Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий
Высшая школа программной инженерии

К у р с о в а я р а б о т а

Алгоритмы работы со словарями

по дисциплине “**Алгоритмы и структуры данных**”

Выполнил

студент гр. Набиуллин Данис Фирдусович ФИО

Руководитель Шемякин Илья Александрович ФИО

«___»_____2022

Санкт-Петербург
2022 г

Введение. Общая постановка задачи:

Тема: Методы разработки алгоритмов.

Вариант 3.1.
Коды Хаффмана.

Реализовать алгоритм кодирования и раскодирования текста по алгоритму Хаффмана.
Проанализировать эффективность сжатия для тестовых файлов.

Содержание

1 Основная часть работы

1. Описание алгоритма решения и используемых структур данных

В качестве основы программы была взята реализация Кодов Хаффмана из расчётно-графической работы курса ТП. Реализована таблица частотностей, позволяющая кодировать и декодировать файл. Добавлена таблица кодировки. собственная реализация очереди с приоритетом, реализация узлов в дереве Хаффмана.

2. Анализ алгоритмов

h – высота дерева

n – количество узлов

deleteHuffmanNode – O(h)

swap – O(1)

addPriorityQueue – O(n)

getPriorityQueue – O(1)

3. Описание спецификации программы (детальные требования)

Требование	Детальные требования	Данные	Ожидаемый результат
Пользователь, для выполнения кодирования и декодирования текста, должен использовать «file.txt»	Пользователь, для выполнения кодирования и декодирования текста, должен использовать «file.txt»	Lor abcdefghijklmnopqrstu vwxyzABCDEFGHIJK LMNOPQRSTUVWX YZ abcdefghijklmnopqrstu vwxyzABCDEFGHIJK LMNOPQRSTUVWX YZ. abcdefghijklm nopqrstuvwxyzABCDE FGHIJKLMNOPQRST UVWXYZ, abcdefghijklmnopqrstu vwxyz saf asdf sa dfasdfa sdf asd fasdgaf t sdfg s	Программа создаст файл “encodedFile.txt”, в котором будет записан закодированный текст, файл “frequencyTable_encodedFile.txt", в котором будут записаны частоты для каждого символа, файл “decodedFile.txt”, в котором будет записан декодированный текст

4. Описание программы (структура программы, форматы входных и выходных данных)

В качестве основы программы была взята реализация кодов Хаффмана из расчётно-графической работы курса АиСД и помещена в файл main.cpp. Был пересоздан ряд структур, алгоритмов и переименованы переменные. Были реализованы структуры PriorityQueue, PriorityQueueNode, HuffmanNode. Была реализована декодировка файла.

Файлы PriorityQueue.h и PriorityQueue.cpp содержат реализацию структур PriorityQueue и PriorityQueueNode.

Файлы HuffmanNode.h и HuffmanNode.cpp содержат реализацию структуры HuffmanNode.

Файл main.cpp содержит реализацию кодированию и декодирования текста.

Файл file.txt содержит исходный текст, который необходимо закодировать

Файл encodedFile.txt содержит закодированный текст

Файл frequencyTable_encodedFile.txt содержит таблицу частотностей для закодированного текста

Файл decodedFile.txt содержит раскодированный фай

Заключение

В ходе выполнения работы была «адаптирована» основа программы — коды Хаффмана из РГР по ТП, заменены контейнеры из библиотеки STL на собственноручно написанные.

