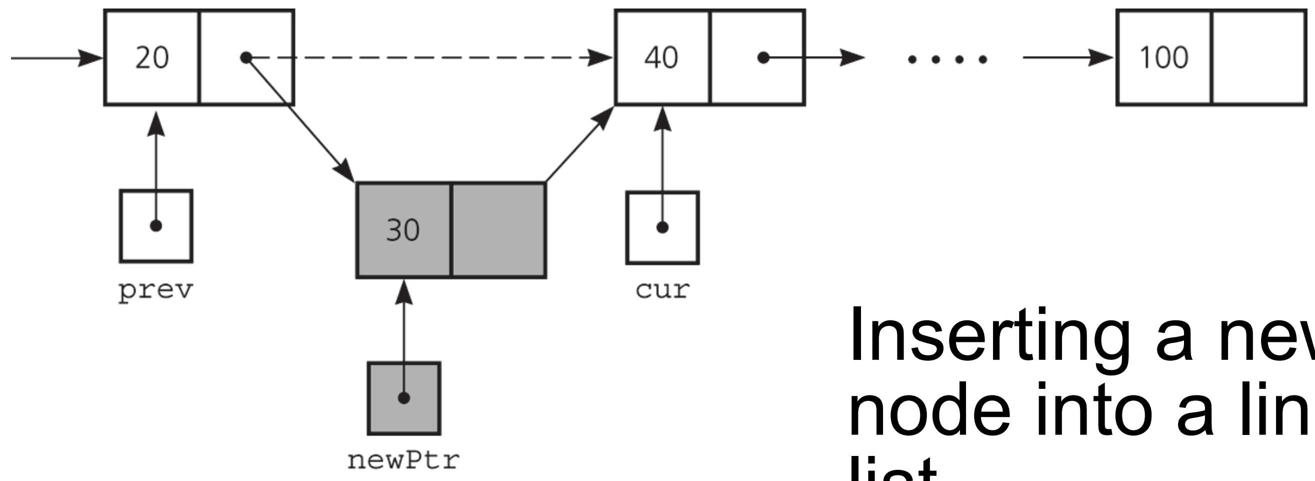
- Deleting the first node

```
head=head->next;
```

Insert a Node into a Linked List

- To insert a node between two nodes

```
newPtr->next = cur;
prev->next = newPtr;
```

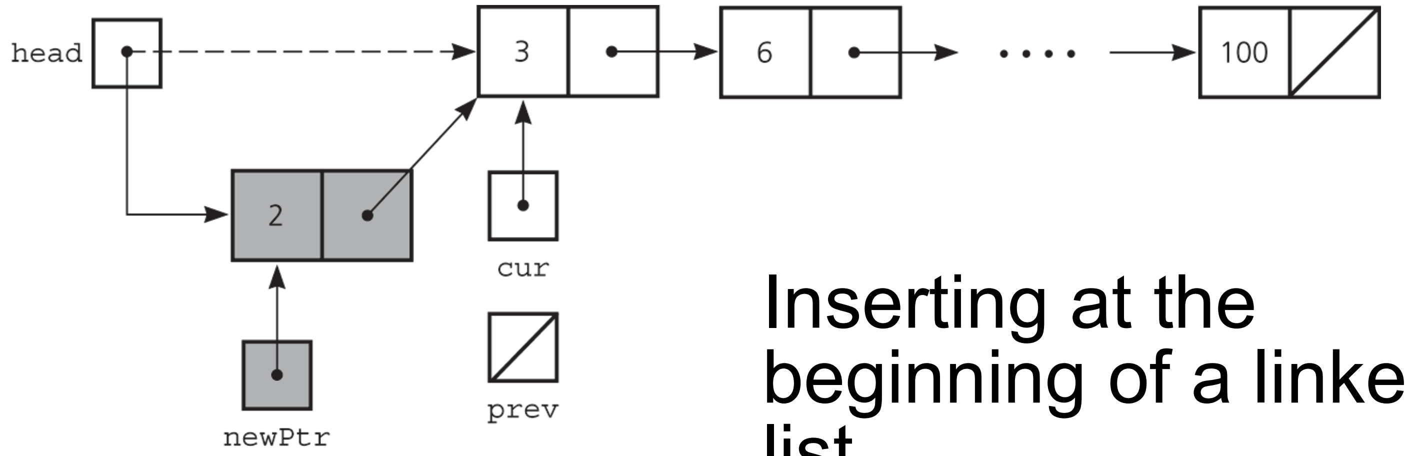


Inserting a new node into a linked list

Insert a Node into a Linked List

- To insert a node at the beginning of a linked list

```
newPtr->next = head;
head = newPtr;
```

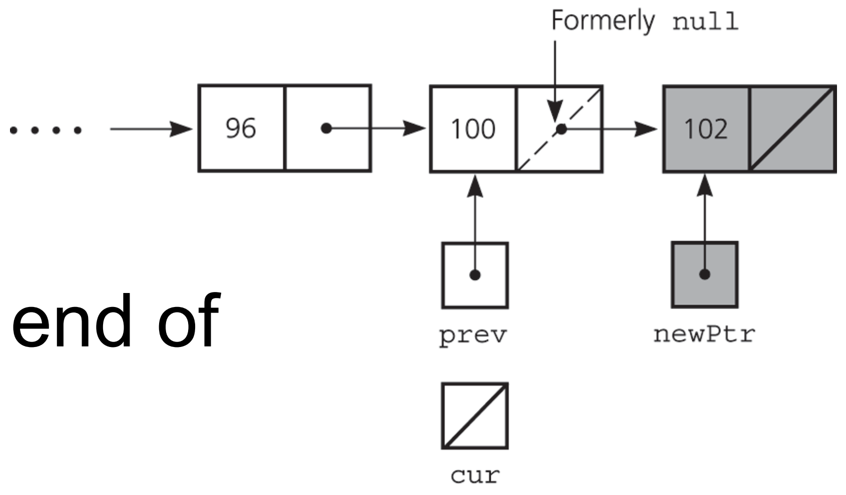


Inserting at the beginning of a linked list

Insert a Node into a Linked List

- Inserting at the end of a linked list is not a special case if `cur` is *NULL*

```
newPtr->next = cur;
prev->next = newPtr;
```



Inserting at the end of a linked list

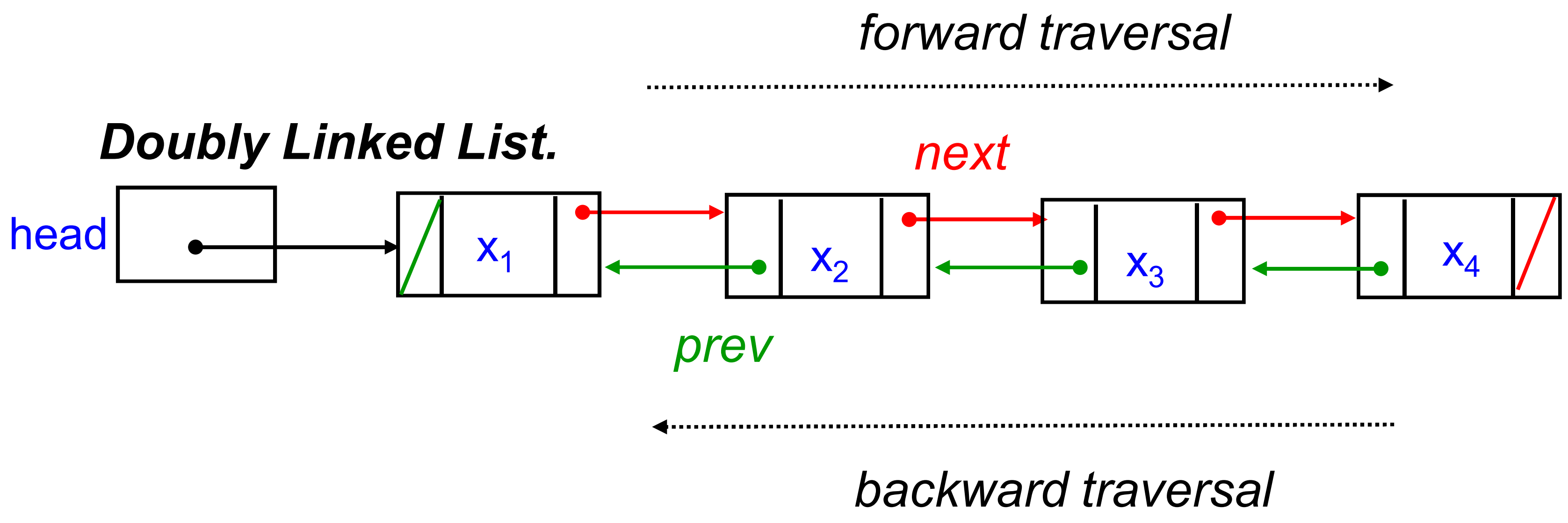
An ADT Interface for List

- Functions
  - `isEmpty`
  - `getLength`
  - `insert`
  - `delete`
  - `Lookup`
  - …

- Data Members
  - `head`
  - `Size`
- Local variables to member functions
  - `cur`
  - `prev`

Doubly Liked Lists

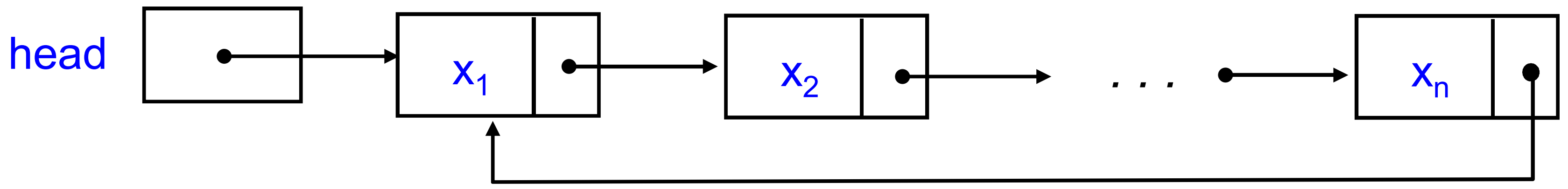
- Frequently, we need to traverse a sequence in BOTH directions efficiently
- *Solution* : Use doubly-linked list where each node has two pointers



Circular Linked Lists

- May need to cycle through a list repeatedly, e.g. round robin system for a shared resource
- *Solution* : Have the last node point to the first node

Circular Linked List.

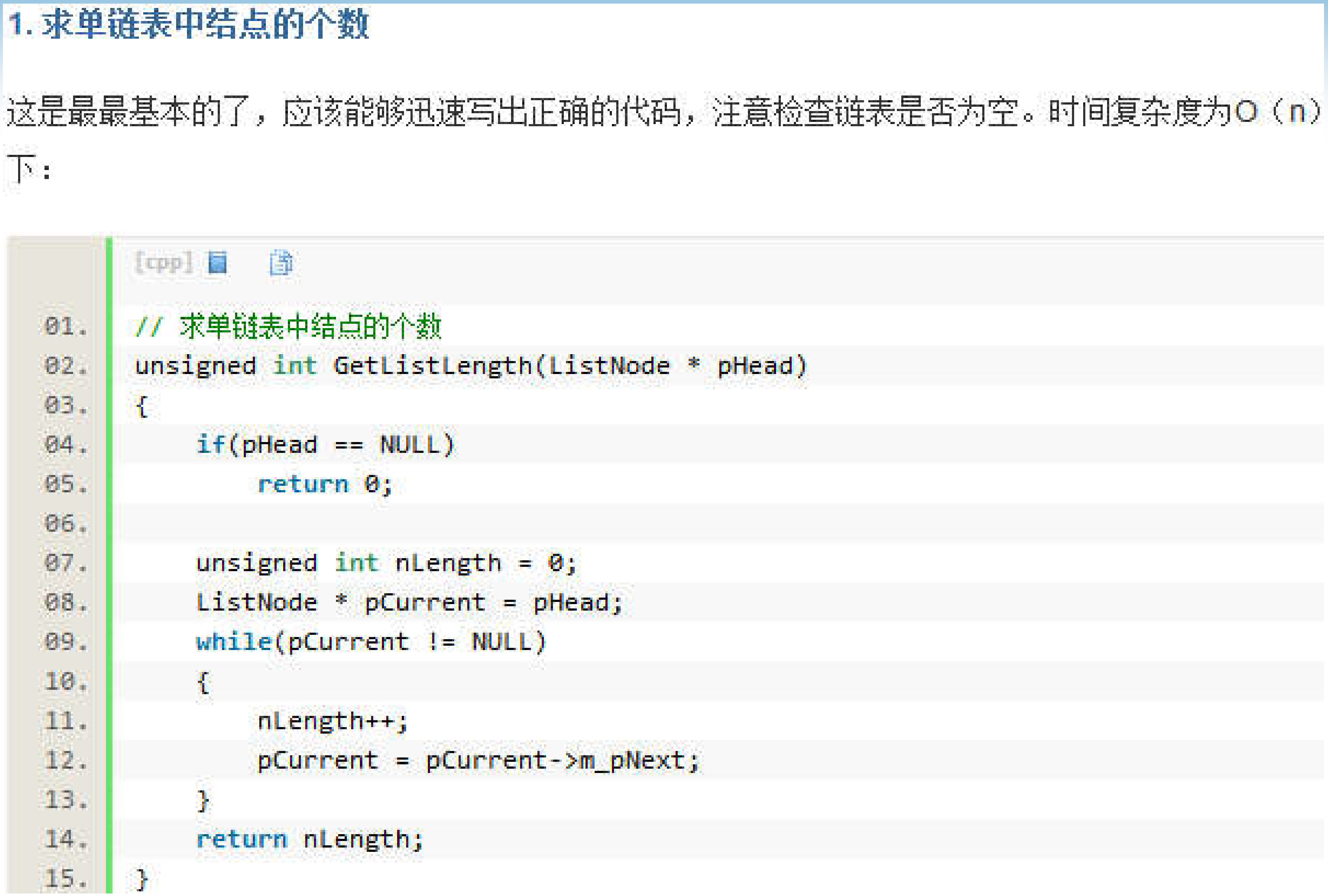


- 链表结点声明如下：
- struct ListNode

  {

      int m_nKey;
      ListNode * m_pNext;

- };



