数据结构 作业6

王华强 2016K8009929035

第6次作业---

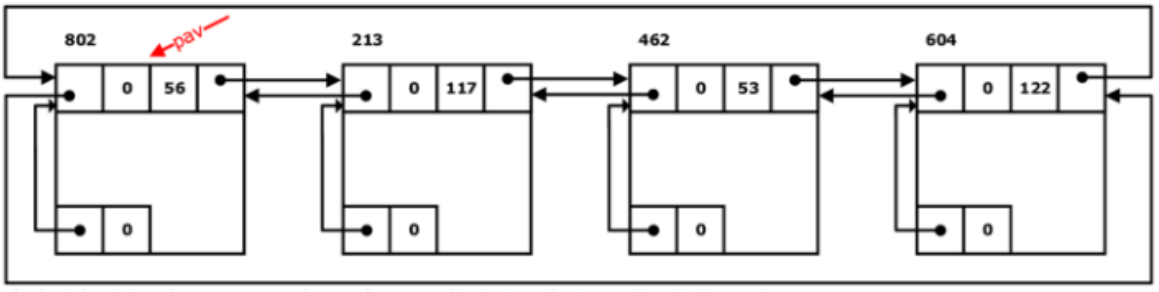
第8章动态存储管理：8.1, 8.7； 8.13, 8.14, 8.15

第9章查找： 9.1, 9.14, 9.19, 9.24； 9.29, 9.31, 9.33, 9.35, 9.38, 9.42, 9.43

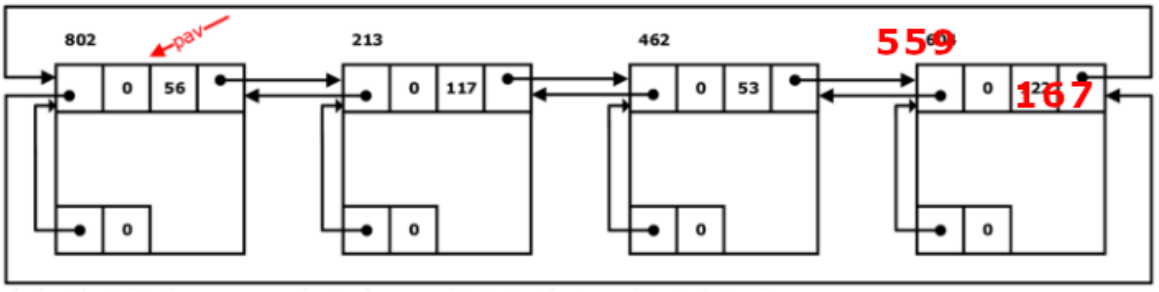
动态存储管理

8.1 假设利用边界标识法首次适配策略分配，已知在某个时刻的可利用空间表的状态如下图所示：

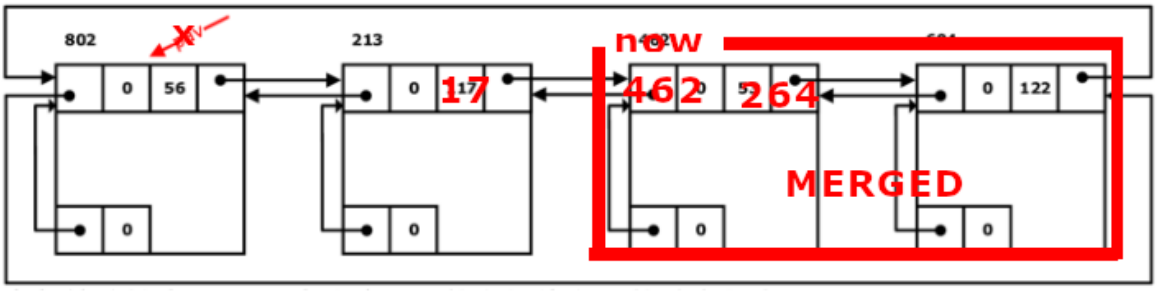
原始状态:



(1)画出当系统回收一个起始地址为559、大小为45的空闲块之后的链表状态；



(2)画出系统继而在接受存储块大小为100的请求之后，又回收一块起始地址为515、大小为44的空闲块之后的链表状态。



注意：存储块头部中大小域的值和申请分配的存储量均包括头和尾的存储空间。

8.7 已知一个大小为512字的内存，假设先后有6个用户提出大小分别为23，45，52，100，11和19的分配请求，此后大小为45，52和11的占用块顺序被释放。假设以伙伴系统实现动态存储管理，

(1)画出可利用空间表的初始状态；

2^9整块

(2)画出6个用户进入之后的链表状态以及每个用户所得存储块的起始地址；

用户	请求大小	分配大小	地址
1	23	2^5	0
2	45	2^6	64
3	52	2^6	128
4	100	2^7	256
5	11	2^4	32
6	19	2^5	192

有$2^{7,2}5,2^4$块剩余.