Приложение. Текст программы

main.cpp

```
#include <climits>
#include <fstream>
#include <functional>
#include <iostream>
#include <sstream>
#include <string>
```

```cpp
#include "HuffmanNode.h"
#include "PriorityQueue.h"

namespace nabiullin
{
  std::string reverse(std::string const& s)
  {
    std::string rev(s.rbegin(), s.rend());
    return rev;
  }

  void fillFrequencyTable(unsigned short* frequencyTable, std::string data)
  {
    for (size_t i = 0; i < data.size(); i++)
    {
      frequencyTable[data[i]]++;
    }
  }

  void fillPriorityQueue(PriorityQueue* huffmanQueue, unsigned short* frequencyTable)
  {
    for (size_t i = 0; i < UCHAR_MAX; i++)
    {
      if (frequencyTable[i])
      {
        HuffmanNode* temp = new HuffmanNode(frequencyTable[i], i);
        addPriorityQueue(*huffmanQueue, temp, frequencyTable[i]);
      }
    }
    while (huffmanQueue->size_ > 1)
    {
      unsigned short priority = huffmanQueue->first_->priority_;
      priority += huffmanQueue->first_->next_->priority_;
      HuffmanNode* left_ = getPriorityQueue(*huffmanQueue);
      HuffmanNode* right_ = getPriorityQueue(*huffmanQueue);
      HuffmanNode* temp = new HuffmanNode(left_, right_);
      addPriorityQueue(*huffmanQueue, temp, priority);
    }
  }

  void fillEncodeTable(HuffmanNode* node, std::string* encodeTable, std::string& en-
codedChar, size_t& level)
  {
    if (node->value_ == '\0')
    {
      level++;
      encodedChar += "0";
      fillEncodeTable(node->left_, encodeTable, encodedChar, level);
      encodedChar += "1";
      fillEncodeTable(node->right_, encodeTable, encodedChar, level);
    }
    else
    {
      encodeTable[node->value_] = encodedChar;
    }
    if (encodedChar != "")
    {
      encodedChar.erase(encodedChar.end() - 1, encodedChar.end());
    }
    level--;
  }

  void printEncodedData(std::string* encodeTable, std::string data)
  {
    std::ofstream out("encodedFile.txt");
```

```
    std::string strEncodedCharacters = "";
    bool checker = false;
    for (size_t i = 0; i < data.size(); i++)
    {
      std::string ttt = data;
      unsigned char ctt = ttt[i];
      std::string encodedData = encodeTable[ctt];
      if (strEncodedCharacters.size())
      {
        if ((strEncodedCharacters + encodedData).size() > 8)
        {
          while (strEncodedCharacters.size() > 8)
          {
            unsigned char temp = std::stoi(strEncodedCharacters.substr(0, 8), nullptr,
2);
            out.write((char *)&temp, 1);
            strEncodedCharacters.erase(0, 8);
          }
          while (strEncodedCharacters.size() < 8 && !encodedData.empty())
          {
            strEncodedCharacters += encodedData[0];
            encodedData.erase(encodedData.begin());
          }
          unsigned char temp = std::stoi(strEncodedCharacters, nullptr, 2);
          out.write((char *)&temp, 1);
          checker = false;
          strEncodedCharacters = "";
        }
      }
      strEncodedCharacters += encodedData;
