宏 的模板

哈夫曼

typedef struct HuffNode HTree;

struct HuffNode

{

    int depth;

    int value;

    int parent;

    int lchild, rchild;

    char name[105];

};

void EnHuff( int l, int r )

{

    Huff[Top].value = Huff[l].value + Huff[r].value ;

    Huff[r].parent = Top;

    Huff[l].parent = Top;

    Huff[Top].rchild = r;

    Huff[Top].lchild = l;

    Top++;

}

int GetHuff(int \*l, int \*r)

{

    int i, j;

    int af, bf;

    af = bf = -1;

    i = j = 0;

    for (i = 0; i < Top; i++)

    {

        if (Huff[i].parent == -1)

        {

            af = i;

            break;

        }

    }

    if (af == -1|| af == Top-1 ) return 0;

    for (; i < Top; i++)

    {

        if (Huff[i].parent == -1 && Huff[i].value < Huff[af].value)

            af = i;

    }

if (af == -1) return 0;

    for (j = 0; j < Top; j++)    {

        if (Huff[j].parent == -1 && j != af)    {

            bf = j;

            break;

        }

    }

    if (bf == -1)

        return 0;

    for (; j < Top; j++)    {

        if ( Huff[j].parent == -1 && Huff[j].value < Huff[bf].value && j != af )

            bf = j;

    }

    \*l = af;

    \*r = bf;

    return 1;

}

int Pushlength(int x, int n)

{

    int length = 0;

    for (int i = 0; i < n; i++)

        length += Huff[i].value;

    return length;

}

void HuffDepth(int x)

{

    if (x == -1) return;

    if (Huff[x].parent != -1)

        Huff[x].depth = Huff[Huff[x].parent].depth + 1;

    HuffDepth(Huff[x].lchild);

    HuffDepth(Huff[x].rchild);

}

图和dfs

void dfs(int s, int c) {

if (flag) return;

if (c == n) {

flag = 1;

return;

}

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

if (vis[i] == 0 && gra[s][i]) {

vis[i] = 1;

dfs(i, c + 1);

vis[i] = 0;

}

}

}

int check( )//欧拉回路中检验每点的边数奇偶性

{

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

if( degree[i] % 2 ) return 0;

return 1;

}：

最短路径

void ShortestDist(MGraph Graph, int dist[], int count[], Vertex S)

{

Vertex V = S;

Vertex sure[MaxVertexNum];

for (int i = 0; i < MaxVertexNum; i++) {

dist[i] = INFINITY;

sure[i] = 0;

count[i] = 0;

}

sure[S] = 1;

dist[S] = 0;

count[S] = 1;

while (V != -1) {

for (int i = 0; i < Graph->Nv; i++) {

if (Graph->G[V][i] != INFINITY) {

if (dist[i] > Graph->G[V][i] + dist[V]) {

dist[i] = dist[V] + Graph->G[V][i];

count[i] = count[V];

}

else if (dist[i] == Graph->G[V][i] + dist[V])

count[i] += count[V];

}

}

V = -1;

for (int i = 0; i < MaxVertexNum; i++) {

if (sure[i] == 0) {

if (V == -1 || dist[i] < dist[V]) {

V = i;

}

}

}

if (V != -1)

sure[V] = 1;

}

for (int i = 0; i < MaxVertexNum; i++) {

if (dist[i] == INFINITY) {

dist[i] = -1;

count[i] = 0;

}

}

}

哈希

**线性探测法的查找函数**

#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 \*/

};

Position Find( HashTable H, ElementType Key )

{

//typedef enum { Legitimate, Empty, Deleted } EntryType;

for( int i=0; i<H->TableSize ; i++)

{

if( H->Cells[i].Data == Key ) return i;

}

ElementType hk = Key % H->TableSize;

for( int i=0; i<H->TableSize ; i++ )

{

if( H->Cells[hk].Info == Empty )

return hk;

hk = (hk+1) % H->TableSize;

}

return ERROR;

}

**平方探测法 + 移位法**

// 移位法

int hs(int \*Key){

unsigned int aws = 0; //aws为返回的位置

while (\*Key)

aws = (aws << 5) + \*Key++; //移位

return aws;

}

//平方探测法

int find(int\* Key, int H){

int cNum = 0; // 探测奇点

int Newpos, Curpos = hs(Key) % H;

Newpos = Curpos;

while (vis[Newpos] == 1 && list[Newpos] != Key) ) {

if (++cNum % 2) // 偶数

{

Newpos = Curpos + (cNum + 1) / 2 \* (cNum + 1) / 2;

while (Newpos >= H)

Newpos -= H;

}

else {

Newpos = Curpos - (cNum ) / 2 \* (cNum ) / 2;

while (Newpos < 0)

Newpos += H;

}

}

return Newpos;

}

**分离链接法的删除操作函数**

typedef struct LNode\* PtrToLNode;

struct LNode {

ElementType Data;

PtrToLNode Next;

};

typedef PtrToLNode Position;

typedef PtrToLNode List;

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 \*/

};

bool Delete(HashTable H, ElementType Key){

struct LNode\* p, \* ptre;

int at = Hash(Key, H->TableSize);

p = H->Heads[at].Next;

if (p == NULL) return false;

if (strcmp(p->Data, Key) == 0) {

// printf("%s is deleted from list Heads[%d]\n", Key, at);

H->Heads[at].Next = p->Next;

return true;

}

ptre = p; p = p->Next;

while (p) {

if (strcmp(p->Data, Key) == 0)

break;

ptre = p;

p = p->Next;

}

if (p != NULL) {

// printf("%s is deleted from list Heads[%d]\n", p->Data, at);

ptre->Next = p->Next;

free(p);

return true;

}

return false;

}

字符串处理：

字符串加法：

string str\_add(const char s[], const char t[])

{

string ans;

int er, rr;

er = rr = 0;

int sl = strlen(s);

int tl = strlen(t);

int max = sl > tl ? sl : tl;

int k = 0;

while (max > 0)

{

sl--;

tl--;

max--;

int a = sl >= 0 ? (int)(s[sl] - '0') : 0;

int b = tl >= 0 ? (int)(t[tl] - '0') : 0;

k = a + b + er;

if (k > 9)

{

er = k / 10;

rr = k % 10;

}

else {

er = 0;

rr = k;

}

ans = (char)(rr + '0') + ans;

}

if (er > 0)

ans = (char)(er + '0') + ans;

return ans;

}

数字字符串比较：

int str\_compare(string s, string t)

{

if (s.size() < t.size())

return -1;

else if (s.size() > t.size())

return 1;

else

{

int i = 0, j = 0;

while (i < s.size() && j < t.size())

{

if (s[i] > t[j])

return 1;

else if (s[i] < t[j])

return -1;

else

i++; j++;

}

}

return 0;

}

非math开方：

float dsqrt(float x)

{

float down, up, mid;

int i = 1;

while (1)

{

if (i \* i > x)

{

up = i;

down = i - 1;

break;

}

i++;

}

for (int j = 1; j <= 100; j++)

{

mid = (down + up) / 2;

if ((mid \* mid - x) \* (up \* up - x) < 0) down = mid;

else up = mid;

}

return mid;

}