(3)画出在回收三个用户释放的存储块之后的链表状态。

有$2^{5,2}5,2^{6,2}6,2^7$块剩余.

8.13 试完成边界标志法和依首次适配策略进行分配相应的回收释放块的算法。

```
Space mfree(Space &memory)
{
    //case 1: the only block
    if(memory->last==memory)
    {
        memory->tag = 0;
        return memory;
    }

    //case last is used
    if(memory->last->tag)
    {

        //case next is used
        if(memory->next->tag)
        {//this only
            memory->tag = 0;
            return memory;
        }else
        //case last is unused
        {//this and last
            memory->last->size += memory->size;
            memory->last->next = memory->next;

            foot(memory)->tag = 0;
            foot(memory)->head= memory->last;

            memory->next->last = memory->last;
             memory=memory->last;
             return memory;
        }
    }else
    //case next is unused
    {
        if(memory->last->tag)
        //case last is used
        {//this and next merged
            memory->next->next->last = memory;

            memory->size += memory->next->size;
            memory->tag = 0;
            memory->next = memory->next->next;

            foot(memory)->tag = 0;
            foot(memory)->head = memory;

            return memory;
        }else
        //case last is unused
        {//this last and next is merged
            memory->last->size += memory->size;
            memory->last->size += memory->next->size;

            memory->last->next = memory->next->next;

            memory->last = memory->last->last;

            //next
            memory->next->next->last = memory->last;
            memory = memory->last;
            return memory;
        }
    }
    return nullptr;
}
```

8.14 试完成伙伴管理系统的存储回收算法。

```
#include<vector>

    struct buddy_node
{
    struct buddy_node* last;
    struct buddy_node* next;
    int tag;
    int size;
};

struct buddy_table
{
    int size;
    struct buddy_node* baseptr;
    vector<struct buddy_node *> list;
};

#include<algorithm>
#include<cmath>

int fastpow(int base, int index)
{
    if(!index)
        return 1;
    int sigindex = index & 1;
    int temp = base;
    while (index > 1)
    {
        index >>= 1;
        temp *= temp;
    }
    if(sigindex)
        return temp * base;
    return temp;
}

struct buddy_table init_buddy_system(int size)
{
    auto r = (struct buddy_node *)malloc(fastpow(2, size)*sizeof(buddy_node));
    if(r)
    {
        struct buddy_table result;
        result.size = size;
        while(size-->0)
        {
            result.list.push_back(nullptr);
        }
        result.list.push_back(r);
        //list[0] is useless and list[i] is size i;
        result.baseptr = r;
        return result;
    }
}

struct buddy_node *malloc_buddy(int size, struct buddy_table &buddy_table)
{
    //there is an error in the book's code, the inserting process for the remain block in ti the blank table is wror
    if(size>buddy_table.size)
        return nullptr;
    if(buddy_table.list[size])
    {
        auto a = buddy_table.list[size];
        if (buddy_table.list[size]->next == buddy_table.list[size])
            buddy_table.list[size] = nullptr;
        else
        {
```

```
            buddy_table.list[size]->last->next = buddy_table.list[size]->next;
            buddy_table.list[size]->next->last = buddy_table.list[size]->last;
            buddy_table.list[size] = buddy_table.list[size]->next;
        }
        return a;
    }
    if(size<buddy_table.size)
    {
        struct buddy_node *result;
        if (result=malloc_buddy(size, buddy_table))
        {
            //use the first half;
            int realsize = fastpow(2, size);

            result->size = realsize;
            result->tag = 1;

            auto n = result->next;
            result->next = result + realsize;
            result->next->next = n;
            result->next->last = result;
            result->next->size = size;
            result->next->tag = 0;

            //add the other half to the table;
            if(buddy_table.list[size])
            {
                result->next->next = buddy_table.list[size];
                result->next->last = buddy_table.list[size];
                buddy_table.list[size]->next = result;
            }
            else
            {
                buddy_table.list[size] = result->next;
                result->next->next = result->next;
            }

            return result;
        }
    }
    return nullptr;
}

struct buddy_node* merge_buddy(struct buddy_node* tgt, struct buddy_table buddy_table)
{
    //merge two node with the same size and tag == 0;
    //return the point of merged node;
    //only merge one level;
    //if failed, return nullptr;

    //find buddy node
    auto rsize = tgt - buddy_table.baseptr;
    struct buddy_node *buddy;
    auto p = rsize % fastpow(2, tgt->size + 1);
    if (p)
    {
        //this is upper half;
        buddy=tgt+fastpow(2,tgt->size);
    }
    else
    {
        //this is the back half
        buddy=tgt-fastpow(2,tgt->size);
    }

    if(tgt->tag||buddy->tag)
    {
```

```
        return nullptr;
    }

    //del tgt

    if (tgt->next == tgt)
    {
        buddy_table.list[tgt->size] = nullptr;
    }
    else
    {
        tgt->last->next = tgt->next;
        tgt->next->last = tgt->last;
    }

    //del buddy
    if(buddy->next == buddy)
    {
        buddy_table.list[buddy->size] = nullptr;
    }
    else
    {
        buddy->last->next = buddy->next;
        buddy->next->last = buddy->last;
    }

    //merge
    struct buddy_node *head = (struct buddy_node *)(p - rsize % fastpow(2, tgt->size + 1));
    head->tag = 0;
    head->size = tgt->size+1;

    //add merge result
    if(buddy_table.list[head->size])
    {
        head->last = buddy_table.list[head->size];
        head->next = buddy_table.list[head->size]->next;
        buddy_table.list[head->size]->next = head;
    }else
    {
        buddy_table.list[head->size] = head;
        head->next = head;
    }

    //return
    return head;
}

struct buddy_node* free_buddy(struct buddy_node* tgt, struct buddy_table& table)
{
    tgt->tag = 0;
    while(tgt=merge_buddy(tgt,table))
    {
        //loop;
    }
    return tgt;
}
```

