# 数据结构

## 第1章 绪论

**数据**(Data)是对客观事物的符号表示，在计算机科学中是指所有能输入到计算机中并被计算机程序处理的符号的总称。

**数据元素**(Data Element)是数据的基本单位，在计算机程序中通常作为一个整体进行考虑和处理。数据元素可由若干个**数据项**(Data Item)组成。数据项是数据的不可分割的最小单位。

**数据结构**(Data Structure)是相互之间存在某种关系的数据元素的集合。数据结构的三要素：逻辑结构、物理结构和数据的运算。

结构中定义的“关系”描述的是数据元素之间的逻辑关系，称为为数据的**逻辑结构**。逻辑结构分为集合、线性结构和非线性结构。线性表是典型的线性结构；树和图是典型的非线性结构。

数据结构在计算机中的表示（又称映像）称为数据的物理结构，又称**存储结构**。典型的存储结构有4种，顺序存储，链式存储，索引存储和散列存储。

数据的运算是在数据的逻辑结构上定义的操作算法，如查找、插入、删除等。



**数据类型**(Data Type)是一个值的集合和定义在这个值集上的一组操作的总称。数据类型分为原子类型和结构类型。

**算法**是对特定问题求解步骤的一种描述，它是指令的有限序列。

算法的五大特性：有穷性、确定性、可行性、输入、输出。

算法评价：时间复杂度，空间复杂度。

评价好的算法有四个方面：正确性，易读性，健壮性，时空效率。

算法与程序的区别：（1）一个程序不一定满足有穷性，而算法必须有穷；（2）算法代表了对问题的解，而程序则是算法在计算机上的特定的实现。

本笔记的所有代码，都包含一个公共的头文件 config.h。

/\*\* @file config.h

\* @brief 公共头文件

\* @author soulmachine@gmail.com

\* @date 2010-8-3

\* @version 0.1

\* @note 无

\*/

#ifndef \_CONFIG\_H\_

#define \_CONFIG\_H\_

#ifdef \_\_cplusplus

extern "C"

{

#endif

#ifndef NULL

#ifdef \_\_cplusplus

#define NULL 0

#else

#define NULL ((void \*)0)

#endif

#endif

#ifndef \_\_cplusplus

typedef char bool;

#define false 0

#define true (!false)

#endif

/\* no error \*/

#define SUCCESS 0

/\* 错误码\*/

#define ERR\_NULL\_POINTER 1

#define ERR\_MALLOC\_FAILED 2

#define ERR\_OTHER 3

#ifdef \_\_cplusplus

}

#endif /\* end of \_cplusplus \*/

#endif /\* end of \_CONFIG\_H\_ \*/

所有代码都是纯C，基于对象（不是面向对象），不透明结构体。

必须背诵的代码：二叉树的前序遍历(pre\_order)，中序遍历(in\_order)，折半查找(binary\_search)，快速排序(pre\_order)。其他算法不需要背代码但要记住算法流程，会画图画表格。

## 第2章 线性表

### 2.1 线性表的定义

线性结构：①存在唯一一个被称作“第一个”的数据元素；②存在为一个被称作“最后一个”的数据元素；③除第一个元素外，集合中的每个元素均只有一个直接前驱；④除最后一个元素外，集合中的每个元素均只有一个后继（共四个唯一）。



### 2.2 线性表的顺序存储

1. 查找

最好情况：查找的元素就在表头（即i=0），仅需比较一次，时间复杂度为O(1)。

最坏情况：查找的元素在表尾（即i=n-1），需比较n次，时间复杂度为O(n)。

平均情况： 设pi(pi=1/n)是查找的元素在第i个位置上的概率，则查找某个元素所需的平均比较次数为



2. 插入

最好情况：在表尾插入（即i=n），无需移动元素，时间复杂度为O(1)。

最坏情况：在表头插入（即i=0），所有元素都需要后移一格，时间复杂度为O(n)。

平均情况： 设pi(pi=1/(n+1))是在第i个位置上插入一个元素的概率，则插入一个元素所需的平均移动次数为



3. 删除

最好情况：删除表尾元素（即i=n-1），无需移动元素，时间复杂度为O(1)。

最坏情况：删除表头元素（即i=0），除第一个元素外的所有元素都需要前移一格，时间复杂度为O(n)。

平均情况： 设pi(pi=1/n)是删除第i个位置上元素的概率，则删除一个元素所需的平均移动次数为



### 2.3 线性表的链式存储

单链表的建立、查找、插入、删除运算。

/\*\* @file single\_list.h

\* @brief 单链表

\* @author soulmachine@gmail.com

\* @date 2010-7-30

\* @version 0.1

\* @note 单链表的头文件

\*/

#ifndef \_SINGLE\_LIST\_H\_

#define \_SINGLE\_LIST\_H\_

#ifdef \_\_cplusplus

extern "C"

{

#endif

#include "config.h"

/\*\* 链表结点所带的数据，这里用int仅作演示. \*/

typedef int single\_list\_node\_data\_t;

/\*\* 单链表结点，不透明的结构体. \*/

typedef struct single\_list\_node\_t single\_list\_node\_t;

/\*\*

\* @brief 建立单链表，头插法

\* @param[in] l 带头节点的单链表的头指针的指针

\* @param[in] data 原始数据

\* @param[in] data 元素个数

\* @return 头节点的头指针，失败则返回NULL

\* @remarks 无

\*/

extern single\_list\_node\_t\* single\_list\_create(single\_list\_node\_t \*\*l,

const int data[], int n);

/\*\*

\* @brief 查找

\* @param[in] l 带头节点的单链表的头指针

\* @param[in] k 查找第k个元素

\* @return 返回指向第k个元素的地址，失败则返回NULL

\* @note k是从1开始而不是0

\* @remarks 无

\*/

extern single\_list\_node\_t\* single\_list\_find(

const single\_list\_node\_t \*l, int k);

/\*\*

\* @brief 插入

\* @param[in] l 带头节点的点链表的头指针

\* @param[in] k 将新元素插入到第k个位置

\* @param[in] elem 要插入的元素

\* @return 成功返回SUCCESS，失败返回错误码

\* @note k是从开始而不是

\* @remarks 无

\*/

int single\_list\_insert(single\_list\_node\_t \*l, int k,

single\_list\_node\_data\_t elem);

/\*\*

\* @brief 删除

\* @param[in] l 带头节点的点链表的头指针

\* @param[in] k 查找第k个元素

\* @return 成功返回被删节点的地址，失败返回NULL

\* @note 无

\* @remarks 无

\*/

extern single\_list\_node\_t\* single\_list\_remove(single\_list\_node\_t \*l,

int k);

#ifdef \_\_cplusplus

}

#endif

#endif /\* end of \_SINGLE\_LIST\_H\_ \*/

/\*\* @file single\_list.c

\* @brief 单链表

\* @author soulmachine@gmail.com

\* @date 2010-7-30

\* @version 0.1

\* @note 单链表的源文件

\*/

#include <stdlib.h>

#include "single\_list.h"

/\*

\*@struct

\*@brief 单链表节点

\*@author soulmachine@gmail.com

\*@note 无

\*/

struct single\_list\_node\_t {

/\* 节点中存放实际数据\*/

single\_list\_node\_data\_t data;

/\* 指向下一个节点\*/

single\_list\_node\_t \*next;

};

single\_list\_node\_t\* single\_list\_create(single\_list\_node\_t \*\*l,

const int data[], int n)

{

int i;

single\_list\_node\_t \*ph;

if(l == NULL){

return NULL;

} else{ // 分配头节点

\*l =(single\_list\_node\_t\*)malloc(sizeof(single\_list\_node\_t));

(\*l)->next = NULL;

ph = \*l;

}

/\* 插入数据\*/

for(i = 0; i < n; i++){

single\_list\_node\_t \*p = (single\_list\_node\_t\*)malloc(sizeof(

single\_list\_node\_t));

p->data = data[i];

p->next = ph->next;

ph->next = p;

}

return ph;

}

single\_list\_node\_t\* single\_list\_find(const single\_list\_node\_t \*l, int k)

{

const single\_list\_node\_t\* p = l;

int i =0;

while(p != NULL && i < k) {

p = p->next;

i++;

}

return (single\_list\_node\_t\*)p;

}

int single\_list\_insert(single\_list\_node\_t \*l, int k,

single\_list\_node\_data\_t elem)

{

single\_list\_node\_t \*p, \*q;

p = single\_list\_find(l, k -1);

if(p == NULL) return ERR\_NULL\_POINTER;

q = (single\_list\_node\_t \*) malloc(sizeof(single\_list\_node\_t));

if(q == NULL) return ERR\_MALLOC\_FAILED;

q->data = elem;

q->next = p->next;

p->next = q;

return SUCCESS;

}

single\_list\_node\_t\* single\_list\_remove(single\_list\_node\_t \*l, int k)

{

single\_list\_node\_t \*p, \*q;

p = single\_list\_find(l, k -1);

if(p == NULL) {

return NULL;

}

q = p->next;

p->next = q->next;

return q;

}

## 第3章 栈和队列

### 3.1 栈和队列的定义

栈只能在表的一端插入和删除，先进后出（LIFO, Last In, First Out）。

队列只能在表的一端（队尾rear）插入，另一端（队头front）删除，先进先出（FIFO, First In, First Out）。

出栈序列的数目：





为了判断队列是空还是满，有3种处理方式：

1、浪费一个元素空间，以区分队空和队满。空，Q.front==Q.rear；满，(Q.rear+1)%n==Q.front。

2、增设一个size数据成员，记录实际元素个数。空，size==0，满，size==n。

3、增设一个标志域。空，Q.rear==Q.front&&tag==0；满，Q.rear==Q.front&&tag==1。

最常用的是第一种方式。

### 3.2 栈和队列的顺序存储

栈的C语言实现如下。

/\*\* @file stack.h

\* @brief 栈.

\* @author soulmachine@gmail.com

\* @date 2010-7-31

\* @version 0.1

\* @note 栈的头文件.

\*/

#ifndef \_STACK\_H\_

#define \_STACK\_H\_

#ifdef \_\_cplusplus

extern "C"

{

#endif

#include "config.h"

/\*\* 栈的不透明结构体. \*/

typedef struct stack\_t stack\_t;

/\*\*

\* @brief 初始化栈.

\* @param[inout] s 栈对象的二级指针

\* @param[in] elem\_size 单个元素的大小

\* @return 成功返回0，失败返回错误码

\* @note 本函数会分配内存，创建一个stack\_t对象，将其地址

\* 存放在s所指向的指针中

\* @remarks 无

\*/

extern int stack\_init(stack\_t \*\*s, int elem\_size);

/\*\*

\* @brief 释放栈的内存空间.

\* @param[inout] s 栈对象的二级指针

\* @return 成功返回0，失败返回错误码

\* @note 释放掉栈的数据所占用的内存块，并释放栈对象本身

\* 所的内存

\* @remarks 无

\*/

extern int stack\_uninit(stack\_t \*\*s);

/\*\*

\* @brief 清空元素，但不释放内存.

\* @param[in] s 指向栈对象的指针

\* @return 无

\* @note 无

\* @remarks 无

\*/

extern void stack\_clear(stack\_t \*s);

/\*\*

\* @brief 判断栈是否为空.

\* @param[in] s 指向栈对象的指针

\* @return 是空，返回TRUE，否则返回FALSE

\* @note 无

\* @remarks 无

\*/

extern bool stack\_is\_empty(const stack\_t \*s);

/\*\*

\* @brief 判断栈是否已满.

\* @param[in] s 指向栈对象的指针

\* @return 满，返回TRUE，否则返回FALSE

\* @note 无

\* @remarks 无

\*/

extern bool stack\_is\_full(const stack\_t \*s);

/\*\*

\* @brief 进栈.

\* @param[in] s 指向栈对象的指针

\* @param[in] x 要进栈的元素

\* @return 成功返回0，失败返回错误码

\* @note 无

\* @remarks 无

\*/

extern int stack\_push(stack\_t \*s, const void\* x);

/\*\*

\* @brief 退栈.

\* @param[in] s 指向栈对象的指针

\* @param[out] x 存放退出栈的元素

\* @return 成功返回0，失败返回错误码

\* @note 无

\* @remarks 无

\*/

extern int stack\_pop(stack\_t \*s, void\* x);

#ifdef \_\_cplusplus

}

#endif

#endif /\* end of \_STACK\_H\_ \*/

/\*\* @file stack.c

\* @brief 栈.

\* @author soulmachine@gmail.com

\* @date 2010-7-31

\* @version 0.1

\* @note 栈的源代码文件

\*/

#include "stack.h"

#include <stdlib.h>

#include <string.h>

/\*

\*@struct

\*@brief 栈的结构体定义

\*@author soulmachine@gmail.com

\*@note 无

\*/

struct stack\_t {

int top; /\* 指向栈顶元素下一个位置\*/

int elem\_size; /\* 单个元素大小\*/

int capacity; /\* 容量大小，以元素为单位\*/

void \*data; /\* 存放数据的内存块\*/

};

int stack\_init(stack\_t \*\*s, int elem\_size)

{

if(s == NULL) return ERR\_NULL\_POINTER;

(\*s) = (stack\_t\*)malloc(sizeof(stack\_t));

if((\*s) == NULL) return ERR\_MALLOC\_FAILED;

(\*s)->top = 0;

(\*s)->elem\_size = elem\_size;

(\*s)->capacity = 4;

(\*s)->data = malloc((\*s)->capacity \* (\*s)->elem\_size);

return SUCCESS;

}

int stack\_uninit(stack\_t \*\*s)

{

if((s !=NULL) && (\*s != NULL)) {

free((\*s)->data);

free((\*s));

(\*s) = NULL;

}

return 0;

}

void stack\_clear(stack\_t \*s)

{

if(s != NULL) s->top = 0;

}

bool stack\_is\_empty(const stack\_t \*s) {

if(s != NULL) {

return s->top == 0;

} else {

return false;

}

}

bool stack\_is\_full(const stack\_t \*s)

{

if(s != NULL) {

return s->top == s->capacity;

} else {

return false;

}

}

int stack\_push(stack\_t \*s, const void\* x)

{

if(stack\_is\_full(s)) {

void\* const tmp = realloc(s->data,

s->capacity \* 2 \* s->elem\_size);

if(tmp == NULL) return ERR\_MALLOC\_FAILED;

s->capacity \*= 2;

s->data = tmp;

}

/\* s->data[(s->top)++] = x; \*/

memcpy((char\*)s->data + s->top \* s->elem\_size,

x, s->elem\_size);

s->top++;

return SUCCESS;

}

int stack\_pop(stack\_t \*s, void\* x)

{

if(!stack\_is\_empty(s)) {

/\* \*x = s->data[--(s->top)]; \*/

s->top--;

memcpy(x, (char\*)s->data + s->top \* s->elem\_size,

s->elem\_size);

return SUCCESS;

} else {

return ERR\_OTHER;

}

}

队列的C语言实现如下。

/\*\* @file queue.h

\* @brief 队列.

\* @author soulmachine@gmail.com

\* @date 2010-7-30

\* @version 0.1

\* @note 实现了一个循环队列

\*/

#ifndef \_QUEUE\_H\_

#define \_QUEUE\_H\_

#ifdef \_\_cplusplus

extern "C"

{

#endif

#include "config.h"

/\*\* 队列的不透明结构体. \*/

typedef struct queue\_t queue\_t;

/\*\*

\* @brief 初始化队列.

\* @param[out] q 队列结构体的二级指针

\* @return 成功返回0，失败返回错误码

\* @note 本函数会分配内存，创建一个queue\_t对象，将其地址

\* 存放在h所指向的指针中

\* @remarks 无

\*/

extern int queue\_init(queue\_t \*\*q, int elem\_size);

/\*\*

\* @brief 释放队列的内存空间.

\* @return 成功返回0，失败返回错误码

\* @note 释放掉队列的数据所占用的内存块，并释放队列对象

\* 本身所占用的内存

\* @remarks 无

\*/

extern int queue\_uninit(queue\_t \*\*q);

/\*\*

\* @brief 清空元素，但不释放内存.

\* @param[in] q 指向队列结构体的指针

\* @return 无

\* @note 无

\* @remarks 无

\*/

extern void queue\_clear(queue\_t \*q);

/\*\*

\* @brief 判断队列是否为空.

\* @param[in] q 指向队列结构体的指针

\* @return 是空，返回TRUE，否则返回FALSE

\* @note 无

\* @remarks 无

\*/

extern bool queue\_is\_empty(const queue\_t \*q);

/\*\*

\* @brief 判断队列是否已满.

\* @param[in] q 指向队列结构体的指针

\* @return 满，返回TRUE，否则返回FALSE

\* @note 无

\* @remarks 无

\*/

extern bool queue\_is\_full(const queue\_t \*q);

/\*\*

\* @brief 在队尾添加元素.

\* @param[in] q 指向队列结构体的指针

\* @param[in] x 要添加的元素

\* @return 成功返回0，失败返回错误码

\* @note 无

\* @remarks 无

\*/

extern int queue\_enter(queue\_t \*q, const void \*x);

/\*\*

\* @brief 在队头删除元素.

\* @param[in] q 指向队列结构体的指针

\* @param[out] x 存放退出队列的元素

\* @return 成功返回0，失败返回错误码

\* @note 无

\* @remarks 无

\*/

extern int queue\_detach(queue\_t \*q, void \*x);

#ifdef \_\_cplusplus

}

#endif

#endif /\* end of \_QUEUE\_H\_ \*/

/\*\* @file queue.c

\* @brief 队列.

\* @author soulmachine@gmail.com

\* @date 2010-7-30

\* @version 0.1

\* @note 实现了一个循环队列

\*/

#include "queue.h"

#include <malloc.h> /\* for malloc(), free() \*/

#include <string.h> /\* for memcpy() \*/

/\*

\*@struct

\*@brief 队列的结构体定义.

