****

**计算机体系结构实验报告**

|  |  |
| --- | --- |
| 学生姓名： | 谭哲文 |
| 学 号： | 8202191123 |
| 专业班级： | 计科2105班 |
| 指导教师： | 余腊生 |
| 学 院： | 计算机学院 |
| 完成时间： | 2023/12/31 |

**实验1 对指令操作码进行霍夫曼编码**

**一、实验目的**

了解和掌握指令编码的基本要求和基本原理。

**二、实验内容**

使用编程工具编写一个程序，对一组指令进行霍夫曼编码，并输出最后的编码结果以及对指令码的长度进行评价。与扩展操作码和等长编码进行比较。

**问题描述以及问题分析：**

我们举例说明此问题，例如：

有一组指令的操作码共分七类，它们出现概率如下表所示：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P4 | P5 | P6 | P7 |
| 0.45 | 0.30 | 0.15 | 0.05 | 0.03 | 0.01 | 0.01 |

对此组指令进行HUFFMAN编码正如下图所示：

0.01

0.01

0.03

0.05

0.15

0.30

0.45

0 1

0.02

0 1`

0.05

0 1

0.10

0 1

0.25

0 1

0.55

0 1

1.00

**图1**

最后得到的HUFFMAN编码如下表所示：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P4 | P5 | P6 | P7 |
| 0 | 10 | 110 | 1110 | 11110 | 111110 | 111111 |
| 1 | 2 | 3 | 4 | 5 | 6 | 6 |

最短编码长度为：

H=0.45\*1+0.30\*2+0.15\*3+0.05\*4+0.03\*5+0.01\*6+0.01\*6=-1.97.

要对指令的操作码进行HUFFMAN编码，只要根据指令的各类操作码的出现概率构造HUFFMAN树再进行HUFFAM编码。此过程的难点构造HUFFMAN树，进行HUFFAM编码只要对你所生成的HUFFMAN树进行中序遍历即可完成编码工作。

**三、设计描述**

依据实验内容需要设计一个用于构造Huffman编码的程序。首先输入指令集，接着构造Huffman树，然后根据Huffman树生成Huffman编码，最后输出Huffman编码并计算指令用Huffman编码的平均编码字长。可以使用了链表来存储指令信息，并使用多个函数来实现Huffman编码的构造和输出。

构建哈夫曼树的过程如下：

1. 将所有字符及其出现频率作为节点加入优先队列中。

2. 取出优先队列中频率最小的两个节点，将它们作为左右子树，新建一个节点，其权值为两个子节点的权值之和。

3. 将新建的节点加入优先队列中。

4. 重复步骤 2 和 3，直到优先队列中只剩下一个节点，这个节点就是哈夫曼树的根节点。

**具体分析：**

程序的主函数需要包括五个子模块：

* 输入指令集
* 构造Huffman树
* 生成Huffman编码
* 输出Huffman编码

计算指令用Huffman编码的平均编码字长

其中，输入指令集、构造Huffman树和生成Huffman编码三个子模块为主要的处理过程，输出Huffman编码和计算指令用Huffman编码的平均编码字长为辅助子模块。