      checker = true;
    }
    if (checker)
    {
      unsigned char temp = std::stoi(strEncodedCharacters, nullptr, 2);
      out.write((char *)&temp, 1);
    }
    out.close();
  }

  void printDecodedData(HuffmanNode* node, std::string& data, std::ofstream& out)
  {
    if (node->value_ != '\0')
    {
      out << node->value_;
    }
    else if (data[0] == '0')
    {
      data.erase(data.begin(), data.begin() + 1);
      printDecodedData(node->left_, data, out);
    }
    else if (data[0] == '1')
    {
      data.erase(data.begin(), data.begin() + 1);
      printDecodedData(node->right_, data, out);
    }
  }

  unsigned short* encode(std::string data)
  {
    unsigned short* frequencyTable = new unsigned short[UCHAR_MAX];
    for (int i = 0; i < UCHAR_MAX; ++i)
    {
      frequencyTable[i] = '\0';
    }
```

```
    fillFrequencyTable(frequencyTable, data);
    PriorityQueue* huffmanQueue = new PriorityQueue[1];
    fillPriorityQueue(huffmanQueue, frequencyTable);
    std::string encodeTable[UCHAR_MAX];
    std::string encodedChar = "";
    size_t level = 0;
    fillEncodeTable(huffmanQueue->first_->val_, encodeTable, encodedChar, level);
    printEncodedData(encodeTable, data);
    delete[] huffmanQueue;
    return frequencyTable;
  }

  void decode(std::string data, unsigned short* frequencyTable)
  {
    PriorityQueue* huffmanQueue = new PriorityQueue[1];
    fillPriorityQueue(huffmanQueue, frequencyTable);
    std::ofstream out("decodedFile.txt");
    std::string encodedLine = "";
    for (size_t i = 0; i < data.size(); i++)
    {
      unsigned char temp = data[i];
      std::string strTemp = "";
      for (size_t i = 0; i < 8; i++)
      {
        if (temp % 2)
        {
          strTemp += '1';
        }
        else
        {
          strTemp += '0';
        }
        temp = temp >> 1;
      }
      strTemp = reverse(strTemp);
      encodedLine += strTemp;
    }
    while (encodedLine.size() > 1)
    {
      printDecodedData(huffmanQueue->first_->val_, encodedLine, out);
    }
    out.close();
    delete[] huffmanQueue;
  }
}

int main()
{
  std::string data;

  std::stringstream buffer;
  std::ifstream inEncode("file.txt");
  buffer << inEncode.rdbuf();
  inEncode.close();
  data = buffer.str();
  unsigned short* frequencyTable = nabiullin::encode(data);
  std::ofstream outFrequencyTable("frequencyTable_encodedFile.txt");
  for (size_t i = 0; i < UCHAR_MAX; i++)
  {
    outFrequencyTable << frequencyTable[i] << '\n';
  }
  outFrequencyTable.close();
  std::string encodedData;

  std::ifstream inDecode("encodedFile.txt");
```

```
  std::stringstream ss;
  ss << inDecode.rdbuf();
  inDecode.close();
  encodedData = ss.str();
  std::ifstream inFrequencyTable("frequencyTable_encodedFile.txt");
  for (size_t i = 0; i < UCHAR_MAX; i++)
  {
    inFrequencyTable >> frequencyTable[i];
  }
  inFrequencyTable.close();

  nabiullin::decode(encodedData, frequencyTable);
  delete[] frequencyTable;
  return 0;
}
```