\*@note 无

\*/

struct queue\_t {

int front; /\* 队头\*/

int rear; /\* 队尾\*/

int elem\_size; /\* 单个元素的大小\*/

int capacity; /\* 容量大小，以元素为单位\*/

void \*data; /\* 存放数据的内存块\*/

};

int queue\_init(queue\_t \*\*q, int elem\_size)

{

if(q == NULL) return ERR\_NULL\_POINTER;

(\*q) = (queue\_t\*)malloc(sizeof(queue\_t));

if((\*q) == NULL) return ERR\_MALLOC\_FAILED;

(\*q)->front = 0;

(\*q)->rear = 0;

(\*q)->elem\_size = elem\_size;

(\*q)->capacity = 4;

(\*q)->data = malloc((\*q)->capacity \* (\*q)->elem\_size);

return SUCCESS;

}

int queue\_uninit(queue\_t \*\*q)

{

if((q !=NULL) && (\*q != NULL)) {

free((\*q)->data);

free((\*q));

(\*q) = NULL;

}

return 0;

}

void queue\_clear(queue\_t \*q)

{

if(q != NULL) {

q->front = 0;

q->rear = 0;

}

}

bool queue\_is\_empty(const queue\_t \*q)

{

if(q != NULL) {

return q->front == q->rear;

} else {

return false;

}

}

bool queue\_is\_full(const queue\_t \*q)

{

if(q != NULL) {

return (q->rear + 1) % q->capacity == q->front;

} else {

return false;

}

}

int queue\_enter(queue\_t \*q, const void \*x)

{

if(queue\_is\_full(q)) {

void\* const tmp = malloc(q->capacity \* 2 \* q->elem\_size);

if(tmp == NULL) return ERR\_MALLOC\_FAILED;

if(q->front < q->rear) {

memcpy(tmp,

(char\*)q->data + q->front \* q->elem\_size,

(q->rear - q->front) \* q->elem\_size);

q->rear -= q->front;

q->front = 0;

} else if(q->front > q->rear) {

/\* 拷贝q->front 到q->capacity 之间的数据\*/

memcpy(tmp,

(char\*)q->data + q->front \* q->elem\_size,

(q->capacity - q->front) \* q->elem\_size);

/\* 拷贝q->data[0 到q->data[rear] 之间的数据\*/

memcpy((char\*)tmp +

(q->capacity - q->front) \* q->elem\_size,

q->data, q->rear \* q->elem\_size);

q->rear += q->capacity - q->front;

q->front = 0;

}

free(q->data);

q->data = tmp;

q->capacity \*= 2;

}

/\* q->data[(q->rear)++] = x; \*/

memcpy((char\*)q->data + q->elem\_size \* q->rear,

x, q->elem\_size);

q->rear = (q->rear + 1) % q->capacity;

return SUCCESS;

}

int queue\_detach(queue\_t \*q, void\* x)

{

if(!queue\_is\_empty(q)) {

/\* \*x = q->data[(q->front)++]; \*/

memcpy(x, (char\*)q->data + q->elem\_size \* q->front,

q->elem\_size);

q->front = (q->front + 1) % q->capacity;

return SUCCESS;

} else {

return ERR\_OTHER;

}

}

### 3.3 栈的应用

**1、数制转换**

/\*\* @file base\_trans.c

\* @brief 数制转换

\* @author soulmachine@gmail.com

\* @date 2010-7-30

\* @version 0.1

\* @note 无

\*/

#include "stack.h"

#include <stdio.h>

/\*

\* @brief 将十进制数n转化为d进制

\* @author soulmachine

\* @param[in] n 十进制数n

\* @param[in] d 转化为d进制，比如16进制

\* @return 无

\*/

static void base\_trans(int n, const int d)

{

stack\_t \*s;

int e;

if(!stack\_init(&s, sizeof(int))) return;

while(n != 0) {

e = n % d;

(void)stack\_push(s, &e);

n /= d;

}

while(!stack\_is\_empty(s)) {

(void)stack\_pop(s, &e);

printf("%x", e);

}

stack\_uninit(&s);

return;

}

**2、括号匹配的检验**

[ ( [ ] [ ] ) ]

**3、迷宫求解**

**4、表达式的计算**

中缀表达式转化为后缀表达式，写出处理过程和栈的变化。

**例** 现有一个中缀表达式：a+b\*(c-d)+e/f#，请给出转换为后缀表达式时的处理过程及栈的相应变化（提示：运算符的优先级如下表所示，其中，icp表示正在扫描到的运算符的优先级，isp是该运算符进栈后的优先级，字符’#’为表达式结束符）。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 操作符 | # | ( | \*, / | +, - | ) |
| isp | 0 | 1 | 5 | 3 | 6 |
| icp | 0 | 6 | 4 | 2 | 1 |

【解答】isp是栈内（in stack priority）优先级，icp是栈外（in coming priority）优先级。

左括号’(’的栈外优先级最高，它一来到就立即进栈，但当它进入栈中后，其栈内优先级变得极低，以便括号内的其他操作符进栈。其他操作符进入栈中后优先级都升1，体现在中缀表达式中相同优先级的操作符自左向右计算的要求，让位于栈顶的操作符先退栈并输出。操作符优先级相等的情况只出现在括号配对或栈底的’#’号与输入流最后的’#’号配对时。前者将连续退出位于栈顶的操作符，直到遇到’(’为止。然后将’(‘退栈以对消括号，后者将结束算法。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 步序 | 扫描项 | 项类型 | 动作 | 栈的变化 | 输出 |
| 0 |  |  | ‘#’进栈，读下一符号 | # |  |
| 1 | a | 操作数 | 直接输出，读下一符号 | # | a |
| 2 | + | 操作符 | isp(‘#’)<icp(‘+’)，进栈，读下一符号 | #+ |  |
| 3 | b | 操作数 | 直接输出，读下一符号 | #+ | b |
| 4 | \* | 操作符 | isp(‘+’)<icp(‘\*’)，进栈，读下一符号 | #+\* |  |
| 5 | ( | 操作符 | isp(‘\*’)<icp(‘(’)，进栈，读下一符号 | #+\*( |  |
| 6 | c | 操作数 | 直接输出，读下一符号 | #+\*( | c |
| 7 | - | 操作符 | isp(‘(’)<icp(‘-’)，进栈，读下一符号 | #+\*(- |  |
| 8 | d | 操作数 | 直接输出，读下一符号 | #+\*(- | d |
| 9 | ) | 操作符 | isp(‘-’)>icp(‘)’)，退栈输出 | #+\*( | - |
| 10 |  |  | isp(‘(’)==icp(‘)’)，退栈，读下一符号 | #+\* |  |
| 11 | + | 操作符 | isp(‘\*’)>icp(‘+’)，退栈输出 | #+ | \* |
| 12 |  |  | isp(‘+’)>icp(‘+’)，退栈输出 | # | + |
| 13 |  |  | isp(‘#’)<icp(‘+’)，进栈，读下一符号 | #+ |  |
| 14 | e | 操作数 | 直接输出，读下一符号 | #+ | e |
| 15 | / | 操作符 | isp(‘+’)<icp(‘/’)，进栈，读下一符号 | #+/ |  |
| 16 | f | 操作数 | 直接输出，读下一符号 | #+/ | f |
| 17 | # | 操作符 | isp(‘/’)>icp(‘#’)，退栈输出 | #+ |  |
| 18 |  |  | isp(‘+’)>icp(‘#’)，退栈输出 | # | / |
| 19 |  |  | isp(‘#’)=icp(‘#’)，退栈，结束 |  | + |

扫描中缀表达式，将它转化为后缀表达式的算法描述如下：

（1）操作符栈初始化，将结束符‘#’进栈。然后读入中缀表达式字符流的首字符ch。

（2）重复执行以下步骤，直到ch=‘#’，同时栈顶的操作符也是‘#’，停止循环。

a)若ch是操作数直接输出，读入下一个字符ch。

b)若ch是操作符，判断出的ch的优先级icp和当前栈顶操作符op的优先级isp：

* 若icp(ch)>isp(op)，令ch进栈，读入下一个字符ch；
* 若icp(ch)<isp(op)，退栈并输出；
* 若icp(ch)==isp(op)，退栈但不输出，若推出的是‘(’，读入下一字符ch。

（3）算法结束，输出序列即为所需的后缀表达式。

**5、Hanoi塔问题**

（n阶Hanoi塔问题）假设有三个分别命名为X、Y和Z的塔座，在塔座X上插有n个直径大小各不相同、从小到大编号为1，2，...，n的圆盘（如图3.1所示）。

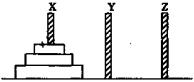


图3.1 Hanoi 塔问题

现要求将X塔上的n个圆盘移动到Z上并仍按同样的顺序叠放，圆盘移动时必须遵循下列规则：

1. 每次只能移动一个圆盘；
2. 圆盘可以插在X、Y和Z中的任一塔座上；
3. 任何时刻都不能将一个较大的圆盘压在较小的圆盘之上。

/\*

\* @brief 将塔座x上按直径有小到大且自上而下编号

\* 为至n的n个圆盘按规则搬到塔座z上，y可用做辅助塔座.

\* @param[in] n 圆盘个数

\* @param[in] x 源塔座

\* @param[in] y 辅助塔座

\* @param[in] z 目标塔座

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void hanoi(int n, char x, char y, char z)

{

if(n == 1) {

/\* 移动操作move(x,n,z)可定义为（c是初始值为的全局

变量，对搬动计数）：

printf("%i. Move disk %i from %c to %c\n",

++c, n, x, z);

\*/

move(1, x, z); /\* 将编号为1的圆盘从x移动到z \*/

return;

} else {

/\* 将x上编号1至n-1的圆盘移到y，z作辅助塔\*/

hanoi(n-1, x, z, y);

move(n,x,z); /\* 将编号为n的圆盘从x移到z \*/

/\* 将y上编号至n-1的圆盘移到z，x作辅助塔\*/

hanoi(n-1, y, x, z);

}

}

### 3.4 队列的应用

**1、打印杨辉三角**

将二项式(a+b)n展开，其系数构成如下图所示杨辉三角（即Pascal’s trigangle）。要求按行将展开式系数的前n行打印出来。

1 1 i=1

1 2 1 2

1 3 3 1 3

1 4 6 4 1 4

1 5 10 10 5 1 5

1 6 15 20 15 6 1 6

图3.2 杨辉三角



图3.3 从第i行数据计算并存放第i+1行数据

/\*

\* @brief 打印杨辉三角系数.

\* @param[in] n 指数

\* @return 无

\* @note 分行打印二项式(a+b)^n展开式的系数。在输出上一行

\* 系数的同时，将其下一行的系数预先计算好，放入队列中。

\* 在各行系数之间插入一个0。

\* @remarks 无

\*/

static void yanghui(int n)

{

int i = 1;

queue\_t \*q;

if(!queue\_init(&q, sizeof(int))) return; /\* 队列初始化\*/

(void) queue\_enter(q, &i); /\* 预先放入第一行的两个系数\*/

(void) queue\_enter(q, &i);

for(i = 1; i <= n; i++) { /\* 逐行处理\*/

int j;

int s = 0;

printf("\n"); /\* 换一行\*/

queue\_enter(q, &s); /\* 在各行间插入一个0\*/

/\* 处理第i行的i+2个系数（包括一个0）\*/

for(j = 1; j <= i+2; j++) {

int t;

int tmp;

(void)queue\_detach(q, &t); /\*读取一个系数，放入t\*/

tmp = s + t; /\* 计算下一行系数，并进队列\*/

(void)queue\_enter(q, &tmp);

s = t; /\* 打印一个系数，第i+2个是0\*/

if(j != i+2) {

printf("%d ",s);

}

}

}

queue\_uninit(&q);

}

### 3.5 特殊矩阵的压缩存储

若值相同的元素或零元素在矩阵中的分布有一定的规律，则称此类矩阵为特殊矩阵。

矩阵中的元素aij(0≤i≤n-1, 0≤j≤n-1)在一维数组中的下标为loc(i, j)。

**1、对称矩阵**

下三角矩阵，行优先：loc(i, j)=1+2+...+i+j=(0≤j≤i≤n-1)

上三角矩阵,行优先：loc(i, j)=n+n-1+...+(n-i+1)+j-i

=(0≤i≤j≤n-1)

2、三对角矩阵

行有限：loc(i, j)=2i+j(0≤i≤n-1, i-1≤j≤i+1)

## 第4章 树

### 4.1 树的基本概念

叶子结点（终端结点），分支结点（非终端结点），双亲，孩子，兄弟，祖先，结点的度，树的度，结点的层次，树的深度，树的高度，有序树，无序树，森林

结点深度、高度和层次：通常认为根结点为第1层（也有人认为根节点在第0层），它的子女为第2层。子女结点的层次为双亲结点层次数加1。书的深度是从根节点开始自顶向下逐层累加的，而根节点的深度为1（也有教材定义根节点的深度为0）；数的高度是从叶节点开始自底向上逐层累加的。书中结点的最大层数称为数的深度（或高度）。

### 4.2 二叉树

#### 4.2.1 二叉树的定义

二叉树，完全二叉树，满二叉树

#### 4.2.2 二叉树的性质

二叉树的性质：设叶子结点数为n0，度为2的结点数为n2，则n=n+1。（因为n=n0+n1+n2，同时n=n1+2n2+1）

完全二叉树的性质：（结点编号从1开始，层次从1开始）

1、若i=1，则结点i为根结点，无双亲；若i>1，则其双亲为；

2、结点i的左孩子为2i，右孩子为2i+1；

3、结点i所在的层次为或。

#### 4.2.3 二叉树的存储结构

顺序存储，一般用于完全二叉树

链式存储：二叉链表，三叉链表

n个结点的二叉链表含有n+1个空指针。

#### 4.2.4 二叉树的遍历

/\*\* @file binary\_tree.h

\* @brief 二叉树

\* @author soulmachine

\* @date 2010-7-31

\* @version 0.1

\* @note 二叉树的头文件

\*/

#ifndef \_BINARY\_TREE\_H\_

#define \_BINARY\_TREE\_H\_

#ifdef \_\_cplusplus

extern "C"

{

#endif

#include "config.h"

/\*\* 二叉树结点，不透明的结构体. \*/

typedef struct binary\_tree\_node\_t binary\_tree\_node\_t;

/\*\*

\* @brief 先序遍历，递归.

\* @param[in] root 根结点

\* @param[in] visit 访问数据元素的函数指针

\* @return 无

\* @note 无

\* @remarks 无

\*/

extern void pre\_order\_r(binary\_tree\_node\_t \*root,

int (\*visit)(void\*));

/\*\*

\* @brief 中序遍历，递归.

\*/

extern void in\_order\_r(binary\_tree\_node\_t \*root,

int (\*visit)(void\*));

/\*\*

\* @brief 后序遍历，递归.

\*/

extern void post\_order\_r(binary\_tree\_node\_t \*root,

int (\*visit)(void\*));

/\*\*

\* @brief 先序遍历，非递归.

\*/

extern void pre\_order(binary\_tree\_node\_t \*root,

int (\*visit)(void\*));

/\*\*

\* @brief 中序遍历，非递归.

\*/

extern void in\_order(binary\_tree\_node\_t \*root,

int (\*visit)(void\*));

/\*\*

\* @brief 后序遍历，非递归.

\*/

extern void post\_order(binary\_tree\_node\_t \*root,

int (\*visit)(void\*));

/\*\*

\* @brief 层次遍历.

\*/

extern void level\_order(binary\_tree\_node\_t \*root,

int (\*visit)(void\*));

#ifdef \_\_cplusplus

}

#endif

#endif /\* end of \_BINARY\_TREE\_H\_ \*/

/\*\* @file binary\_tree.c

\* @brief 二叉树

\* @author soulmachine@gmail.com

\* @date 2010-7-31

\* @version 0.1

\* @note 二叉树的源代码文件

\*/

#include "binary\_tree.h"

#include "stack.h"

#include "queue.h"

/\*

\*@struct

\*@brief 二叉树结点

\*@note 无

\*/

struct binary\_tree\_node\_t {

binary\_tree\_node\_t \*lchild; /\* 左孩子\*/

binary\_tree\_node\_t \*rchild; /\* 右孩子\*/

void\* data; /\* 结点的数据\*/

};

void pre\_order\_r(binary\_tree\_node\_t \*root,

int (\*visit)(void\*))

{

if(root != NULL) {

(void)visit(root->data);

pre\_order\_r(root->lchild, visit);

pre\_order\_r(root->rchild, visit);

}

}

void in\_order\_r(binary\_tree\_node\_t \*root,

int (\*visit)(void\*))

{

if(root != NULL) {

pre\_order\_r(root->lchild, visit);

(void)visit(root->data);

pre\_order\_r(root->rchild, visit);

}

}

void post\_order\_r(binary\_tree\_node\_t \*root,

int (\*visit)(void\*))

{

if(root != NULL) {

pre\_order\_r(root->lchild, visit);

pre\_order\_r(root->rchild, visit);

(void)visit(root->data);

}

}

**void pre\_order(binary\_tree\_node\_t \*root,**

**int (\*visit)(void\*))**

**{**

**binary\_tree\_node\_t \*p;**

**stack\_t \*s;**

**p = root;**

**stack\_init(&s, sizeof(binary\_tree\_node\_t \*));**

**if(p != NULL) {**

**stack\_push(s, &p);**

**}**

**while(!stack\_is\_empty(s)) {**

**stack\_pop(s, &p);**

**visit(p->data);**

**if(p->rchild != NULL) {**

**stack\_push(s, &(p->rchild));**

**}**

**if(p->lchild != NULL) {**

**stack\_push(s, &(p->lchild));**

**}**

**}**

**stack\_uninit(&s);**

**}**

**void in\_order(binary\_tree\_node\_t \*root,**

**int (\*visit)(void\*))**

**{**

**binary\_tree\_node\_t \*p;**

**stack\_t \*s;**

**p = root;**

**stack\_init(&s, sizeof(binary\_tree\_node\_t \*));**

**while(!stack\_is\_empty(s) || p!=NULL) {**

**if(p != NULL) {**

**stack\_push(s, &p);**

**p = p->lchild;**

**} else {**

**stack\_pop(s, &p);**

**visit(p->data);**

**p = p->rchild;**

**}**

**}**

**stack\_uninit(&s);**

**}**

**void post\_order(binary\_tree\_node\_t \*root,**

**int (\*visit)(void\*))**

**{**

**/\* p，正在访问的结点，q，刚刚访问过的结点\*/**

**binary\_tree\_node\_t \*p, \*q;**

**stack\_t \*s;**

**p = root;**

**stack\_init(&s, sizeof(binary\_tree\_node\_t \*)) return;**

**do {**

**while(p != NULL) { /\* 往左下走\*/**

**stack\_push(s, &p);**

**p = p->lchild;**

**}**

**q = NULL;**

**while(!stack\_is\_empty(s)) {**

**stack\_pop(s, &p);**

**/\* 右孩子不存在或已被访问，访问之\*/**

**if(p->rchild == q) {**

**visit(p->data);**

**q = p; /\* 保存刚访问过的结点\*/**

**} else {**

**/\* 当前结点不能访问，需第二次进栈\*/**

**stack\_push(s, &p);**

**/\* 先处理右子树\*/**

**p = p->rchild;**

**break;**

**}**

**}**

**}while(!stack\_is\_empty(s));**

**stack\_uninit(&s);**

**}**

**/\* 跟先序遍历一模一样，唯一的不同是栈换成了队列\*/**

**void level\_order(binary\_tree\_node\_t \*root,**

**int (\*visit)(void\*))**

**{**

**binary\_tree\_node\_t \*p;**

**queue\_t \*q;**

**p = root;**

**queue\_init(&q, sizeof(binary\_tree\_node\_t\*));**

**if(p != NULL) {**

**queue\_enter(q, &p);**

**}**

**while(!queue\_is\_empty(q)) {**

**queue\_detach(q, &p);**

**visit(p->data);**

**if(p->lchild != NULL) { /\*先左后右或先右后左无所谓\*/**

**queue\_enter(q, &(p->lchild));**

**}**

**if(p->rchild != NULL) {**

**queue\_enter(q, &(p->rchild));**

**}**

**}**

**queue\_uninit(&q);**

**}**

#### 4.2.5 线索二叉树



图5.1 中序线索二叉树及其二叉链表示

/\*\* @file thread\_tree.c

\* @brief 线索二叉树.

