11.3.1开放地址法

常用处理冲突的思路

1. 换个位置：开放地址法
2. 同一位置的冲突对象组织在一起：链地址法

开放地址法（ $Open\ Addressing$ ）

一旦产生了冲突，就按照某种规则去寻找另一空地址

1. 若发生了第 i次冲突，试探的下一个地址将增加 d_i，基本公式是 $h_i(key)=(h(key)+d_i)\ mod\ TableSize$，其中 $1\leq i<TableSize$
2. d_i决定了不同的解决冲突方案：线性探测，平方探测，双散列
    1. $d_i=i$
    2. $d_i=\pm i^2$
    3. $d_i=i*h_2(key)$

11.3.2线性探测法

1. 线性探测法（ $LinearProbing$ ）
    1. 形成聚集现象

散列表查找性能分析

1. 成功平均查找长度（ $ASLs$ ）
    1. 冲突次数加1除以元素个数
2. 不成功平均查找长度（ $ASLu$ ）
    1. 看余数要比较几次
    2. 除以的是 mod的数

$TableSize$和要 mod的数可以不同，具体体现在不成功平均查找长度的计算上

11.3.3线性探测-字符串的例子

11.3.4平方探测法

2. 平方探测法（二次探测）

避免了聚集现象，或者说减弱了聚集现象

有定理显示：如果散列表长度 $TableSize$是某个 $4k+3$形式的素数时，平方探测法就可以探查到整个散列表空间

11.3.5平方探测法的实现

```
typedef int Position;
typedef int ElementType;
typedef struct HashTbl *HashTable;
struct HashTbl
{
    int TableSize;
    Cell *TheCells;
}H;

HashTable InitializeTable(int TableSize)
{
    HashTable H;
    int i;
    if(TableSize<MinTableSize)
    {
        ERROR("散列表太小");
        return NULL;
    }
    /* 分配散列表 */
    H=(HashTable)malloc(sizeof(struct HashTbl));
    if(H==NULL)
        FatalError("空间溢出！！");
    H->TableSize=NextPrime(TableSize);
    /* 分配散列表Cells */
    H->TheCells=(Cell *)malloc(sizeof(Cell)*H->TableSize);
    if(H->TheCells==NULL)
        FatalError("空间溢出！！");
    for(i=0;i<H->TableSize;i++)
        H->TheCells[i].Info=Empty;
    return H;
}

Position Find(ElementType Key,HashTable H)  /* 平方探测 */
{
    Position CurrentPos,NewPos;
    int CNum;  /* 记录冲突次数 */
    CNum=0;
    NewPos=CurrentPos=Hash(Key,H->TableSize);
    while(H->TheCells[NewPos].Info!=Empty&&
            H->TheCells[NewPos].Element!=Key)  /* 字符串类型的关键词需要strcmp函数 */
    {
        if(++CNum%2)  /* 判断冲突的奇偶次 */
        {
            NewPos=CurrentPos+(CNum+1)/2*(CNum+1)/2;
            while(NewPos>=H->TableSize)
                NewPos-=H->TableSize;
        }
        else
        {
            NewPos=CurrentPos-CNum/2*CNum/2;
            while(NewPos<0)
                NewPos+=H->TableSize;
        }
    }
    return NewPos;
}

void Insert(ElementType Key,HashTable H)
{  /* 插入操作 */
    Position Pos;
    Pos=Find(Key,H);
    if(H->TheCells[Pos].Info!=Legitimate)
    {  /* 确认在此插入 */
        H->TheCells[Pos].Info=Legitimate;
        H->TheCells[Pos].Element=Key;
        /* 字符串类型的关键词需要strcpy函数 */
    }
}
```

懒惰删除

3. 双散列探测法（ $Double\ Hashing$ ）
    1. 双散列探测法： d_i为 $i*h_2(key)$ ， $h_2(key)$为另一个散列函数
    2. 探测序列成： $h_2(key),2h_2(key),3h_2(key)\cdots$
    3. 对于任意的 key，都有 $h_2(key)\neq 0$
    4. 探测序列还应该保证所有的散列存储单元都应该能够被探测到，选择以下形式有良好的效果