8.15 设被管理空间的上下界地址分别由变量highbound和lowbound给出，形成一个由同样大小的块组成的“堆”。试写一个算法，将所有tag域的值为0的块按始址递增顺序链接成一个可利用空间表（设块大小域为cellsize）。

```
// void *highbound;
// void *lowbound;
// #define cellsize 1000
// #define headsize 2

struct block_head
{
    // int size;
    int tag;
    // struct block_head *last;//
    struct block_head *next;
}

int
algo_8_15(struct block_head *highbound, struct block_head *lowbound, int cellsize)
{
    struct block_head *before = 0;
    int i = 0;
    struct block_head *now = (struct block_head *)(lowbound + i * cellsize);
    while((int)now<highbound)
    {
        if(!now->tag)
        {
            now->last = before;
            before = now;
        }
        i++;
    }
    while(lowbound->tag&&lowbound<highbound)
    {
        lowbound += cellsize;
    }
    if(lowbound==highbound)
        return 0;
    return lowbound;
}
```

查找

9.1 若对大小均为n的有序的顺序表和无序的顺序表分别进行顺序查找，试在下列三种情况下分别讨论两者在等概率时的平均查找长度是否相同?

设顺序为升序.

(1)查找不成功，即表中没有关键字等于给定值K的记录;