\* @author soulmachine@gmail.com

\* @date 2010-7-31

\* @version 0.1

\* @note 线索二叉树的源代码文件

\*/

#include "config.h"

/\* 结点所存储的数据的类型，这里用int仅作演示\*/

typedef int thread\_tree\_node\_data\_t;

typedef struct thread\_tree\_node\_t thread\_tree\_node\_t;

/\*

\*@struct

\*@brief 线索二叉树结点.

\*@note 无

\*/

struct thread\_tree\_node\_t {

int ltag; /\* 1表示是线索，表示是孩子 \*/

int rtag; /\* 1表示是线索，表示是孩子\*/

thread\_tree\_node\_t \*lchild; /\* 左孩子\*/

thread\_tree\_node\_t \*rchild; /\* 右孩子\*/

thread\_tree\_node\_data\_t data; /\* 结点所存放的数据\*/

};

static void in\_thread(thread\_tree\_node\_t \*p,

thread\_tree\_node\_t \*\*pre);

/\*

\* @brief 通过中序遍历建立中序线索二叉树的主过程.

\* @param[in] root 树根

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void create\_in\_thread(thread\_tree\_node\_t \*root)

{

/\* 前驱结点指针\*/

thread\_tree\_node\_t \*pre=NULL;

if(root != NULL) { /\* 非空二叉树，线索化\*/

/\* 中序遍历线索化二叉树\*/

in\_thread(root, &pre);

/\* 处理中序最后一个结点\*/

pre->rchild = NULL;

pre->rtag = 1;

}

}

/\*

\* @brief 通过中序遍历，对二叉树进行线索化，递归形式.

\* @param[in] p 当前要处理的结点

\* @param[inout] pre 当前结点的前驱结点

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void in\_thread(thread\_tree\_node\_t \*p,

thread\_tree\_node\_t \*\*pre);

{

if(p != NULL) {

in\_thread(p->lchild, pre); /\* 递归，左子树线索化\*/

/\* 建立当前结点的前驱线索\*/

if(p->lchild == NULL) {

p->lchild = \*pre;

p->ltag = 1;

}

/\* 建立前驱结点的后继线索\*/

if((\*pre) != NULL &&

(\*pre)->rchild == NULL) {

(\*pre)->rchild = p;

(\*pre)->rtag = 1;

}

(\*pre) = p; /\* 前驱跟上\*/

in\_thread(p->rchild, pre); /\* 递归，右子树线索化\*/

}

}

/\*

\* @brief 寻找线索二叉树的中序下的第一个结点.

\* @param[in] p 线索二叉树中的任意一个结点

\* @return 此线索二叉树的第一个结点

\* @note 无

\* @remarks 无

\*/

static thread\_tree\_node\_t \*first(thread\_tree\_node\_t \*p)

{

if(p == NULL) return NULL;

while(p->ltag == 0) {

p = p->lchild; /\* 最左下结点，不一定是叶结点\*/

}

return p;

}

/\*

\* @brief 求中序线索二叉树中某结点的后继.

\* @param[in] p 某结点

\* @return p的后继

\* @note 无

\* @remarks 无

\*/

static thread\_tree\_node\_t \*next(const thread\_tree\_node\_t \*p)

{

if(p->rtag == 0) {

return first(p->rchild);

} else {

return p->rchild;

}

}

/\*

\* @brief 遍历中序线索二叉树.

\* @param[in] root 树根

\* @param[in] visit 访问结点的数据的函数

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void thread\_in\_order(thread\_tree\_node\_t \*root,

int(\*visit)(thread\_tree\_node\_data\_t\*))

{

thread\_tree\_node\_t \*p;

for(p = first(root); p != NULL; p = next(p)) {

(void)visit(&(p->data));

}

}

中序线索二叉树：结点的后继是右子树中最左下的结点，前驱是左子树中最右下的结点。

后序线索二叉树：

（1）若结点是根结点，则其后继为空；

（2）若结点是双亲的右子树，或是左子树但双亲无右子树，则其后继为双亲结点；

（3）若结点是双亲的左子树且双亲有右子树，则其后继为右子树按后序遍历的第一个结点。

### 4.3 树和森林

#### 4.3.1 树的存储结构

**1、双亲表示法**

用一个数组存储树中的结点，同时在每个结点中附设一个只是气质是该结点的双亲结点在数组中的位置，如图5.2所示。



图5.2 树的双亲表示法

**2、孩子表示法**

把每个结点的孩子结点排列起来，看成是一个线性表，且以单链表做存储结构，则n个结点有n个孩子链表（叶子的孩子链表为空）。而n个头指针又组成一个线性表，为了便于查找，可采用顺序存储结构。



图5.3 树的孩子表示法

**3、孩子兄弟表示法**

又称二叉树表示法，或二叉链表示法，即以二叉链表做存储结构。链表中结点的两个链域分别指向该结点的第一个孩子和下一个兄弟结点。



图5.4 树的孩子兄弟表示法

#### 4.3.2 森林与二叉树的转换

1、森林转换为二叉树

先将森林中的每一棵树转换为二叉树，再将第一棵树的根作为转换后的二叉树的根，第一棵树的左子树作为转换后二叉树根的左子树，第二颗树作为转换后二叉树的右子树，第三棵树作为转换后二叉树根的右子树的右子树，以此类推。

2、二叉树转换为森林

先将二叉树的根及其左子树作为第一棵树的二叉树形式，二叉树根的右子树又可以看作是一个由森林转换后的二叉树，如此递归，直到产生一颗没有右子树的二叉树为止，这样就得到了森林。

#### 4.3.3 树和森林的遍历

1、树的遍历

树的遍历有两类：深度优先遍历和宽度优先遍历。深度优先遍历又可分为两种：先根（次序）遍历和后根（次序）遍历。

树的先根遍历是：先访问树的根结点，然后依次先根遍历根的各棵子树。树的先跟遍历的结果与对应二叉树的先序遍历的结果相同。

树的后根遍历是：先依次后根遍历树根的各棵子树，然后访问根结点。树的后跟遍历的结果与对应二叉树的中序遍历的结果相同。

对于图5.5中的树，



图5.5

先根遍历的序列是ABCDE，后根遍历的序列是BDCEA。

2、森林的遍历

森林的遍历和树一样，有深度优先遍历和宽度优先遍历。深度优先遍历又分为先根（次序）遍历和中根（次序）遍历。

（1）先根遍历森林

①访问森林中第一棵树的根结点；

②先根遍历第一棵树的子树森林；

③先根遍历除去第一棵树后剩余的树所构成的森林。

（2）中根遍历森林

①中根遍历森林中第一棵树的子树森林；

②访问第一棵树的根结点；

③中根遍历除去第一棵树后剩余的树所构成的森林。

对于图5.6中的森林，



图5.6

先根遍历的序列是ABCDEFGHIJ，中根遍历的序列是BCDAFEHJIG。

森林的先根遍历和中根遍历与对应二叉树的先序遍历和中序遍历相同。

### 4.4 树的应用

#### 4.4.1 并查集(Union-Find-Set)

通常用树双亲表示作为并查集的存储结构。每个集合以一棵树表示，数组元素的下标代表元素名，根结点的双亲指针为一个负数，表示集合的元素的个数。



(a)初始化时每个元素自成为一个单元素子集合



(b)初始化时形成的森林的双亲表示

图5.7 并查集的初始化



(a)集合的树的表示



(b)集合S1、S2和S3的双亲表示

图5.8 用树表示并查集



图5.9 两个集合的并

下面是并查集的主要运算的实现。

/\*\* @file union\_find\_set.h

\* @brief 并查集.

\* @author soulmachine@gmail.com

\* @date 2010-8-1

\* @version 0.1

\* @note 并查集的头文件

\*/

#ifndef \_UNION\_FIND\_SET\_H\_

#define \_UNION\_FIND\_SET\_H\_

#ifdef \_\_cplusplus

extern "C"

{

#endif

/\*\*

\* @brief 初始化并查集.

\* @param[in] s 双亲表示法的数组

\* @param[in] n 数组s的元素个数

\* @return 无

\* @note 无

\* @remarks 无

\*/

void ufs\_init(int s[], int n);

/\*\*

\* @brief Find操作.

\* @param[in] s 双亲表示法的数组

\* @param[in] x 要查找的元素

\* @return 包含元素x的树的根

\* @note 查找包含元素x的树的根

\* @remarks 无

\*/

extern int ufs\_find(const int s[], int x);

/\*

\* @brief Union操作.

\* @param[in] s 双亲表示法的数组

\* @param[in] root1 一棵树的根

\* @param[in] root2 另一棵树的根

\* @return 无

\* @note 求两个不相交集合的并集

\* @remarks 无

\*/

extern void ufs\_union(int s[], int root1, int root2);

#ifdef \_\_cplusplus

}

#endif

#endif /\* end of \_UNION\_FIND\_SET\_H\_ \*/

/\*\* @file union\_find\_set.c

\* @brief 并查集.

\* @author soulmachine@gmail.com

\* @date 2010-8-1

\* @version 0.1

\* @note 并查集的实现代码

\*/

#include "union\_find\_set.h"

void ufs\_init(int s[], int n)

{

int i;

for(i = 0; i < n; i++) s[i] = -1;

}

int ufs\_find(const int s[], int x)

{

while(s[x] >= 0) {

x = s[x];

}

return x;

}

void ufs\_union(int s[], int root1, int root2)

{

s[root1] += s[root2];

s[root2] = root1;

}

#### 4.4.2 哈夫曼树(Huffman Tree)和哈夫曼编码

**1. 哈夫曼树**

树中所有结点的带权路径长度之和称为该树的带权路径长度(Weighted Path Length)，记为



其中，wi是第i个叶结点所带的权值，li是该叶结点到根节点的路径长度。

称WPL最小的二叉树为最优二叉树或**哈夫曼树**。

构造哈夫曼树的基本思路（即**Huffman算法**）如下：

（1）根据给定的n个权值{w1,w2,...,wn}，构造具有n棵二叉树的森林F={T1,T2,...,Tn}，其中每棵二叉树Ti只有一个带权值wi的根结点，其左、右子树均为空。

（2）重复一下步骤，直到F中仅剩下一棵树为止：①在F中选取两棵根结点的权值最小的二叉树，作为左、右子树构造一棵新的二叉树。置新的二叉树的跟结点的权值为其左、右子树上根结点的权值之和；②在F中删除这两棵树；③把新的二叉树加入F。

**2. 哈夫曼编码**

对频率高的字符赋以短编码，对频率低的字符赋以长编码，可以使字符串平均编码长度减短，起到压缩数据的效果。

用平均编码长度（实质上就是WPL，只不过权值换成了频率）



来衡量各种编码方案的效率。

如果不存在一个编码是另一个编码的前缀，称这样的编码为前缀编码。用二叉树表示的编码肯定是前缀编码。

用哈夫曼树得到哈夫曼编码是很自然的过程。

究竟是0或1表示左子树还是右子树没有明确规定，并且哈夫曼树不是有序树，因此左右结点的顺序是任意的，所以构造出的哈夫曼树并不唯一，但是各哈夫曼树的WPL相同且为最优。

#### 4.4.3 二叉查找树（二叉排序树）

定义：二叉树或者是一棵空树，或者是一棵具有下列性质的非空二叉树：

(1) 左子树所有结点的值均不大于根结点的值；

(2) 右子树所有结点的值均不小于根结点的值；

(3) 左、右子树分别是又一棵二叉查找树。

如果在二叉查找树上做中序遍历，所得到的序列是一个非递减的有序序列。

下面是二叉查找树的查找、插入和删除操作的实现。

/\*\* @file binary\_search\_tree.c

\* @brief 二叉查找树.

\* @author soulmachine@gmail.com

\* @date 2010-8-1

\* @version 0.1

\* @note 二叉查找树的源代码文件.

\*/

#include "config.h"

#include <malloc.h>

/\* 结点所带数据类型，这里用int仅作演示\*/

typedef int bst\_node\_data\_t;

typedef struct bst\_node\_t bst\_node\_t;

/\*

\*@struct

\*@brief 二叉查找树树结点

\*@note 无

\*/

struct bst\_node\_t {

bst\_node\_t \*lchild; /\* 左孩子\*/

bst\_node\_t \*rchild; /\* 右孩子\*/

bst\_node\_data\_t elem; /\* 结点的数据元素\*/

};

/\*

\* @brief 查找，递归形式.

\* @param[in] root 二叉树的根

\* @param[in] x 要查找的元素

\* @return 元素所在的结点，失败则返回NULL

\* @note 无

\* @remarks 无

\*/

static bst\_node\_t\* bst\_search\_r(bst\_node\_t \*root,

const bst\_node\_data\_t x)

{

bst\_node\_t\* const p = root;

if(p == NULL) {

return NULL;

} else if(x < p->elem) {

return bst\_search\_r(p->lchild, x);

} else if(x > p->elem) {

return bst\_search\_r(p->rchild, x);

} else {

return p;

}

}

/\*

\* @brief 查找，非递归形式.

\* @param[in] root 二叉树的根

\* @param[in] x 要查找的元素

\* @param[out] father x的双亲结点

\* @return 元素所在的结点，失败则返回NULL

\* @note 查找成功，返回x所在的结点的地址，father返回该节

\* 点的双亲结点的地址；查找失败，返回NULL，father返回新

\* 结点应插入的结点的地址，此时新结点应作为叶子结点插入

\* 到father之下。

\* @remarks 无

\*/

static bst\_node\_t\* bst\_search(bst\_node\_t \*root,

const bst\_node\_data\_t x,

bst\_node\_t \*\*father)

{

bst\_node\_t \*p = root;

(\*father) = NULL;

while(p != NULL && p->elem != x) {

(\*father) = p;

if(x < p->elem) {

p = p->lchild;

} else {

p = p->rchild;

}

}

return p;

}

/\*

\* @brief 插入

\* @param[inout] root 二叉树的根

\* @param[in] x 要插入的元素

\* @return 成功返回TRUE，失败返回FALSE

\* @note 无

\* @remarks 无

\*/

static bool bst\_insert(bst\_node\_t \*\*root,

const bst\_node\_data\_t x)

{

const bst\_node\_t \*p;

bst\_node\_t \*f, \*new\_node;

p = bst\_search((\*root), x, &f);

if(p != NULL) { /\* 查找成功，不插入\*/

return true;

}

new\_node = (bst\_node\_t \*)malloc(sizeof(bst\_node\_t));

if(new\_node == NULL) return false;

new\_node->elem = x;

new\_node->lchild = NULL;

new\_node->rchild = NULL;

if(f == NULL) { /\* 空树\*/

(\*root) = new\_node;

} else if(x < f->elem) {

f->lchild = new\_node;

} else {

f->rchild = new\_node;

}

return true;

}

/\*

\* @brief 删除.

\* @param[in] root 二叉树的根

\* @param[in] x 要查找的元素

\* @return 删除成功返回TRUE，失败返

\* 回FALSE

\* @note 无

\* @remarks 无

\*/

static bool bst\_remove(bst\_node\_t \*root,

const bst\_node\_data\_t x)

{

bst\_node\_t \*p, \*f, \*s;

p = bst\_search(root, x, &f); /\*寻找删除结点\*/

if(p == NULL) { /\* 查找失败，不作删除\*/

return false;

}

if(p->lchild != NULL &&

p->rchild != NULL) {

/\* 寻找p的中序前驱s \*/

s = p->rchild;

while(s->lchild != NULL) {

f = s;

s = s->lchild;

}

/\*

// 也可以使用这段代码寻找中序前驱

s = p->lchild;

while(s->rchild != NULL) {

f = s;

s = s->rchild;

} \*/

p->elem = s->elem; /\* 填补到被删除结点\*/

p = s;

}

/\*记录非空子女结点，p 肯定只有一个孩子\*/

if(p->lchild != NULL) {

s = p->lchild;

} else {

s = p->rchild;

}

if(p == root) { /\* 被删除结点为根结点\*/

root = s;

} else if(s->elem < f->elem) {

f->lchild = s;

} else {

f->rchild = s;

}

free(p);

return true;

}

下面分情况讨论删除算法。①如果被删除结点是叶结点，只需将其双亲结点指向它的指针清零，再释放它即可。②如果被删除结点的右子树为空，可以拿它的左子女结点顶替它的位置，再释放它。如果被删结点的左子树为空，可以拿它的右子女结点顶替它的位置，再释放它。这些都比较简单。③如果被删除结点左、右子树都不为空，可以拿它的中序直接后继或直接前驱顶替它的位置，再来处理这个结点的删除问题，这是一个递归处理。参看图5.10。







图5.10 二叉查找树的删除

#### 4.4.4 平衡二叉树（AVL树）

1. 平衡二叉树的定义

平衡二叉树(Balanced Binary Tree或Height-Balanced Tree)又称AVL树。它或者是一棵空树，或者是一棵具有下列性质的二叉查找树：它的左子树和右子树都是平衡二叉树，且左右子树的高度之差的绝对值不超过1。

若将二叉树上结点的平衡因子(Balanced Factor, BF)定义为该结点左子树的高度减去右子树的高度，则平衡二叉树上所有结点的平衡因子只可能是-1、0、1。

设高为h平衡二叉树的最少结点数为Nh，则有

N0=0, N1=1, Nh=Nh-1+ Nh-1+1

设平衡二叉树的深度最小的叶子结点所在层次为Lh，则有

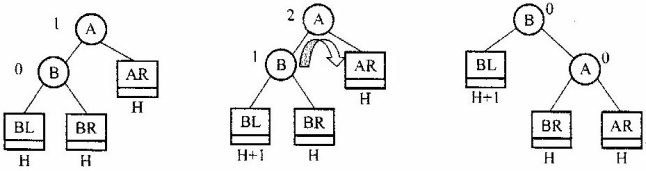
L1=1, L2=2, Lh=min{Lh-1, Lh-2}+1= Lh-2+1

n个结点的平衡二叉树高度不超过

2. 平衡二叉树的插入

在平衡二叉树上的查找与插入过程与二叉查找树，但在新结点作为叶结点插入后，必须沿着从插入位置到根节点的路径回溯，检查此路径上的某个结点是否变得不平衡。一般可将失去平衡后进行调整的情况归纳为下列4种。