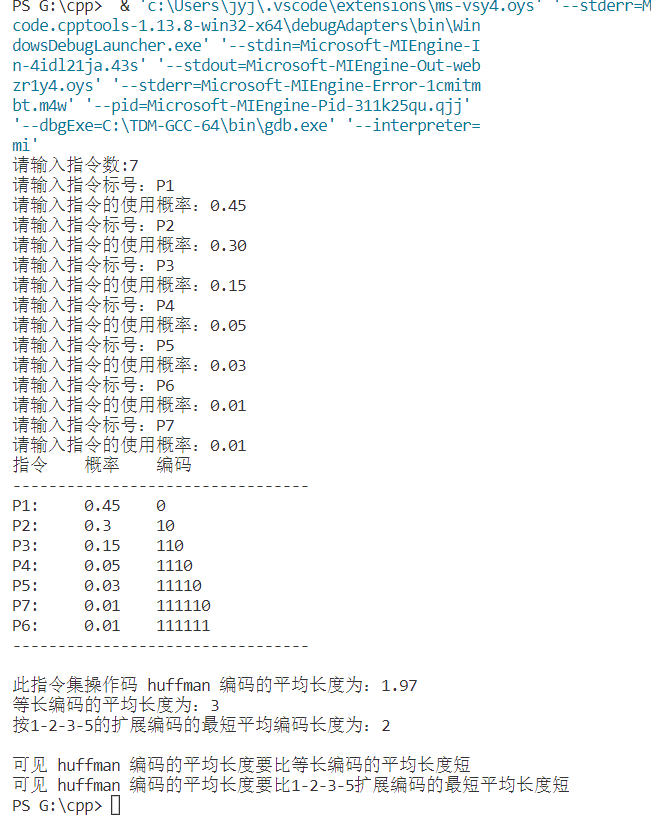
* 输入指令集：该模块主要输入指令的标号和指令使用的概率，并将指令按照概率大小从小到大排序，用于后续构建Huffman树的过程。
* 构造Huffman树：该模块根据输入的指令概率，构建Huffman树，并返回Huffman树的根节点。
* 生成Huffman编码：该模块根据Huffman树，递归生成Huffman编码。
* 输出Huffman编码：该模块输出生成的Huffman编码。模块中包含一个链表变量head，每个节点存储了一个指令的信息，包括指令标号，指令使用概率，以及该指令对应的Huffman编码。遍历链表，依次输出每个指令的信息即可。