        1. $h_2(key)=p-(key \bmod p)$
        2. 其中： $p<TableSize$,p、$TableSize$都是素数

4. 再散列（ $Rehashing$ ）
    1. 当散列表元素太多的时候（即装填因子 α太大 ），查找效率会下降
        1. 实用最大装填因子（ $0.5 \leq \alpha \leq 0.85$ ）
    2. 当装填因子过大时，解决的方法是加倍扩大散列表，这个过程叫做（再散列 $Rehashing$ ）

11.3.6分离链接法

分离链接法（ $Seperate\ Chaining$ ）

将相应位置上冲突的所有关键词都存储在同一个单链表中

```
typedef int ElementType;
typedef struct ListNode *Postion,*List;
struct ListNode
{
    ElementType Element;
    Postion Next;
};

typedef struct HashTbl *HashTable;
struct HashTbl
{
    int TableSize;
    List TheLists;
};

Postion Find(ElementType Key,HashTable H)
{
    Postion P;
    int Pos;

    Pos=Hash(Key,H->TableSize);
    P=H->TheLists[Pos].Next;
    while(P!=NULL&&strcmp(P->Element,Key))
        P=P->Next;
    return P;
}
```

创建开放定址法的散列表

```c
#define MAXTABLESIZE 100000 /* 允许开辟的最大散列表长度 */
typedef int ElementType;    /* 关键词类型用整型 */
typedef int Index;          /* 散列地址类型 */
typedef Index Position;     /* 数据所在位置与散列地址是同一类型 */
/* 散列单元状态类型，分别对应：有合法元素、空单元、有已删除元素 */
typedef enum { Legitimate, Empty, Deleted } EntryType;

typedef struct HashEntry Cell; /* 散列表单元类型 */
struct HashEntry{
    ElementType Data; /* 存放元素 */
    EntryType Info;   /* 单元状态 */
};

typedef struct TblNode *HashTable; /* 散列表类型 */
struct TblNode {   /* 散列表结点定义 */
    int TableSize; /* 表的最大长度 */
    Cell *Cells;   /* 存放散列单元数据的数组 */
};

int NextPrime( int N )
{ /* 返回大于N且不超过MAXTABLESIZE的最小素数 */
    int i, p = (N%2)? N+2 : N+1; /*从大于N的下一个奇数开始 */

    while( p <= MAXTABLESIZE ) {
        for( i=(int)sqrt(p); i>2; i-- )
            if ( !(p%i) ) break; /* p不是素数 */
        if ( i==2 ) break; /* for正常结束，说明p是素数 */
        else  p += 2; /* 否则试探下一个奇数 */
    }
    return p;
}

HashTable CreateTable( int TableSize )
{
    HashTable H;
    int i;

    H = (HashTable)malloc(sizeof(struct TblNode));
    /* 保证散列表最大长度是素数 */
    H->TableSize = NextPrime(TableSize);
    /* 声明单元数组 */
    H->Cells = (Cell *)malloc(H->TableSize*sizeof(Cell));
    /* 初始化单元状态为“空单元” */
    for( i=0; i<H->TableSize; i++ )
        H->Cells[i].Info = Empty;

    return H;
}
```