（1）LL型，右旋

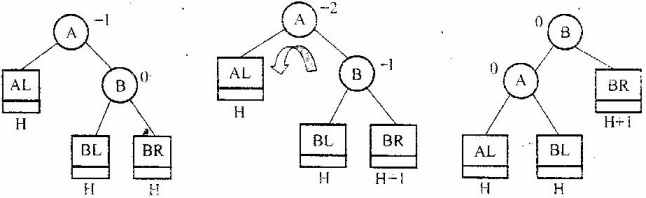


(a) (b) (c)

图4.14 LL平衡旋转

(a) 插入结点前 (b) 插入结点导致不平衡 (c) LL旋转

（2）RR型，左旋



(a) (b) (c)

图4.15 RR平衡旋转

(a) 插入结点前 (b) 插入结点导致不平衡 (c) RR旋转

（3）LR型，先左旋后右旋

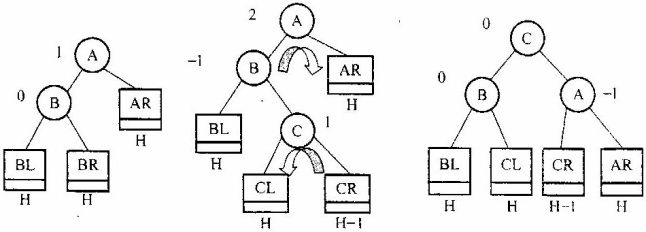


图4.16 LR平衡旋转

(a) 插入结点前 (b) 插入结点导致不平衡 (c) LR旋转

（4）RL型，先右旋后左旋

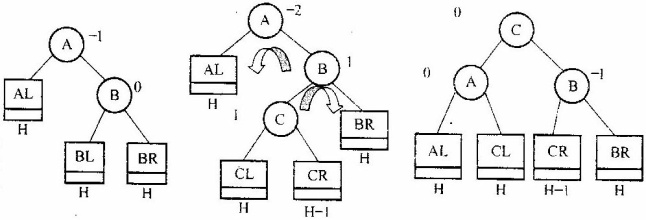


图4.17 RL平衡旋转

(a) 插入结点前 (b) 插入结点导致不平衡 (c) RL旋转

#### 4.4.5 堆(Heap)

设完全二叉树中各结点的值均不小于（不大于）其双亲结点的值，且存储结构为顺序存储，则称此二叉树为小根堆（大根堆）。

堆的性质（结点编号从0开始，层次从0开始）：

（1）若i=0，结点i是根结点，无双亲；否则结

点i的双亲结点为；

（2）结点i的左孩子为2i+1，右孩子为2i+2；若2i+1>n-1，则无左孩子，若2i+2>n-1，则无右孩子；

（3）结点i所在的层次为或。

下面是小根堆的主要操作的实现。

/\*\* @file heap.h

\* @brief 小根堆.

\* @author soulmachine@gmail.com

\* @date 2010-8-1

\* @version 0.1

\* @note 小根堆的头文件

\*/

#ifndef \_HEAP\_H\_

#define \_HEAP\_H\_

#ifdef \_\_cplusplus

extern "C"

{

#endif

#include "config.h"

/\*\* 堆结构体，不透明. \*/

typedef struct heap\_t heap\_t;

/\*\*

\* @brief 堆的初始化.

\* @param[out] h 堆结构体的二级指针

\* @param[in] elem\_size 单个元素的大小

\* @param[in] cmp cmp 比较函数，小于返回-1，等于，

\* 大于，反过来则是大根堆

\* @return 成功返回0，失败返回错误码

\* @note 本函数会分配内存，创建一个

\* min\_heap\_t对象，将其地址存放在

\* h所指向的指针中

\* @remarks 无

\*/

extern int heap\_init(heap\_t \*\*h, int elem\_size,

int (\*cmp)(const void\*, const void\*));

/\*\*

\* @brief 释放堆所占用的内存空间.

\* @param[inout] h 堆对象的二级指针

\* @return 成功返回0，失败返回错误码

\* @note 释放栈对象本身所占用的内存

\* @remarks 无

\*/

extern int heap\_uninit(heap\_t \*\*s);

/\*\*

\* @brief 堆的建立.

\* @param[out] h 堆对象的二级指针

\* @param[in] arr 存放初始数据的数组

\* @param[in] n 数组的大小

\* @return 成功返回0，失败返回错误码

\* @note 无

\* @remarks 无

\*/

extern int heap\_create(heap\_t \*h, const void \*arr, int n);

/\*\*

\* @brief 清空数据，但不释放内存.

\* @param[in] h 指向堆对象的指针

\* @return 无

\* @note 无

\* @remarks 无

\*/

extern void heap\_clear(heap\_t \*h);

/\*\*

\* @brief 判断堆是否为空.

\* @param[in] h 指向堆对象的指针

\* @return 是空，返回true，否则返回false

\* @note 无

\* @remarks 无

\*/

extern bool heap\_is\_empty(const heap\_t \*h);

/\*\*

\* @brief 判断堆是否已满.

\* @param[in] h 指向堆对象的指针

\* @return 满，返回true，否则返回false

\* @note 无

\* @remarks 无

\*/

extern bool heap\_is\_full(const heap\_t \*h);

/\*\*

\* @brief 在尾部插入一个元素.

\* @param[in] h 堆对象的指针

\* @param[in] x 要插入的元素

\* @return 成功返回0，失败返回错误码

\* @note 无

\* @remarks 无

\*/

extern int heap\_insert(heap\_t \*h, const void \*x);

/\*\*

\* @brief 删除堆顶.

\* @param[in] h 堆对象的指针

\* @param[in] x 保存删除掉的元素

\* @return 成功返回0，失败返回错误码

\* @note 无

\* @remarks 无

\*/

extern int heap\_remove(heap\_t \*h, void \*x);

#ifdef \_\_cplusplus

}

#endif

#endif /\* end of \_HEAP\_H\_ \*/

/\*\* @file heap.c

\* @brief 小根堆.

\* @author soulmachine@gmail.com

\* @date 2010-8-1

\* @version 0.1

\* @note 小根堆的源代码文件

\*/

#include "heap.h"

#include <stdlib.h> /\* for malloc() \*/

#include <string.h> /\* for memcpy() \*/

/\*

\*@struct

\*@brief 堆的结点.

\*@note 无

\*/

struct heap\_t {

int size; /\* 实际元素个数\*/

int capacity; /\* 容量大小，以元素为单位\*/

int elem\_size; /\* 单个元素的大小\*/

int (\*cmp)(const void\*, const void\*); /\*元素的比较函数\*/

void \*data; /\* 堆的数组\*/

};

int heap\_init(heap\_t \*\*h, int elem\_size,

int (\*cmp)(const void\*, const void\*))

{

if(h == NULL) return ERR\_NULL\_POINTER;

\*h = (heap\_t\*)malloc(sizeof(heap\_t));

if(\*h == NULL) return ERR\_MALLOC\_FAILED;

(\*h)->size = 0;

(\*h)->capacity = 0;

(\*h)->elem\_size = elem\_size;

(\*h)->cmp = cmp;

(\*h)->data = NULL;

return SUCCESS;

}

int heap\_uninit(heap\_t \*\*s)

{

if((s !=NULL) && (\*s != NULL)) {

free((\*s)->data);

free((\*s));

(\*s) = NULL;

}

return 0;

}

/\*

\* @brief 小根堆的自上向下筛选算法.

\* @param[in] h 堆对象的指针

\* @param[in] start 开始结点

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void heap\_sift\_down(const heap\_t \*h,int start)

{

int i = start;

int j;

/\* data\_t tmp = h->data[start]; \*/

void\* const tmp = malloc(h->elem\_size);

if(tmp == NULL) return /\* ERR\_MALLOC\_FAILED \*/;

memcpy(tmp, (char\*)h->data + start \* h->elem\_size,

h->elem\_size);

for(j = 2 \* i + 1; j < h->size; j = 2 \* j + 1) {

if(j < (h->size - 1) &&

/\* h->data[j] > h->data[j + 1] \*/

h->cmp((char\*)h->data + j \* h->elem\_size,

(char\*)h->data + (j + 1) \* h->elem\_size) > 0) {

j++; /\* j 指向两子女中小者\*/

}

/\* tmp <= h->data[j] \*/

if(h->cmp(tmp, (char\*)h->data + j \* h->elem\_size) <= 0) {

break;

} else {

/\* h->data[i] = h->data[j]; \*/

memcpy((char\*)h->data + i \* h->elem\_size,

(char\*)h->data + j \* h->elem\_size,

h->elem\_size);

i = j;

}

}

/\* h->data[i] = tmp; \*/

memcpy((char\*)h->data + i \* h->elem\_size, tmp, h->elem\_size);

free(tmp);

}

/\*

\* @brief 小根堆的自下向上筛选算法.

\* @param[in] h 堆对象的指针

\* @param[in] start 开始结点

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void heap\_sift\_up(const heap\_t \*h, int start)

{

int j = start;

int i= (j - 1) / 2;

/\* data\_t tmp = h->data[start]; \*/

void\* const tmp = malloc(h->elem\_size);

/\* \*tmp = h-data[start]; \*/

if(tmp == NULL) return /\* ERR\_MALLOC\_FAILED \*/;

memcpy(tmp, (char\*)h->data + start \* h->elem\_size, h->elem\_size);

while(j > 0) {

/\* h->data[i] <= tmp \*/

if(h->cmp((char\*)h->data + i \* h->elem\_size, tmp) <= 0) {

break;

} else {

/\* h->data[j] = h->data[i]; \*/

memcpy((char\*)h->data + j \* h->elem\_size,

(char\*)h->data + i \* h->elem\_size,

h->elem\_size);

j = i;

i = (i - 1) / 2;

}

}

/\* h->data[j] = tmp; \*/

memcpy((char\*)h->data + j \* h->elem\_size, tmp, h->elem\_size);

free(tmp);

}

int heap\_create(heap\_t \*h, const void \*arr, int n)

{

int i;

if(h->capacity < n) {/\*空间不够，重新分配\*/

free(h->data);

h->data = malloc(n \* h->elem\_size);

if(h->data == NULL) return ERR\_MALLOC\_FAILED;

h->capacity = n;

}

/\* 复制数组到堆中\*/

h->size = n;

memcpy(h->data, arr, n \* h->elem\_size);

i = (h->size - 2)/2; /\* 找最初调整位置：最后分支结点\*/

while(i >= 0) { /\* 自底向上逐步扩大形成堆\*/

heap\_sift\_down(h, i);

i--;

}

return SUCCESS;

}

void heap\_clear(heap\_t \*h)

{

if(h != NULL) h->size = 0;

}

bool heap\_is\_empty(const heap\_t \*h)

{

if(h != NULL) {

return h->size == 0;

} else {

return false;

}

}

bool heap\_is\_full(const heap\_t \*h)

{

if(h != NULL) {

return h->size == h->capacity;

} else {

return false;

}

}

int heap\_insert(heap\_t \*h, const void\* x)

{

if(heap\_is\_full(h)) { /\*已经满，再分配内存\*/

void\* const tmp =

realloc(h->data, h->capacity \* h->elem\_size \* 2);

if(tmp == NULL) return ERR\_MALLOC\_FAILED;

h->data = tmp;

h->capacity \*= 2;

}

/\* h->data[h->size] = x; \*/

memcpy((char\*)h->data + h->size \* h->elem\_size,

x, h->elem\_size);

heap\_sift\_up(h, h->size);

h->size++;

return SUCCESS;

}

int heap\_remove(heap\_t \*h, void\* x)

{

if(heap\_is\_empty(h)) return ERR\_OTHER;

/\* \*x = h->data[0]; \*/

memcpy(x, h->data, h->elem\_size);

/\* h->data[0] = h->data[h->size - 1]; \*/

memcpy(h->data, (char\*)h->data + (h->size - 1) \* h->elem\_size,

h->elem\_size);

h->size --;

heap\_sift\_down(h, 0);

return SUCCESS;

}

建堆的过程如下图所示。









图5.11 自下而上逐步调整为小根堆

## 第5章 图

### 5.1 图的基本概念

顶点(vertex)，弧(arc)，弧头(head)，弧尾(tail)，有向图(digraph)，无向图(undigraph)，稀疏图(sparse graph)，稠密图(dense graph)，完全图(completed graph)，有向完全图(completed digraph)，有向无环图(directed acycline graph)

邻接点(adjacent)，度(degree)，路径(path)，简单路径，环(cycle)，连通图(connected graph)，连通分量(connected component )，强连通图，生成树，生成森林

**强连通图**：在有向图G中，如果对于每一对Vi，Vj∈V，≠，从Vi到Vj和从Vj到Vi都存在路径，则称G是强连通图。

有向图的弧一般用尖括号表示，如<x, y>表示顶点x指向顶点y的有向弧。无向图一般用圆括号表示，如(x, y)。

几个注意的地方：

1. 线性表和树可以为空，图不能为空。
2. 连通性一般指的是无向图，在有向图中一般考虑的是强连通性。

### 5.2 图的存储结构及基本操作

图的信息包括两部分，顶点的信息和边的信息，因此无论采用什么存储结构，都要完整、准确地反映这两部分的信息。

#### 5.2.1 邻接矩阵

图的邻接矩阵是一个n阶方阵，它的定义为



图6.1所示的有向图和无向图的邻接矩阵分别为A和B。





图6.1 有向图和无向图

网络（带权图）的邻接矩阵可定义为：



图6.2所示是网络及其邻接矩阵C的示例。



图6.2 一个网络及其邻接矩阵表示

#### 5.2.2 邻接表

邻接表(Adjacency List)是邻接矩阵的改进。它把邻接矩阵的n行改为n个单链表，把同一个顶点出发的边链接在一个单链表中（称为出边表），单链表的每一个结点代表一条边，叫做边结点，结点中保存有与改变相关联的另一顶点（对有向图来说就是弧头）。

图6.3所示是一个无向图及其邻接表。



图6.3 图的邻接表表示

有向图还有逆邻接表，即边结点存放的是弧尾，此时的单链表叫做入边表。

下面是图的存储结构及基本操作的C语言实现。

/\*\* @file graph.h

\* @brief 图

\* @author soulmachine@gmail.com

\* @date 2010-8-2

\* @version 0.1

\* @note 包含邻接矩阵，邻接表及其主要函数

\*/

#ifndef \_GRAPH\_H\_

#define \_GRAPH\_H\_

#ifdef \_\_cplusplus

extern "C"

{

#endif

#include "config.h"

#include <limits.h> /\* for INT\_MAX \*/

/\*\* 无穷大. \*/

#define GRAPH\_INFINITY INT\_MAX

/\*\* 顶点数的最大值\*/

#define MAX\_VERTICES\_NUM 100

/\*\* 边的权值，对无权图，用或表示是否相邻；对有权图，则为权值. \*/

typedef int graph\_weigh\_t;

/\*\* 顶点信息. \*/

typedef char graph\_vertex\_data\_t;

/\*\*

\*@struct

\*@brief 邻接矩阵.

\*/

typedef struct mgraph\_t {

int vertices\_num; /\* 顶点数\*/

int edges\_num; /\* 边数\*/

/\* 顶点表，存放顶点的信息，如名字等\*/

graph\_vertex\_data\_t vertices[MAX\_VERTICES\_NUM];

/\* 邻接矩阵，存放边的信息，如权重等\*/

graph\_weigh\_t edges[MAX\_VERTICES\_NUM][MAX\_VERTICES\_NUM] ;

int kind; /\* 图(网)的种类，表示有向，表示无向\*/

}mgraph\_t;

/\*

\*@struct

\*@brief 边..

\*/

typedef struct enode\_t{

int dest; /\* 弧尾的下标\*/

graph\_weigh\_t weight; /\* 本边的权值\*/

struct enode\_t \*next; /\* 指向下一条边\*/

}enode\_t;

/\*

\*@struct

\*@brief 顶点.

\*/

typedef struct vnode\_t{

graph\_vertex\_data\_t data; /\* 顶点信息，如名字等\*/

enode\_t \*first\_edge; /\* 指向依附于该点的第一条边\*/

}vnode\_t;

/\*\*

\*@struct

\*@brief 邻接表.

\*/

struct algraph\_t {

int vertices\_num; /\* 顶点数\*/

int edges\_num; /\* 边数\*/

/\* 顶点表（各出边表的头结点）\*/

vnode\_t vertices[MAX\_VERTICES\_NUM];

int kind; /\* 图(网)的种类，表示有向，表示无向\*/

};

/\*\*

\* @brief 取顶点v的第一个邻接点.

\* @param[in] g 邻接矩阵结构体的指针

\* @param[in] v 顶点位置

\* @return 顶点v的第一个邻接点，找不到则返回-1

\* @note 无

\* @remarks 无

\*/

extern int mgraph\_get\_first\_neighbor(const mgraph\_t \*g, int v);

/\*\*

\* @brief 取顶点v在邻接点w的下一个邻接点.

\* @param[in] g 邻接矩阵结构体的指针

\* @param[in] v 顶点位置

\* @param[in] w 顶点v的某个邻接点

\* @return 顶点v在邻接点w的下一个邻接点，找不到则返回-1

\* @note 无

\* @remarks 无

\*/

extern int mgraph\_get\_next\_neighbor(const mgraph\_t \*g,

int v, int w);

/\*\*

\* @brief 获取图的顶点数.

\* @author soulmachine@gmail.com

\* @param[in] g 图对象的指针

\* @return 该图的顶点数

\* @note 无

\* @remarks 无

\*/

extern int mgraph\_get\_vertices\_number(const mgraph\_t \*g);

/\*\*

\* @brief 获取图的边数.

\* @param[in] g 图对象的指针

\* @return 该图的边数

\* @note 无

\* @remarks 无

\*/

extern int mgraph\_get\_edges\_number(const mgraph\_t \*g);

/\*\*

\* @brief 获取边的权值.

\* @param[in] g 图对象的指针

\* @param[in] v 顶点v

\* @param[in] w 顶点w

\* @return 边的权值，或无穷大表示u,v 之间没有边

\* @note 无

\* @remarks 无

\*/

extern graph\_weigh\_t mgraph\_get\_weight(const mgraph\_t \*g,

int v, int w);

/\*\*

\* @brief 获取顶点信息.

