\*\*\*\*\*\*\*\*\*\*哈弗曼树（Huffman）实现\*\*\*\*\*\*\*\*\*\*

1. **堆的物理实现(heap.h)：**

// Heap class 堆类

template <typename E, typename Comp> class heap {

private:

E\* Heap; // Pointer to the heap array 堆数组指针

int maxsize; // Maximum size of the heap 最大容量

int n; // Number of elements now in the heap 元素当前位置

// Helper function to put element in its correct place 下拉函数

void siftdown(int pos) {

while (!isLeaf(pos)) { //Stop if pos is a leaf 是叶节点就停止下拉

int j = leftchild(pos);

int rc = rightchild(pos);

if ((rc < n) && Comp::prior(Heap[rc], Heap[j]))

j = rc; // Set j to greater child's value 将j赋值为较大子节点的值

if (Comp::prior(Heap[pos], Heap[j])) return;

swap(Heap, pos, j);

pos = j; // Move down 下拉

}

}

public:

heap(E\* h, int num, int max) // Constructor构造函数

{ Heap = h; n = num; maxsize = max; buildHeap(); }

int size() const // Return current heap size 返回当前堆大小

{ return n; }

bool isLeaf(int pos) const

{ return (pos >= n/2) && (pos < n); }

int leftchild(int pos) const

{ return 2\*pos + 1; } // Return leftchild position 返回左孩子位置

int rightchild(int pos) const

{ return 2\*pos + 2; } // Return rightchild position 返回右孩子位置

int parent(int pos) const // Return parent position 返回父结点位置

{ return (pos-1)/2; }

void buildHeap() // Heapify contents of Heap 建堆

{ for (int i=n/2-1; i>=0; i--) siftdown(i); }

// Insert "it" into the heap 将it插入堆

void insert(const E& it) {

Assert(n < maxsize, "Heap is full");

int curr = n++;

Heap[curr] = it; // Start at end of heap 先插在末尾

// Now sift up until curr's parent > curr 上移直到当前节点父结点大于当前节点

while ((curr!=0) &&

(Comp::prior(Heap[curr], Heap[parent(curr)]))) {

swap(Heap, curr, parent(curr));

curr = parent(curr);

}

}

// Remove first value 删除最大值

E removefirst() {

Assert (n > 0, "Heap is empty");

swap(Heap, 0, --n); // Swap first with last value 将最上面节点与最后一个节点交换

if (n != 0) siftdown(0); // Siftdown new root val 下拉根节点

return Heap[n]; // Return deleted value 返回删除节点

}

// Remove and return element at specified position 删除任意位置节点

E remove(int pos) {

Assert((pos >= 0) && (pos < n), "Bad position");

if (pos == (n-1)) n--; // Last element, no work to do 是最后一个节点则不处理

else

{

swap(Heap, pos, --n);

while ((pos != 0) &&

(Comp::prior(Heap[pos], Heap[parent(pos)]))) {

swap(Heap, pos, parent(pos));

pos = parent(pos);

}

if (n != 0) siftdown(pos);

}

return Heap[n];

}

};

1. **哈弗曼节点实现(huffnode.h)：**

// Huffman tree node abstract base class哈弗曼节点抽象类

template <typename E> class HuffNode {

public:

virtual ~HuffNode() {}

virtual int weight() = 0;

virtual bool isLeaf() = 0;

};

// Internal node subclass 内部节点子类

template <typename E> class IntlNode : public HuffNode<E> {

private:

HuffNode<E>\* lc; // Left child

HuffNode<E>\* rc; // Right child

int wgt; // Subtree weight

public:

IntlNode(HuffNode<E>\* l, HuffNode<E>\* r)

{ wgt = l->weight() + r->weight(); lc = l; rc = r; }

int weight() { return wgt; }

bool isLeaf() { return false; }

HuffNode<E>\* left() const { return lc; }

void setLeft(HuffNode<E>\* b)

{ lc = (HuffNode<E>\*)b; }

HuffNode<E>\* right() const { return rc; }

void setRight(HuffNode<E>\* b)

{ rc = (HuffNode<E>\*)b; }

};

