重庆大学本科学生实验项目任务书

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 实验题目 | 哈夫曼树及其应用 | | | |
| 实验时间 | 20210520 | 实验地点 | | DS1401 |
| 实验性质 | □验证性 √设计性 □综合性 | | | |
| 实验目的  1. 掌握哈夫曼树的定义及存储；  2. 掌握哈夫曼树的构造算法；  3. 掌握哈夫曼编码的生成算法。 | | | | |
| 实验内容：   1. 从终端输入若干字符，统计字符出现的频率，将字符出现的频率作为结点的权值，建立哈夫曼树，对各字符进行哈夫曼编码，然后对输入字符串进行编码和解码。 2. 要求输出：中序打印输出哈夫曼树；输出各字符的哈夫曼编码表；解码后的字符串   3. 最后提交完整的实验报告和源程序。 | | | | |
| 参考资料：   * + 《数据结构（ C 语言版）》，严蔚敏，吴伟民编著，清华大学出版社，2007年第1版 | | | | |
| 任务下达日期 2021 年 5月 10 日 | | | 完成日期 2021 年 5 月 20 日 | |

说明：学院、专业、年级均填全称，如：计算机学院、计算机科学与技术、2021

1. 实验代码

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\* 树的宏定义 \*/

#define STACK\_INIT\_SIZE 100 // 初始分配量

#define STACKINCREMENT 10 // 分配增量

/\* 栈的宏定义 \*/

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

/\* 状态码 \*/

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

typedef int Status;

typedef int TElemType;

typedef int SElemType;

/\* 栈的存储表示 \*/

typedef struct{

SElemType \*base; // 在栈构造之前和销毁之后, base的值为NULL

SElemType \*top; // 栈顶指针

int stacksize; // 当前已分配的存储空间, 以元素为单位

}SqStack;

/\* 初始化顺序栈 \*/

Status InitStack(SqStack \*S){

// 构造一个空栈S

S->base = (SElemType \* )malloc(STACK\_INIT\_SIZE \* sizeof(SElemType));

if (!S->base) exit(OVERFLOW); // 存储分配失败

S->top = S->base;

S->stacksize = STACK\_INIT\_SIZE;

return OK;

}// InitStack

/\* 判断栈S是否为空 \*/

Status StackEmpty(SqStack S){

if (S.base == S.top) return TRUE;

else return FALSE;

}// StackEmpty

/\* 取栈顶元素 \*/

Status GetTop(SqStack S, SElemType \*e){

// 若栈不空, 则用e返回S的栈顶元素, 并返回OK; 否则返回ERROR

if (S.top == S.base) return ERROR;

\*e = \*(S.top - 1);

return OK;

}// GetTop

/\* 压栈 \*/

Status Push(SqStack \*S, SElemType e){

// 插入元素e为新的栈顶元素

if ((\*S).top - (\*S).base >= (\*S).stacksize){ // 栈满, 追加存储空间

(\*S).base = (SElemType \* )realloc((\*S).base, ((\*S).stacksize + STACKINCREMENT) \* sizeof(SElemType));

if (!(\*S).base) exit(OVERFLOW); // 存储分配失败

(\*S).top = (\*S).base + (\*S).stacksize;

(\*S).stacksize += STACKINCREMENT;

}

\*(\*S).top++ = e;

return OK;

}// Push

/\* 弹栈 \*/

Status Pop(SqStack \*S, SElemType \*e){

// 若栈不空, 则删除S的栈顶元素, 用e返回其值, 并返回OK; 否则返回ERROR

if ((\*S).top == (\*S).base) return ERROR;

\*e = \*(--(\*S).top);

return OK;

}// Pop

// -------------以上为栈操作的定义------------------

/\* 哈夫曼树的存储表示 \*/

typedef struct{

unsigned int weight; // 结点权重

unsigned int parent; // 结点双亲

unsigned int lchild; // 结点左孩子

unsigned int rchild; // 结点右孩子

}HTNode, \*HuffmanTree; // 动态分配存储哈夫曼树

/\* 哈夫曼编码的存储表示 \*/

typedef char \* \* HuffmanCode; // 动态分配数组存储哈夫曼编码表

/\* 建立哈夫曼树过程中的选择操作 \*/

void Select(HuffmanTree HT, int i, int \*s1, int \*s2){

// 在HT[1..i-1]选择parent为0且weight最小的两个结点, 其序号分别为s1和s2.

