# 山东大学 计算机科学与技术 学院

## 数据结构与算法 课程实验报告

| 学号：201700140056 | 姓名：李港 | 班级：跟 18.2（17.4） |
|---|---|---|
| 实验题目：实验十 堆及其应用 | | |
| 实验学时：2h | 实验日期：2019.11.14 | |

**实验目的：**

1、掌握堆结构的定义、描述方法、操作定义及实现。
2、掌握堆结构的应用。
3、掌握霍夫曼树的建立以及计算霍夫曼长度的简便方法。

**软件开发工具：**

Virtual Studio 2019

**1. 实验内容**

1. 创建最小堆类。最小堆的存储结构使用数组。提供操作:插入、删除、初始化。
2. 接收键盘录入的一系列整数，以文本形式输出其对应的最小堆；
3. 对建立好的最小堆，键盘输入插入元素，输出插入操作完成后的堆（可以文本形式表示）；键盘输入删除元素，输出删除操作完成后的堆；
4. 键盘输入 n，随机生成 n 个 0~1000 之间的整数；输出堆排序的排序过程。
5. 键盘输入字符个数 n，以（c，w）形式依次字符和字符出现的频率，字符互不相同，输出 Huffman 树（可用文本形式）和每个字符的 Huffman 编码。

**2. 数据结构与算法描述（整体思路描述，所需要的数据结构与算法）**

**总体思路：**

1. 采用原生数组存储堆，采用之前实验中的二叉树结构存储霍夫曼树。
2. 采用最小堆和霍夫曼节点结构体参与来构建霍夫曼树
3. 最小堆提供 top、pop、push 等常用操作
4. 最小堆长度可自动扩展

**数据结构：**

1. 采用原生数组存储堆。
    1. 左子节点索引 = 当前节点索引*2
    2. 右子节点索引 = 当前节点索引*2+1
2. 采用之前实验中的二叉树结构存储霍夫曼树。

**算法：**

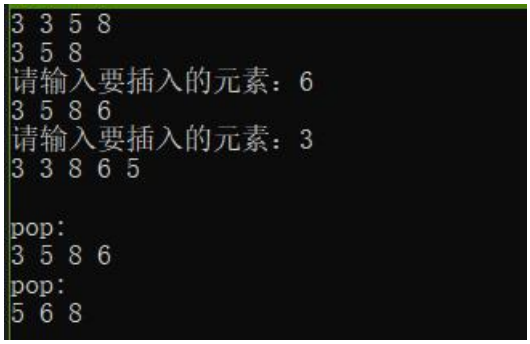
1. 构建霍夫曼树与计算霍夫曼长度采用不同方法：
    1. 构建霍夫曼树：
        1. 首先统计序列中各元素的出现频率，构建霍夫曼节点数组
        2. 然后以出现频率为关键词，将霍夫曼节点数组初始化为最小堆，注意这里允许重复
        3. 取出堆顶两元素，将频率求和，将两元素中包含的二叉树构建为新的二叉树，插入到堆中
        4. 循环元素种类数次，即完成二叉树的构建，返回堆顶元素中的数组，即为所求数组
    2. 计算霍夫曼长度：
        1. 计算霍夫曼长度与构建霍夫曼树采用不同方法

2. 首先使用构建频率数组，将元素作为索引，统计各元素出现次数
3. 使用最小堆，将频率数插入到最小堆中。
4. 然后不断取集合中全权值最小的两个频率值（也即堆顶两个），加和之后插入到堆中
5. 如此反复直到堆中只剩一个频率值，该频率值就是整个序列的霍夫曼编码长度。

2. 最小堆的操作
   1. 初始化 initialize
      1. 传入元素数组与数组长度，对每一个元素进行一次入堆操作，如此一来原先无序的元素便在最小堆中正确排列。
   2. 插入 push
      1. 首先判断缓冲区长度，缓冲区满则扩充长度
      2. 首先默认将新元素插入到数组末尾
      3. 然后依次对比该新元素与其父元素的大小，若新元素更小，则将两元素互换，直到新元素的父元素不小于新元素。
      4. 注意：无需判断新元素的兄弟元素，因为只要新元素比父元素小，则一定比兄弟元素小。
   3. 删除 pop
      1. 该操作比插入更复杂，因为需要考虑兄弟元素。
      2. 首先删除顶部元素，然后将堆底元素放置到堆顶
      3. 首先选出两子树中较小的元素（若没有右子树，则不必进行判断，直接选择左子树即可）
      4. 随后比较新根部元素与子树中较小者，若新根元素更小，则将两元素置换
      5. 继续进行上述操作，直到没有子树或根比子树小。
   4. 清空_clear
      1. 因为堆选用数组存储，只需删除整个数组并将长度置就可以清空堆。
   5. 长度扩展_extLength
      1. 在插入时判断堆元素数与缓冲区长度，若缓冲区长度不足，则申请两倍长的缓冲区，并复制原来的缓冲区内容到新缓冲区。