\* @param[in] g 图对象的指针

\* @param[in] v 顶点v

\* @return 顶点信息,如名字

\* @note 无

\* @remarks 无

\*/

extern graph\_vertex\_data\_t\* mgraph\_get\_vertex\_data(

const mgraph\_t \*g,

int v);

/\* 这里使用邻接矩阵，也可以切换成邻接表\*/

typedef mgraph\_t graph\_t;

#define graph\_get\_first\_neighbor mgraph\_get\_first\_neighbor

#define graph\_get\_next\_neighbor mgraph\_get\_next\_neighbor

#define graph\_get\_vertices\_number mgraph\_get\_vertices\_number

#define graph\_get\_edges\_number mgraph\_get\_edges\_number

#define graph\_get\_weight mgraph\_get\_weight

#define graph\_get\_vertex\_data mgraph\_get\_vertex\_data

#ifdef \_\_cplusplus

}

#endif /\* end of \_cplusplus \*/

#endif /\* end of \_GRAPH\_H\_ \*/

### 5.3 图的遍历

#### 5.3.1 深度优先搜索(Depth First Search)

/\*\* @file graph\_problems.c

\* @brief 图的各种经典问题.

\* @author soulmachine@gmail.com

\* @date 2010-8-2

\* @version 0.1

\* @note 无

\*/

#include "graph.h"

#include "stack.h"

#include "queue.h"

#include "heap.h"

#include "union\_find\_set.h"

#include <stdlib.h> /\* for malloc() \*/

#include <stdio.h> /\* for printf() \*/

/\*

\* @brief 深度优先搜索，递归形式

\* @param[in] g 图对象的指针

\* @param[in] v 出发顶点

\* @param[in] visited 记录顶点是否被访问过的数组，初始

\* 全为FALSE

\* @param[in] visit 访问函数

\* @return 无

\* @note 从顶点位置v出发，以深度优先的次序访问所有尚未访

\* 问过的邻接点。算法中用到一个辅助数组visited，对已访问

\* 过的顶点做标记。

\* @remarks 无

\*/

static void graph\_dfs\_r(graph\_t \*g, int v, bool visited[],

int (\*visit)(graph\_vertex\_data\_t \*))

{

int w;

(void)visit(graph\_get\_vertex\_data(g, v));

visited[v]= true;

for(w = graph\_get\_first\_neighbor(g, v); w != -1;

w = graph\_get\_next\_neighbor(g, v, w)) {

if(!visited[w]) {

graph\_dfs\_r(g, w, visited, visit);

}

}

}

/\*

\* @brief 深度优先搜索，非递归

\* @param[in] g 图对象的指针

\* @param[in] v 出发顶点

\* @param[in] visit 访问函数

\* @return 无

\* @note 非递归算法中使用了一个栈s，记忆下一步可能访问的

\* 所有邻接顶点，同时使用了一个辅助数组bool in\_stack[]，

\* in\_stack[i]记忆顶点i是否在栈内或曾经在栈内。若是，以

\* 后它不能进栈。

\* @remarks 无

\*/

static void graph\_dfs(graph\_t \*g, int v,

int (\*visit)(graph\_vertex\_data\_t \*))

{

int i, w, n;

stack\_t \*s;

bool \*in\_stack;

n = graph\_get\_vertices\_number(g);

in\_stack = (bool \*)malloc(n \* sizeof(bool));

if(in\_stack == NULL) return;

for(i = 0; i < n; i++) in\_stack[i] = false;

if(!stack\_init(&s,sizeof(int))) {

free(in\_stack);

in\_stack = NULL;

return;

}

(void)stack\_push(s, &v);

in\_stack[v] = true;

while(!stack\_is\_empty(s)) {

(void)stack\_pop(s, &w);

(void)visit(graph\_get\_vertex\_data(g, w));

for(i = graph\_get\_first\_neighbor(g, w); i != -1;

i = graph\_get\_next\_neighbor(g, w, i)) {

if(!in\_stack[i]) {

(void)stack\_push(s, &i);

in\_stack[i] = true;

}

}

}

free(in\_stack);

stack\_uninit(&s);

}

算法分析：该算法需要把每个顶点都访问一次，每访问一个顶点需要查找其邻接点。当采用邻接矩阵时，查找邻接点所需时间为O(n)，总的时间复杂度为O(n2)；当采用邻接表时，查找邻接点所需时间为O(e)，总的时间复杂度为O(n+e)。因为是递归算法，需要一个递归栈，它的空间复杂度为O(n)。

查找邻接点的代码为：

for(i = graph\_get\_first\_neighbor(g, w); i != -1;

i = graph\_get\_next\_neighbor(g, w, i)) {

/\* 代码段 \*/

}

对于邻接矩阵，等价于下面的代码：

for(i = 0; i < g->vertices\_num; i++) {

if(g->edges[w][i] == -1) {

/\* 代码段 \*/

}

}

对于邻接表，等价于下面的代码：

for(i = g->vertices[w].first\_edge;i != NULL;i= i->next) {

/\* 代码段 \*/

}

#### 5.3.2 广度优先搜索(Breadth First Search)

/\*

\* @brief 广度优先搜索

\* @param[in] g 图对象的指针

\* @param[in] v 出发顶点

\* @param[in] visit 访问函数

\* @return 无

\* @note 广度优先搜索是一种分层的查找过程，每向前走一步

\* 可能访问一批顶点，不像深度优先搜索那样有往回退的情

\* 况。因此广度优先搜索不是一个递归的过程。为了实现逐层

\* 访问，代码中使用了一个队列，用于记忆正在访问的这一层

\* 和上一层的顶点，以便于向下一层访问。

\* @remarks 无

\*/

static void graph\_bfs(graph\_t \*g, int v,

int (\*visit)(graph\_vertex\_data\_t \*))

{

int i, w, n;

queue\_t \*q;

bool \*visited;

n = graph\_get\_vertices\_number(g);

visited = (bool \*)malloc(n \* sizeof(bool));

if(visited == NULL) return;

for(i = 0; i < n; i++) visited[i] = false;

if(!queue\_init(&q, sizeof(int))){

free(visited);

return;

}

(void)visit(graph\_get\_vertex\_data(g, v));

visited[v] = true;

(void)queue\_enter(q, &v);

while(!queue\_is\_empty(q)) {

(void)queue\_detach(q, &w);

for(i = graph\_get\_first\_neighbor(g, w); i != -1;

i = graph\_get\_next\_neighbor(g, w, i)) {

if(!visited[i]) {

(void)visit(graph\_get\_vertex\_data(g, i));

visited[i] = true;

(void)queue\_enter(q, &i);

}

}

}

free(visited);

queue\_uninit(&q);

}

算法分析：遍历图的过程实质上是对每个顶点查找其邻接点的过程。因此广度优先搜索的时间复杂度和深度优先搜索相同，两者的不同之处仅仅在于访问顶点的顺序不同。

### 5.4 图的基本应用

#### 5.4.1 最小生成树

“最小”指的是边的权值之和最小。

构造最小生成树可以有多种算法。其中多数算法利用了最小生成树的下列一种简称为MST的性质：假设N=(V, E)是一个连通网，U是顶点集V的一个非空子集。若(u, v)是一条具有最小权值的边，其中u∈U，v∈V-U，则必存在一颗包含边(u, v)的最小生成树。

Prim算法和Kruskal算法是两个利用MST性质构造最小生成树的算法。

**1. Prim算法**

假设N=(V, E)是一个连通网，TE是N上最小生成树中边的集合。算法从U={u0}(u0∈V), TE={}开始，重复执行下述操作：在所有u∈U，v∈V-U的边(u, v)∈E中找一条代价最小的边(u0, v0)并入集合TE，同时v0并入U，直至U=V为止。此时TE中必有n-1条边，则T=(V, TE)为N的最小生成树。

为实现这个算法需附设一个数组closedge，以记录从U到V-U具有最小代价的边。对每个顶点vi∈V-U，在辅助数组中存在一个相应分量closedge[i-1]，它包括两个域，其中lowcost存储该边上的权。显然，

closedge[i].lowcost=Min{cost(u, vi)} {u∈U}







图6.4 Prim算法构造最小生成树的过程

adjvex域存储该边依附的在U中的顶点。例如，图6.4所示为按Prim算法构造网的一棵最小生成树的过程，在构造过程中辅助数组中各分量值的变化如表6.1所示。

表6.1 图6.4构造最小生成树过程中辅助数组的变化

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i  closedge | 1 | 2 | 3 | 4 | 5 | U | V-U | k |
| adjvex  lowcost | v0  6 | v0  1 | v0  5 |  |  | { v0} | { v1,v2,v3,v4,v5 } | 2 |
| adjvex  lowcost | v2  5 | 0 | v1  5 | v2  6 | v2  4 | { v0,v2} | { v1,v3,v4,v5 } | 5 |
| adjvex  lowcost | v2  5 | 0 | v6  2 | v2  6 | 0 | { v0,v2,v5} | { v1,v3,v4 } | 3 |
| adjvex  lowcost | v2  5 | 0 | 0 | v2  6 | 0 | { v0,v2,v5,v3} | { v1,v4 } | 1 |
| adjvex  lowcost | 0 | 0 | 0 | v1  3 | 0 | { v0,v2,v5,v3,v1} | {v4} | 4 |
| adjvex  lowcost | 0 | 0 | 0 | 0 | 0 | { v0,v2,v5,v3,v1, v4} | {} |  |

该算法的C语言实现如下。

typedef struct closedge\_t{

int adjvex; /\* 弧头，属于U \*/

graph\_weight\_t lowcost;

}closedge\_t;

static int minimum(const closedge\_t closedge[], int n)

{

int i;

int min\_value = closedge[0].lowcost;

int min\_loc = 0;

for(i = 1; i < n; i++) {

if(min\_value > closedge[i].lowcost) {

min\_value = closedge[i].lowcost;

min\_loc = i;

}

}

return min\_loc;

}

/\*

\* @brief 从顶点u出发构造图g的最小生成树TE，输出TE的边.

\* @param[in] g 图对象的指针

\* @param[in] u 出发顶点

\* @return 无

\* @note 本算法适用于邻接矩阵。

\* @remarks 无

\*/

static void mgraph\_prim(const mgraph\_t \*g, int u)

{

int i, j;

const int n = g->vertices\_num;

/\* closedge[n]，记录从顶点集U到V-U的\*/

closedge\_t\* const closedge = (closedge\_t\*)

malloc(n \* sizeof(closedge\_t));

if(closedge == NULL) return /\* ERR\_MALLOC\_FAILED \*/;

/\* 辅助数组初始化\*/

for(i = 0; i< n && i != u; i++) {

closedge[i].adjvex = u;

closedge[i].lowcost = g->edges[u][i];

}

closedge[u].lowcost = 0; /\* 初始, U={u} \*/

for(i = 0; i< n && i != u; i++) { /\* 其余的n-1个顶点\*/

/\* 求出TE的下一个顶点k \*/

const int k = minimum(closedge, n);

/\*输出生成树TE的边，即此边加入TE\*/

printf("%d ---> %d\n", closedge[k].adjvex, k);

closedge[k].lowcost = 0; /\* 顶点k并入U \*/

/\* 更新k的邻接点的值，不相邻为无穷大\*/

for(j = 0; j < n; j++) {

const int w = g->edges[k][j];

if(w < closedge[j].lowcost) {

closedge[j].adjvex = k;

closedge[j].lowcost = w;

}

}

}

free(closedge);

}

算法分析：假设网中有n个顶点，则第一个进行初始化的循环语句的频度为n，第二个循环语句的频度为n-1。其中有两个内循环：其一是在closedge[v].lowcost中求最小值，其频度为n-1；其二是重新选择具有最小代价的边，其频度为n。因此Prim算法的时间复杂度为O(n2)，与网中边数无关，因此适用于求边稠密的网的最小生成树。

Prim算法的另一种实现是使用小根堆，其流程是：小根堆中存储一个端点在生成树中，另一个端点不在生成树的边，每次从小根堆的堆顶可选出权值最小的边(u, v)，将其从堆中推出，加入生成树中。然后将新出现的所有一个端点在生成树中，一个端点不在生成树的边都插入小根堆中。下一轮迭代中，下一条满足要求的边又上升到堆顶。如此重复n-1次，最后建立起该图的最小生成树。该算法的C代码实现如下。

typedef struct edge\_t{

int tail, head;

graph\_weigh\_t cost;

}edge\_t;

static int edge\_cmp(const void \*e1, const void \*e2)

{

const edge\_t\* const e11 = (const edge\_t \*)e1;

const edge\_t\* const e22 = (const edge\_t \*)e2;

return e11->cost - e22->cost;

}

/\*

\* @brief 从顶点u出发构造图g的最小生成树TE，输出TE的边.

\* @param[in] g 图对象的指针

\* @param[in] u 出发顶点

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void graph\_prim(const graph\_t \*g, int u)

{

int i, count = 1;

edge\_t ed;

heap\_t \*h;

const int n = graph\_get\_vertices\_number(g);

/\* 判断顶点是否已经加入最小生成树\*/

bool\* const is\_in\_u = (bool \*)malloc(n \* sizeof(bool));

if(is\_in\_u == NULL) return /\* ERR\_MALLOC\_FAILED \*/;

for(i = 0; i < n; i++) is\_in\_u[i] = false;

if(!heap\_init(&h, sizeof(edge\_t), edge\_cmp)) {

free(is\_in\_u);

return /\* ERR\_OTHER \*/;

}

/\* 开始顶点加入U(所以count初始为) \*/

is\_in\_u[u] = true;

do

{

int v;

for(v = graph\_get\_first\_neighbor(g, u); v >= 0;

v = graph\_get\_next\_neighbor(g, u, v)) {

/\* 若v不在生成树，(u,v)加入堆\*/

if(!is\_in\_u[v]) {

ed.tail = u;

ed.head = v;

/\* tail在树内，head不在树内\*/

ed.cost = graph\_get\_weight(g, u, v);

(void)heap\_insert(h, &ed);

}

}

while(!heap\_is\_empty(h) && count < n) {

/\* 从堆中退出最小权值边，存入ed \*/

(void)heap\_remove(h, &ed);

if(!is\_in\_u[ed.head]) {

/\* 输出生成树TE的边，即此边加入TE \*/

printf("%d ---> %d\n", ed.tail, ed.head);

u = ed.head;

/\* u并入到生成树的顶点集合U \*/

is\_in\_u[u] = true;

count++;

break;

}

}

} while (count < n);

free(is\_in\_u);

heap\_uninit(&h);

}

算法分析：该算法迭代次数为O(n)，每次迭代将平均e/n条边插入最小堆中，e条边冲堆中删除，堆的插入和删除操作时间复杂度均为O(log2e)，则总的时间复杂度为 O(elog2e)。

**2. Kruskal算法**

假设连通网N={V, E}，则令最小生成树的初始状态为只有n个顶点而无边的非连通图T=(V, {})，图中每个顶点自成一个连通分量。在E中选择代价最小的边，若该边依附的顶点落在T中不同的连通分量上，则将此边加入到T中，否则舍去此边而选择下一条代价最小的边。依次类推，直至T中所有顶点都在同一连通分量上为止。

图6.5所示为Kruskal算法构造一棵最小生成树的过程。







图6.5 Kruskal算法构造最小生成树的过程

下面是Kruskal算法的C语言实现。

/\*

\* @brief Kruskal算法求图的最小生成树.

\* @param[in] g 图对象的指针

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void graph\_kruskal(const graph\_t \*g)

{

edge\_t ed;

int u, v, count;

const int n = graph\_get\_vertices\_number(g);

heap\_t \*h;

int \*ufs;

ufs = (int \*)malloc(n \* sizeof(int));

if(ufs == NULL) return /\* ERR\_MALLOC\_FAILED \*/;

ufs\_init(ufs, n);

if(!heap\_init(&h, sizeof(edge\_t), edge\_cmp)) {

free(ufs);

return /\* ERR\_OTHER \*/;

}

/\* 把所有边插入堆中\*/

for(u = 0; u < n; u++) {

for(v = u + 1; v < n; v++) {

const int w = graph\_get\_weight(g, u, v);

if(w < GRAPH\_INFINITY) {

ed.tail = u;

ed.head = v;

ed.cost = w;

(void)heap\_insert(h, &ed);

}

}

}

count = 0; /\* 最小生成树的边数\*/

while (count < n - 1)

{

/\* 从堆中退出最小权值边，存入ed \*/

(void)heap\_remove(h, &ed);

/\* 取两顶点所在集合的根\*/

u = ufs\_find(ufs, ed.tail);

v = ufs\_find(ufs, ed.head);

if(u != v) { /\* 不是同一集合，说明不连通\*/

ufs\_union(ufs, u, v); /\* 合并，连通成一个分量\*/

/\* 输出生成树TE的边，即此边加入TE \*/

printf("%d ---> %d\n", ed.tail, ed.head);

count++;

}

}

free(ufs);

heap\_uninit(&h);

}

算法分析：

如果采用邻接矩阵作为图的存储结构，则在建立小根堆时需要检测图的邻接矩阵，这需要O(n2)的时间。此外，需要将e条边组成初始的小根堆。如果直接从空堆开始，依次插入各边，需要O(elog2e)的时间。在构造最小生成树的过程中，需要进行O(e)次出堆操作heap\_remove()、2e次并查集的ufs\_find()操作以及n-1次ufs\_union()操作，计算时间分别为O(elog2e)、O(elog2n)和O(n)，所以总时间为O(n2+elog2e)。

如果采用邻接表作为图的存储结构，则在建立小根堆时需要检测图的邻接表，这需要O(n+e)的时间。为建成初始的小根堆，需要O(elog2e)的时间。在构造最小生成树的过程中，需要进行O(e)次出堆操作heap\_remove()、2e次并查集的ufs\_find()操作以及n-1次ufs\_union()操作，计算时间分别为O(elog2e)、O(elog2n)和O(n)，所以总时间为O(n+elog2e)。

#### 5.4.2 最短路径

**1. 单源最短路径(Dijkstra算法)**

假设S为已求得最短路径的点的集合，则可证明：下一条最短路径（设其终点为x）或者是弧(v, x)，或者是中间只经过S中的顶点而最后到达顶点x的路径。

Dijkstra算法流程如下：

1、S为已找到从v出发的最短路径的终点的集合，它的初始状态为空集。dist[i]存放的是v到vi的最短路径长度，根据前面所述性质，dist[i]=min{dist[i],weight(v,vi)}。path[i]存放的是最短路径上指向vi的弧尾顶点。那么从v出发到图上其余vi的最短路径长度的初值为：

dist[i] = weight(v, vi) vi∈V

2、选择vj，使得

dist[j]=min{dist[j], weight(v, vj)|vi∈V-S}

将vj加入到S，

S = S U {vj}

3、修改从v出发到集合V-S上任一顶点vk可达的最短路径长度，并记录下这条边。

if(dist[j] + weight(j, k) < dist[k]) {

dist[k] = dist[j] + weight(j, k);

path[k] = j; /\* 修改到k的最短路径 \*/

}

4、重复2，3共n-1次。

例如，对图6.6所示的有向图及其邻接矩阵运行运行Dijkstra算法，



图6.6 有向图及其邻接矩阵

运算过程中v0到其余个顶点的最短路近，dist[]向量的变化情况如下表所示：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 终点 | i=1 | i=2 | i=3 | i=4 | i=5 |
| v1 | ∝ | ∝ | ∝ | ∝ | ∝ |
| v2 | 10  (v0,v2) |  |  |  |  |
| v3 | ∝ | 60  (v0,v2,v3) | 50  (v0,v4,v5) |  |  |
| v4 | 30  (v0,v4) | 30  (v0,v4) |  |  |  |
| v5 | 100  (v0,v5) | 100  (v0,v5) | 90  (v0,v4,v5) | 60  (v0,v4,v3, v5) |  |
| vj | v2 | v4 | v3 | v5 |  |
| S | {v0,v2} | {v0,v2,v4} | {v0,v2,v3,v4} | {v0,v2,v3,v4,v5} |  |

Dijkstra算法的C语言实现如下。

/\*

\* @brief Dijkstra算法求最短路径.