// Leaf node subclass 叶节点子类

template <typename E> class LeafNode : public HuffNode<E> {

private:

E it; // Value

int wgt; // Weight

public:

LeafNode(const E& val, int freq) // Constructor构造函数

{ it = val; wgt = freq; }

int weight() { return wgt; }

E val() { return it; }

bool isLeaf() { return true; }

};

1. **哈弗曼树实现(hufftree.h)：**

template <typename E>

class HuffTree {

private:

HuffNode<E>\* Root; // Tree root 树根节点

public:

HuffTree(E& val, int freq) // Leaf constructor叶节点构造函数

{ Root = new LeafNode<E>(val, freq); }

// Internal node constructor内部节点构造函数

HuffTree(HuffTree<E>\* l, HuffTree<E>\* r)

{ Root = new IntlNode<E>(l->root(), r->root()); }

~HuffTree() {} // Destructor析构函数

HuffNode<E>\* root() { return Root; } // Get root 获得根节点

int weight() { return Root->weight(); } // Root weight 获得根权值

};

1. **哈弗曼编码表实现(codetable.h)：**

#define MAXCODELEN 20 // Max length of a huffman code 哈弗曼编码最大长度

#define CODETABLELEN 100 // Maximum number of codes storable 编码表最大长度

// CodeTable maps objects to their associated codes.编码表：将元素和编码一一对应

template <typename E>

class CodeTable {

private:

E\* obs;

char\*\* codes; // Associated code values 对应的编码值

int currsize; // Current number of objects in table 编码表当前元素个数

int maxsize; // Max objects permitted in table 表中最大容量

public:

CodeTable(int size) {

obs = new E[size];

codes = new char\*[size];

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

codes[i] = new char[MAXCODELEN+1];

for(int j=0; j<=MAXCODELEN; j++)

codes[i][j] = '-';

codes[i][MAXCODELEN] = '\0';

}

maxsize = size; currsize = 0;

}

void addobject(E& obj) {

Assert(currsize < maxsize, "CodeTable is full!");

obs[currsize++] = obj;

}

char\* getcode(E obj) {

for (int i=0; i<currsize; i++)

if(obj == obs[i]) return codes[i];

return NULL;

}

};

1. **常用函数类（book.h）：**

//book.h文件中定义了常用的swap、Assert函数

#include <iostream>

#include <cstdlib>

using std::cout;

using std::endl;

using std::string;

using std::ostream;

using namespace std;

const int defaultSize = 10;

//若val为0即断言错误，则输出字符串s并异常终止程序

void Assert(bool val, string s) {

if (!val) { // Assertion failed -- close the program

cout << "Assertion Failed: " << s << endl;

exit(-1);

}

}

// Swap two elements in a generic array 交换数组中两个元素

template<typename E>

inline void swap(E A[], int i, int j) {

E temp = A[i];

A[i] = A[j];

A[j] = temp;

}

1. **主函数测试类（huffman.cpp）:**

#include <cstring>

#include "book.h"

#include "heap.h"

#include "codetable.h"

#include "huffnode.h"

#include "hufftree.h"

//测试函数声明

void do\_commands(HuffTree<char>\* theTree,CodeTable<char>\* theTable, FILE \*fp);

int read\_freqs(CodeTable<char>\* ct, FILE\* fp);

void buildcode(HuffNode<char>\* root, CodeTable<char>\* ct,char\* prefix, int level, double& total);

void decode(HuffTree<char>\* theTree, char\* code, char& msg, int& cnt);

//"<<"运算符重载

template <typename E>

ostream& operator << (ostream& s, HuffNode<E>\* z)

{

if (z->isLeaf())

return s << ((LeafNode<E>\*)z)->val();

else

return s << z->weight();

}

// Comparator for the heap 比较两个堆根节点大小并返回布尔值

class minTreeComp {

public:

static bool prior(HuffTree<char>\* x, HuffTree<char>\* y)

{ return x->weight() < y->weight(); }

};

// Space for the heap's array 建立存储堆的数组

HuffTree<char>\*\* TreeArray = NULL;