**3. 测试结果（测试输入，测试输出）**

1. 验收展示：
   1. 根据输入构建堆，并执行插入删除操作




   2. 随机初始化堆，图中连续出现的行代表一次插入动作的排序过程

```
请输入要初始化的元素个数：10
41

41 449

41 449 328

41 449 328 474

41 449 328 474 150
41 150 328 474 449

41 150 328 474 449 709

41 150 328 474 449 709 467

41 150 328 474 449 709 467 329
41 150 328 329 449 709 467 474

41 150 328 329 449 709 467 474 936

41 150 328 329 449 709 467 474 936 440
41 150 328 329 440 709 467 474 936 449
```

3. 构建霍夫曼树

输入的数据：

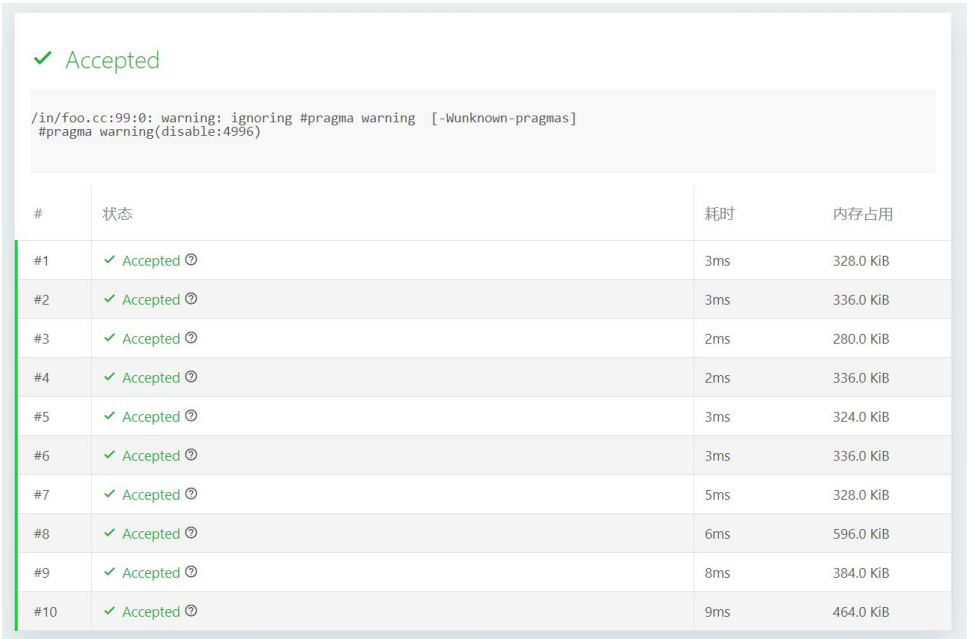
```
int ws[10] = {0, 9,9,9,9,9,9,9 };
char chars[10] = {'-', 'a','b','c','d','e' };
```