1. 求单链表中结点的个数

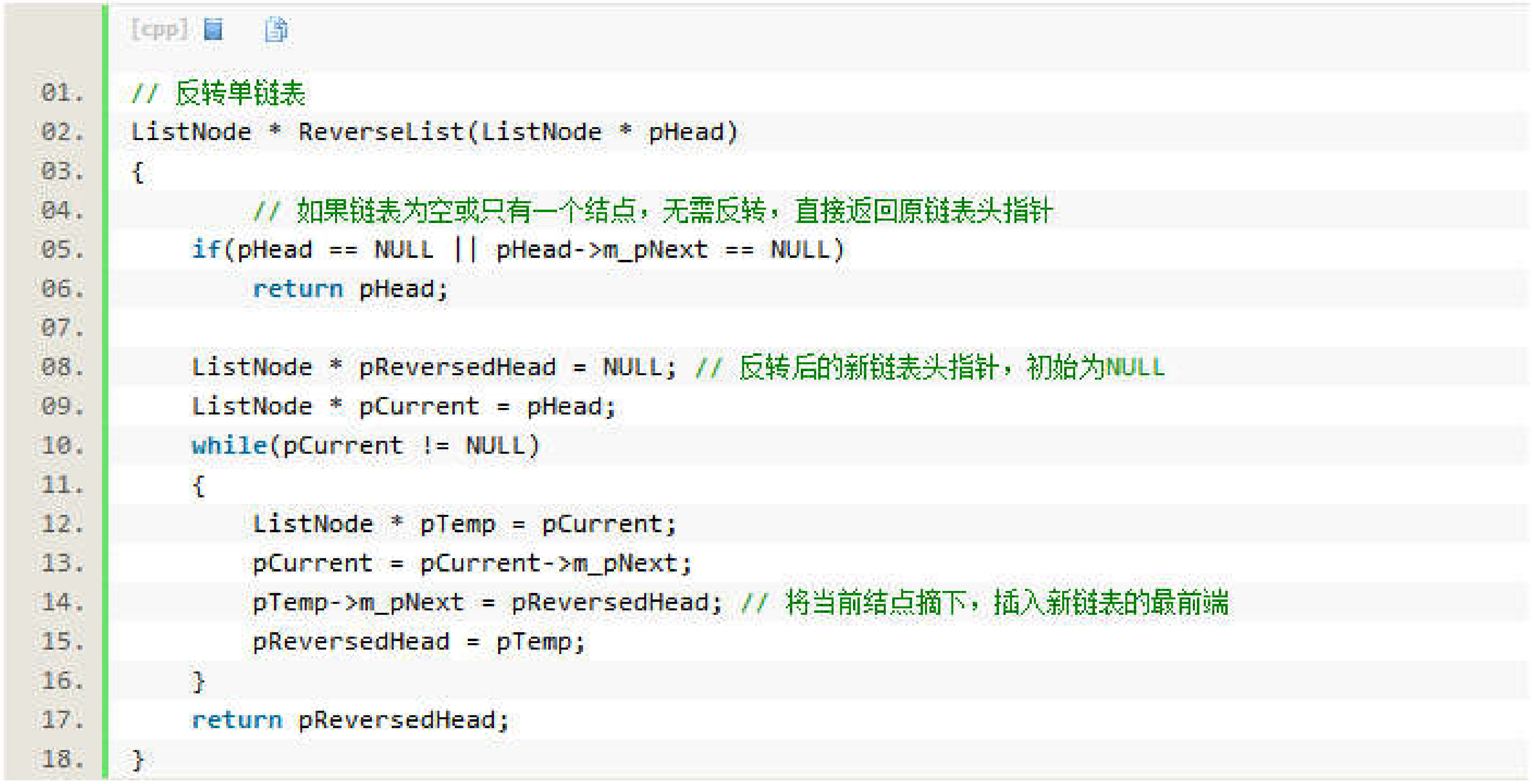
这是最最基本的了，应该能够迅速写出正确的代码，注意检查链表是否为空。时间复杂度为O（n）

下：

```cpp
// 求单链表中结点的个数
unsigned int GetListLength(ListNode * pHead)
{
    if(pHead == NULL)
        return 0;

    unsigned int nLength = 0;
    ListNode * pCurrent = pHead;
    while(pCurrent != NULL)
    {
        nLength++;
        pCurrent = pCurrent->m_pNext;
    }
    return nLength;
}
```

2. 将单链表反转

从头到尾遍历原链表，每遍历一个结点，将其摘下放在新链表的最前端。注意链表为空和只有一个结点的情况。时间复杂度为O（n）。参考代码如下：

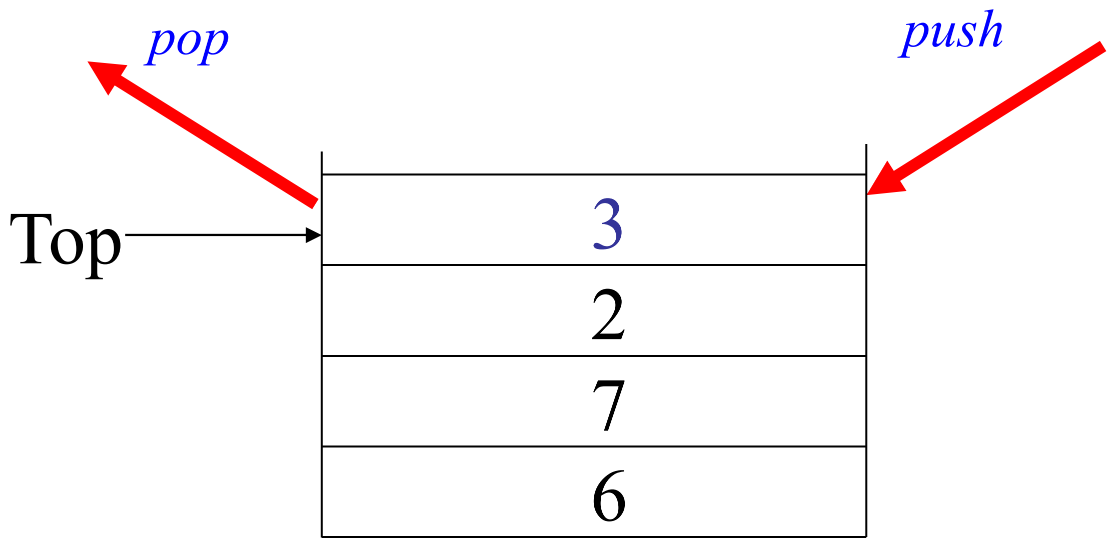


```cpp
// 反转单链表
ListNode * ReverseList(ListNode * pHead)
{
        // 如果链表为空或只有一个结点，无需反转，直接返回原链表头指针
    if(pHead == NULL || pHead->m_pNext == NULL)
        return pHead;

    ListNode * pReversedHead = NULL; // 反转后的新链表头指针，初始为NULL
    ListNode * pCurrent = pHead;
    while(pCurrent != NULL)
    {
        ListNode * pTemp = pCurrent;
        pCurrent = pCurrent->m_pNext;
        pTemp->m_pNext = pReversedHead; // 将当前结点摘下，插入新链表的最前端
        pReversedHead = pTemp;
    }
    return pReversedHead;
}
```

Stacks （栈）

What is a Stack?

- A *stack* is a list with the restriction that insertions and deletions can be performed in only one position, called the *top*.
- The operations: push (insert) and pop (delete)



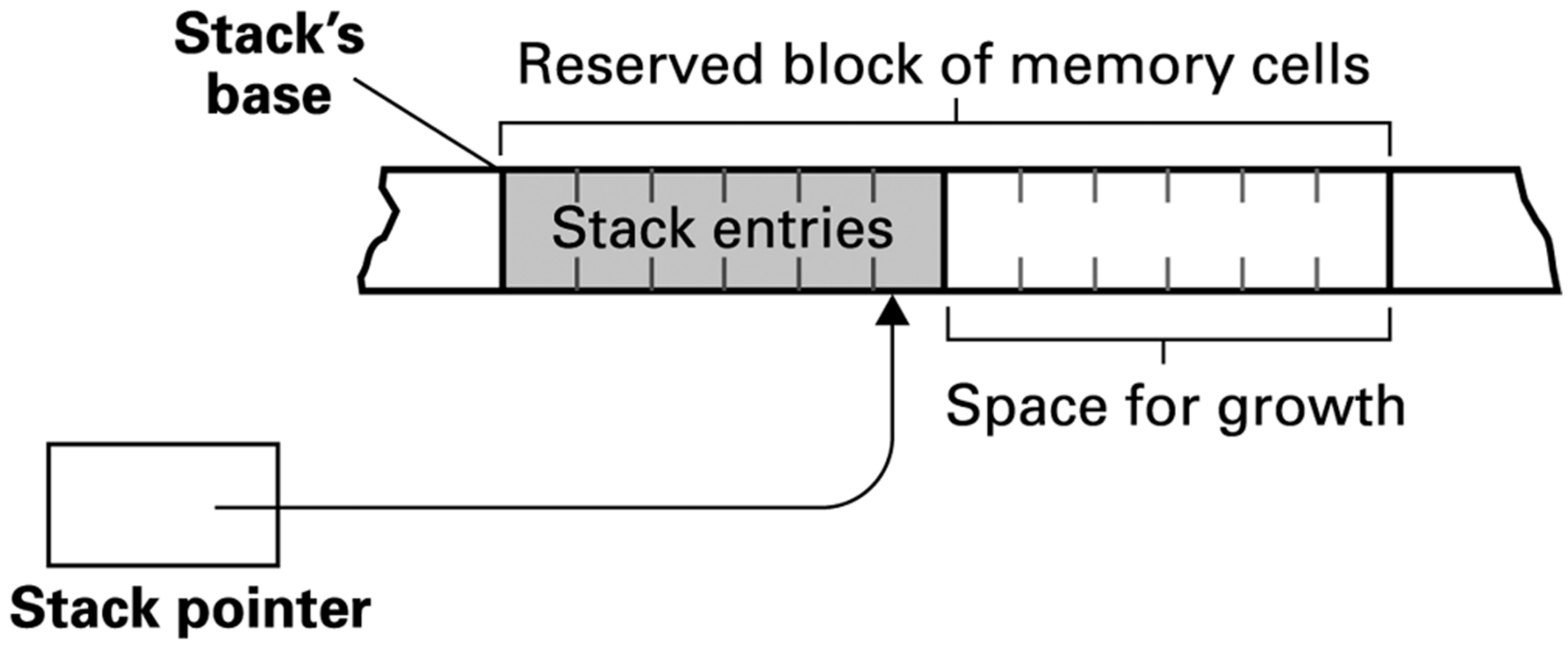
Terminology for Stacks

- **Stack**（栈）: A list in which entries are removed and inserted only at the head
- **LIFO:** Last-in-first-out
- **Top**（栈顶）: The head of list (stack)
- **Bottom** or **base**（栈底）: The tail of list (stack)
- **Pop**（出栈）: To remove the entry at the top
- **Push**（入栈）: To insert an entry at the top

Stacks usually stored as contiguous lists
只能在表头进行添加删除

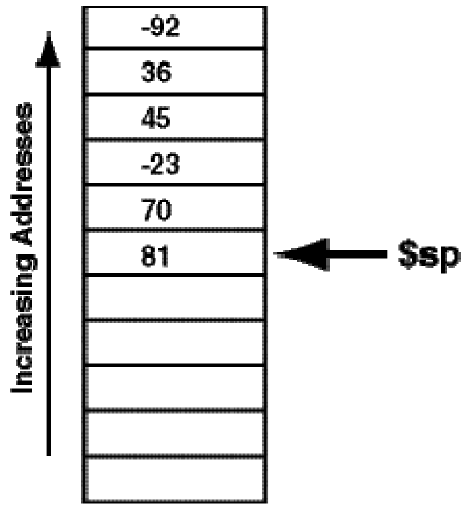
- Stack用途：编译器中的词法分析器、Java虚拟机、软件中的撤销操作、浏览器中的回退操作，编译器中的函数调用实现
- 实例：利用stack 处理多余无效的请求，比如用户长按键盘，或者在很短的时间内连续按某一个功能键，我们需要过滤到这些无效的请求。将所有的请求都压入到堆中，然后要处理的时候Pop出来一个，这个就是最新的一次请求

Figure 8.12 **A stack in memory**

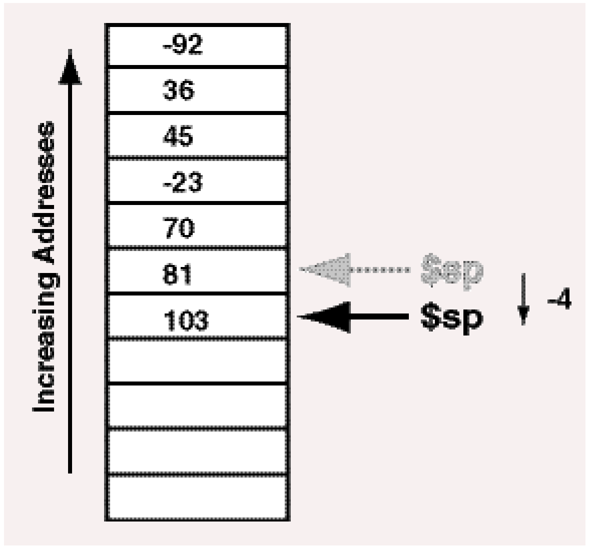


Stack

- Stack-like behavior is sometimes called "LIFO" for Last In First Out.
- The top item of the stack is 81. The bottom of the stack contains the integer -92
- Stack pointer $sp always points to the top of the stack.



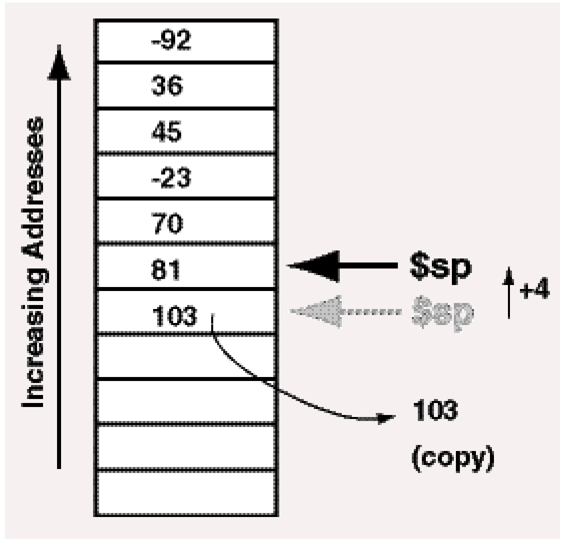
- To **push** an item onto the stack, first **subtract** 4 from the stack pointer, then store the item at the address in the stack pointer



MIPS (32-bit) example

```
                        # PUSH the item in $t0:
subu  $sp,$sp,4         #   point to the place for the new item,
sw    $t0,($sp)         #   store the contents of $t0 as the new top.
```

- To **pop** the top item from a stack, copy the item pointed at by the stack pointer, then **add** 4 to the stack pointer.



```
                    # POP the item into $t0:
lw    $t0,($sp)     #   Copy top the item to $t0.
addu $sp,$sp,4      #   Point to the item beneath the old top.
```

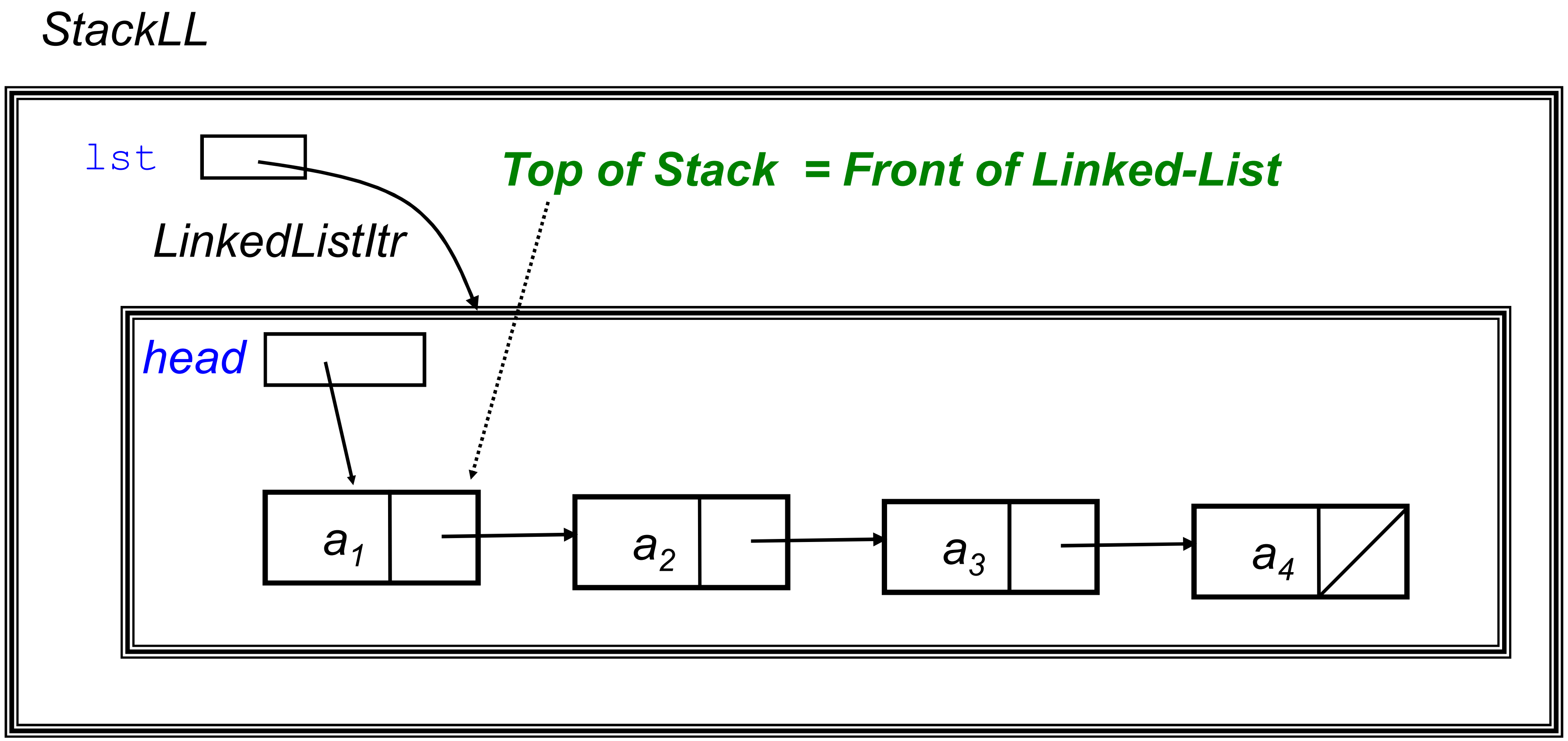
Stack ADT Interface

- The main functions in the Stack ADT are (S is the stack)

```
boolean isEmpty();              // return true if empty
boolean isFull(S);              // return true if full
void push(S, item);             // insert item into stack
void pop(S);                    // remove most recent item
void clear(S);                  // remove all items from stack
Item top(S);                    // retrieve most recent item
Item topAndPop(S);              // return & remove most recent item
```