- 若首元素<要查找的元素: 两者相同, 均要查找n个元素
- 若首元素>要查找的元素: 两者不同, 分别为1个元素, n个元素

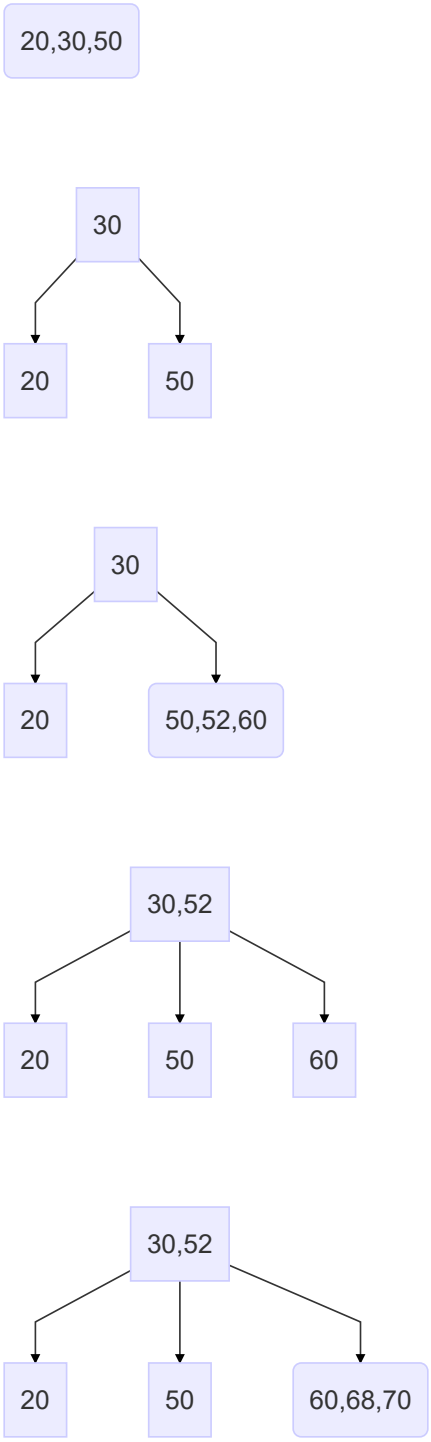
(2)查找成功，且表中只有一个关键字等于给定值K的记录;

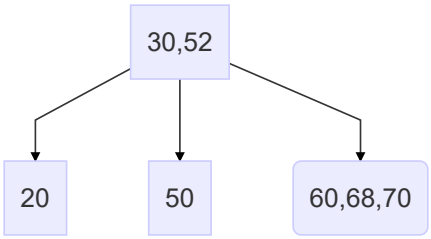
相同, 在有序表和无序表中关键字在各个位置的期望都相同, 由于是顺序查找, 故平均查找长度相同.

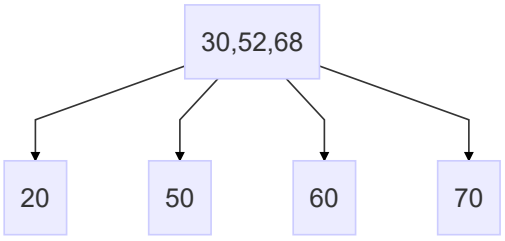
(3)查找成功，且表中有若干个关键字等于给定值K的记录，一次查找要求找出所有记录。此时的平均查找长度应考虑找到所有记录时所用的比较次数

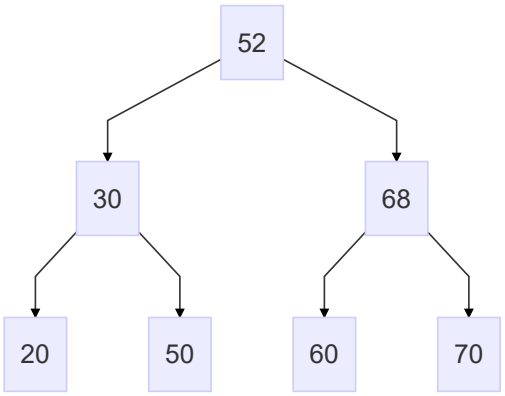
不相同. 无序表要求遍历所有元素(n), 有序表只要找到一个比待查找元素大的即可终止查找.

9.14 试从空树开始，画出按以下次序向2-3树即3阶B-树中插入关键码的建树过程：20,30,50,52,60,68,70。如果此后删除50和68，画出每一步执行后2-3树的状态

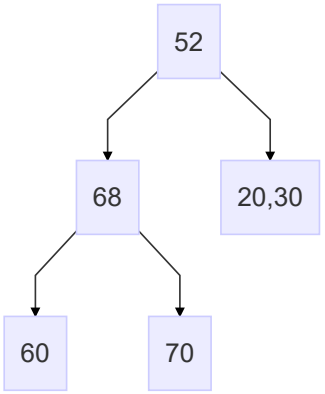


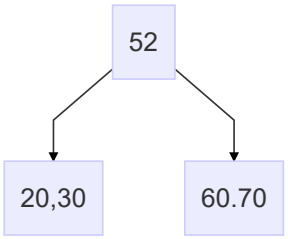

30,52
20
50
60,68,70


30,52,68
20
50
60
70


52
30
68
20
50
60
70

删除部分:


52
68
20,30
60
70


52
20,30
60.70

9.19 选取哈希函数H(k)=(3k) MOD 11。用开放定址法处理冲突，di=i((7k) MOD 10+1) (i=1,2,3, …)。试在0~10的散列地址空间中对关键字序列(22, 41, 53, 46, 30, 13, 01, 67)造哈希表，并求等概率情况下查找成功时的平均查找长度。

hash	key	length
0	22	1
1	30	2
2	41	1
3	01	1
4	13	3
5	53	1
6	46	1
7		
8		
9		
10	67	2

> 平均查找长度为1.5

9.24 某校学生学号由8位十进制数字组成:C1C2C3C4C5C6C7C8。C1C2为入学时年份的后两位；C3C4为系别：00~24分别代表该校的25个系；C5为0或1，0表示本科生，1表示研究生；C6C7C8为对某级某系某类学生的顺序编号：对于本科生，它不超过199，对于研究生，它不超过049，共有4个年级，四年级学生1996年入学。

(1)当在校生人数达极限情况时，将他们的学号散列到0~24999的地址空间，问装载因子是多少?

`4*25*(199+49)=24800`

装载因子为`24800/24999=0.827`

(2)求一个无冲突的哈希函数H1，它将在校生学号散列到。0~24999的地址空间。其簇聚性如何?

`H1=(C1C2-96)*(25*(199+49))+C3C4*(199+49)+C5*199+C6C7C8`

所有相同的类别聚集在同一个区域.

(3)设在校生总数为15000人，散列地址空间为,0~19999，你是否能找到一个(2)中要求的H1？若不能，试设计一个哈希函数H2及其解决冲突的方法，使得多数学号可只经一次散列得到（可设各系各年级本科生平均人数为130，研究生平均人数为20）。

`H2=(C1C2-96)*(25*150)+C3C4*(150)+C5*150+C6C7C8%(C5?20:150)`

使用rehash法处理冲突.

`H2rehash(i)=C1C2*C3C4*(C5+1)*C6C7C8%4999+15000+i`

不能, 散列地址空间至少要24800.

(4)用算法描述语言表达H2，并写出相应的查找函数。

`H2=(C1C2-96)*(25*150)+C3C4*(150)+C5*150+C6C7C8%(C5?20:150)`

```
int find(long int id)
{
    // int C1=id%100000000-id%10000000;
    // int C2=id%10000000-id%1000000;
    // int C3=id%1000000-id%100000;
    // int C4=id%100000-id%10000;
    int C5=id%10000-id%1000;
    // int C6=id%1000-id%100;
    // int C7=id%100-id%10;
    // int C8=id%10;
    int C1C2=id%100000000-id%1000000;
    int C3C4=id%100000000-id%10000;
    int C6C7C8=id%1000;
    int H2=(C1C2-96)*(25*150)+C3C4*(150)+C5*150+C6C7C8%(C5?20:150);
    if(hashhit(H2))
    {
        //return result;
    }
    int i=0;
    do
    {
        H2=C1C2*C3C4*(C5+1)*C6C7C8%4999+15000+i;
    }while(!hashhit(H2));
}
```

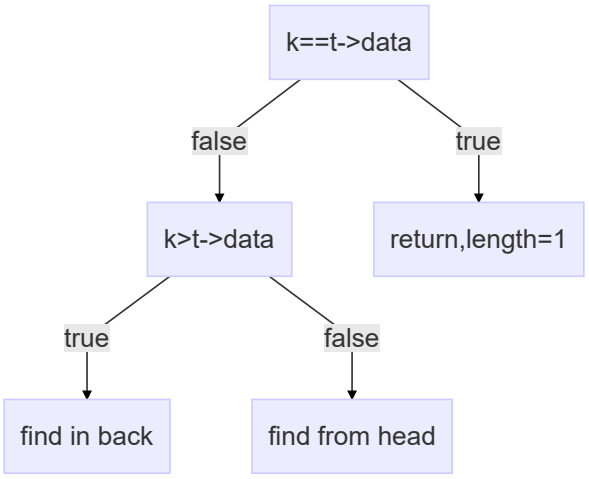
9.29 已知一非空有序表，表中记录按关键字递增排列，以不带头结点的单循环链表作存储结构，外设两个指针h和t，其中h始终指向关键字最小的结点，t则在表中浮动，其初始位置和h相同，在每次查找之后指向刚查到的结点。查找算法的策略是：首先将给定值K和t->key进行比较，若相等，则查找成功；否则因K小于或大于t->key而从h所指结点或t所指结点的后继结点起进行查找。