平方探测法的查找与插入

```
Position Find( HashTable H, ElementType Key )
{
    Position CurrentPos, NewPos;
    int CNum = 0; /* 记录冲突次数 */

    NewPos = CurrentPos = Hash( Key, H->TableSize ); /* 初始散列位置 */
    /* 当该位置的单元非空，并且不是要找的元素时，发生冲突 */
    while( H->Cells[NewPos].Info!=Empty && H->Cells[NewPos].Data!=Key ) {
                                        /* 字符串类型的关键词需要 strcmp 函数!! */
        /* 统计1次冲突，并判断奇偶次 */
        if( ++CNum%2 ){ /* 奇数次冲突 */
            NewPos = CurrentPos + (CNum+1)*(CNum+1)/4; /* 增量为+[(CNum+1)/2]^2 */
            if ( NewPos >= H->TableSize )
                NewPos = NewPos % H->TableSize; /* 调整为合法地址 */
        }
        else { /* 偶数次冲突 */
            NewPos = CurrentPos - CNum*CNum/4; /* 增量为-(CNum/2)^2 */
            while( NewPos < 0 )
                NewPos += H->TableSize; /* 调整为合法地址 */
        }
    }
    return NewPos; /* 此时NewPos或者是Key的位置，或者是一个空单元的位置（表示找不到）*/
}

bool Insert( HashTable H, ElementType Key )
{
    Position Pos = Find( H, Key ); /* 先检查Key是否已经存在 */

    if( H->Cells[Pos].Info != Legitimate ) { /* 如果这个单元没有被占，说明Key可以插入在此 */
        H->Cells[Pos].Info = Legitimate;
        H->Cells[Pos].Data = Key;
        /*字符串类型的关键词需要 strcpy 函数!! */
        return true;
    }
    else {
        printf("键值已存在");
        return false;
    }
}
```

分离链接法的散列表实现

```
#define KEYLENGTH 15                    /* 关键词字符串的最大长度 */
typedef char ElementType[KEYLENGTH+1]; /* 关键词类型用字符串 */
typedef int Index;                      /* 散列地址类型 */
/******** 以下是单链表的定义 ********/
typedef struct LNode *PtrToLNode;
struct LNode {
    ElementType Data;
    PtrToLNode Next;
};
typedef PtrToLNode Position;
typedef PtrToLNode List;
/******** 以上是单链表的定义 ********/

typedef struct TblNode *HashTable; /* 散列表类型 */
struct TblNode {   /* 散列表结点定义 */
    int TableSize; /* 表的最大长度 */
    List Heads;    /* 指向链表头结点的数组 */
};

HashTable CreateTable( int TableSize )
{
    HashTable H;
    int i;

    H = (HashTable)malloc(sizeof(struct TblNode));
    /* 保证散列表最大长度是素数，具体见代码5.3 */
    H->TableSize = NextPrime(TableSize);

    /* 以下分配链表头结点数组 */
    H->Heads = (List)malloc(H->TableSize*sizeof(struct LNode));
    /* 初始化表头结点 */
    for( i=0; i<H->TableSize; i++ ) {
         H->Heads[i].Data[0] = '\0';
         H->Heads[i].Next = NULL;
    }
```

```
    }

    return H;
}

Position Find( HashTable H, ElementType Key )
{
    Position P;
    Index Pos;

    Pos = Hash( Key, H->TableSize ); /* 初始散列位置 */
    P = H->Heads[Pos].Next; /* 从该链表的第1个结点开始 */
    /* 当未到表尾，并且Key未找到时 */
    while( P && strcmp(P->Data, Key) )
        P = P->Next;

    return P; /* 此时P或者指向找到的结点，或者为NULL */
}

bool Insert( HashTable H, ElementType Key )
{
    Position P, NewCell;
    Index Pos;

    P = Find( H, Key );
    if ( !P ) { /* 关键词未找到，可以插入 */
        NewCell = (Position)malloc(sizeof(struct LNode));
        strcpy(NewCell->Data, Key);
        Pos = Hash( Key, H->TableSize ); /* 初始散列位置 */
        /* 将NewCell插入为H->Heads[Pos]链表的第1个结点 */
        NewCell->Next = H->Heads[Pos].Next;
        H->Heads[Pos].Next = NewCell;
        return true;
    }
    else { /* 关键词已存在 */
        printf("键值已存在");
        return false;
    }
}

void DestroyTable( HashTable H )
{
    int i;
    Position P, Tmp;

    /* 释放每个链表的结点 */
    for( i=0; i<H->TableSize; i++ ) {
        P = H->Heads[i].Next;
        while( P ) {
            Tmp = P->Next;
            free( P );
            P = Tmp;
        }
    }
    free( H->Heads ); /* 释放头结点数组 */
    free( H );        /* 释放散列表结点 */
}
```