// Build a Huffman tree from a collection of frequencies

//用已收集的频数构建哈弗曼树

template <typename E> HuffTree<E>\* buildHuff(HuffTree<E>\*\* TreeArray, int count) {

heap<HuffTree<E>\*,minTreeComp>\* forest =

new heap<HuffTree<E>\*, minTreeComp>(TreeArray,count, count);

HuffTree<char> \*temp1, \*temp2, \*temp3 = NULL;

while (forest->size() > 1) {

temp1 = forest->removefirst(); //Pull first two trees off the list

temp2 = forest->removefirst(); //取出最前面的两棵树

temp3 = new HuffTree<E>(temp1, temp2);//合并两棵树

forest->insert(temp3); // Put the new tree back on list 将新树加入列表

delete temp1; // Must delete the remnants of the trees we created

delete temp2; // 删除原来的两棵树

}

return temp3;

}

//主函数测试部分

int main() {

// This will be the eventual Huffman tree

//存储最终建成的哈弗曼树

HuffTree<char>\* theTree;

CodeTable<char>\* theTable = new CodeTable<char>(CODETABLELEN);

// Working storage for the tree traversal that builds the code table

//存储树遍历的值来构建编码表

char prefix[MAXCODELEN+1];//prefix 前缀

// total is used to calculate the average code length

double total = 0;

FILE \*fp; // The file pointer文件指针

Assert((fp = fopen("huffbook.huff", "rt")) != NULL, "No such file");

// Now, read in the list of frequencies, and initialize the forest of Huffman trees.

//读取频数文件列表，初始化哈弗曼树森林(即多个单结点）

cout << "读取文件中每个元素及其对应的频数\n";

int count = read\_freqs(theTable, fp);

// Now, build the tree.构建哈弗曼树

theTree = buildHuff<char>(TreeArray, count);

cout << "已构建哈弗曼树\n";

// Now, output the tree, which also creates the code table.

//输出已构建好编码表的哈弗曼树

cout << "输出哈弗曼树：\n";

buildcode(theTree->root(), theTable, prefix, 0, total);

cout << "平均编码长度为："

<< total/(double)theTree->weight() << "\n";

// Finally, do the encode/decode commands to test the system.

//最后，测试编码和解码命令

do\_commands(theTree, theTable, fp);

return 0;

}

//do\_commands命令函数执行decode(解码）、encode(编码)进行功能测试

void do\_commands(HuffTree<char>\* theTree,CodeTable<char>\* theTable, FILE \*fp)

{

int currchar;

char buff[80];

while(fgets(buff, 99, fp)) {

if(strncmp(buff, "decode", 6) == 0) {

for (currchar=0; buff[currchar] != '"'; currchar++);

cout << "解码：" << &buff[currchar++];

while (buff[currchar] != '"') {

int cnt = 0;

char msg;

decode(theTree, &buff[currchar], msg, cnt);

cout << msg << endl;

currchar += cnt;

}

}

else if(strncmp(buff, "encode", 6) == 0) {

for (currchar=0; buff[currchar] != '"'; currchar++);

cout << "编码：" << &buff[currchar++];

for(; buff[currchar] != '"' ; currchar++)

if (buff[currchar] == ' ') cout << ' ';

else cout << theTable->getcode(buff[currchar]);

}

}

}

void decode(HuffTree<char>\* theTree, char\* code, char& msg, int& cnt)

{

HuffNode<char>\* currnode = theTree->root();

while (!currnode->isLeaf()) {

cnt++;

if (code[cnt-1] == '0') currnode = ((IntlNode<char>\*)currnode)->left();

else if (code[cnt-1] == '1') currnode = ((IntlNode<char>\*)currnode)->right();

else Assert(false, "Bad code character");

}

msg = ((LeafNode<char>\*)currnode)->val();

}

// Read the list of frequencies, make the forest, and set the list of

//entries into the code table. 读取频数文件内容，初始化单个数据节点

int read\_freqs(CodeTable<char>\* ct, FILE\* fp)

{

// Read a list of strings and frequencies from standard input,

// building a list of Huffman coding tree nodes

//从标准输入读取一组字符和频数，并创建哈弗曼编码节点列表

char buff[100];

char buff2[100];

char \*ptr;

char \*ptr2;

int freq;