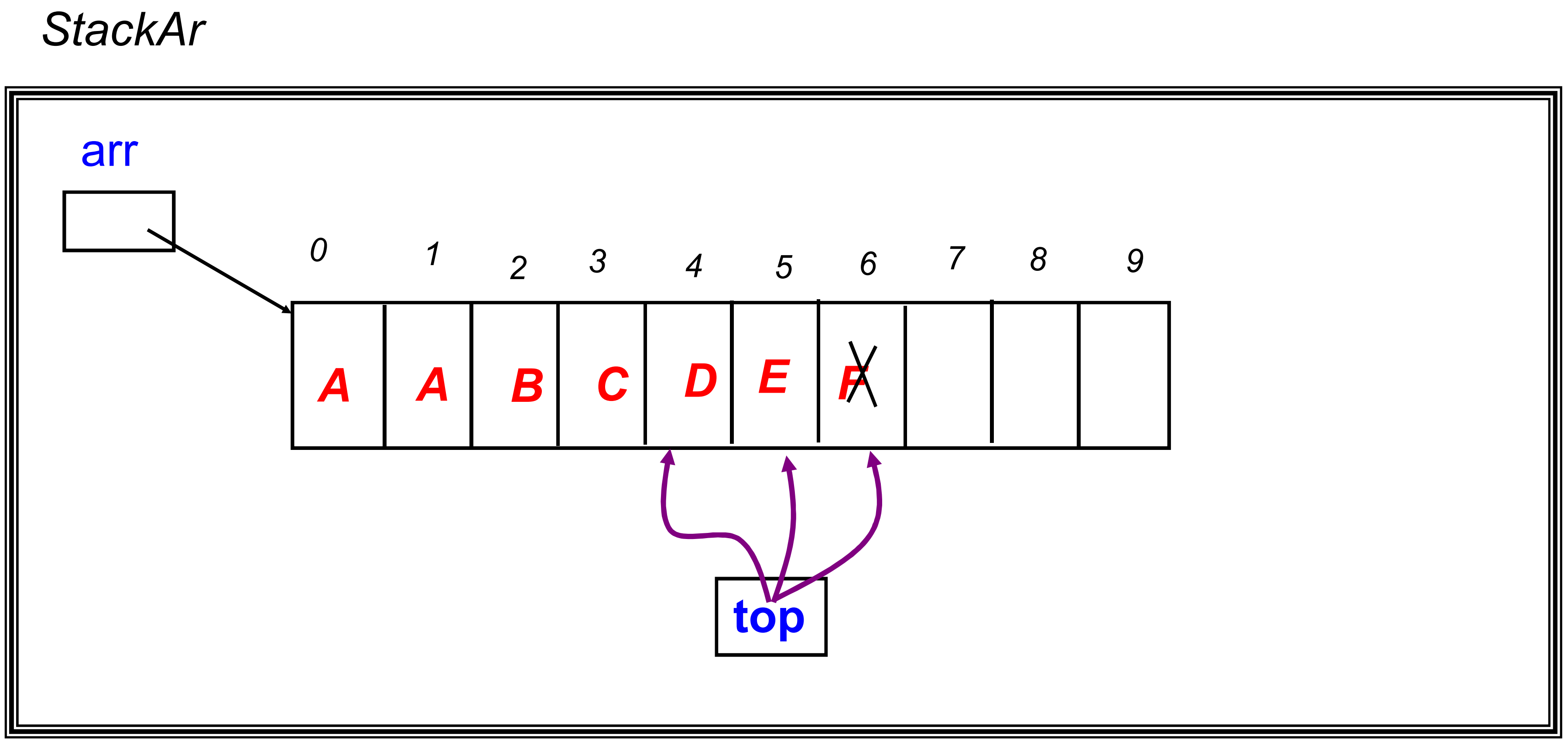
Implementation by Linked Lists

- Can use a Linked List as implementation of stack

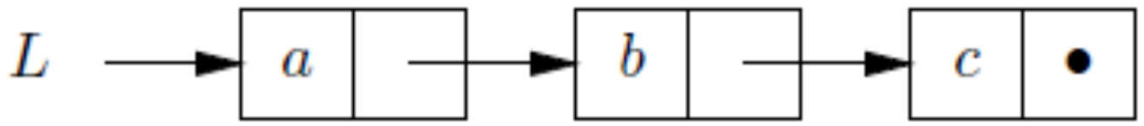


Implementation by Array

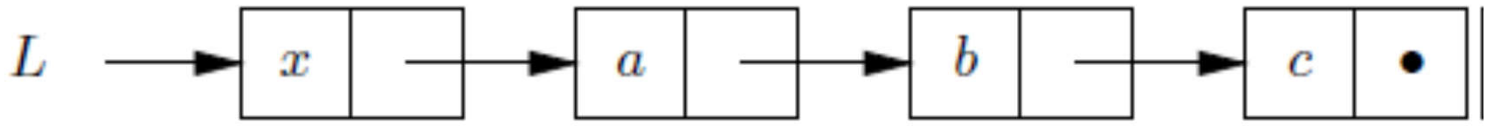
- use Array with a top index pointer as an implementation of stack



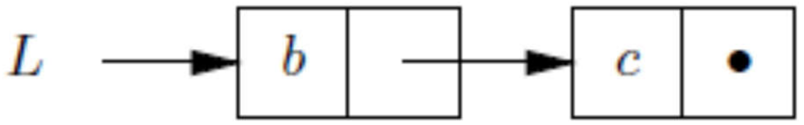
Effects



(a) List L.



(b) After executing $push(x, L)$.

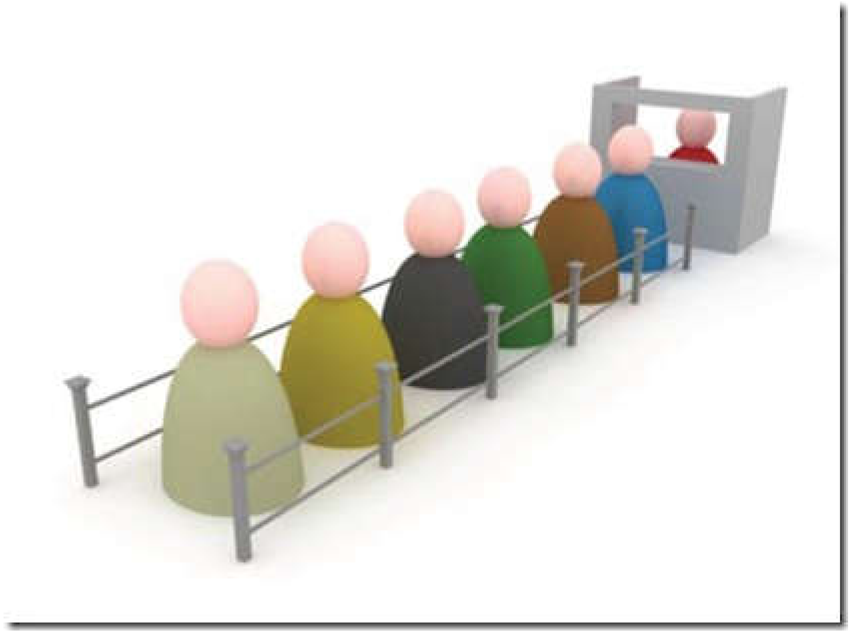


(c) After executing $pop(L, \cancel{x})$ on list L of (a).

Summary： stack

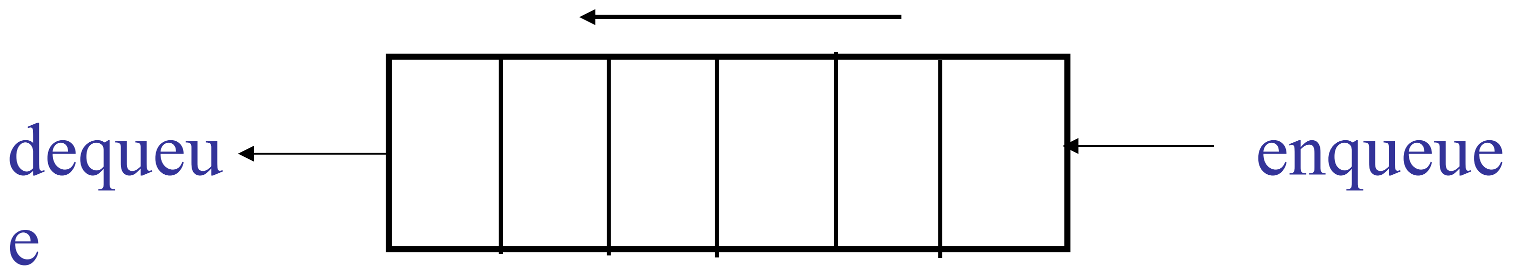
- The ADT stack operations have a last-in, first-out (LIFO) behavior
- Stack has many applications
  - algorithms that operate on algebraic expressions
  - a strong relationship between recursion and stacks exists
- Stack can be implemented using arrays or linked lists

Queues



What is a Queue?

- Like stacks, queues are lists. With a queue, however, insertion is done at one end whereas deletion is done at the other end.
- Queues implement the FIFO (first-in first-out) policy. E.g., a printer/job queue!
- Two basic operations of queues:
  - dequeue: remove an item/element from front
  - enqueue: add an item/element at the back



Terminology for Queues

- **Queue（队列）:** A list in which entries are removed at the head and are inserted at the tail
- **FIFO:** First-in-first-out

Storing Queues

- Queues usually stored as **Circular Queues**
  - Stored in a contiguous block in which the first entry is considered to follow the last entry
  - Prevents a queue from crawling out of its allotted storage space

- 应用：
  - 播放器上的播放列表
  - 打印机的打印队列
  - 呼叫中心用户等待时间模拟

Queue ADT

- Queues implement the FIFO (first-in first-out) policy
  - An example is the printer/job queue!