**四、程序清单**

|  |
| --- |
| #include <iostream>  #include <cmath>  #include <cstring>  using namespace std;  const int N = 8; // huffman编码最大长度  class huff\_p  {  public:      huff\_p \*r\_child; // 大概率的节点      huff\_p \*l\_child; // 小概率的节点      char op\_mask[3]; // 指令标号      float p;         // 指令使用概率  };  class f\_min\_p  {  public:      f\_min\_p \*next;      char op\_mask[3]; // 指令标号      float p;         // 指令使用概率      huff\_p \*huf\_p;  };  // huff\_man code  class huff\_code  {  public:      huff\_code \*next;      float p;      char op\_mask[3];      char code[N]; // huffman 编码  };  f\_min\_p \*input\_instruct\_set();             // 输入指令集子模块；  huff\_p \*creat\_huffman\_tree(f\_min\_p \*head); // 构造huffman树;  f\_min\_p \*fin\_min(f\_min\_p \*h);  f\_min\_p \*del\_min(f\_min\_p \*h, f\_min\_p \*p);  void insert\_n(f\_min\_p \*h, f\_min\_p \*p);  huff\_p \*creat\_huffp(f\_min\_p \*p);  void creat\_huffman\_code(huff\_p \*h1, huff\_code \*h); // 生成 huffman 编码;  void r\_find(huff\_p \*p1, char code[], int i, huff\_code \*h);  void output\_huffman(huff\_code \*head);  // 输出huffman编码;  void cal\_sort\_length(huff\_code \*head); // 计算指令用huffman编码的平均编码字长  void print(huff\_p \*h1);  int main()  {      f\_min\_p \*h, \*h1;      huff\_p \*root;      huff\_code \*head, \*pl;      h = input\_instruct\_set();      h1 = h;      root = creat\_huffman\_tree(h1);      head = new huff\_code;      head->next = NULL;      creat\_huffman\_code(root, head);      output\_huffman(head);      cal\_sort\_length(head);      pl = head->next;      while (pl)      {          delete head;          head = pl;          pl = pl->next;      }      return 0;  }  f\_min\_p \*input\_instruct\_set()  {      f\_min\_p \*head;      f\_min\_p \*h;      h = new f\_min\_p;      h->next = NULL;      h->huf\_p = NULL;      head = h;      int n;      cout << "请输入指令数:";      cin >> n;      cout << "请输入指令标号：";      cin >> h->op\_mask;      cout << "请输入指令的使用概率：";      cin >> h->p;      f\_min\_p \*point;      f\_min\_p \*p1 = head;      for (int i = 0; i < n - 1; i++)      {          point = new f\_min\_p;          cout << "请输入指令标号：";          cin >> point->op\_mask;          point->op\_mask[2] = '\0';          cout << "请输入指令的使用概率：";          cin >> point->p;          point->huf\_p = NULL;          point->next = p1->next;          p1->next = point;          p1 = point;      }      return head;  }  huff\_p \*creat\_huffman\_tree(f\_min\_p \*h)  {      f\_min\_p \*h1, \*min1, \*min2, \*comb;      huff\_p \*head, \*rd, \*ld, \*parent;      h1 = h;      min1 = fin\_min(h1);      ld = creat\_huffp(min1);      h1 = del\_min(h1, min1);      if (h1->next)      {          min2 = fin\_min(h1);      }      else      {          min2 = h1;      }      rd = creat\_huffp(min2);      comb = new f\_min\_p;      comb->next = NULL;      comb->p = rd->p + ld->p;      comb->op\_mask[0] = '\0';      comb->op\_mask[1] = '\0';      parent = creat\_huffp(comb);      insert\_n(h1, comb);      if (h1->next != NULL)      {          h1 = del\_min(h1, min2);      }      parent->l\_child = ld;      parent->r\_child = rd;      comb->huf\_p = parent;      head = parent;      while (h1->next != NULL)      {          min1 = fin\_min(h1);          if (min1->huf\_p == NULL)          {              ld = creat\_huffp(min1);          }          else          {              ld = min1->huf\_p;          }          h1 = del\_min(h1, min1);          if (h1->next)          {              min2 = fin\_min(h1);          }          else          {              min2 = h1;          }          if (min2->huf\_p == NULL)          {              rd = creat\_huffp(min2);          }          else          {              rd = min2->huf\_p;          }          comb = new f\_min\_p;          comb->next = NULL;          comb->p = rd->p + ld->p;          comb->op\_mask[0] = '\0';          comb->op\_mask[1] = '\0';          parent = creat\_huffp(comb);          if (h1 != NULL)          {              insert\_n(h1, comb);          }          if (h1->next != NULL)          {              h1 = del\_min(h1, min2);          }          if (h1->next == NULL && ld->p < rd->p)          {              huff\_p \*tmp = ld;              ld = rd;              rd = tmp;          }          parent->l\_child = ld;          parent->r\_child = rd;          comb->huf\_p = parent;          head = parent;          if (h1->next == NULL)          {              break;          }      }      delete comb;      return head;  }  f\_min\_p \*fin\_min(f\_min\_p \*h)  {      f\_min\_p \*h1, \*p1;      h1 = h;      p1 = h1;      float min = h1->p;      h1 = h1->next;      while (h1)      {          if (min > (h1->p))          {              min = h1->p;              p1 = h1;          }          h1 = h1->next;      }      return p1;  }  f\_min\_p \*del\_min(f\_min\_p \*h, f\_min\_p \*p)  {      f\_min\_p \*p1, \*p2;      p1 = h;      p2 = h;      if (h == p)      {          h = h->next;          delete p;      }      else      {          while (p1->next != NULL)          {              p1 = p1->next;              if (p1 == p)              {                  p2->next = p1->next;                  delete p;                  break;              }              p2 = p1;          }      }      return h;  }  void insert\_n(f\_min\_p \*h, f\_min\_p \*p1)  {      p1->next = h->next;      h->next = p1;  }  huff\_p \*creat\_huffp(f\_min\_p \*d)  {      huff\_p \*p1;      p1 = new huff\_p;      p1->l\_child = NULL;      p1->r\_child = NULL;      p1->p = d->p;      p1->op\_mask[0] = d->op\_mask[0];      p1->op\_mask[1] = d->op\_mask[1];      return p1;  }  void r\_find(huff\_p \*p1, char code[], int i, huff\_code \*h)  {      if (p1->l\_child)      {          code[i] = '1';          r\_find(p1->l\_child, code, i + 1, h);      }      if (p1->op\_mask[0] != '\0')      {          huff\_code \*p2 = new huff\_code;          p2->op\_mask[0] = p1->op\_mask[0];          p2->op\_mask[1] = p1->op\_mask[1];          p1->op\_mask[2] = '\0';          p2->p = p1->p;          int j = 0;          for (; j < i; j++)          {              p2->code[j] = code[j];          }          p2->code[j] = '\0';          p2->next = h->next;          h->next = p2;      }      if (p1->r\_child)      {          code[i] = '0';          r\_find(p1->r\_child, code, i + 1, h);      }      delete p1;  }  void creat\_huffman\_code(huff\_p \*h1, huff\_code \*h)  {      int i = 0;      char code[N] = {'\0'};      r\_find(h1, code, i, h);  }  void output\_huffman(huff\_code \*head)  {      huff\_code \*h = head->next;      cout << "指令\t"           << "概率\t"           << "编码" << endl;      cout << "---------------------------------" << endl;      while (h)      {          h->op\_mask[2] = '\0';          cout << h->op\_mask << ":\t" << h->p << "\t" << h->code << endl;          h = h->next;      }      cout << "---------------------------------" << endl;      cout << endl;  }  void cal\_sort\_length(huff\_code \*head)  {      huff\_code \*h = head->next;      double j = 0;      float one\_length = 0;      float per\_length = 0;      float ext\_length = 0; // 按1-2-3-5扩展编码的最小长度为。      while (h)      {          float length = 0;          int i = 0;          while (h->code[i] != '\0')          {              length++;              i++;          }          one\_length = h->p \* length;          per\_length = per\_length + one\_length;          h = h->next;          j++;      }      int i1 = int(j);      huff\_code \*p2 = head->next;      float \*p\_a = new float[i1];      // sort指令概率      int i0 = 0;      while (p2)      {          p\_a[i0++] = p2->p;          p2 = p2->next;      }      float max, temp;      int l;      for (int s = 0; s < i1; s++)      {          max = p\_a[s];          l = s;          for (int k = s + 1; k < i1; k++)          {              if (max < p\_a[k])              {                  max = p\_a[k];                  l = k;              }          }          temp = p\_a[s];          p\_a[s] = max;          p\_a[l] = temp;      }      // 计算1-2-3-5扩展编码的最短平均长度      float \*code\_len = new float[i1];      code\_len[0] = 1;      code\_len[1] = 2;      code\_len[2] = 3;      code\_len[3] = 5;      for (int i = 4; i < j; i++)      {          code\_len[i] = 5;      }      l = 0;      while (l < i1)      {          ext\_length = ext\_length + code\_len[l] \* p\_a[l];          l++;      }      // 计算等长编码平均长度；      int q\_length = ceil(log10(j) / log10(2));      cout << "此指令集操作码 huffman 编码的平均长度为：" << per\_length << endl;      cout << "等长编码的平均长度为：" << q\_length << endl;      cout << "按1-2-3-5的扩展编码的最短平均编码长度为：" << ext\_length;      cout << endl;      cout << endl;      if (q\_length > per\_length)      {          cout << "可见 huffman 编码的平均长度要比等长编码的平均长度短" << endl;      }      else      {          cout << "huffman 编码有问题请仔细查看算法，以及输入的指令集的概率之和是否大于1。" << endl;      }      if (ext\_length > per\_length)      {          cout << "可见 huffman 编码的平均长度要比1-2-3-5扩展编码的最短平均长度短" << endl;      }      else      {          cout << "huffman 编码有问题请仔细查看算法，以及输入的指令集的概率之和是否大于1。" << endl;      }  } |