(1)按上述查找过程编写查找算法;

```
typedef struct node{
    int data;
    struct node* next;
}*pnode;

pnode find(int k, pnode h, pnode t)
{
    if(k==t->data)return k;
    if(k>t->data)
    {
        return find(k,h,t+1);
    }
    return find(k,h+1,k);
}
```

(2)画出描述此查找过程的判定树，并分析在等概率查找时查找成功的平均查找长度（假设表长为n，待查关键码K等于每个结点关键码的概率为1/n，每次查找都是成功的，因此在查找时，t指向每个结点的概率也为1/n）。



$$ASL(avesearchlength) = \frac{\sum_{i=0}^{n-1}(1+(n-i+1)(n-i)/2+(i)(i+1)/2)}{n}$$

9.31 试写一个判别给定二叉树是否为二叉排序树的算法，设此二叉树以二叉链表作存储结构。且树中结点的关键字均不同。

```
bool check(bintree* root, char type,int var)
{
    switch(type)
    {
        case '<':
            if(root->val>var)return false;
        break;

        case '>':
            if(root->val<var)return false;
        break;
    }
    if(root->left)if!(check(root->left, '<', root->val))return false;
    if(root->right)if(!check(root->right, '>', root->val))return false;
    return true;
}
```

9.33 编写递归算法，从大到小输出给定二叉排序树中所有关键字不小于x的数据元素。要求你的算法的时间复杂度为O(log2n+m)，其中n为排序树中所含结点数，m为输出的关键字个数。

```
int print_maxtomin(bintree* root, int x)
{
    if(root->right)print_maxtomin(root->right);
    if(root->val>=x)
    {
        printf("%d\n",root->val);
        if(root->left)print_maxtomin(root->left);
    }
}

int print_nolessthan(bintree* root, int x)
{
    while(root!-0&&root<x)root=root->right;
    if(root==0)return 0;
    print_maxtomin(root,x);
    return 0;
}
```

9.35 假设二叉排序树以后继线索链表作存储结构，编写输出该二叉排序树中所有大于a且小于b的关键字的算法

```
int algo_9_35(bintree* root, int a, int b)
{
    //find a
    while(root->val>a)
    {
        if(root->left)
        {
            root=root->left;
        }
        else
        {
            break;
        }
    }
    while(root->val<b)
    {
        printf("%d\n",root->val);
        root=root->next;
    }
    return 0;
}
```

9.38 试写一算法，将两棵二叉排序树合并为一棵二叉排序树。

```
int insert_bintree(bintree* &root, int val)
{
    if(root)
    {
        if(root->val==val)return 0;
        if(root->val>val)return insert_bintree(root->left, val);
        if(root->val<val)return insert_bintree(root->right, val);
    }
    root=new struct sbintree;
    root->val=val;
    root->left=0;
    root->right=0;
}
```

9.42 假设Trie树上叶子结点的最大层次为h，同义词放在同一叶子结点中，试写在Trie树中插入一个关键字的算法。

伪代码实现:

使用的函数基于search.cc

```
int add_to_trie(trietree* t, int h, char* word)
{
    char* head=word;
    while(*word!=0&&h>0)
    {
        if(t->exist(*word))
        {
            t=t->next(*word);
            h--;
        }
    }
    {
        t->addword(head);
    }
}
```

9.43 同9.42的假设，试写在Trie树中删除一个关键字的算法。

伪代码实现:

```
int del_in_trie(trietree* &t, int h, char* word, char* wordhead)
{
    while(*word!=0&&h>0)
    {
        if(t->exist(*word))
        {
            if(del_in_trie(t,h-1,word+1,wordhead))
            {
                t->delword(wordhead);
                if(t->empty())
                {
                    delete t;
                    t=0;
                    return 1;
                }
            }
        }
    }
    if(!*word||h==0)
    {
        t->delword(wordhead);
        if(t->empty())
        {
            delete t;
            t=0;
            return 1;
        }
    }
    return 0;
}
```