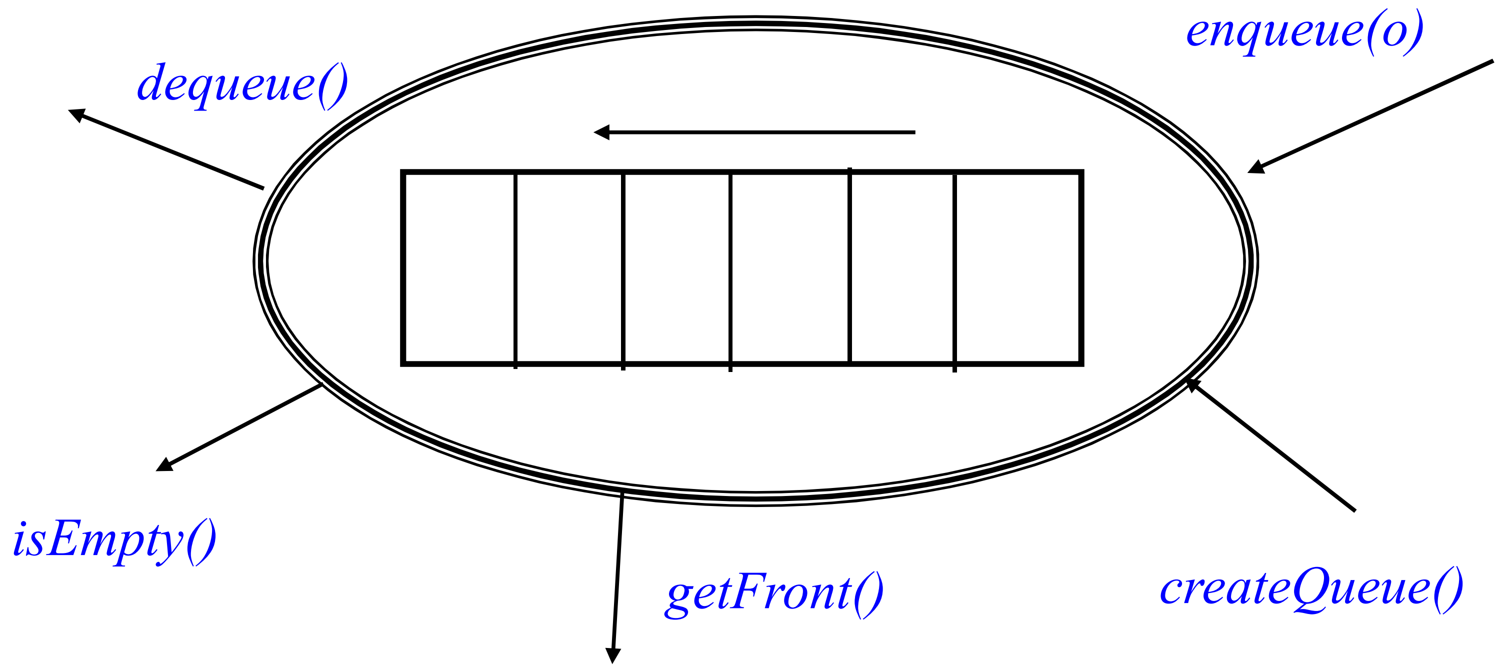


Figure 8.13 **A queue implementation with head and tail pointers**

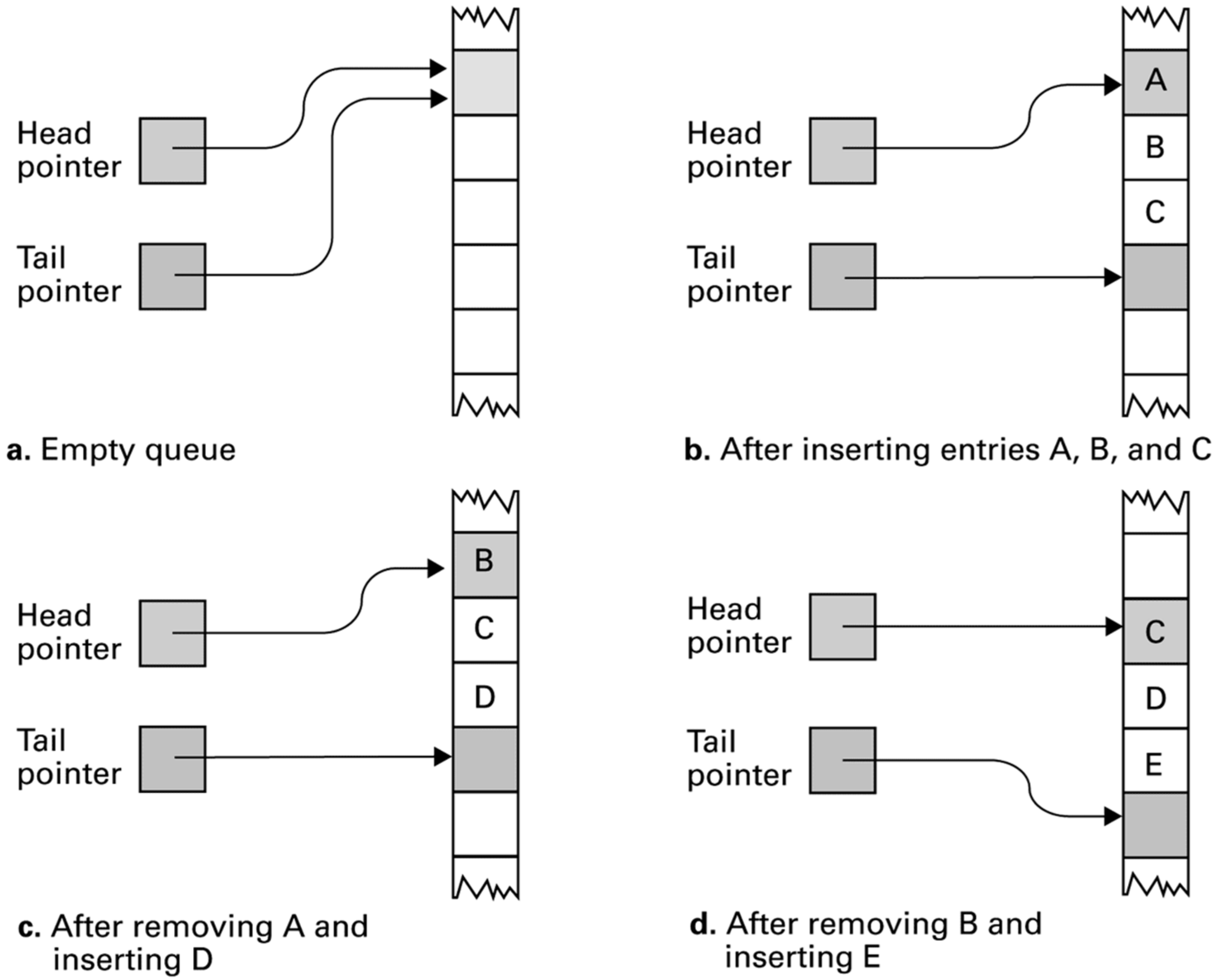
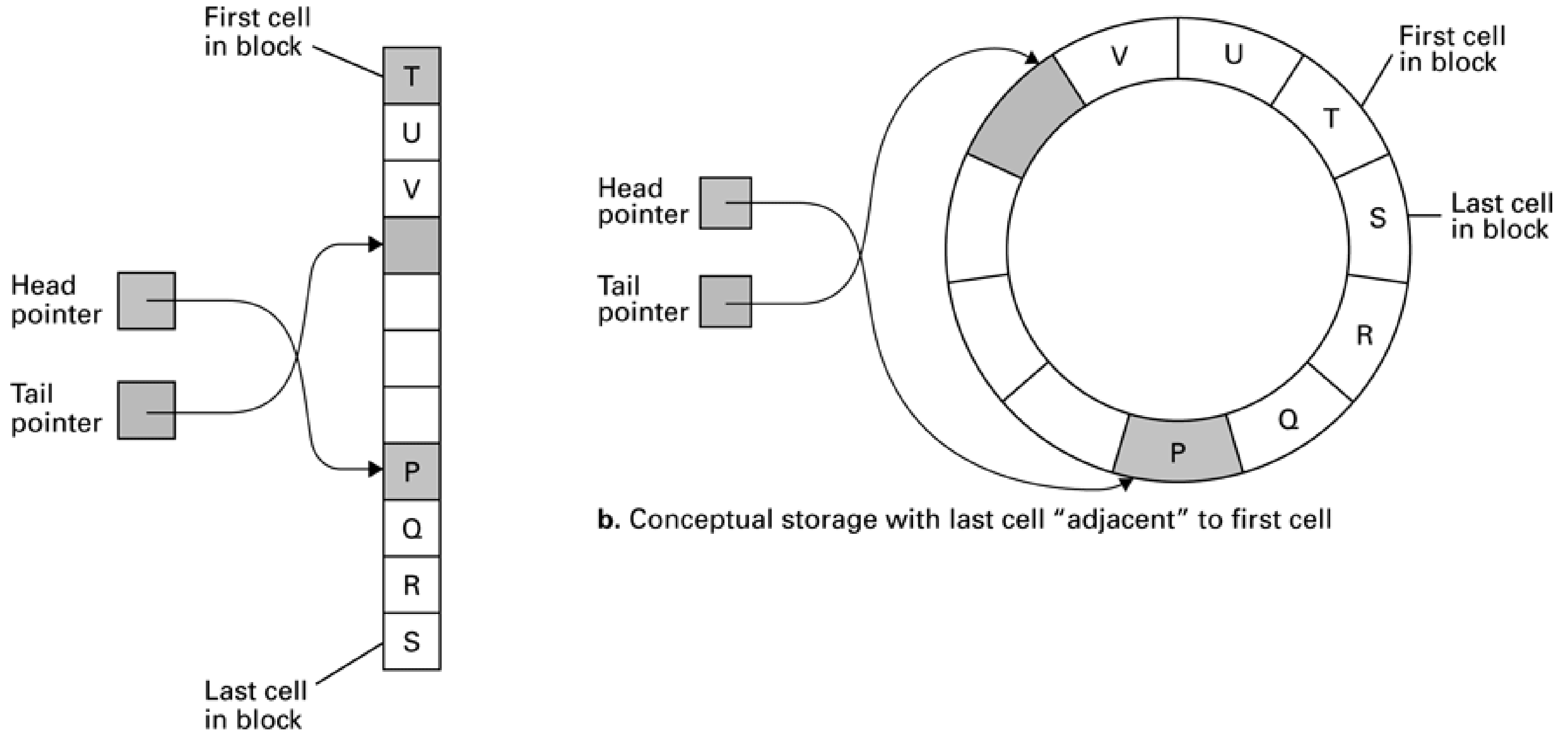


Figure 8.14 **A circular queue containing the letters P through V**

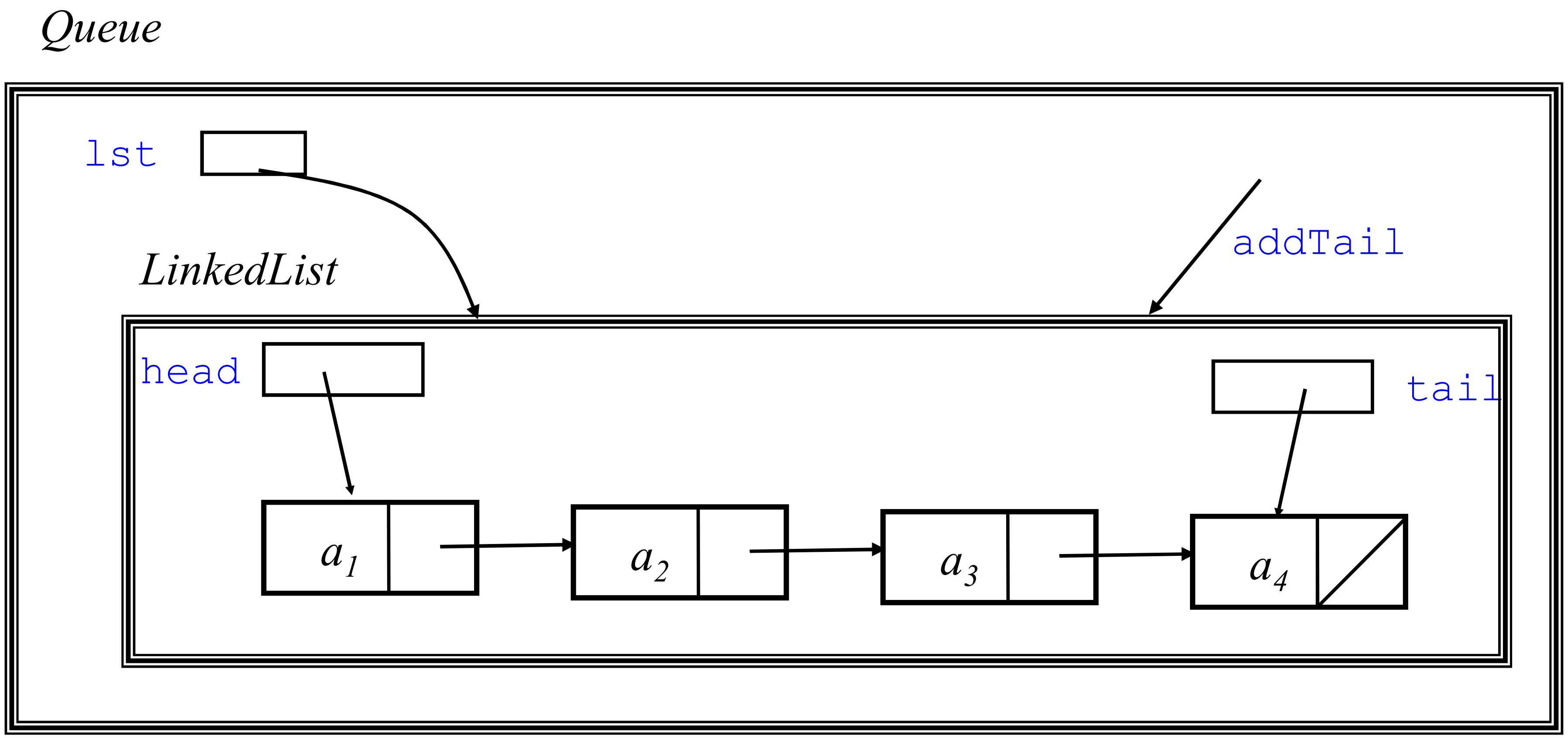


a. Queue as actually stored

b. Conceptual storage with last cell "adjacent" to first cell

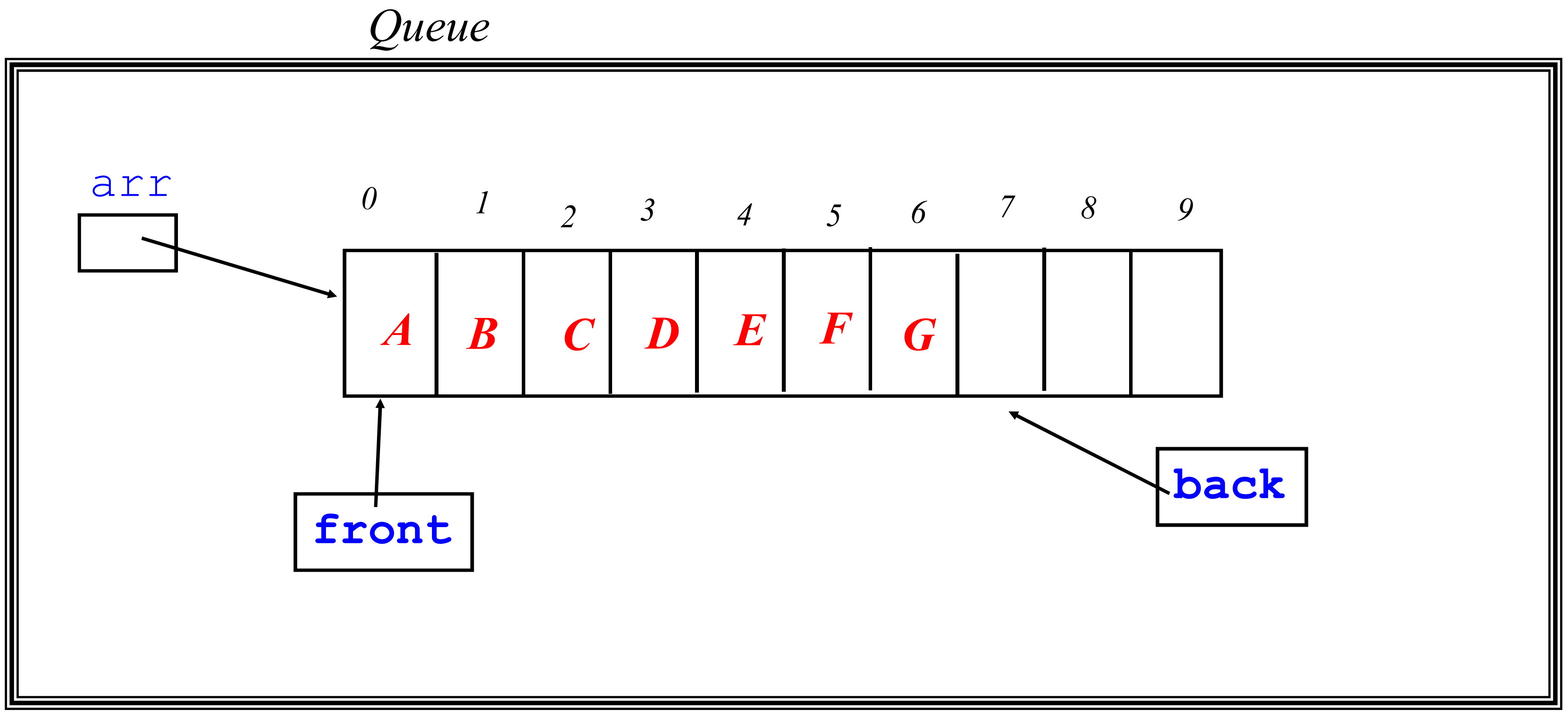
Implementation of Queue (Linked List)

- Can use LinkedListItr as underlying implementation of Queues



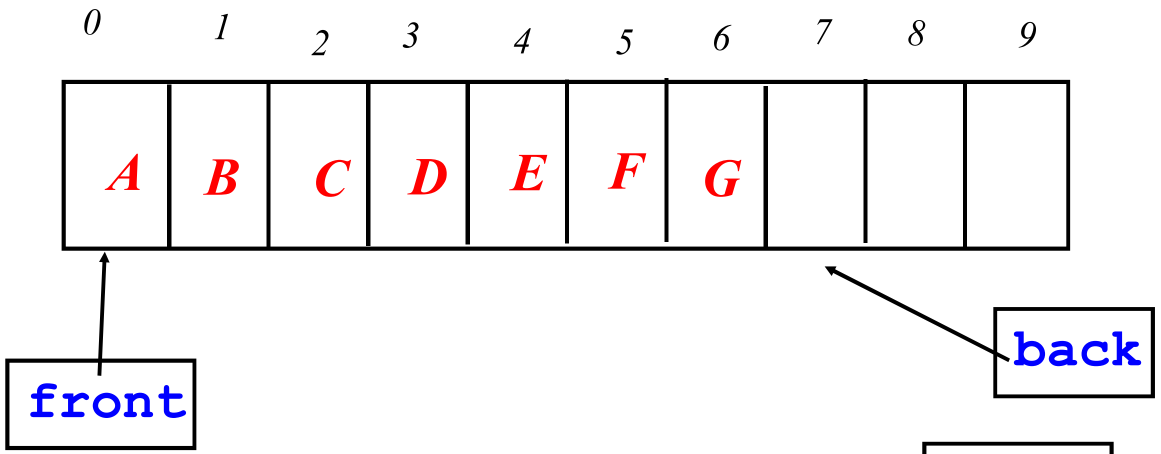
Implementation of Queue (Array)

- use Array with front and back pointers as implementation of queue

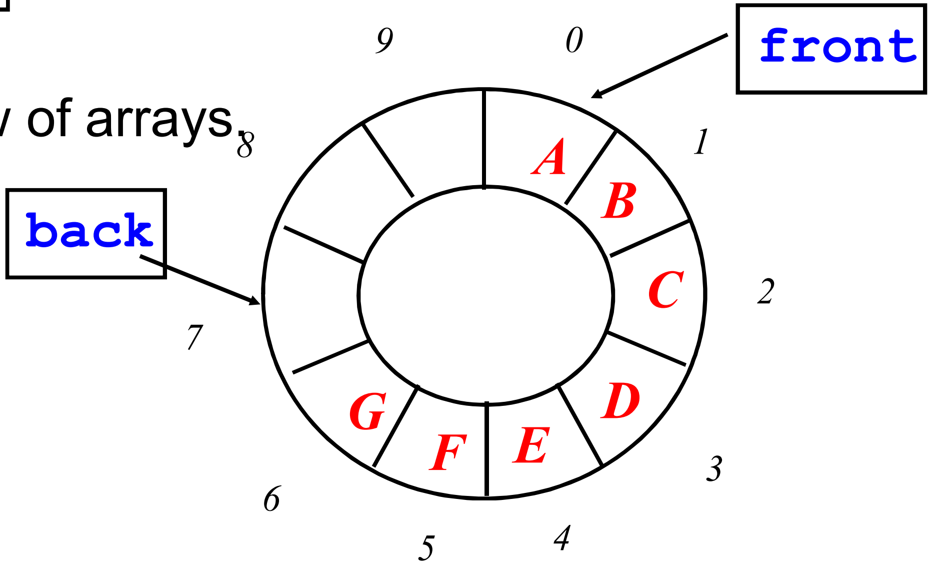


Circular Array

- To implement queue, it is best to view arrays as circular structure



Circular view of arrays.