PriorityQueue.h

```
#ifndef PRIORITY_QUEUE_H
#define PRIORITY_QUEUE_H

#include "HuffmanNode.h"

namespace nabiullin
{
  struct PriorityQueueNode
  {
    HuffmanNode* val_;
    unsigned short priority_;
    PriorityQueueNode* next_;
    PriorityQueueNode() = default;
    PriorityQueueNode(HuffmanNode* val, unsigned short priority, PriorityQueueNode*
next = nullptr) :
        val_(val), priority_(priority), next_(next)
    {}
    ~PriorityQueueNode(){
      delete val_;
    }
  };
  struct PriorityQueue
  {
    size_t size_;
    PriorityQueueNode* first_;
    PriorityQueue(size_t size = 0, PriorityQueueNode* first = nullptr) : size_(size),
first_(first) {}
    ~PriorityQueue();
  };
  void addPriorityQueue(PriorityQueue& queue, HuffmanNode* val, unsigned short prior-
ity);
  HuffmanNode* getPriorityQueue(PriorityQueue& queue);
}
#endif
```

PriorityQueue.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include "PriorityQueue.h"

namespace nabiullin
{
```

```
  PriorityQueue::~PriorityQueue()
  {
    while (this->first_)
    {
      PriorityQueueNode* tmp = this->first_->next_;
      delete this->first_;
      this->first_ = tmp;
    }
  }

  void addPriorityQueue(PriorityQueue& queue, HuffmanNode* val, unsigned short prior-
ity)
  {
    PriorityQueueNode* aux = new PriorityQueueNode(val, priority);
    if (queue.size_ == 0 || queue.first_ == nullptr)
    {
      aux->next_ = nullptr;
      queue.first_ = aux;
      queue.size_ = 1;
      return;
    }
    else
    {

      if (priority <= queue.first_->priority_)
      {
        aux->next_ = queue.first_;
        queue.first_ = aux;
        queue.size_++;
        return;
      }
      else
      {
        PriorityQueueNode* iterator = queue.first_;
        while (iterator->next_ != nullptr)
        {
          if (priority <= iterator->next_->priority_)
          {
            aux->next_ = iterator->next_;
            iterator->next_ = aux;
            queue.size_++;
            return;
          }
          iterator = iterator->next_;
        }
        if (iterator->next_ == nullptr)
        {
          aux->next_ = nullptr;
          iterator->next_ = aux;
          queue.size_++;
          return;
        }
      }
    }
  }

  HuffmanNode* getPriorityQueue(PriorityQueue& queue)
  {
    HuffmanNode* temp = new HuffmanNode(queue.first_->val_);
    if (queue.size_ > 0)
    {
      PriorityQueueNode* temp = queue.first_;
      queue.first_ = queue.first_->next_;
      queue.size_--;
      delete temp;
```

```
    }
    return temp;
  }
}
```

readyMadeTests.h

```
#ifndef _READY_MADE_TEST_H
#define _READY_MADE_TEST_H

void testDeleteNodes();
void testGetCountNodes();
void testIterativeSearch();
void testGetHeight();
void testInorderWalk();
void testGetCountKey();
void testPrintThreeMostCommonKeys();
void testExceptions(int count = 10, int range = 10);
template<typename T>
void printKey(T& key);

#endif
```

HuffmanNode.h

```
#ifndef HUFFMAN_NODE_H
#define HUFFMAN_NODE_H

#include <string>

namespace nabiullin
{
  struct HuffmanNode
  {
    size_t key_;
    unsigned char value_;
    HuffmanNode* left_;
    HuffmanNode* right_;
    HuffmanNode() = default;
    HuffmanNode(size_t key, char value, HuffmanNode* left = nullptr, HuffmanNode*
right = nullptr) :
        key_(key), value_(value), left_(left), right_(right)
    {}
    HuffmanNode(HuffmanNode* left, HuffmanNode* right)
    {
      value_ = '\0';
      left_ = left;
      right_ = right;
      key_ = left->key_ + right->key_;
    }
    HuffmanNode(HuffmanNode* src)
    {
      swap(*src);
    }
    ~HuffmanNode();
    void deleteHuffmanNode(HuffmanNode* node);
    void swap(HuffmanNode& src);
    HuffmanNode& operator=(HuffmanNode&& src) noexcept;
  };

  bool operator>(const HuffmanNode&, const HuffmanNode&);
  bool operator<(const HuffmanNode&, const HuffmanNode&);
}
#endif
```

HuffmanNode.cpp

```
#include <utility>
#include "HuffmanNode.h"

nabiullin::HuffmanNode::~HuffmanNode()
{
  deleteHuffmanNode(this);
}

void nabiullin::HuffmanNode::deleteHuffmanNode(HuffmanNode* node)
{
  if (node != nullptr)
  {
    deleteHuffmanNode(node->left_);
    deleteHuffmanNode(node->right_);
  }
}

nabiullin::HuffmanNode& nabiullin::HuffmanNode::operator=(HuffmanNode&& src) noexcept
{
  if (this != &src)
  {
    swap(src);
  }
  return *this;
}

void nabiullin::HuffmanNode::swap(HuffmanNode& src)
{
  key_ = src.key_;
  value_ = src.value_;
  std::swap(left_, src.left_);
  std::swap(right_, src.right_);
}

bool nabiullin::operator>(const HuffmanNode& first, const HuffmanNode& second)
{
  return first.key_ > second.key_;
}

bool nabiullin::operator<(const HuffmanNode& first, const HuffmanNode& second)
{
  return first.key_ < second.key_;
}
```