Assert(fgets(buff, 99, fp) != NULL, "Couldn't read character count");//读取元素总个数

ptr = buff;

Assert(isdigit(\*ptr) != 0, "Must be a digit here.");

int count = atoi(ptr);

TreeArray = new HuffTree<char>\*[count];

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

//文件中元素个数不够则断言终止

Assert(fgets(buff, 99, fp) != NULL, "Ran out of codes too early");

// process the entry, creating a new HuffTree

//处理每个入口，对读取到的字符创建一棵哈弗曼树

for(ptr=buff; \*ptr==' '; ptr++);

Assert(\*ptr == '"', "First char was not a quote mark.");

for (ptr2=buff2,ptr++; \*ptr!='"'; ptr++)

\*ptr2++ = \*ptr;

\*ptr2 = '\0'; // End of string 字符串结束标志

for (ptr++; \*ptr==' '; ptr++);

Assert(isdigit(\*ptr) != 0, "Must be a digit here.");

freq = atoi(ptr);

ct->addobject(buff2[0]);

TreeArray[i] = new HuffTree<char>(buff2[0], freq);

}

return count;

}

//建立哈弗曼编码

void buildcode(HuffNode<char>\* root, CodeTable<char>\* ct,char\* prefix, int level, double& total)

{

if (root->isLeaf()) {

cout << ((LeafNode<char>\*)root)->val() << "\t" << prefix << "\n";

strcpy(ct->getcode(((LeafNode<char>\*)root)->val()), prefix);

total += level \* root->weight();

}

else{

prefix[level] = '0';

prefix[level+1] = '\0';

buildcode(((IntlNode<char>\*)root)->left(), ct, prefix, level+1, total);

prefix[level] = '1';

prefix[level+1] = '\0';

buildcode(((IntlNode<char>\*)root)->right(), ct, prefix, level+1, total);

prefix[level] = '\0';

}

}

1. **程序所用测试文件内容（huffbook.huff）：**

8

"c" 32

"l" 42

"d" 42

"e" 120

"f" 24

"k" 7

"u" 37

"z" 2

decode "10100101"

decode "11111100111100111100"

decode "1011001110111101"

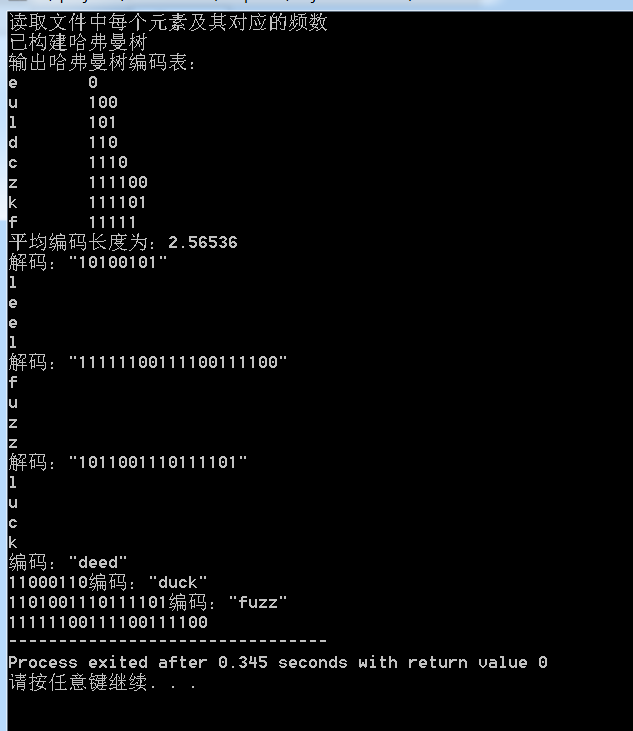
encode "deed"

encode "duck"

encode "fuzz"

encode "fuzz"

1. **测试结果示例：**



**【附录】**

版本信息声明：

Dev-C++ 5.11

TDM-GCC 4.9.2 64-bit Release

资料整理人：  
物联网工程1402班 宁静仪

物联网工程1402班 吴彦彦