// 结点s1权值小于等于s2权值

int min1, min2;

int k, j=0;

for (k=0; k<=i-1; ++k){

if (HT[k].parent == 0){ // 只在未接入哈夫曼树和新组成子树中进行选择

if (j == 0){ // 从起始结点开始, 假设其权值最小

min1 = HT[k].weight; \*s1 = k;

}// if

else if (j == 1){ // 第二结点次之, 并与第一个结点比较

if (HT[k].weight < min1){

min2 = min1; \*s2 = \*s1;

min1 = HT[k].weight; \*s1 =k;

}// if

else{

min2 = HT[k].weight; \*s2 = k;

}// else

}// else if

else{ // 剩下结点依次与s1和s2对应权值比较, 最小的赋给s1, 原s1成为s2

// 中间比s2小但比s1大或等于s1的结点成为s2

if (HT[k].weight < min1){

min2 = min1; \*s2 = \*s1;

min1 = HT[k].weight; \*s1 =k;

}// if

else if (HT[k].weight >= min1 && HT[k].weight < min2){

min2 = HT[k].weight; \*s2 = k;

}// else if

}// else

++j;

}// if

}// for

}// Select

/\* 哈夫曼编码的算法表示 \*/

void HuffmanCoding(HuffmanTree \*HT, HuffmanCode \*HC, int \*w, int n){

// w存放n个字符的权值(均>0), 构造哈夫曼树HT, 并求出n个字符的哈夫曼编码HC.

if (n <= 1) return;

int m = 2\*n-1; // m为结点总个数, 包括根节点和各个结点左右孩子

(\*HT) = (HuffmanTree)malloc(m \* sizeof(HTNode)); // 0号单元未用

HuffmanTree p = (\*HT);

int i; // i为对应数组下标, 仅在建立哈夫曼树过程中使用i

for (i=0; i<n; ++i, ++p, ++w){

// 0号单元未使用, 前n个字符对应结点存储在HT[1..n]中

// 这些结点都是哈夫曼树的叶子结点

p->weight = \*w;

p->parent = 0;

p->lchild = -1;

p->rchild = -1;

}// for

for (; i<m; ++i, ++p){

// 剩下结点起始权值均为0, 后作为双亲结点

// 最后一个结点为根节点

p->weight = 0;

p->parent = 0;

p->lchild = 0;

p->rchild = 0;

}// for

for (i=n; i<m; ++i){ // 建哈夫曼树

// 在HT[1..i-1]选择parent为0且weight最小的两个结点, 其序号分别为s1和s2.

int s1, s2;

Select((\*HT), i, &s1, &s2);

(\*HT)[s1].parent = i; (\*HT)[s2].parent = i;

(\*HT)[i].lchild = s1; (\*HT)[i].rchild = s2;

(\*HT)[i].weight = (\*HT)[s1].weight + (\*HT)[s2].weight;

}// for

// ---从叶子到根逆向求每个字符的哈夫曼编码---

(\*HC) = (HuffmanCode)malloc(n \* sizeof(char\*)); // 分配n个字符编码的头指针向量

char \*cd = (char \* )malloc(n \* sizeof(char)); // 分配求编码的工作空间

cd[n-1] = '\0'; // 编码结束符

int start, c, f; // 用作建立哈夫曼编码

for (i=0; i<n; ++i){ // 逐个字符求哈夫曼编码

start = n - 1; // 编码结束符位置

for (c=i, f=(\*HT)[i].parent; f!=0; c=f, f=(\*HT)[f].parent){ // 从叶子到根逆向求编码

if ((\*HT)[f].lchild == c) cd[--start] = '0';

else cd[--start] = '1';

}// for

(\*HC)[i] = (char \* )malloc((n-start) \* sizeof(char)); // 为第i个字符编码分配空间

strcpy((\*HC)[i], &cd[start]); // 从cd复制编码(串)到HC

}// for

free(cd); // 释放工作空间

}// HuffmanCoding

/\* 中序遍历二叉树 \*/

Status InOrderTraverse(HuffmanTree HT, int root){

// 采用哈夫曼树存储结构

// 中序遍历哈夫曼树HT的非递归算法, 对每个数据元素遍历, 此处遍历为打印

SqStack S;