\* @param[in] g 图对象的指针

\* @param[in] v 从顶点v出发

\* @param[out] dist dist[i]存放的是v到vi的最短路径长度

\* @param[out] path path[i]存放的是最短路径上指向vi的弧尾顶

\* 点

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void mgraph\_dijkstra(const mgraph\_t \*g, int v,

int dist[], int path[])

{

int i, j, u;

graph\_weight\_t w, min;

const int n = g->vertices\_num;

bool\* const S = (bool\*)malloc( n \* sizeof(bool));

if(S == NULL) return /\* ERR\_MALLOC\_FAILED \*/;

for(i = 0; i < n && i !=v; i++) {

dist[i] = g->edges[v][i];

S[i] = false;

if(dist[i] < GRAPH\_INFINITY) {

path[i] = v;

} else {

path[i] = -1; /\* 没有顶点指向i \*/

}

}

S[v] = true; /\* 初始化，顶点v加入S \*/

dist[v] = 0;

path[v] = -1;

for(i = 0; i < n && i !=v; i++) {

/\* 选不再S中的最短路径顶点u \*/

min = GRAPH\_INFINITY;

for(j = 0; j < n; j++) {

if(!S[j] && dist[j] < min) {

u = j;

min = dist[j];

}

}

S[u] = true;

for(j = 0; j < n; j++) {

w = g->edges[u][j];

/\* 顶点j未就加入S，且经过u到j可缩短路径\*/

if(!S[j] && w < GRAPH\_INFINITY &&

dist[u] + w < dist[j]) {

dist[j] = dist[u] + w;

path[j] = u; /\* 修改到j的最短路径\*/

}

}

}

free(S);

}

算法分析：该算法包含了两个并列的for循环，第一个for循环做辅助数组的初始化工作，计算时间为O(n)，第二个for循环是二重嵌套循环，进行最短路径的求解工作，由于对图中几乎每个顶点都要做计算，每个顶点的又要对集合S内的顶点进行检测，对集合V-S内中的顶点进行修改，所以运算时间复杂度为O(n2)。算法总的时间复杂度为O(n2)。

**2. 每点最短路径(Floyd算法)**

Floyd算法的基本思想是：假设求从定点vi到vj的最短路径。初始时，若vi与vj之间存在边，则最短路径长度为此边的权值；若不存在边，则最短路径长度为无穷大。以后逐步在路径中加入顶点k(k=0,1,...,n-1)作为中间顶点，如果加入中间顶点后，得到的路径比原来的路径长度减少了，则以新路径代替原路径。

首先比较(vi,vj)和(vi,v0,vj)的路径长度，取较短者为从vi到vj的中间顶点的序号不大于0的最短路径。如果(vi,v0,vj)较短，则取(vi,v0,vj)作为最短路径。假如在路径上再增加一个顶点v1，也就是说，如果(vi,...,v1)和(v1,...,vj)分别是当前找到的中间定点的序号不大于0的最短路径，那么(vi,...,v1,...,vj)就有可能是从vi到vj的中间顶点的序号不大于1的最短路径，将它和已经得到的从vi到vj的中间顶点的序号不大于0的最短路径相比较，选出较短者作为从vi到vj的中间顶点的序号不大于1的最短路径。再增加一个顶点v2，继续进行试探，依此类推。一般的，若(vi,...,vk)和(vk,...,vj)分别是从vi到vk和从vk到vj的中间定点的序号不大于k-1的最短路径，则将(vi,...,vk,...,vj)和已经得到的从vi到vj的中间顶点的序号不大于k-1的最短路径相比，较短者便是从vi到vj的中间顶点的序号不大于k的最短路径。这样，在经过n次比较后，最后求得的必是从vi到vj的最短路径。

现定义一个n阶方阵序列，

D(-1), D(0) , D(1),..., , D(k),..., , D(n-1)

其中

D(-1) [i][j]=g->m[i][j],

D(k)[i][j]=Min{D(k-1)[i][j], D(k-1) [i][k] + D(k-1) [k][j]}

0≤k≤n-1

上述公式中，D(k)[i][j]是从vi到vj的中间顶点的序号不大于k的最短路径的长度；D(n-1)[i][j]是从vi到vj的最短路径的长度。

例如，对图6.7所示的有向图及其邻接矩阵运行Floyd算法，



图6.7 有向图及其邻接矩阵

运算过程中矩阵D的变化如表6.2所示。

表6.2 Floyd算法过程中方阵和最短路径的变化

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| D | D(-1) | | | D(0) | | | D(1) | | | D(2) | | |
| 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 |
| 0 | 0 | 4 | 11 | 0 | 4 | 11 | 0 | 4 | 6 | 0 | 4 | 6 |
| 1 | 6 | 0 | 2 | 6 | 0 | 2 | 6 | 0 | 2 | 5 | 0 | 2 |
| 2 | 3 | ∞ | 0 | 3 | 7 | 0 | 3 | 7 | 0 | 3 | 7 | 0 |
| P | P(-1) | | | P (0) | | | P (1) | | | P (2) | | |
| 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 |
| 0 |  | AB | AC |  | AB | AC |  | AB | ABC |  | AB | ABC |
| 1 | BA |  | BC | BA |  | BC | BA |  | BC | BCA |  | BC |
| 2 | CA |  |  | CA | CAB |  | CA | CAB |  | CA | CAB |  |

Floyd算法的C语言实现如下。

/\*

\* @brief Floyd算法求每点之间最短路径.

\* @param[in] g 图对象的指针

\* @param[out] d d[i][j]是顶点i和j之间最短路径长度

\* @param[out] p p[i][j]是最短路径上i和j之间的顶点

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void mgraph\_floyd(const mgraph\_t \*g,

int d[MAX\_VERTICES\_NUM][MAX\_VERTICES\_NUM],

int p[MAX\_VERTICES\_NUM][MAX\_VERTICES\_NUM])

{

int i, j, k;

const int n = g->vertices\_num;

for(i = 0; i < n; i++) {

for(j = 0; j < n; j++) {

d[i][j] = g->edges[i][j];

if(i != j && d[i][j] < GRAPH\_INFINITY) {

p[i][j] = i;

} else {

p[i][j] = -1;

}

}

}

for(k = 0; k < n; k++) {

for(i = 0; i < n; i++) {

for(j = 0; j < n; j++) {

/\* i到j的路径上加入顶点k可以缩短路径长度\*/

if(d[i][k] + d[k][j] < d[i][j]) {

d[i][j] = d[i][k] + d[k][j];

p[i][j] = k;

}

}

}

}

}

算法分析：该算法中有两个并列的for循环，第一个循环是个二重循环，用于初始化方阵D；第二个循环是个三重循环，逐步生成D(0), D(1) ,...,D(n-1)。所以算法总的时间复杂度为O(n3)。

#### 5.4.3 拓扑排序

由某个集合上的一个偏序得到该集合上的一个全序，这个操作称为**拓扑排序**。

拓扑序列的特点是：若有向边<Vi, Vj>是途中的弧，则在序列中顶点Vi必须排在顶点Vj之前。

如果用有向图表示一个工程，顶点表示活动，用有向边<vi,vj>表示活动必须先于活动进行。这种有向图叫做顶点表示活动的网络(Activity On Vertext Network)，简称**AOV网络**。

检测AOV网络是否存在环的方法是对AOV网络构造其顶点的拓扑有序序列。拓扑排序的基本步骤是：

1、在有向图中选一个没有前驱的顶点且输出之；

2、从图中删除该顶点和所有以它为尾的弧线。

重复以上两步，直至全部顶点输出，或当前图中不存在无前驱的顶点为止（这中情况说明图中存在环）。

拓扑排序的C语言实现如下。

/\*

\* @brief 拓扑排序.

\* @param[in] g 图对象的指针

\* @param[in] in\_degree in\_degree[i]是顶点i的入度

\* @param[out] v 保存拓扑排序的结果

\* @return 无环返回true，有环返回false

\* @note 无

\* @remarks 无

\*/

static bool graph\_topological\_sort(const graph\_t \*g,

int in\_degree[],

int v[])

{

int i, u;

int count = 0; /\* 拓扑序列的元素个数\*/

const int n = graph\_get\_vertices\_number(g);

stack\_t \*s;

if(!stack\_init(&s, sizeof(int))) return false;

for(i = 0; i < n; i ++) {

if(in\_degree[i] == 0)

(void)stack\_push(s, &i);

}

while(!stack\_is\_empty(s)) {

(void)stack\_pop(s, &u);

v[count++] = u;

for(i = graph\_get\_first\_neighbor(g, u); i != -1;

i = graph\_get\_next\_neighbor(g, u, i)){

if(--in\_degree[i] == 0)

(void)stack\_push(s, &i);

}

}

if(count != n) { /\* 有环\*/

return false;

} else { /\* 无环\*/

return true;

}

}

算法分析：对有n个顶点和e条边的AOV网络而言，求各顶点的入度所需时间为O(e)，建立零入度顶点栈所需时间为O(n)；在拓扑排序过程中，若有向图无环，每个顶进一次栈出一次栈，顶点入度减1的操作共执行了e次。所以总的时间复杂度为O(n+e)。

当有向图中无环时，也可以利用深度优先搜索进行拓扑排序。因为图中无环，深度优先遍历不会死循环。进行深度优先遍历时，最先退出DFS函数的顶点即为出度为零的顶点，是拓扑有序序列的最后一个顶点。由此，按退出DFS函数的先后次序记录下来的顶点序列即为逆向的拓扑有序序列。

#### 5.4.4 关键路径

用有向边上的权值表示活动的持续时间，用顶点表示时间，这样的有向图叫做边表示的活动网络(Activity On Edge Network)，简称**AOE网络**。

路径最长的路径叫做关键路径(Critical Path)。假设开始点为v1，从v1到vi的最长路径长度叫做事件vi的最早发生时间。这个事件决定了所有以vi为尾的弧所表示的活动的最早开始时间。我们用e(i)表示活动ai的最早开始时间。还可以定义一个活动的最迟开始时间l(i)，这是在不推迟整个工程完成的前提下，活动ai最迟必须开始进行的时间。两者之差l(i)-e(i)意味着完成活动ai的时间余量。我们把l(i)=e(i)的活动叫做关键活动。

设活动ai由弧<j, k>表示，为了求得活动的e(i)和l(i)，首先应求得事件的最早发生时间ve(j)和最迟发生时间vl(j)，其持续时间记为dut(<j, k>)，则有如下关系

e(i)=ve(j)

l(i)=vl(k)-dut(<j, k>)

求ve(j)和vl(k)需分两步进行：

(1)从ve(0)=0开始向前递推

ve(j)=Max{ve(i)+dut(<i, j>)} <i, j>∈T

其中T是所有以顶点j为弧头的边的集合。

(2)从vl(n-1)=ve(n-1)起向后递推

vl(j)=Min{ vl(k)-dut(<j, k>)} <j, k>∈S

其中S是所有以顶点j为弧尾的边的集合。



图6.8 AOE网络及其关键路径

例如，对图6.8(a)所示的AOE网络计算结果如图6.9所示，可见a2、a5和a7为关键活动，组成一条从源点到汇点的关键路径，如图6.8(b)所示。

|  |  |
| --- | --- |
| 顶点 ve vl | 活动 e l l-e |
| v1 0 0  v2 3 4  v3 2 2  v4 6 6  v5 6 7  v6 8 8 | a1 0 1 1  a2 0 0 0  a3 3 4 1  a4 3 4 1  a5 2 2 0  a6 2 5 3  a7 6 6 0  a8 6 7 1 |

图6.9 图6.8(a)所示AOE网络的关键路径的计算过程

## 第6章 查找

### 6.1 查找的基本概念

查找表（Search Table）是由同一类型的数据元素（或记录）构成的集合。由于“集合”中的数据元素之间存在着完全松散的关系，因此查找表是一种非常灵便的数据结构。

对查找表经常进行的操作有：（1）查询某个“特定的”的数据元素是否在查找表中；（2）检索某个“特定的”数据元素的各种属性；（3）在查找表中插入一个数据元素；（4）从查找表中删除某个数据元素。若对查找表只作前两种统称为“查找”的操作，则称此类查找表为**静态查找表**（Static Search Table）。若在查找过程中同时插入查找表中不存在的数据元素，或者从查找表中删除已存在的某个数据元素，则称此类表为**动态查找表**（Dynamic Search Table）。

衡量一个查找算法的时间效率的标准是：在查找过程中关键字的平均比较次数，这个标准也称为平均查找长度（Average Search Length, ASL）。

### 6.2 静态查找表

#### 6.2.1 顺序查找

/\*\* @file search.c

\* @brief 查找的各种算法.

\* @author soulmachine@gmail.com

\* @date 2010-8-7

\* @version 0.1

\* @note 无

\*/

/\* 要查找的数据元素的类型\*/

typedef int data\_t;

/\*

\* @brief 使用监视哨的顺序查找算法.

\* @param[in] a 存放数据元素的数组

\* @param[in] n 数组的元素个数

\* @param[in] x 要查找的元素

\* @return 查找成功则返回元素所在下标，否则返回n

\* @note 由于末尾位置要存放哨兵，所以数组l的容量大小应

\* 该为n+1

\* @remarks a[n]起到了监视哨的作用，目的在于免去查找过

\* 程中每次都要检测整个表是否查找完毕(i < n)，这样能使

\* 每次循环所需的比较次数由次降为次。

\*/

static int seq\_search(data\_t a[], int n, data\_t x)

{

int i = 0;

a[n] = x; /\* 哨兵\*/

while(a[i] != x) i++;

return i;

}

算法分析：

ASL成功=，ASL失败=n+1

/\*

\* @brief 不使用监视哨的顺序查找算法.

\* @param[in] a 存放数据元素的数组

\* @param[in] n 数组的元素个数

\* @param[in] x 要查找的元素

\* @return 查找成功则返回元素所在下标，否则返回n

\* @note 无

\* @remarks 无

\*/

static int seq\_search1(const data\_t a[], int n, data\_t x)

{

int i = 0;

while(i < n && a[i] != x) i++;

return i;

}

/\*

\* @brief 有序顺序表的顺序查找算法.

\* @param[in] a 存放数据元素的数组，已排好序

\* @param[in] n 数组的元素个数

\* @param[in] x 要查找的元素

\* @return 查找成功则返回元素所在下标，否则返回-1

\* @note 无

\* @remarks 无

\*/

static int seq\_search2(const data\_t a[], int n, data\_t x)

{

int i;

for(i = 0; i < n; i++) {

if(a[i] == x) return i;

else if(a[i] > x) break;

}

return -1;

}

算法分析：

ASL成功=，

ASL失败=

有时，表中各个元素的的查找概率并不相等。因此若能预先得知每个元素的查找概率，可以先对元素按查找概率进行排序，使表中的元素从大到小排列，以便提高查找效率。

然而，在一般情况下，元素的查找概率预先无法测定。为了提高查找效率，我们可以在每个元素中附设一个访问频度域，并使顺序表中的元素始终保持按访问频度从大到小的次序排列，使得查找概率大的元素在查找过程中不断前移。

#### 6.2.2 折半查找

**/\***

**\* @brief 有序顺序表的折半查找算法.**

**\* @param[in] a 存放数据元素的数组，已排好序**

**\* @param[in] n 数组的元素个数**

**\* @param[in] x 要查找的元素**

**\* @return 查找成功则返回元素所在下标，否则返回-1**

**\* @note 无**

**\* @remarks 无**

**\*/**

**int binary\_search(const data\_t a[], int n, data\_t x)**

**{**

**int left = 0, right = n -1, mid;**

**while(left <= right) {**

**mid = (left + right) / 2;**

**if(x > a[mid]) {**

**left = mid + 1;**

**} else if(x < a[mid]) {**

**right = mid - 1;**

**} else {**

**return mid;**

**}**

**}**

**return -1;**

**}**

算法分析：

ASL成功=

折半查找算法仅适用于有序的顺序表，如果想要在有序链表上实施折半查找，那就要讨论跳表了，这已超出考试大纲范围。

在有序的静态查找表上进行查找的方法除折半查找外，还有斐波那契查找和插值查找，已超出考试大纲范围。

### 6.3 二叉查找树和平衡二叉树

见4.4.3节和4.4.4节。

### 6.4 B树

#### 6.4.1 B树的概念

一棵m阶的B树，或为空树，或为满足下列特性的m叉树：

（1）每个结点至多有m棵子树；

（2）根结点至少有两棵子树；

（3）除根之外的所有结点至少有棵子树；

（4）每个结点都包含如下信息

(n, P0, K1, P1, K2, P2, ..., Kn, Pn)

其中，n是结点内关键字的实际个数，Ki(i=1, 2, ..., n)为关键字，且Ki<Ki+1；Pi(i=0, 1, ..., n)为指向子树根结点的指针，且指针Pi所指子树中所有的关键字均小于Ki+1，大于Ki。

（5）所有的失败结点都位于同一层，它们是查找失败时查找指针到达的结点。所有失败结点都是空结点，指向它们的指针都为空。

注意，B树的高度h不包括失败结点，最底层的结点为叶结点。

图7.1是一棵3阶B树。



图7.1 一棵3阶B树

#### 6.4.2 B树的查找

B树的查找过程是一个在结点内查找和循某一条路径向下一层查找交替进行的过程。例如，在图7.1所示的B树中查找关键字值等于35的元素，需要从根开始查找。首先从磁盘读入结点a，沿30右侧的子树指针找到结点c，再从磁盘读入结点c，沿40左侧的子树指针找到结点f，最后读入结点f，在结点f中找到关键字35。