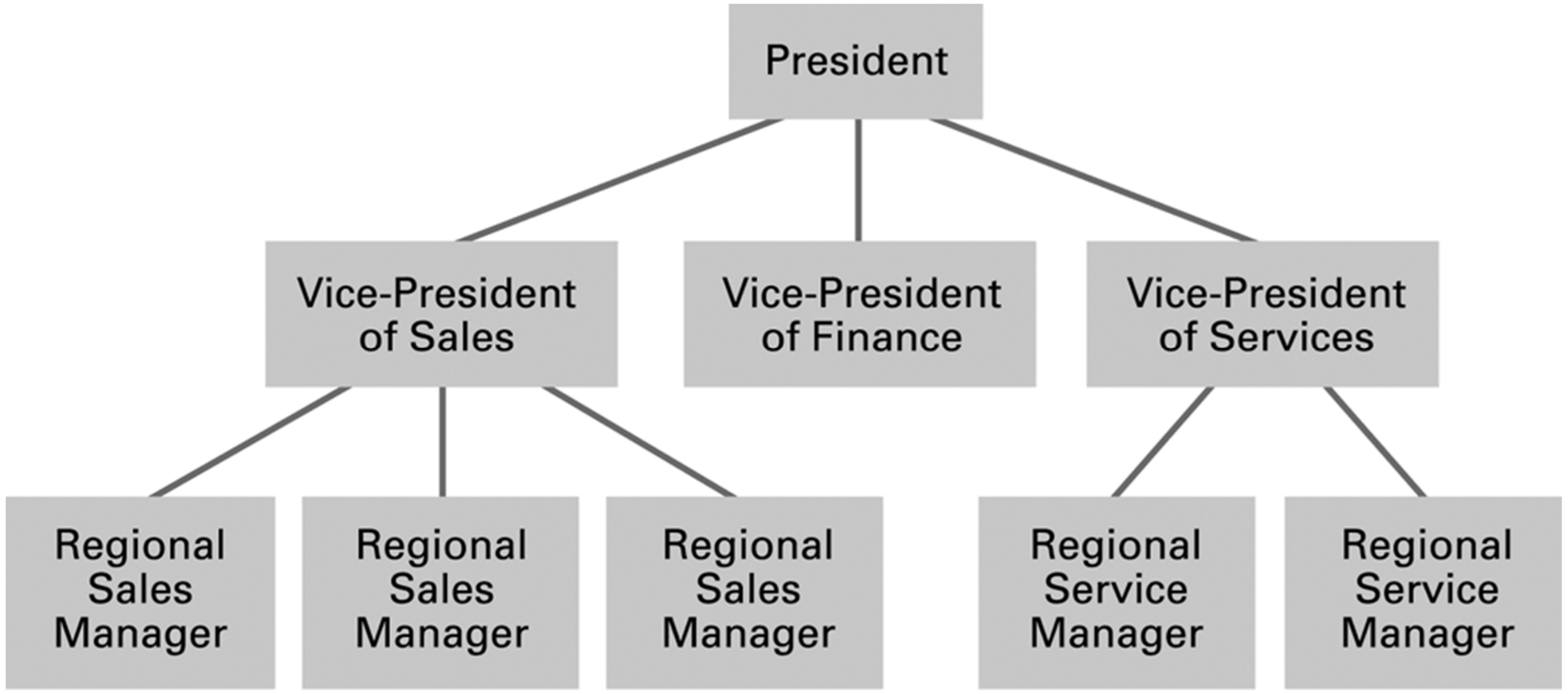


Summary: queue

- The definition of the queue operations gives the ADT queue first-in, first-out (FIFO) behavior
- The queue can be implemented by linked lists or by arrays
- There are many applications
  - Printer queues,
  - Telecommunication queues,
  - Simulations,
  - Etc.

Tree

Figure 8.2 **An example of an organization chart**



Terminology for a Tree

- **Tree**（树）: A collection of data whose entries have a hierarchical organization
- **Node**（结点）: An entry in a tree
- **Root node**（根节点）: The node at the top
- **Terminal** or **leaf node**（终端/叶子结点）: A node at the bottom

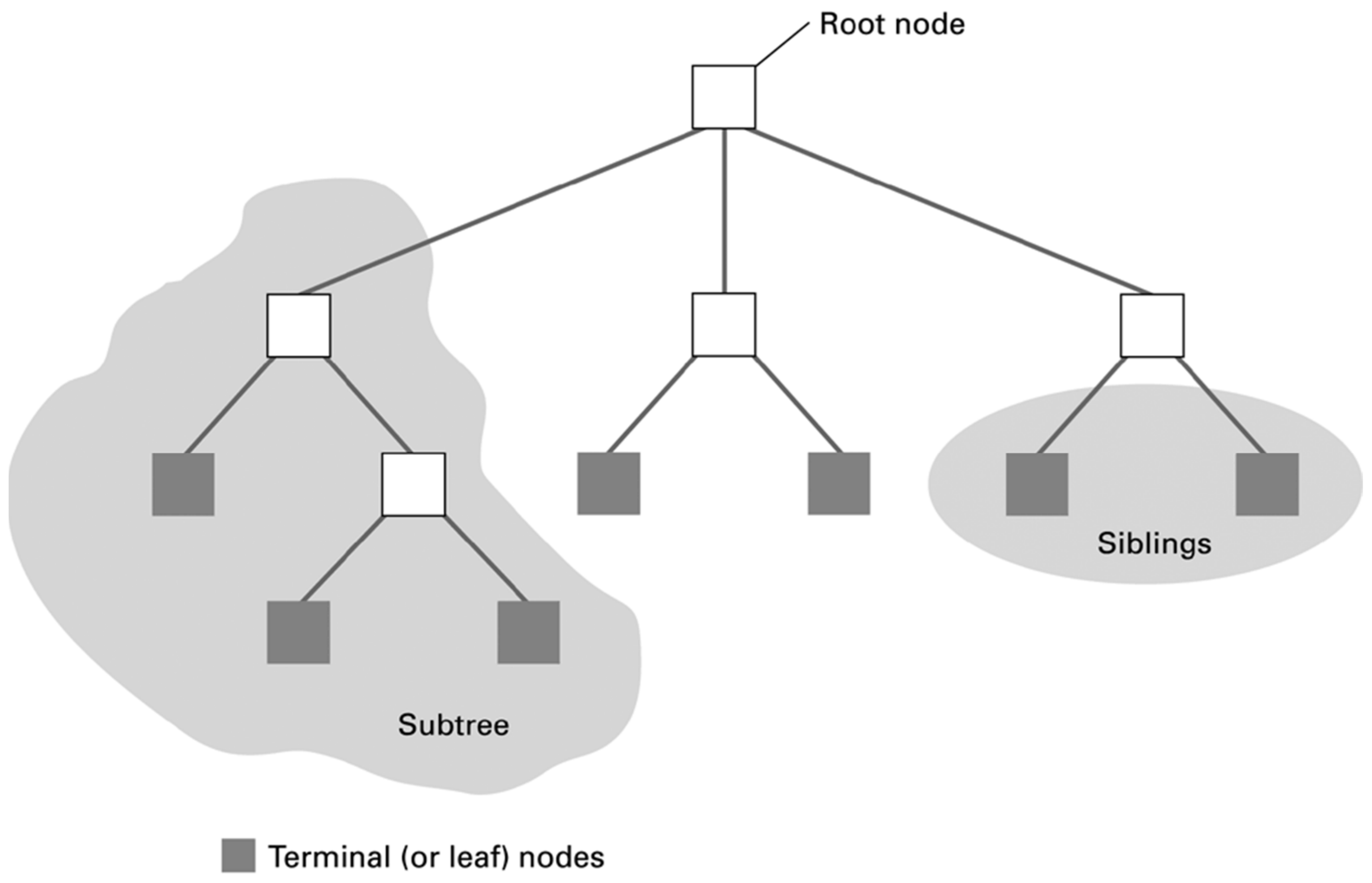
Terminology for a Tree (continued)

- **Parent:** The node immediately above a specified node
- **Child:** A node immediately below a specified node
- **Ancestor:** Parent, parent of parent, etc.
- **Descendent:** Child, child of child, etc.
- **Siblings:** Nodes sharing a common parent

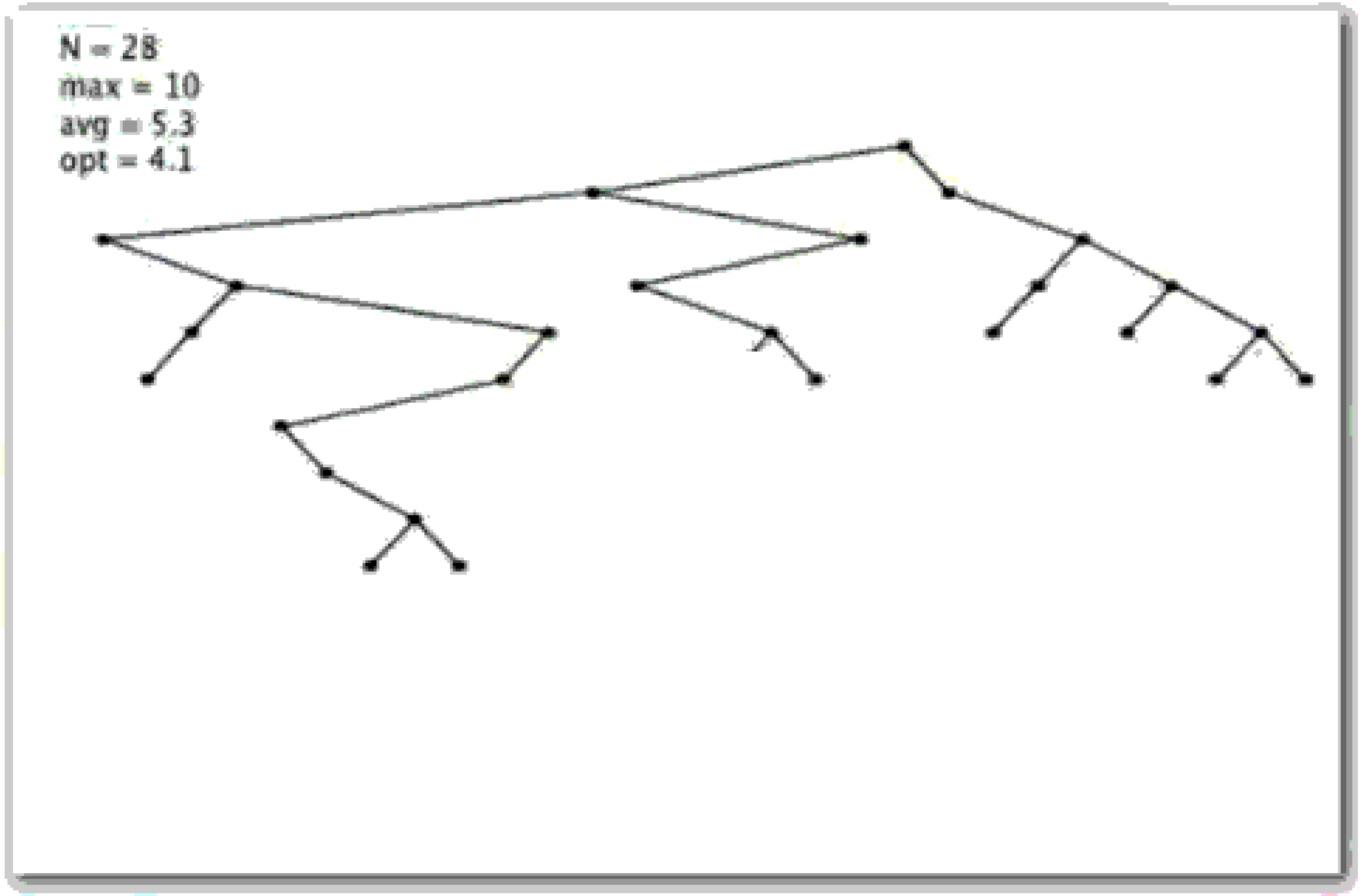
Terminology for a Tree (continued)

- **Binary tree**（二叉树）**:** A tree in which every node has at most two children
- **Depth**（深度）**:** The number of nodes in longest path from root to leaf

Figure 8.3 **Tree terminology**



随机插入形成树的动画



Storing Binary Trees

- Linked structure
  - Each node = data cells + two child pointers
  - Accessed via a pointer to root node
- Contiguous array structure
  - A[1] = root node
  - A[2],A[3] = children of A[1]
  - A[4],A[5],A[6],A[7] = children of A[2] and A[3]

Figure 8.15 **The structure of a node in a binary tree**

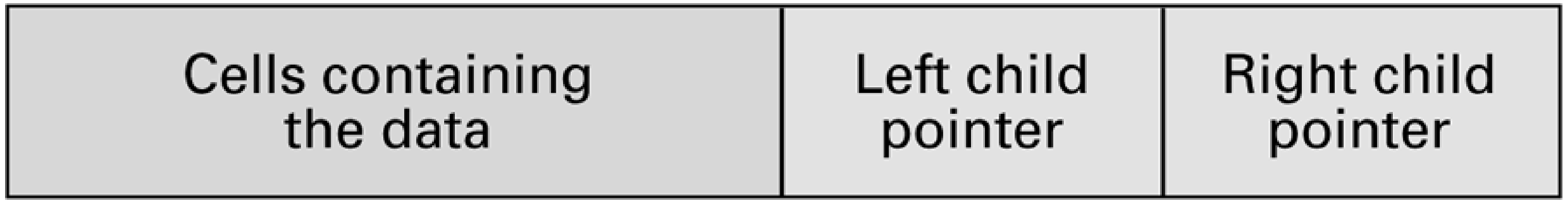


Figure 8.16 **The conceptual and actual organization of a binary tree using a linked storage system**