**五、结果分析**

**输出结果：**



**调试分析：**

按照预期得到的HUFFMAN编码如下表所示：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P4 | P5 | P6 | P7 |
| 0 | 10 | 110 | 1110 | 11110 | 111110 | 111111 |
| 1 | 2 | 3 | 4 | 5 | 6 | 6 |

最短编码长度为：

H=0.45\*1+0.30\*2+0.15\*3+0.05\*4+0.03\*5+0.01\*6+0.01\*6=-1.97.

经过结果比对，输出结果正确。

**六、实验心得**

这次实验的要求是使用Huffman编码的方法来压缩指令的编码长度。Huffman编码是一种编码方法，它根据每个指令的使用概率来确定每个指令的编码长度。指令使用较高的概率对应的编码长度较短，使用较低的概率对应的编码长度较长。这样，在指令流中使用较多的指令就可以用较短的编码表示，从而达到压缩的目的。

在本次实验中，我首先输入了指令集，然后利用指令的使用概率来构造Huffman树。接着利用Huffman树生成了Huffman编码，并输出了生成的编码。最后再统计指令流用Huffman编码的平均编码字长。实验结果可见，使用huffman树对字符串进行编码和解码后，可以大大减少存储空间，提高存储效率。

通过本次的实验，我对Huffman编码有了更加深入的理解，并且也学会了如何使用C++语言来实现这个算法。