#### 6.4.3 B树的插入

B树是从空树起，逐个插入关键字而生成的。在B树，每个

非失败结点的关键字个数都在[-1, m-1]之间。插入是在某

个叶结点（即第h层的结点）开始的。如果关键字插入后结点中的关键字个数到达m，则结点需要“分裂”，否则可以直接插入。结点“分裂”的原则是：

设结点p中已经有m-1个关键字，当再插入一个关键字后结点的状态为：

(m, P0, K1, P1, K2, P2, ..., Km, Pm)

这时必须把结点p分裂成两个结点p和q，它们包含的信息分别为：

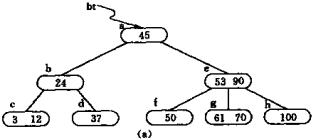
结点p：(-1, P0, K1, P1, K2, P2, ..., )

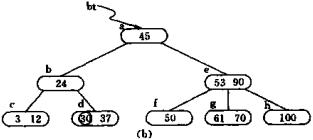
结点q：(m-, , ..., Km, Pm)

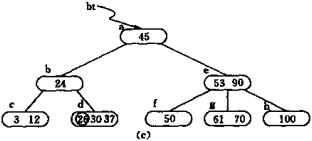
位于中间的关键字与指向新结点q的指针一起插入到这两

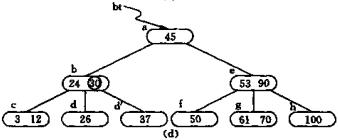
个结点的双亲结点中去。

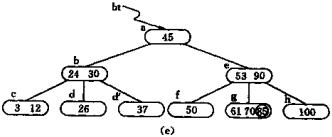
例如，图7.2展示了在B树中插入结点的过程。

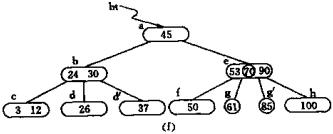


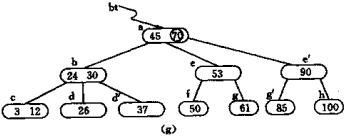


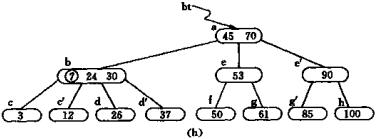


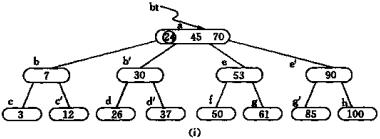












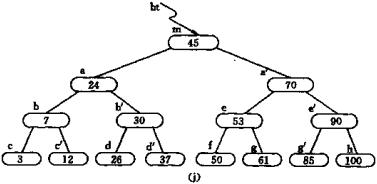


图7.2 在B树中插入结点（省略失败结点）

(a)一颗2-3树；(b)插入30之后；(c)-(d)插入26之后；

(e)-(g)插入85之后；(h)-(j)插入7之后

#### 6.4.4 B树的删除

想要在B树上删除一个关键字，先要在B树上找到这个关键字所在的结点。若该结点不是叶结点，设被删关键字为Ki，可以用Pi所指子树中的最小关键字x来替代Ki，然后在x所在的叶结点中中删除x。现在问题归于在叶结点中删除关键字了。

在叶结点上删除关键字有4中情况：

(1) 若被删关键字所在叶结点同时又是根结点且该结点关键字个数n≥2，则直接删去该关键字并将修改后的结点写回磁盘，删除结束。

(2) 若被删关键字所在叶结点不是根结点且该结点关键字个数

n≥，则直接删去该关键字并将修改后的结点写回磁盘，删

除结束。

(3) 若被删关键字所在叶结点关键字个数n=-1，且与该结点相邻的右兄弟（或左兄弟）结点的关键字个数n≥，则

需将其兄弟结点中的最小（或最大）关键字上移至双亲结点中，而将双亲结点中小于（或大于）且紧靠该上移关键字的关键字下移至被删关键字所在结点。

(4) 若被删关键字所在叶结点与其相邻的兄弟结点的关键字个数

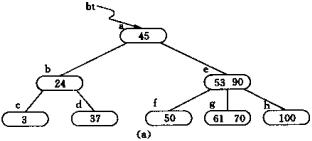
均等于-1，则需要个并这两个结点。假如该结点有右兄弟

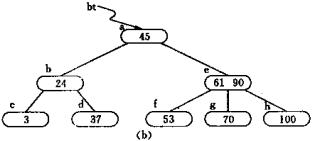
且由其双亲结点中的指针Pi所指向，则删去该关键字后，被删关键字所在叶结点中剩余的关键字和指针，加上双亲结点中的关键字Ki一起，合并到Pi所指兄弟结点中去（若没有右兄弟，则合并至左兄弟结点）。例如，在图7.3(b)所示B树中删除53，应先删除f结点，并将f中的剩余信息（指针空）和双亲结点e中的61合并到右兄弟结点g中，删除后的树如图7.3(c)所示。如果因此使双

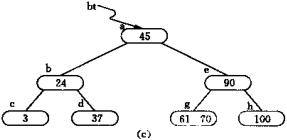
亲结点的关键字个数小于-1，则依次作类似处理。例如，

在图7.3(c)的B树中删去关键字37之后，双亲结点中b中剩余信息应和其双亲结点a中关键字一起合并至右兄弟结点e中，删除后的B树如图7.3(d)所示。

图7.3展示了在B树中删除关键字的过程。







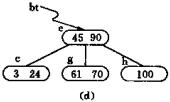


图7.3 在B树中删除关键字

(a)删除50 (b)删除53 (c)删除37

### 6.5 B+树

#### 6.5.1 B+树的概念

B+是B树的变形，它与B树的不同之处在于：

（1）叶结点包含了全部关键字以及指向这些关键字记录的指针，且叶结点本身按关键字从小到大顺序链接。

（2）上层的非叶结点可以看成是索引部分，结点中仅含有其子树中的最大关键字。

一棵m阶的B树，或为空树，或为满足下列特性的m叉树：

（1）每个结点至多有m棵子树；

（2）根结点至少有一棵子树；

（3）除根之外的所有结点至少有棵子树；

（4）每个非叶结点都包含如下信息

(n, K1, P1, K2, P2, ..., Kn, Pn)

其中，n是结点内关键字的实际个数，Ki(i=1, 2, ..., n)为关键字，且Ki<Ki+1；Pi(i=1, ..., n)为指向子树根结点的指针，且Ki是指针Pi所指子树中的最大关键字。

（5）所有的叶结点都位于同一层，按从小到大的顺序存放全部关键字，各个叶结点顺序链接。

图7.4是一棵4阶B+树。



图7.4 一棵4阶B+树

#### 6.5.2 B+树的查找

在B+树中有两个头指针：一个指向B+树的根结点，一个指向关键字最小的叶结点。因此，可以对B+树进行两种查找运算：一种是循叶结点自己拉起的链表顺序查找，另一种是从根结点开始，进行自顶向下，直至叶结点的随机查找。

在B+树上进行随机查找的过程基本上与B树类似。只是在查找过程中，如果非叶结点上的的关键字等于给定值，查找并不停止，而是继续沿右指针向下，一直查到叶结点上的这个关键字。因此，在B+树中不论查找成功与否，每次查找都是走了一条从根到叶结点的路径。

#### 6.5.3 B+树的插入

B+树的插入首先在叶结点上进行。每插入一个关键字后都要判断该结点的关键字个数n是否超出上限m。如果没有超过，插入结束；否则可将结点分裂为两个，它们所包含的关键字个数分

别为和。然后把这两个结点的最大关

键字和结点地址插入到他们的双亲结点中。在非叶结点中关键字的插入与叶结点类似，如果非叶结点中的关键字个数超过了m，也要进行结点分裂。如果需要做根结点分裂，必须创建新的双亲结点，作为树的新根。这样树的高度就增加了一层。图7.5通过在一棵4阶B+树上连续插入13个关键字，描述了在插入过程中B+树的变化。



图7.5 B+树插入的示例

#### 6.5.4 B+树的删除

B+树的删除首先在叶结点上进行。如果在该叶结点上删除一个关键字后，结点的关键字个数仍然不少于，删除结束。例如，在图7.6所示的B+树中删除关键字15后，虽然删除的是该节点的最大关键字，但因在其上层的副本只是起了一个引导查找的“分界关键字”的作用，所以即使树中已经删除了关键字15，但上层的副本仍然可以保留，如图7.6(a)。

如果在叶结点上删除一个关键字后，该节点的关键字个数小于，必须作结点的调整或合并工作。例如，在图7.6(a)所示的B+树中从叶结点删除关键字10后，该结点的关键字个数n=1< = 2，这时看它的右兄弟结点，发现右兄弟结点的关键字个数n=3>2，因此可以右兄弟结点的最左关键字移过来，使得两个结点的关键字个数都在允许范围之内，如图7.6(b)。

如果在图7.6(b)所示的B+树上再删除叶结点的关键字72，该结点的关键字个数减为1，小于下限，其右兄弟结点中关键字个数为2，已经到达下限，因此必须进行结点合并。所有关键字都集中于左边的结点，将右边结点删去。这样导致其上层的非叶结点中的子树棵数n(=1)<，必须与其做兄弟结点进行调整。其做兄弟结点中的子树棵数n(=3)>，可以将它属下的最右的一个叶结点移到右边的非叶结点下，同时修改各非叶结点相应的“分界关键字”，从而达到新的平衡。如图7.6(c)所示。







图7.6 B+树删除的示例

### 6.6 散列表(Hash Table)

#### 6.6.1 散列函数的概念

前面讨论的各种数据结构，数据元素的关键字值和在结构中的位置不存在确定关系，因此在结构中查找数据元素需要经过一系列的比较。理想的情况是希望不经过任何比较，一次访问便能得到所查的数据元素，那就必须在数据元素的关键字和位置之间建立一个确定的对应关系f，使每个关键字和结构中唯一一个位置相对应，因而在查找时，只需要根据关键字值K计算f(K)，就得到了该元素所在位置。在此，我们称这个对应关系f为**散列函数**，按这个思想建立的表为**散列表**。

散列方法需要解决两个问题：

（1）对于给定的一个关键字集合，选择一个计算简单且地址分布均匀的散列函数，避免或尽量减少冲突；

（2）一旦发生冲突，要有系统的解决冲突的方案。

#### 6.6.2 常用的散列函数

1、直接定址法

hash(key) = a \* key + b

直接定址法所得的地址与关键字一一对应，因此不会发生冲突，它适合于关键字的分布基本连续的情况。若关键字分布不连续，空位较多，将造成存储空间的浪费。

2、除留余数法

hash(key) = key % p

这种方法的关键是选好p，使得每一个关键字映射到散列表上的任意地址的概率相等，从而尽可能地减少冲突。

一般的经验是选p为质数或不包含小于20的质因数的合数。

3、数字分析法

假设关键字是以r为基的数，关键字集合已知，则可取关键字的若干位构成地址。

4、平方取中法

取关键字平方的中间几位数作为地址，通常不一定能知道关键字的全部情况，取其中哪几位也不好确定，而一个数平方后的中间几位数和每一位都相关，由此得到的地址分布比较均匀。

5、折叠法

把关键字分割成尾数相同的几部分（最后一部分可能短一些），将它们的叠加和作为散列地址。

有两种叠加方法：（1）移位法，各部分的最后一位对齐相加；（2）分界法，各部分不折断，沿各部分的分界来回折叠，对齐相加。

折叠法适合于关键字位数很多，而且每一位上数字分布大致均匀的情况。

#### 6.6.3 解决冲突的方案

**1、开地址法**

所谓开地址法指的是散列表的空闲位置既向它的同义词开放，也向它的非同义词开放，这里的同义词指的是那些散列地址相同的不同关键字。

Hi = [hash(key) + di] % m

（1）线性探测法

di = 1, 2, 3, ... , n-1

（2）二次探测法

di = ±12, ±22, ... , ±k2 (k ≤ m / 2)

线性探测法可以保证，只要表未满，总能找到一个空闲地址，二次探测法只有在表长为形如4n+3的素数时才可能。

可以证明，当表的长度m为质数且表的装载因子α不超过0.5时，新的表项一定能够插入，而且任何一个位置不会被探测两次。因此，只要表中至少有一半空的，就不会有表满问题。在查找时可以不考虑表装满的情况；但在插入时必须却确保表的装载因子α不超过0.5。如果超过，必须将表长度扩充一倍，进行表的分裂。

（3）双散列法

di = hash2(key) \* i 1<hash2(key)<m，且与m互质

例如，hash(key) = key % m, hash2(key) = key % (m-1) + 1

这种方法不会产生堆积问题，但增加了计算时间。

注意，在开地址的情况下，不能随便删除表中已有元素。因为若删除元素会影响其他元素的查找。所以若想删除一个元素时，给它做一个删除标记deleted。这样做的副作用是，在执行多次删除后，表面上看起来散列表很满，实际上很多位置没有利用，因此需要定期维护散列表，把加了deleted标记的元素删除。

**2、拉链法**

把同义词存储在一个线性表中，这个线性表也称为一个桶。散列表的每个地址对应一个桶。

实验表明，拉链法优于线性探测法或双散列法；除留余数法优于其他散列函数，最差的是折叠法。当装载因子α较高时，因选择的散列函数不同，散列表的查找性能差别很大。因此，在一般情况下多选用除留余数法。

散列表的装载因子α定义为：



表7.1 各种方法处理冲突时的平均查找长度

|  |  |  |  |
| --- | --- | --- | --- |
| 处理冲突的方法 | | 平均查找长度ASL | |
| 查找成功Sn | 查找不成功(插入新记录)Un |
| 开地址法 | 线性探测法 |  |  |
| 二次探测法  双散列法 |  |  |
| 拉链法 | |  |  |

## 第7章 内部排序

### 7.1 排序的基本概念

**排序**是将一组任意排列的元素重新排列成一个按关键字有序的序列。

如果在记录序列中有两个记录Ri和Rj，它们的关键字Ki=Kj，且在排序之前，记录Ri排在Rj前面。如果在排序之后，记录Ri仍然在记录Rj的前面，则称这个排序方法是**稳定的**，否则称这个排序方法是**不稳定的**。

排序方法根据在排序过程中数据记录是否完全在内存，分为两大类：**内部排序**和**外部排序**。内部排序是指在排序期间数据记录全部存放在内存的排序；外部排序是指在排序期间全部记录不能同时存放在内存，在排序过程中需要不断在内、外存之间移动的排序。

### 7.2 插入排序

#### 7.2.1 直接插入排序

直接插入排序的基本思想是：把数组a[n]中待排序的n个元素看成为一个有序表和一个无序表，开始时有序表中只包含一个元素a[0]，无序表中包含有n-1个元素a[1]~a[n-1]，排序过程中每次从无序表中取出第一个元素，把它插入到有序表中的适当位置，使之成为新的有序表，这样经过n-1次插入后，无序表就变为空表，有序表中就包含了全部n个元素，至此排序完毕。在有序表中寻找插入位置是采用从后向前的顺序查找的方法。

直接插入排序的C语言实现如下。

/\*\* @file sort.c

\* @brief 各种排序算法.

\* @author soulmachine@gmail.com

\* @date 2010-8-9

\* @version 0.1

\* @note 无

\*/

#include <stdio.h>

#include "heap.h"

/\* 待排序元素的数据类型\*/

typedef int data\_t;

/\*

\* @brief 直接插入排序.

\* @param[inout] a 待排序元素序列

\* @param[in] start 开始位置

\* @param[in] end 结束位置，最后一个元素后一个位置

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void straight\_insert\_sort(data\_t a[], int start, int end)

{

data\_t tmp;

int i, j;

for(i = start + 1; i < end; i++) {

tmp = a[i];

for(j = i - 1; tmp < a[j] && j >= start; j--) {

a[j + 1] = a[j];

}

a[j + 1] = tmp;

}

}

最好情况下，即初始序列有序，每次循环只需与前一个元素比较一次，不需移动，总的比较次数为n+1，移动次数为0。

最坏情况下，即初始序列为逆序，每个元素都需要与前面的所有元素进行比较，前面的所有元素都要后移，总共的

元素比较次数

元素移动次数

直接插入排序的时间复杂度为O(n2)。它是稳定的。

#### 7.2.2 折半插入排序

在查找插入位置时，若改为折半查找，就是折半插入排序。

折半插入排序的C语言实现如下。

/\*

\* @brief 折半插入排序.

\* @param[inout] a 待排序元素序列

\* @param[in] start 开始位置

\* @param[in] end 结束位置，最后一个元素后一个位置

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void binary\_insert\_sort(data\_t a[], int start, int end)

{

data\_t tmp;

int i, j, left, right, mid;

for(i = start + 1; i < end; i++) {

tmp = a[i];

left = start;

right = i - 1;

while(left <= right) {

mid = (left + right) / 2;

if(tmp < a[mid]) {

right = mid - 1;

} else {

left = mid + 1;

}

}

for(j = i - 1;j >= left; j--) {

a[j + 1] = a[j];

}

a[left] = tmp;

}

}

比较次数为O(nlog2n)，移动次数与直接插入排序相同。它是稳定的。

#### 7.2.3 希尔(Shell)插入排序

从对直接插入排序的分析得知，其算法时间复杂度为O(n2)但是，若待排序记录序列为“正序”时，其时间复杂度可提高至O(n)。由此可设想，若待排序记录序列按关键字“基本有序”，即序列中具有下列特性

Ri.key < Max{Rj.key} j<i

的记录较少时，直接插入排序的效率就可大大提高，从另一方面来看，由于直接插入排序算法简单，则在n值很小时效率也比较高。希尔排序正是从这两点分析出发对直接插入排序进行改进得到的一种插入排序方法。

希尔排序又称缩小增量排序。该方法的基本思想是：设待排

序元素序列有n个元素，首先取一个整数gap=作为间

隔，将全部元素分为gap个子序列，所有距离为gap的元素放在同一个子序列中，在每一个子序列中分别施行直接插入排序。然

后缩小间隔gap，取gap=，重复上述的子序列划分

和排序工作，直到最后取gap=1，将所有元素放在同一个序列中排序为止。

图8.1展示了希尔排序的过程。

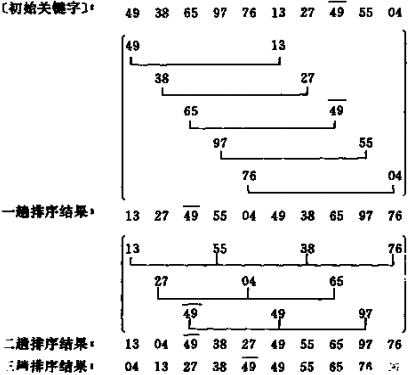


图8.1 希尔排序示例

希尔排序的C语言实现如下。

/\*

\* @brief 一趟希尔插入排序.