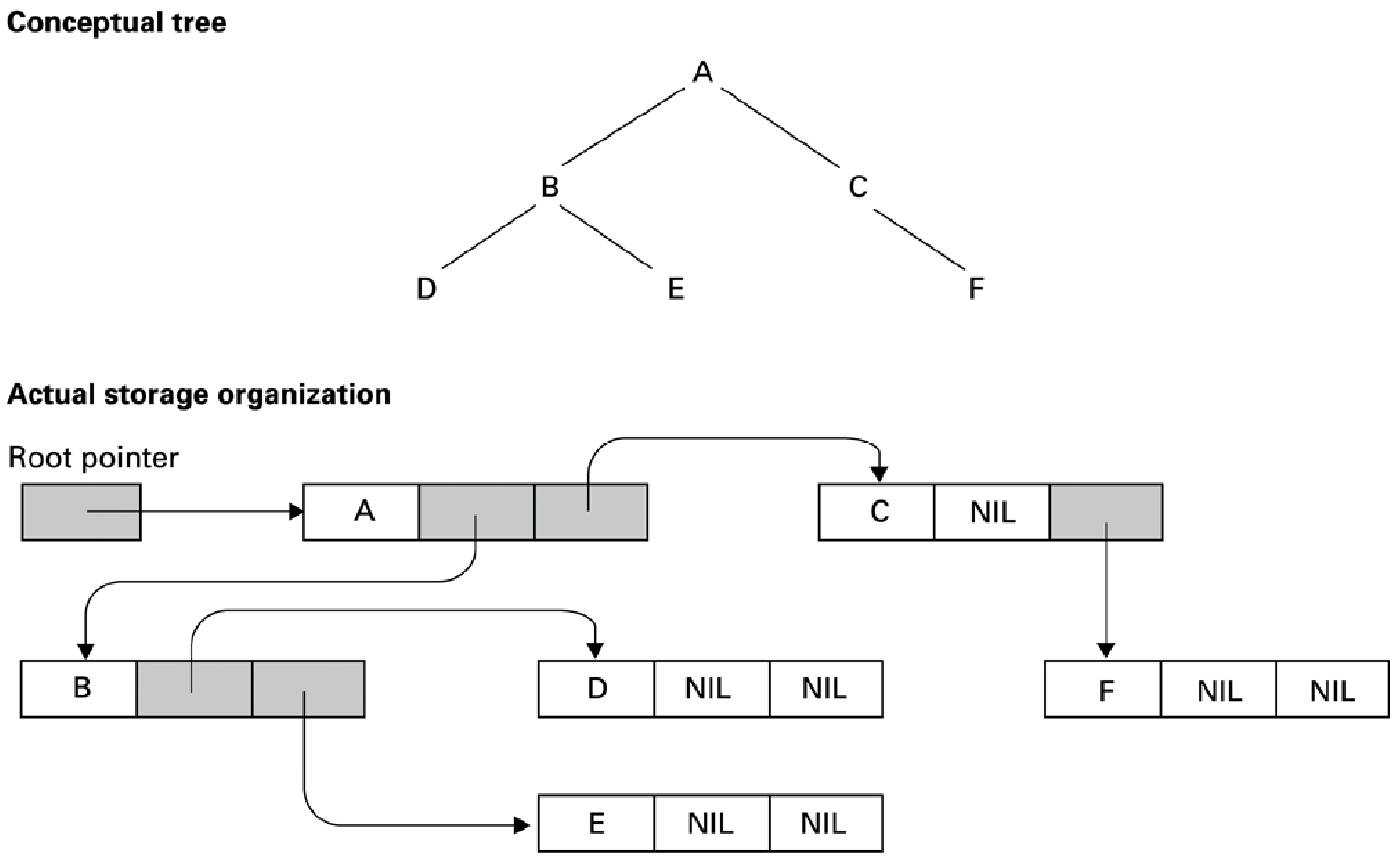


Figure 8.17 A tree stored without pointers

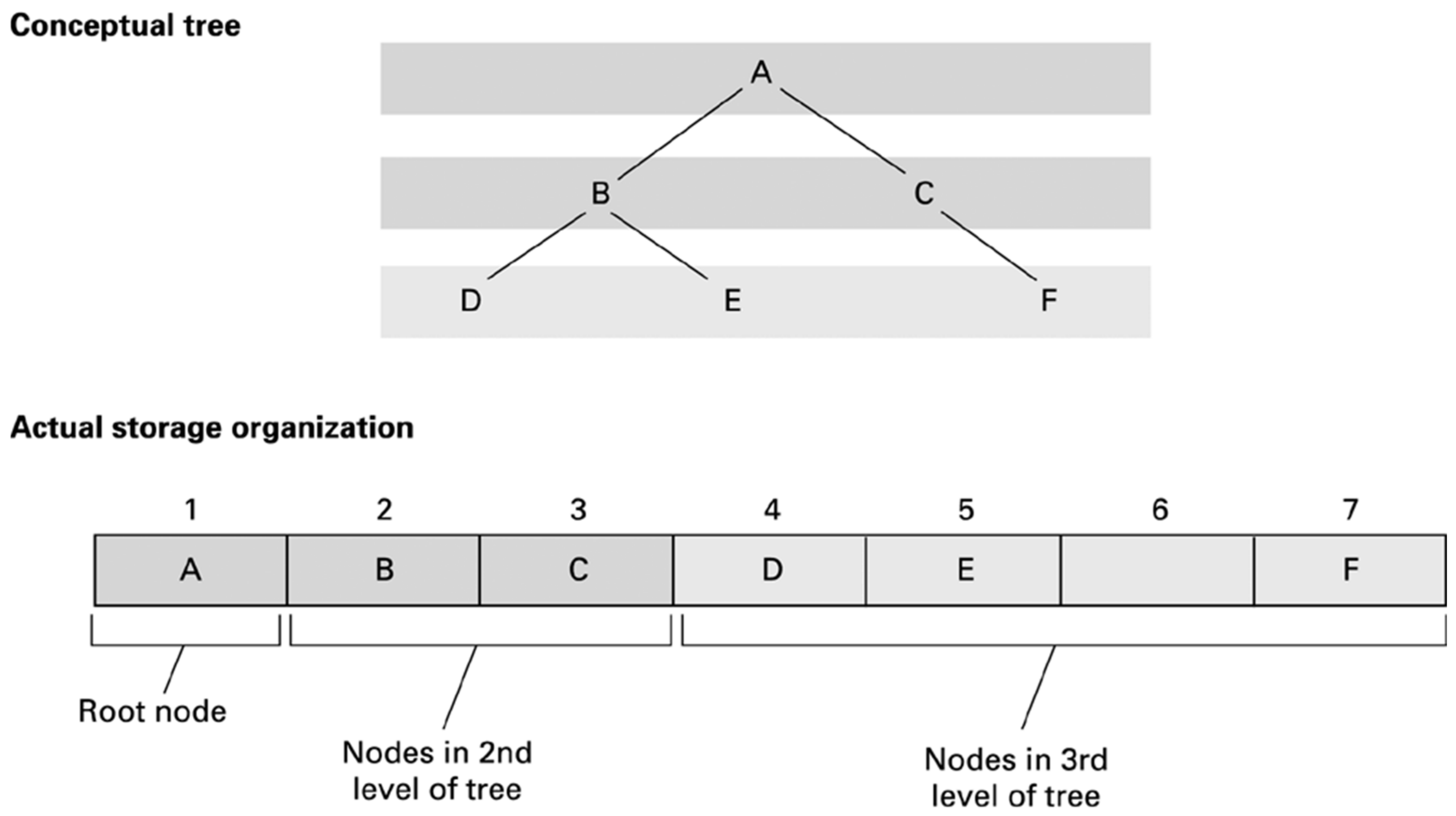
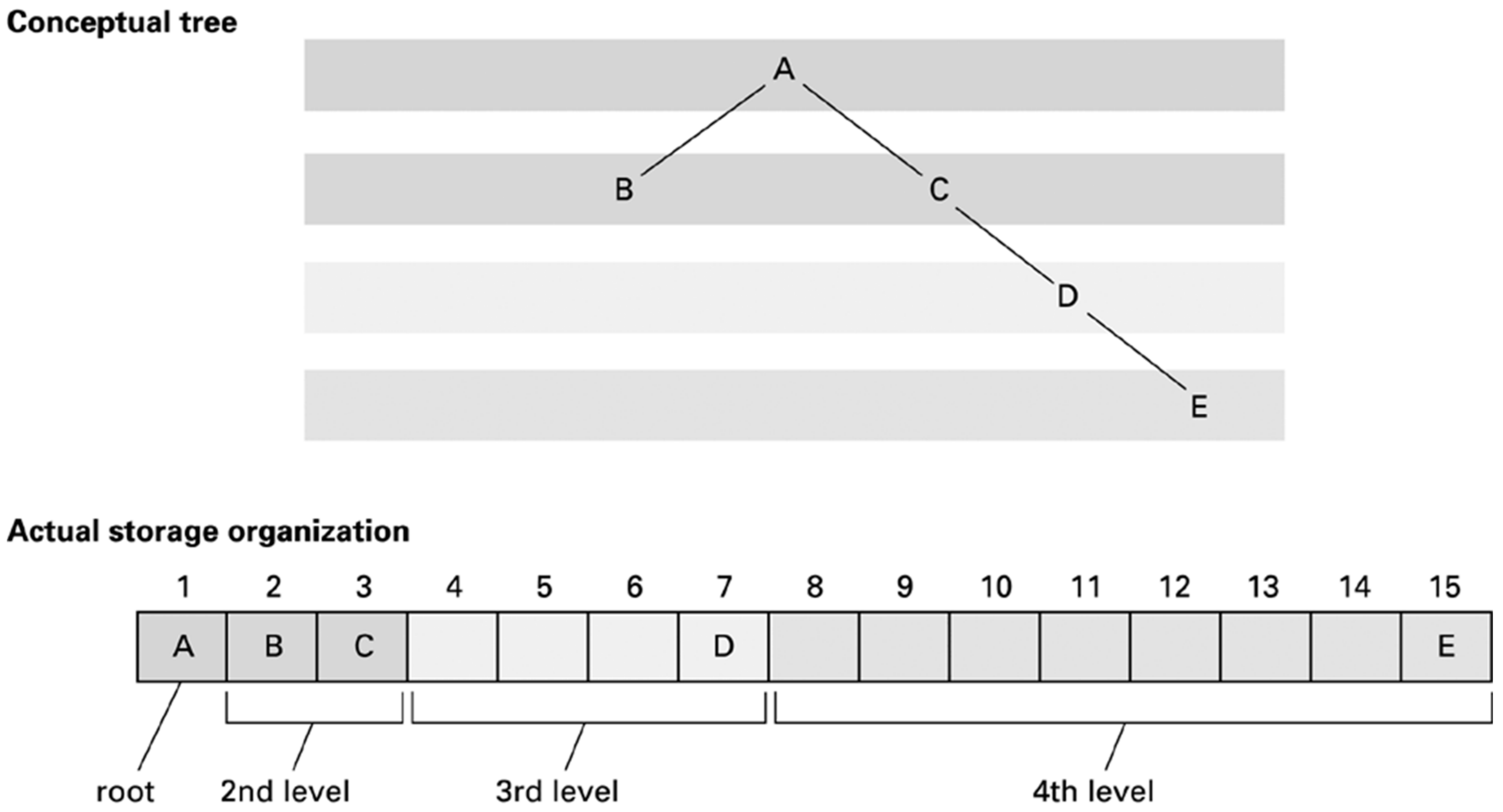

Conceptual tree
A
B
C
D
E
F
Actual storage organization
1
2
3
4
5
6
7
A
B
C
D
E
F
Root node
Nodes in 2nd level of tree
Nodes in 3rd level of tree

Figure 8.18 A sparse, unbalanced tree shown in its conceptual form and as it would be stored without pointers



- 二叉树是一种特殊的树，在二叉树中每个节点最多有两个子节点，一般称为左子节点和右子节点（或左孩子和右孩子
- 二叉树是递归定义的，因此，与二叉树有关的题目基本都可以用递归思想解决
- 二叉树节点定义如下：

```
struct BinaryTreeNode
{
    int m_nValue;
    BinaryTreeNode* m_pLeft;
    BinaryTreeNode* m_pRight;
};
```

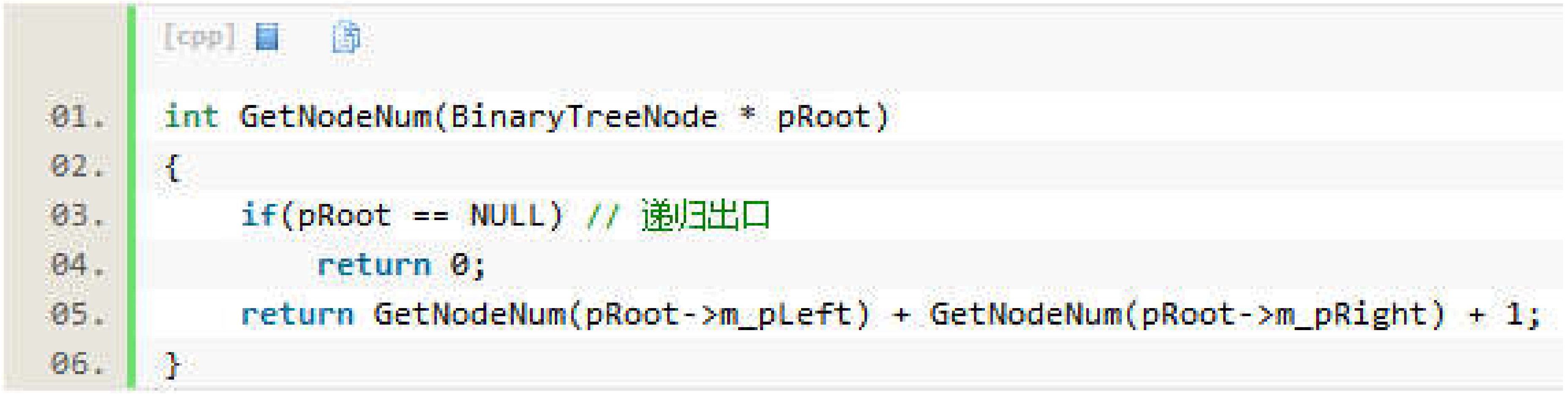
1. 求二叉树中的节点个数

递归解法：

（1）如果二叉树为空，节点个数为0

（2）如果二叉树不为空，二叉树节点个数 = 左子树节点个数 + 右子树节点个数 + 1

参考代码如下：



```cpp
int GetNodeNum(BinaryTreeNode * pRoot)
{
    if(pRoot == NULL) // 递归出口
        return 0;
    return GetNodeNum(pRoot->m_pLeft) + GetNodeNum(pRoot->m_pRight) + 1;
}
```

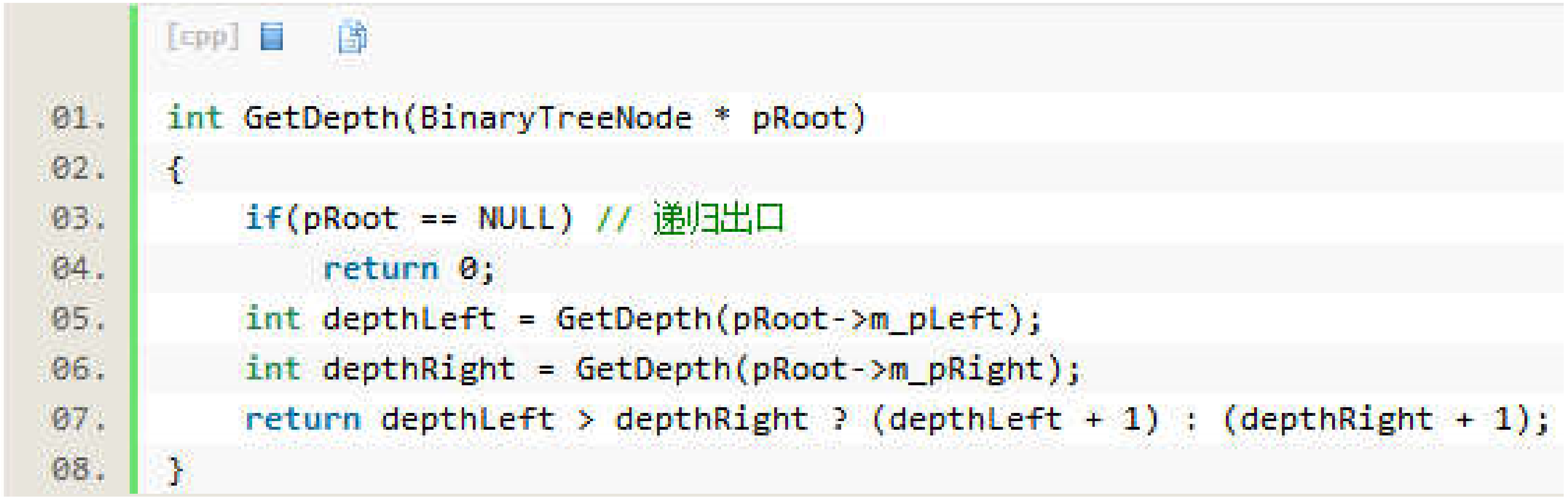
2. 求二叉树的深度

递归解法：

（1）如果二叉树为空，二叉树的深度为0

（2）如果二叉树不为空，二叉树的深度 = max(左子树深度， 右子树深度) + 1

参考代码如下：



[cpp]