InitStack(&S);

int i = root;

Push(&S, i); // 根指针进栈

while (!StackEmpty(S)){

while (i != -1){

GetTop(S, &i);

Push(&S, HT[i].lchild);

}// while

Pop(&S, &i);

if (!StackEmpty(S)){

Pop(&S, &i);

if (i == -1){

Push(&S, HT[i].rchild);

continue;

}// if

printf("%d, ", HT[i].weight);

Push(&S, HT[i].rchild);

}// if

}// while

}// InOrderTraverse

int main(){

// 从终端输入若干字符, 统计字符出现的频率, 将字符出现的频率作为结点的权值

// 建立哈夫曼树, 对各字符进行哈夫曼编码, 然后对输入字符串进行编码和解码

// 读取统计字符串

char s[100];

printf("请输入字符串:\n");

scanf("%s", &s);

int len = strlen(s); // len存储字符串的长度

// printf("%s ", s); printf("%d", len);

// 统计字符串中字符数目

int z[30]; // z用来统计字符数目

for (int i=0; i<30; ++i){

z[i] = 0;

}// for

for (int i=0; i<=len; ++i){ // 统计字符数目

z[s[i]-'a'+1]++;

}// for

int cnt = 0; // cnt为字符种类数目

for (int i=1; i<=26; ++i){

// printf("%d", z[i]);

if (z[i]) cnt++;

}// for

int x[len+1]; // x存储各字符出现次数, 即权值

int j = 0; char str[cnt]; // str存储出现的字符种类

for (int i=1; i<=26; ++i){

if(z[i]){

x[j] = z[i];

printf("%c : %d ", 'a'+i-1, z[i]);

str[j] = 'a'+i-1; // 将出现的字符种类存储在str中

j++;

}// if

}// for

printf("\n");

printf("\n你输入的字符种类个数分别为:\n");

for (int j=0; j<cnt; ++j){ // 打印x检查各权值是否正确

printf("%d", x[j]);

}

printf("\n");

// 建立哈夫曼树和初始化哈夫曼编码

HuffmanTree HT;

HuffmanCode HC;

HuffmanCoding(&HT, &HC, x, cnt);

// 打印哈夫曼树

printf("\n该哈夫曼树结构为:\n");

printf(" i weight parent lchild rchild \n");

for(int i=0; i<2\*cnt-1; ++i)

{

printf("%2d %5d %8d %8d %8d\n", i, HT[i].weight, HT[i].parent, HT[i].lchild, HT[i].rchild);

}// for

// 中序遍历打印哈夫曼树

printf("\n");

int root = 2 \* cnt - 2;

printf("\n中序遍历哈夫曼树为:\n");

InOrderTraverse(HT, root);

// 查询哈夫曼编码是否建立成功

printf("\n");

printf("\n各字符对应的编码为:\n");

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

printf("%c %s\n", str[i], HC[i]);

}// for

// 存储哈夫曼编码至一个数组中并打印

printf("\n");

printf("\n该字符串的哈夫曼编码为:\n");

char c[300]; // c用来存储读取字符串对应地哈夫曼编码

int index = 0; int index1;

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

for (int j=0; j<cnt; ++j){

if (s[i] == str[j]){ // 找到在存储字符种类数组中的位置

index1 = 0;

while (HC[j][index1] != '\0'){ // 哈夫曼编码中有对应的标识

c[index] = HC[j][index1]; // 将哈夫曼编码存储至c中

printf("%c", c[index]);

index++;

index1++;

}// while

}// if

}// for

}// for

// 打印出哈夫曼编码的译码

printf("\n");

int p;

int i = 0;

printf("\n该段哈夫曼编码的译码为:\n");

while(c[i] != '\0'){

p = 2\*cnt-2;

while (HT[p].lchild != -1 || HT[p].rchild != -1){

if (c[i] == '0') p = HT[p].lchild; // 左走

else p = HT[p].rchild; // 右走

++i;

}// while

printf("%c", str[p]);

}// while

printf("\n执行完毕.\n");

system("pause");

return 0;

}// main

1. 实验结果