\* @param[inout] a 待排序元素序列

\* @param[in] start 开始位置

\* @param[in] end 结束位置，最后一个元素后一个位置

\* @param[in] gap 间隔

\* @return 无

\* @note 和一趟直接插入排序相比，仅有一点不同，就是前后

\* 元素的间距是gap而不是1

\* @remarks 无

\*/

static void shell\_insert(data\_t a[], int start, int end, int gap)

{

data\_t tmp;

int i, j;

for(i = start + gap; i < end; i++) {

tmp = a[i];

for(j = i - gap; tmp < a[j] && j >= start; j -= gap) {

a[j + gap] = a[j];

}

a[j + gap] = tmp;

}

}

/\*

\* @brief 希尔排序.

\* @param[inout] a 待排序元素序列

\* @param[in] start 开始位置

\* @param[in] end 结束位置，最后一个元素后一个位置

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void shell\_sort(data\_t a[], int start, int end)

{

int gap = end - start;

while(gap > 1) {

gap = gap / 3 + 1;

shell\_insert(a, start, end, gap);

}

}

希尔排序的分析是一个复杂的问题。到目前为止尚未有人求出一种最好的增量序列。

一种简单的取法是gap[0]=n/3+1, gap[k]=gap[k-1]/3+1。

对于规模较大的序列(n≥1000)，希尔排序具有很高的效率。并且希尔排序算法的代码简单，容易执行，所以很多排序应用程序都选用了希尔排序算法。希尔排序是不稳定的。

### 7.3 交换排序

#### 7.3.1 冒泡排序

冒泡排序的基本方法是：设待排序元素序列的元素个数为n，从后向前两两比较相邻元素的值，如果发生逆序（即前一个比后一个大），则交换它们，直到序列比较完。我们称它为一趟冒泡，结果是最小的元素交换到待排序序列的第一个位置，其他元素也都向排序的最终位置移动。下一趟冒泡时前一趟确定的最小元素不参加比较，待排序序列减少一个元素，一趟冒泡的结果又把序列中最小的元素交换到待排序序列的第一个位置。这样最多做n-1趟冒泡就能把所有元素排好序。

冒泡排序的C语言实现如下。

/\*

\* @brief 冒泡排序.

\* @param[inout] a 待排序元素序列

\* @param[in] start 开始位置

\* @param[in] end 结束位置，最后一个元素后一个位置

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void bubble\_sort(data\_t a[], int start, int end)

{

int exchange; /\* 是否发生交换\*/

data\_t tmp;

int i, j;

for(i = start; i < end - 1; i++) {

exchange = 0;

for(j = end - 1; j > i; j--) { /\* 发生逆序，交换\*/

if(a[j - 1] > a[j]) {

tmp = a[j - 1];

a[j - 1] = a[j];

a[j] = tmp;

exchange = 1;

}

}

if(exchange == 0) return; /\* 本趟无逆序，停止处理\*/

}

}

若初始序列有序，则只需要一趟冒泡就可以结束，共需要n-1次比较和0次数据交换。若初始序列为逆序，则需要n-1次冒泡，总共的

元素比较次数

元素移动次数

因此，冒泡排序的时间复杂度为O(n2)。它是稳定的。

#### 7.3.2 快速排序

快速排序的基本思想是任取待排序元素序列中的某个元素（例如取第一个元素）作为基准，按照该元素的关键字大小，将整个元素序列划分为左右两个子序列：左侧子序列中所有元素的关键字都小于基准元素的关键字，右侧子序列中所有元素的关键字都大于或等于基准元素的关键字，基准元素则排在这两个子序列中间（这也是该元素最终应该安放的位置）。然后分别对这两个子序列重复施行上述算法，直到所有的元素都排在相应位置为止。

一趟快排的过程如图8.2(a)所示。整个快速排序的过程可递归，如图8.2(b)所示。

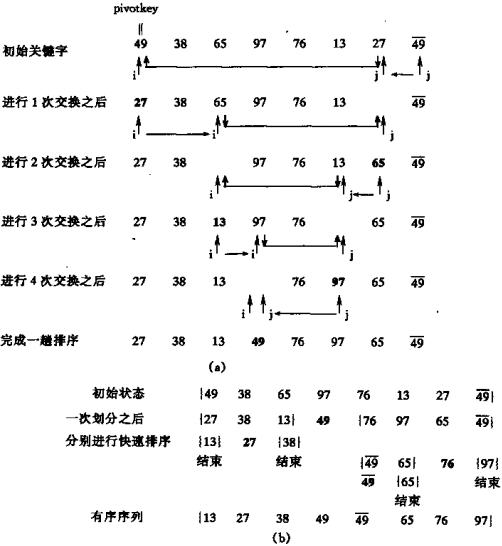


图8.2 快速排序示例

(a)一趟快排过程；(b)排序的全过程

快速排序的C语言实现如下。

**/\***

**\* @brief 一趟划分.**

**\* @param[inout] a 待排序元素序列**

**\* @param[in] start 开始位置**

**\* @param[in] end 结束位置，最后一个元素后一个位置**

**\* @return 基准元素的新位置**

**\* @note 无**

**\* @remarks 无**

**\*/**

**static int partition(data\_t a[], int start, int end)**

**{**

**int i = start;**

**int j = end - 1;**

**const data\_t pivot = a[i];**

**while(i < j) {**

**while(i < j && a[j] >= pivot) j--;**

**a[i] = a[j];**

**while(i < j && a[i] <= pivot) i++;**

**a[j] = a[i];**

**}**

**a[i] = pivot;**

**return i;**

**}**

**/\***

**\* @brief 快速排序.**

**\* @param[inout] a 待排序元素序列**

**\* @param[in] start 开始位置**

**\* @param[in] end 结束位置，最后一个元素后一个位置**

**\* @return 无**

**\* @note 无**

**\* @remarks 无**

**\*/**

**static void quick\_sort(data\_t a[], int start, int end)**

**{**

**if(start < end - 1) { /\* 至少两个元素\*/**

**const int pivot\_pos = partition(a, start, end);**

**quick\_sort(a, start, pivot\_pos);**

**quick\_sort(a, pivot\_pos + 1, end);**

**}**

**}**

快速排序的平均计算时间为O(nlog2n)，由于需要递归栈，存储开销为O(log2n)。在最坏的情况下，即元素序列已经有序时，快速排序退化为冒泡排序。

实验表明，在所有同数量级（O(nlog2n)）的排序方法中，快速排序的平均性能最好。

快速排序是不稳定的。

### 7.4 选择排序

选择排序的基本思想是：每一趟在后面n-i(i=1, 2, ..., n-2)个元素中选取最小的元素作为有序序列的第i个元素。

#### 7.4.1 简单选择排序

简单选择排序也叫直接选择排序，其基本步骤是：

（1）在一组元素a[i]~a[n-1]中选择最小的元素；

（2）若它不是这组元素中的第一个元素，则将它与这组元素的第一个元素对调；

（3）在剩下的a[i+1]~a[n-1]中重复执行以上两步，直到剩余元素只有一个为止。

简单选择排序的C语言实现如下。

/\*

\* @brief 简单选择排序.

\* @param[inout] a 待排序元素序列

\* @param[in] start 开始位置

\* @param[in] end 结束位置，最后一个元素后一个位置

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void select\_sort(data\_t a[], int start, int end)

{

data\_t tmp;

int i, j, k;

for(i = start; i < end; i++) {

k = i;

/\* 在a[i]到a[end-1]中寻找最小元素\*/

for(j = i + 1; j < end; j++)

if(a[j] < a[k]) k = j;

/\* 交换\*/

if(k != i) {

tmp = a[i];

a[i] = a[k];

a[k]= tmp;

}

}

}

简单选择排序的比较次数与元素的初始排列无关，总是n(n-1)/2次，移动次数与初始排列有关，当元素有序排列时，移动次数为0，最坏情况是逆序时，每一趟都要进行交换，总的移动次数为3n(n-1)/2。

简单选择排序是不稳定的。

#### 7.4.2 堆排序

堆排序的C语言实现如下（在heap.h和heap.c文件中）。

/\*\*

\* @brief 堆排序.

\* @param[inout] a 待排序元素序列

\* @param[inout] elem\_size 单个元素大小

\* @param[in] n 元素个数

\* @param[in] cmp cmp 比较函数，小于返回-1，等于，

\* 大于

\* @return 无

\* @note 无

\* @remarks 无。

\*/

extern void heap\_sort(void \*a, int elem\_size, int n,

int (\*cmp)(const void\*, const void\*));

void heap\_sort(void \*a, int elem\_size, int n,

int (\*cmp)(const void\*, const void\*))

{

int i;

heap\_t h = {0, 0, 0, NULL, NULL};

void \*tmp = malloc(elem\_size);

if(tmp == NULL) return /\* ERR\_MALLOC\_FAILED \*/;

/\* 替代heap\_init() \*/

h.size = n;

h.elem\_size = elem\_size;

h.cmp = cmp;

/\* 替代heap\_create() \*/

h.data = a;

i = (h.size - 2)/2; /\* 找最初调整位置：最后分支结点\*/

while(i >= 0) { /\* 自底向上逐步扩大形成堆\*/

heap\_sift\_down(&h, i);

i--;

}

for(i = h.size - 1; i > 0; i--) {

/\* tmp = h.data[i]; \*/

memcpy(tmp, (char\*)h.data + i \* h.elem\_size, h.elem\_size);

/\* h.data[i] = h.data[0]; \*/

memcpy((char\*)h.data + i \* h.elem\_size, h.data,

h.elem\_size);

/\* h.data[0] = tmp; \*/

memcpy(h.data, tmp, h.elem\_size);

h.size = i; /\* 相当于h.size -- \*/

heap\_sift\_down(&h, 0);

}

free(tmp);

}

堆排序的时间复杂度为O(nlog2n)，最坏情况下也是O(nlog2n)，这是相对于快速排序最大的优点。堆排序是不稳定的。

### 7.5 归并排序

所谓“归并”，就是将两个或两个以上的有序序列合并成一个有序序列。考研大纲只要求二路归并排序。

图8.3是一个二路归并排序的例子。

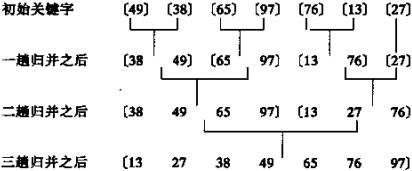


图8.3 二路归并排序示例

二路归并排序的C语言实现如下。

/\*

\* @brief 将两个有序表合并成一个新的有序表

\* @param[inout] a 待排序元素序列，包含两个有序表

\* @param[in] tmp 与a等长的辅助数组

\* @param[in] start a[start]~a[mid-1]为第一个有序表

\* @param[in] mid 分界点

\* @param[in] end a[mid]~a[end-1]为第二个有序表

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void merge(data\_t a[], data\_t tmp[], int start, int mid, int end)

{

int i, j, k;

for(i = 0; i < end; i++) tmp[i] = a[i];

/\* i, j是检测指针，k是存放指针\*/

for(i = start, j = mid, k = start; i < mid && j < end; k++) {

if(tmp[i] < tmp[j]) {

a[k] = tmp[i++];

} else {

a[k] = tmp[j++];

}

}

/\* 若第一个表未检测完，复制\*/

while(i < mid) a[k++] = tmp[i++];

/\* 若第二个表未检测完，复制\*/

while(j < end) a[k++] = tmp[j++];

}

/\*

\* @brief 归并排序.

\* @param[inout] a 待排序元素序列

\* @param[in] tmp 与a等长的辅助数组

\* @param[in] start 开始位置

\* @param[in] end 结束位置，最后一个元素后一个位置

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void merge\_sort(data\_t a[], data\_t tmp[], int start, int end)

{

if(start < end - 1) {

const int mid = (start + end) / 2;

merge\_sort(a, tmp, start, mid);

merge\_sort(a, tmp, mid, end);

merge(a, tmp, start, mid, end);

}

}

二路归并排序最好、最坏和平均时间复杂度都是O(nlog2n)。

归并排序的缺点是它需要一个与待排序数组一样大辅助数组空间。归并排序是稳定的。

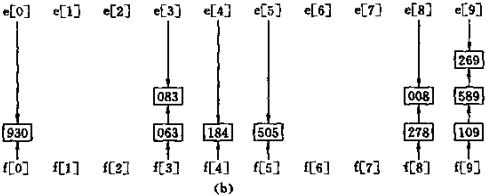
### 7.6 基数排序

利用多关键字实现对单关键字排序的算法就称为基数排序。

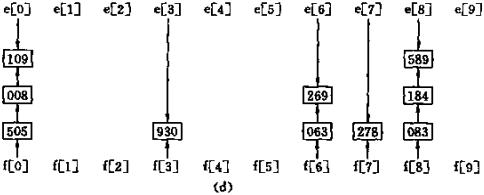
有两种顺序，最高位优先MSD(Most Significant Digit first)和最低位优先LSD(Least Significant Digit first)。

下面介绍“LSD链式基数排序”。首先以静态链表存储n个待排元素，并令表头指针指向第一个元素，即A[1]到A[n]存放元素，A[0]为表头结点，这样元素在重排时不必移动元素，只需要修改各个元素的link指针即可，如图8.4(a)所示。每个位设置一个桶（跟散列桶一样），桶采用静态链表结构，同时设置两个数组f[RADIX]和r[RADIX]，记录每个桶的头指针和尾指针。排序过程就是d（关键字位数）趟“分配”、“收集”的过程，如图8.4所示。

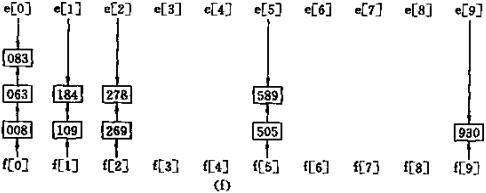
Snap6.jpg



Snap8.jpg



Snap10.jpg



Snap12.jpg

图8.4 LSD链式基数排序示例

(a)初始状态；(b)第一趟分配之后；(c)第一趟收集之后；

(d) 第二趟分配之后；(e)第二趟收集之后；

(f) 第三趟分配之后；(g) 第三趟收集之后的有序文件

LSD链式基数排序的C语言实现如下。

/\*\* @file radix\_sort.c

\* @brief LSD链式基数排序.

\* @author soulmachine@gmail.com

\* @date 2010-8-8

\* @version 0.1

\* @note 无

\*/

#include "config.h"

#include <stdio.h> /\* for printf() \*/

/\* 关键字基数，此时是十进制\*/

#define RADIX 10

/\*

\*@struct

\*@brief 静态链表结点.

\*@author soulmachine@gmail.com

\*@note 无

\*/

typedef struct static\_list\_node\_t {

int key; /\* 关键字\*/

int link; /\* 下一个节点\*/

}static\_list\_node\_t;

/\*

\* @brief 打印静态链表.

\* @param[in] a 静态链表数组

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void static\_list\_print(const static\_list\_node\_t a[])

{

int i = a[0].link;

while(i != 0) {

printf("%d ", a[i].key);

i = a[i].link;

}

}

/\*

\* @brief 获取十进制整数的某一位数字.

\* @param[in] n 整数

\* @param[in] i 第i位

\* @return 整数n第i位的数字

\* @note 无

\* @remarks 无

\*/

static int get\_digit(int n, int i)

{

int j;

for(j = 1; j < i; j++) {

n /= 10;

}

return n % 10;

}

/\*

\* @brief LSD链式基数排序.

\* @param[in] a 静态链表，a[0]是头指针

\* @param[in] n 待排序元素的个数

\* @param[in] d 最大整数的位数

\* @return 无

\* @note 无

\* @remarks 无

\*/

static void radix\_sort(static\_list\_node\_t a[], int n, int d)

{

int i, j, k, current, last;

int rear[RADIX], front[RADIX];

for(i = 0; i < n; i++) a[i].link = i + 1;

a[n].link = 0;

for(i = 0; i < d; i++) {

/\* 分配\*/

for(j = 0; j < RADIX; j++) front[j] = 0;

for(current = a[0].link; current != 0;

current = a[current].link) {

k = get\_digit(a[current].key, i + 1);

if(front[k] == 0) {

front[k] = current;

rear[k] = current;

} else {

a[rear[k]].link = current;

rear[k] = current;

}

}

/\* 收集\*/

j = 0;

while(front[j] == 0) j++;

a[0].link = current = front[j];

last = rear[j];

for(j = j + 1; j < RADIX; j++) {

if(front[j] != 0) {

a[last].link = front[j];

last = rear[j];

}

}

a[last].link = 0;

}

}

static void radix\_sort\_test(void)

{

static\_list\_node\_t a[] = {{0,0}/\* 头指针\*/, {278,0}, {109,0},

{63,0}, {930,0}, {589,0}, {184,0}, {505,0}, {269,0},

{8,0}, {83,0}};

radix\_sort(a, 10, 3);

static\_list\_print(a);

}

算法分析：对于有n个元素的链表，每趟进行“分配”的while循环需要执行n次，进行“收集”的for循环需要执行RADIX次，需要重复执行d趟“分配”和“收集”，所以总的时间复杂度是O(d(n+RADIX))。基数排序适合于关键字位数较少的情况。它是稳定的。

### 7.7 各种内部排序方法的比较

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 排序方法 | 平均时间 | 最坏情况 | 辅助存储 | 是否稳定 |
| 直接插入排序 | O(n2) | O(n2) | O(1) | 是 |
| 折半插入排序 | O(n2) | O(n2) | O(1) | 是 |
| 希尔排序 | —— | —— | O(1) | 否 |
| 冒泡排序 | O(n2) | O(n2) | O(1) | 是 |
| 快速排序 | O(nlog2n) | O(n2) | O(log2n) | 否 |
| 简单选择排序 | O(n2) | O(n2) | O(1) | 否 |
| 堆排序 | O(nlog2n) | O(nlog2n) | O(1) | 否 |
| 二路归并 | O(nlog2n) | O(nlog2n) | O(n) | 是 |
| 基数排序 | O(d(n+RADIX)) | O(d(n+RADIX)) | O(RADIX) | 是 |

## 参考资料

[1] 数据结构（C语言版），严蔚敏，吴伟民，清华大学出版社，2003

[2] 计算机学科专业基础综合考试大纲解析（2010年版），高等教育出版社，2009

由于时间仓促，错误和疏漏之处在所难免，欢迎414的同学指出错误。若

(1) 指出10个以上错误；或

(2) 添加5个以上知识点；

(3) 添加3个以上经典例题。

将会被载入~~史册~~感谢名单。

版本：2.1，最后更新日期：2010/12/02

官方地址：<http://www.gotothu.com> （未来清华人）

作者：戴方勤

个人博客：<http://www.yanjiuyanjiu.com/> （研究研究）

本手册遵循创作共享协议2.0，禁止一切商业用途。