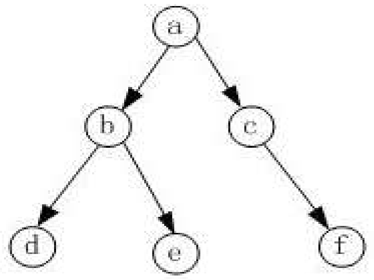
```cpp
int GetDepth(BinaryTreeNode * pRoot)
{
    if(pRoot == NULL) // 递归出口
        return 0;
    int depthLeft = GetDepth(pRoot->m_pLeft);
    int depthRight = GetDepth(pRoot->m_pRight);
    return depthLeft > depthRight ? (depthLeft + 1) : (depthRight + 1);
}
```

二叉树遍历：

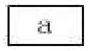
沿着某条搜索路线，依次对树中每个结点均做一次且仅做一次访问

设置一个队列，然后只要队列不为空，将对首元素的左右孩子加入队列（如果左右孩子不为空），然后将队列的首元素出对即可，如下图所示：

二叉树如下图所示：



那么，整个过程如下：



a	a入队
a b c	a出队之前，将a的左右孩子 b, c入队
b c	a出队
b c d e	b出队之前，将b的左右孩子 d, e入队
c d e	b出队
c d e f	c出队之前，将c的右孩子f入队(c的左孩子为空)
d e f	c出队
e f	d出队
f	e出队
	f出队

自然，就输出了a,b,c,d,e,f

3. 前序遍历，中序遍历，后序遍历

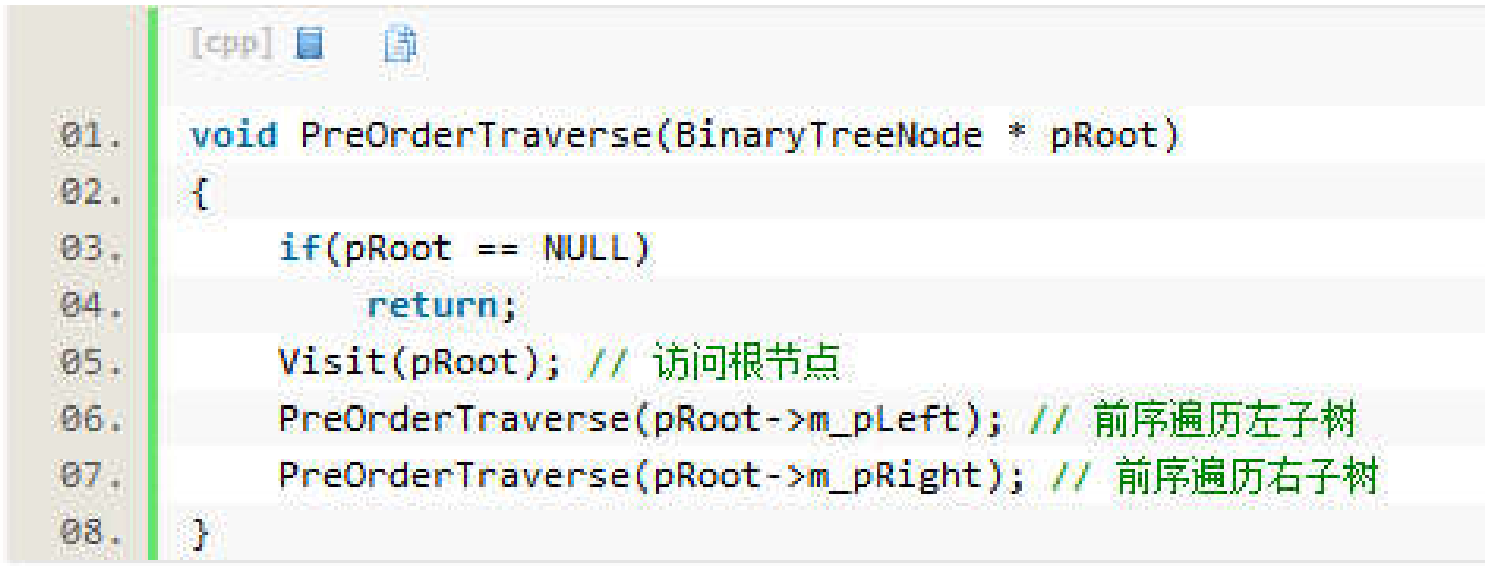
前序遍历首先访问根结点然后遍历左子树，最后遍历右子树。在遍历左、右子树时，仍然先访问根结点，然后遍历左子树，最后遍历右子树。

前序遍历递归解法：

（1）如果二叉树为空，空操作

（2）如果二叉树不为空，访问根节点，前序遍历左子树，前序遍历右子树

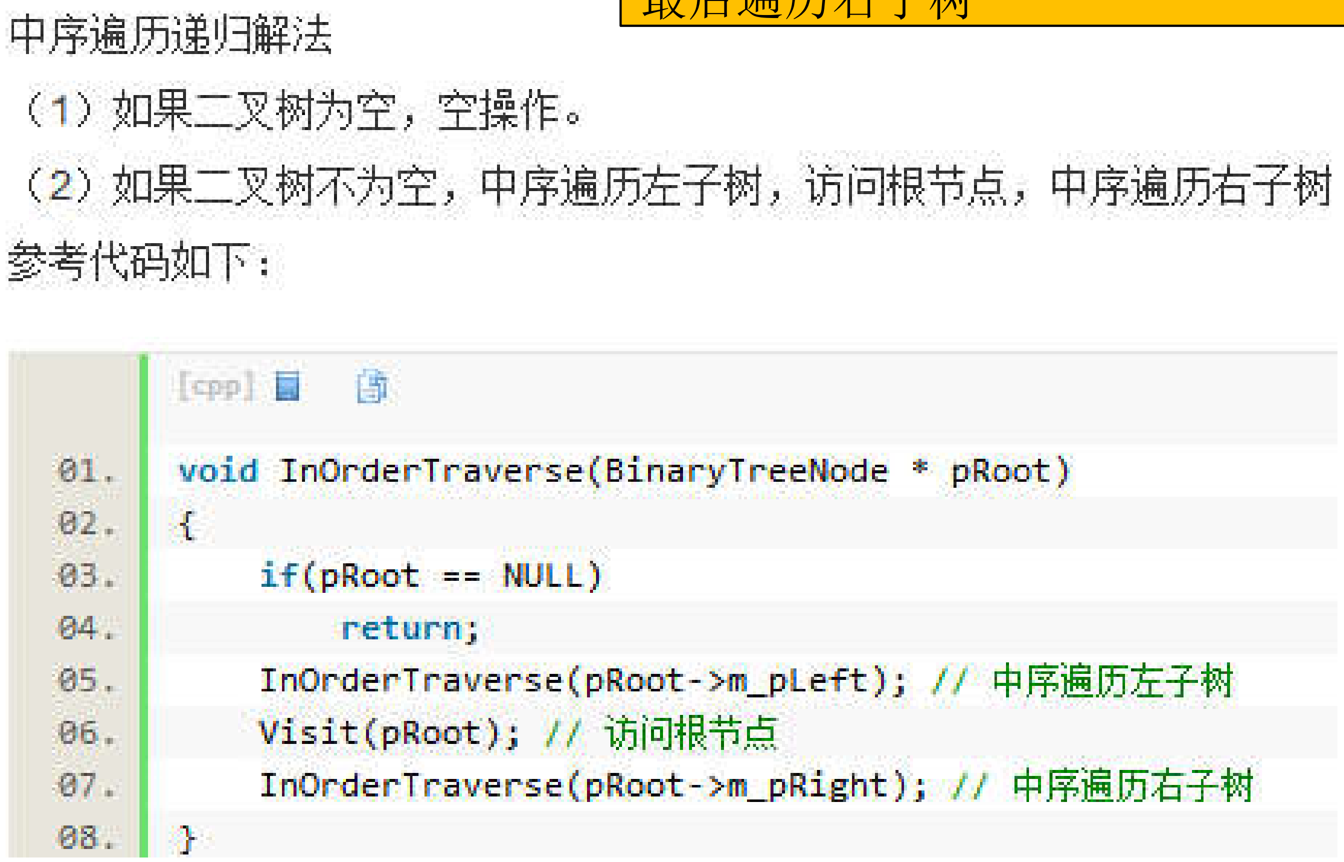
参考代码如下：



[cpp]

```cpp
void PreOrderTraverse(BinaryTreeNode * pRoot)
{
    if(pRoot == NULL)
        return;
    Visit(pRoot); // 访问根节点
    PreOrderTraverse(pRoot->m_pLeft); // 前序遍历左子树
    PreOrderTraverse(pRoot->m_pRight); // 前序遍历右子树
}
```

中序遍历首先遍历左子树，然后访问根结点，最后遍历右子树。在遍历左、右子树时，仍然先遍历左子树，再访问根结点，最后遍历右子树



中序遍历递归解法

（1）如果二叉树为空，空操作。

（2）如果二叉树不为空，中序遍历左子树，访问根节点，中序遍历右子树

参考代码如下：

[cpp]

```cpp
void InOrderTraverse(BinaryTreeNode * pRoot)
{
    if(pRoot == NULL)
        return;
    InOrderTraverse(pRoot->m_pLeft); // 中序遍历左子树
    Visit(pRoot); // 访问根节点
    InOrderTraverse(pRoot->m_pRight); // 中序遍历右子树
}
```

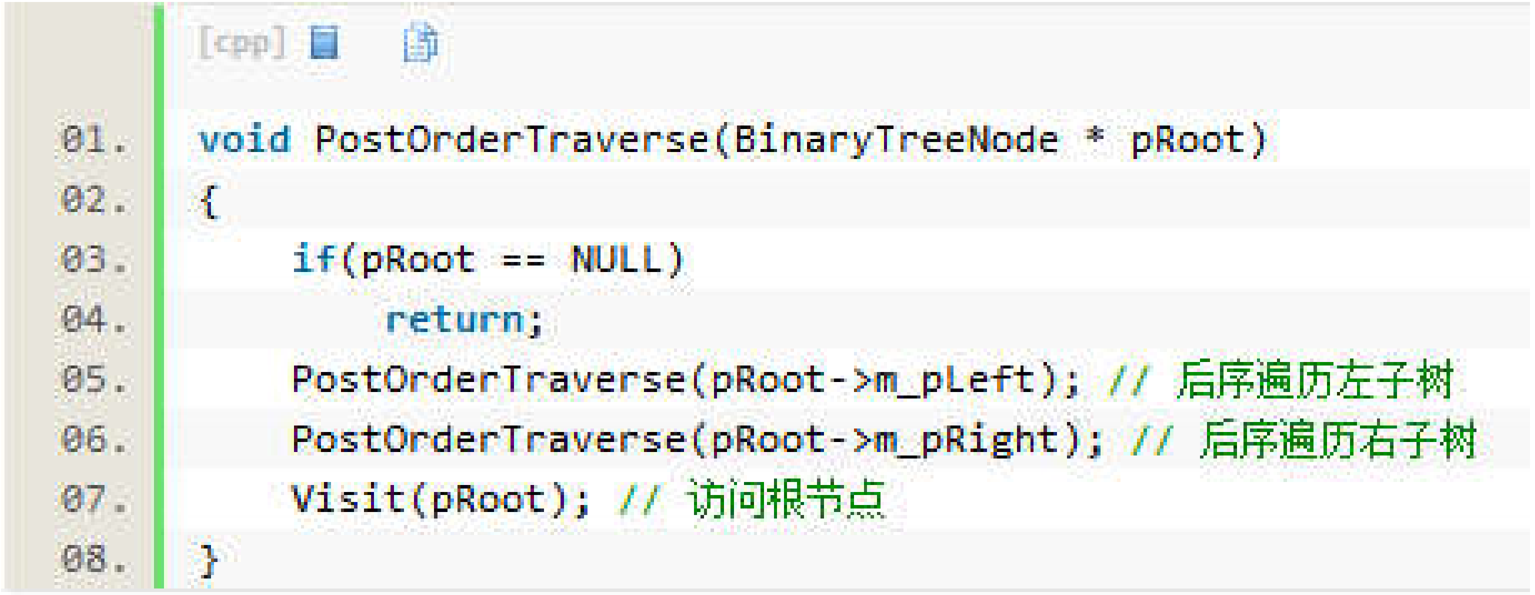
后序遍历首先遍历左子树，然后遍历右子树，最后访问根结点。在遍历左、右子树时，仍然先遍历左子树，再遍历右子树，最后访问根结点。

后序遍历递归解法

（1）如果二叉树为空，空操作

（2）如果二叉树不为空，后序遍历左子树，后序遍历右子树，访问根节点

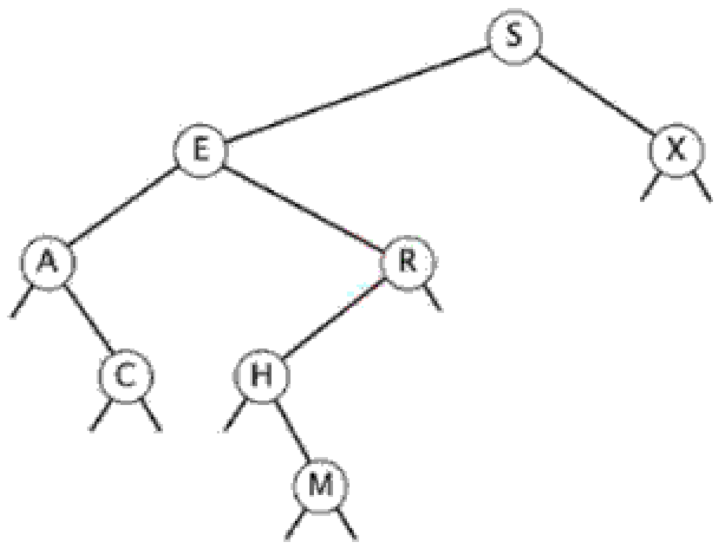
参考代码如下：



[cpp]

```cpp
void PostOrderTraverse(BinaryTreeNode * pRoot)
{
    if(pRoot == NULL)
        return;
    PostOrderTraverse(pRoot->m_pLeft); // 后序遍历左子树
    PostOrderTraverse(pRoot->m_pRight); // 后序遍历右子树
    Visit(pRoot); // 访问根节点
}
```

insert G



Manipulating Data Structures

- Ideally, a data structure should be manipulated solely by pre-defined procedures.
  - Example: A stack typically needs at least `push` and `pop` procedures.
  - The data structure along with these procedures constitutes a complete abstract tool.

Figure 8.19 **A procedure for printing a linked list**

```
procedure PrintList (List)
CurrentPointer ← head pointer of List.
while (CurrentPointer is not NIL) do
    (Print the name in the entry pointed to by CurrentPointer;
     Observe the value in the pointer cell of the List entry
      pointed to by CurrentPointer, and reassign CurrentPointer
      to be that value.)
```

Case Study

Problem: Construct an abstract tool consisting of a list of names in alphabetical order along with the operations search, print, and insert.

Figure 8.20 **The letters A through M arranged in an ordered tree**

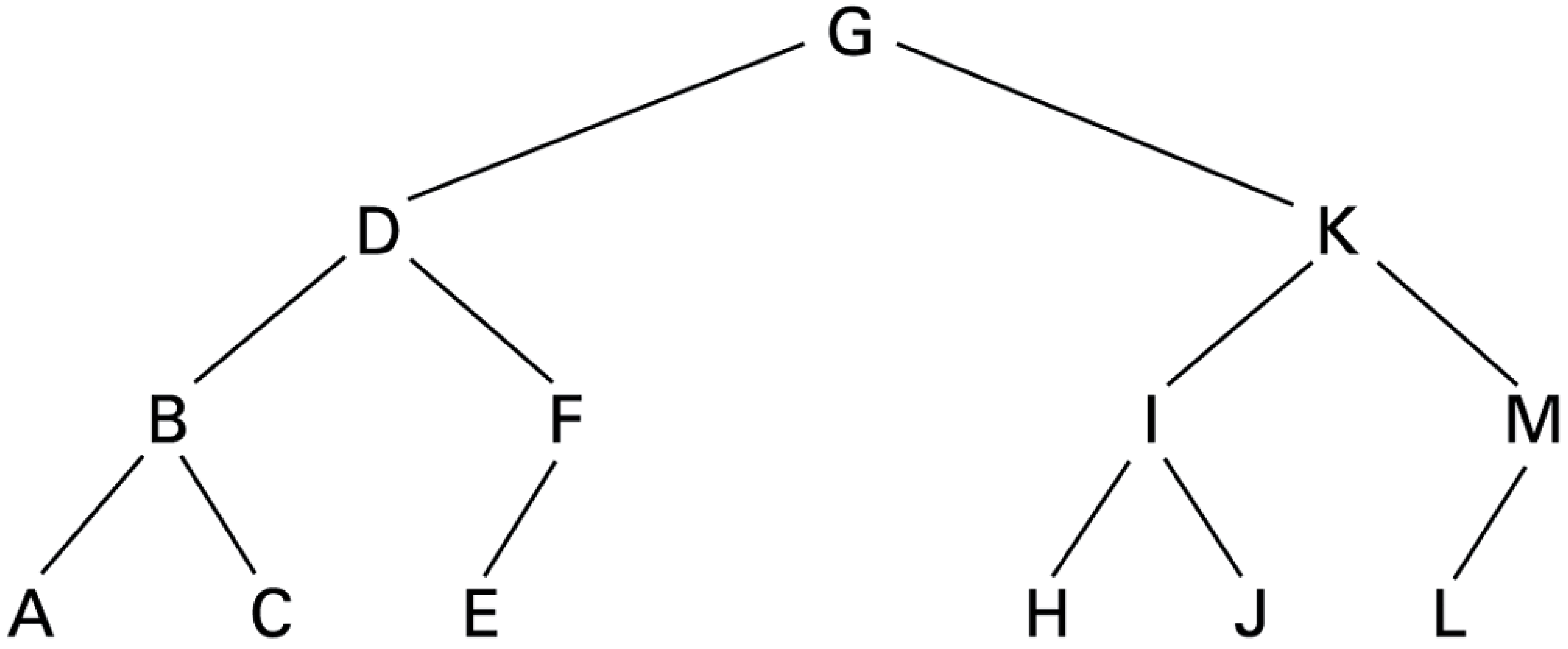


Figure 8.21 **The binary search as it would appear if the list were implemented as a linked binary tree**

```
procedure Search(Tree, TargetValue)

if (root pointer of Tree = NIL)
  then
     (declare the search a failure)
  else
     (execute the block of instructions below that is
      associated with the appropriate case)
      case 1: TargetValue = value of root node
                (Report that the search succeeded)
      case 2: TargetValue < value of root node
                (Apply the procedure Search to see if
                   TargetValue is in the subtree identified
                    by the root's left child pointer and
                    report the result of that search)
      case 3: TargetValue > value of root node
                (Apply the procedure Search to see if
                    TargetValue is in the subtree identified
                    by the root's right child pointer and
                    report the result of that search)
     ) end if
```

Figure 8.22 **The successively smaller trees considered by the procedure in Figure 8.18 when searching for the letter J**

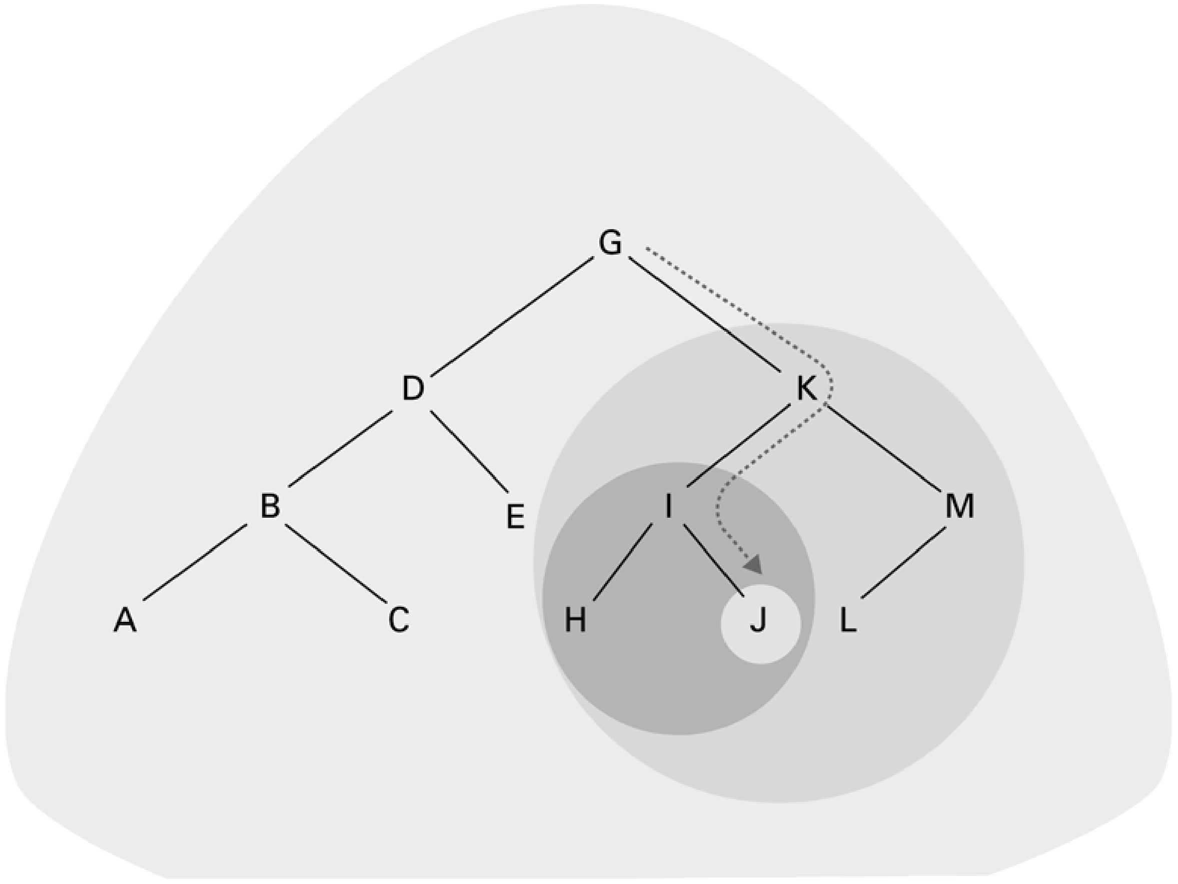


Figure 8.23 **Printing a search tree in alphabetical order**

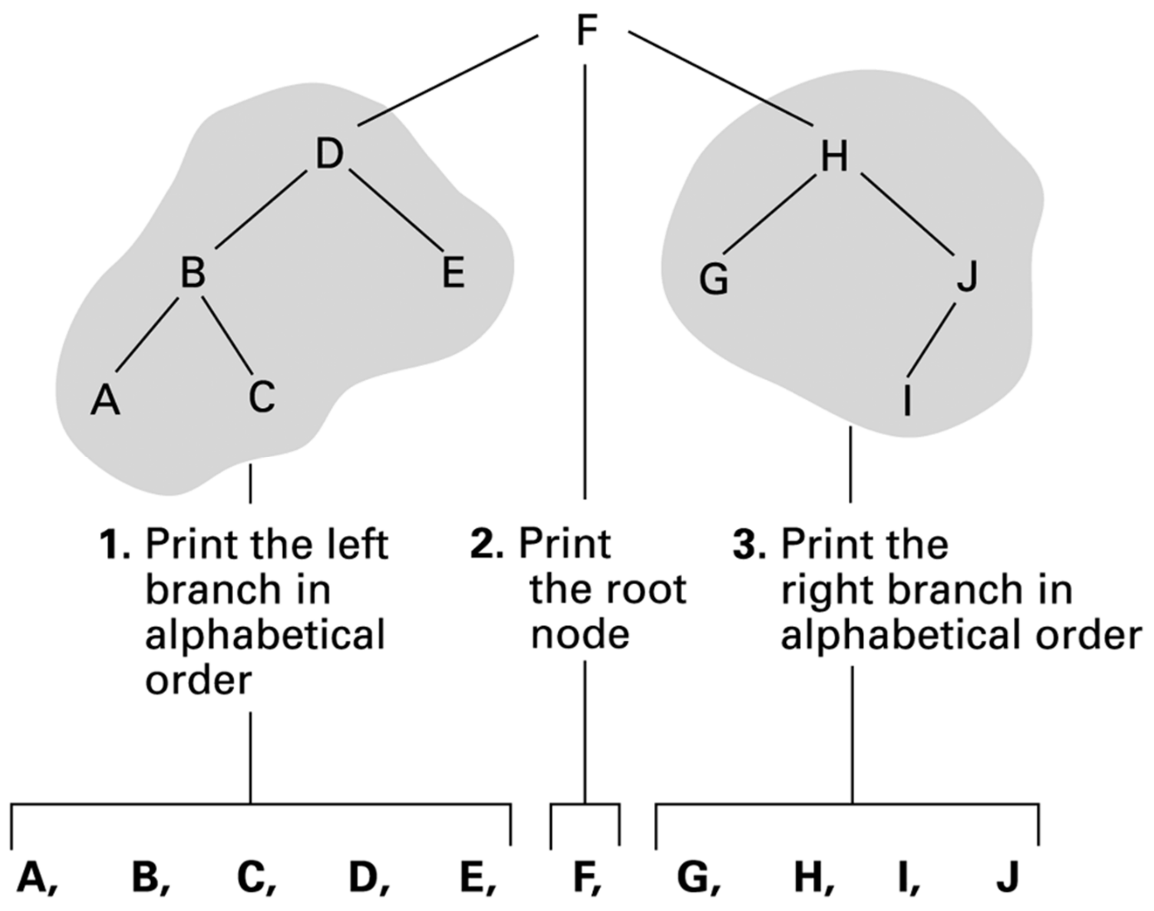
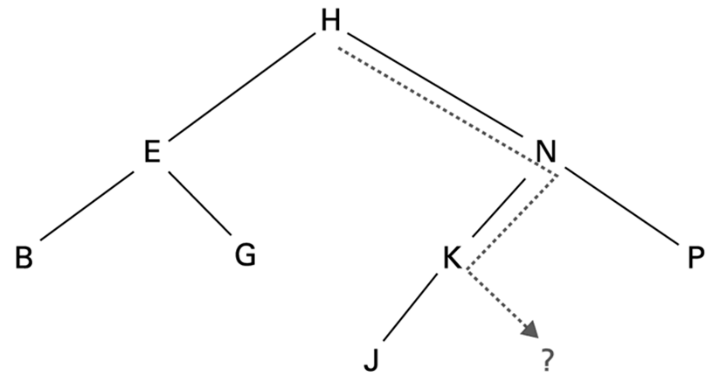

F
D
H
B
E
G
J
A
C
I
1. Print the left branch in alphabetical order
2. Print the root node
3. Print the right branch in alphabetical order
A, B, C, D, E, F, G, H, I, J

Figure 8.24 **A procedure for printing the data in a binary tree**

```
procedure PrintTree (Tree)

if (Tree is not empty)
   then (Apply the procedure PrintTree to the tree that
              appears as the left branch in Tree;
          Print the root node of Tree;
          Apply the procedure PrintTree to the tree that
              appears as the right branch in Tree)
```

Figure 8.25 **Inserting the entry M into the list B, E, G, H, J, K, N, P stored as a tree**

a. Search for the new entry until its absence is detected



b. This is the position in which the new entry should be attached

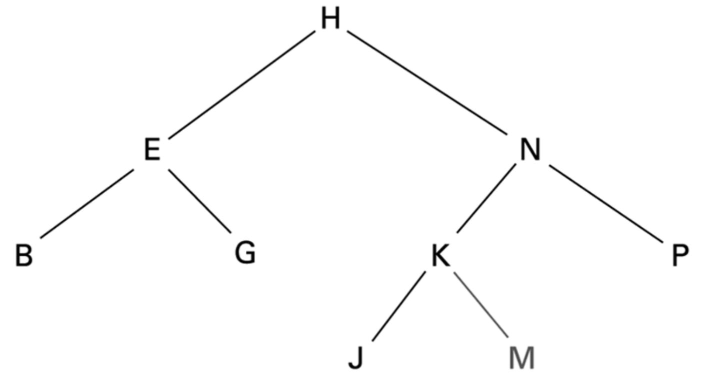


Figure 8.26 **A procedure for inserting a new entry in a list stored as a binary tree**

```
procedure Insert(Tree, NewValue)

if (root pointer of Tree = NIL)
  (set the root pointer to point to a new leaf
            containing NewValue)
  else (execute the block of instructions below that is
            associated with the appropriate case)
              case 1: NewValue = value of root node
                      (Do nothing)
              case 2: NewValue < value of root node
                      (if (left child pointer of root node = NIL)
                                then (set that pointer to point to a new
                                           leaf node containing NewValue)
                                else (apply the procedure Insert to insert
                                           NewValue into the subtree identified
                                           by the left child pointer)
              case 3: NewValue > value of root node
                      (if (right child pointer of root node = NIL)
                                then (set that pointer to point to a new
                                           leaf node containing NewValue)
                                else (apply the procedure Insert to insert
                                           NewValue into the subtree identified
                                           by the right child pointer)
          ) end if
```

User-defined Data Type

- A template for a heterogeneous structure
- Example:

```
define type EmployeeType to be
{char     Name[25];
 int      Age;
 real     SkillRating;
}
```

Abstract Data Type

- A user-defined data type with procedures for access and manipulation
- Example:

```
define type StackType to be
{int StackEntries[20];
 int StackPointer = 0;
 procedure push(value)
    {StackEntries[StackPointer] ← value;
     StackPointer ¬ StackPointer + 1;
    }
 procedure pop . . .
}
```

Figure 8.27 **A stack of integers implemented in Java and C#**

```
class StackOfIntegers
{private int[] StackEntries = new int[20];
 private int StackPointer = 0;


 public void push(int NewEntry)
 {if (StackPointer < 20)
    StackEntries[StackPointer++] = NewEntry;
 }

 public int pop()
 {if (StackPointer > 0) return StackEntries[--StackPointer];
  else return 0;
 }
}
```

Key points

- Arrays, lists, trees, stacks and queues
- Static and dynamic structures, pointers
- Storing arrays, lists
- Storing stacks and queues
- Storing binary trees