构建的霍夫曼树：

```
输出霍夫曼树结构：
      a e c   d b
输出霍夫曼编码长度：
12
```

2. 平台提交

1. 最小堆的插入删除：

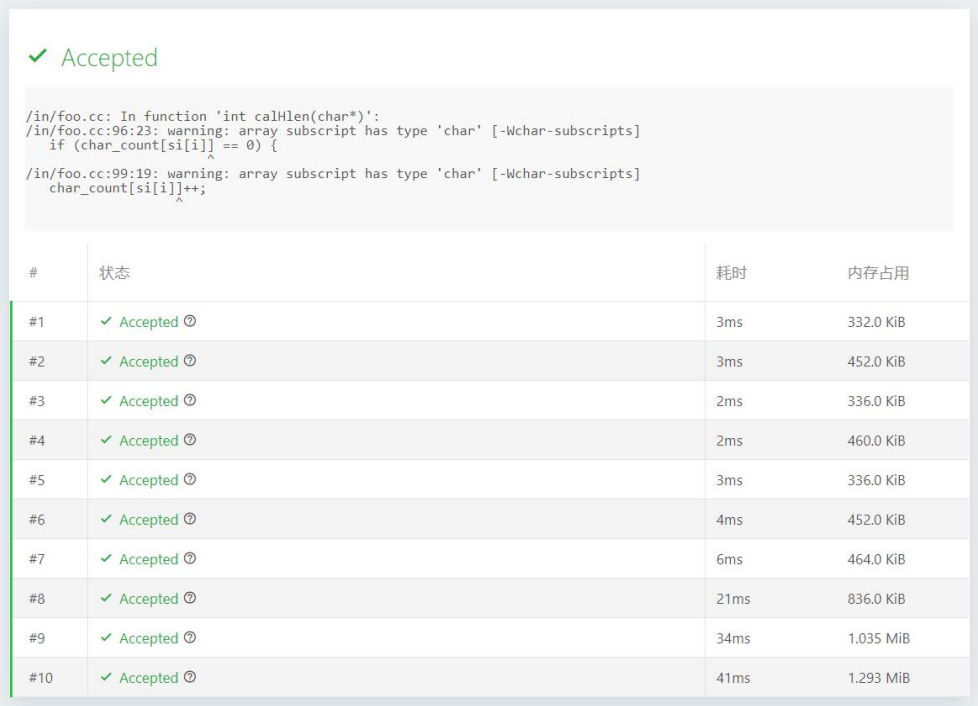


✓ Accepted

```
/in/foo.cc:99:0: warning: ignoring #pragma warning  [-Wunknown-pragmas]
 #pragma warning(disable:4996)
```

| # | 状态 | 耗时 | 内存占用 |
|---|---|---|---|
| #1 | ✓ Accepted | 3ms | 328.0 KiB |
| #2 | ✓ Accepted | 3ms | 336.0 KiB |
| #3 | ✓ Accepted | 2ms | 280.0 KiB |
| #4 | ✓ Accepted | 2ms | 336.0 KiB |
| #5 | ✓ Accepted | 3ms | 324.0 KiB |
| #6 | ✓ Accepted | 3ms | 336.0 KiB |
| #7 | ✓ Accepted | 5ms | 328.0 KiB |
| #8 | ✓ Accepted | 6ms | 596.0 KiB |
| #9 | ✓ Accepted | 8ms | 384.0 KiB |
| #10 | ✓ Accepted | 9ms | 464.0 KiB |

2. 最小堆的插入删除：




4. **分析与探讨（结果分析，若存在问题，探讨解决问题的途径）**

本实验最终结果正确，在实验过程中有以下问题或心得：

1. OJ 第二题一开始准备使用定长数组，后来发现题目中测试数据过大，造成数组越界，因此改用变长数组。
2. 第一题有个 sizes 写成了 deepthes，导致结果错误。
3. 后来在计算节点数时忘记在某个条件分支处加一，导致结果错误。
4. 根据题目要求适当地变化数据结构，或进行提前计算结果，可以极大提高算法效率，这也是算法竞赛做题的常用技巧。

5. **附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）**

文件 1 main.h

```
#include"huffManTree.h"
#include<iostream>
using namespace std;



/*1、创建最小堆类。最小堆的存储结构使用数组。提供操作:插入、删除、初始化。
2、接收键盘录入的一系列整数，以文本形式输出其对应的最小堆；
3、对建立好的最小堆，键盘输入插入元素，输出插入操作完成后的堆（可以文本形式表示）；键盘输入删除元素，输出删除操作完成后的堆；
4、键盘输入 n，随机生成 n 个 0~1000 之间的整数；输出堆排序的排序过程。
5、键盘输入字符个数 n，以（c，w）形式依次字符和字符出现的频率，字符互不相同，输出 Huffman 树（可用文本形式）和每个字符的 Huffman 编码。
*/
int main (void) {
    /*这里是 OJ 上第一个题目*/
#pragma warning(disable:4996)
    freopen ("input.txt", "r", stdin);

    minHeap<int> h;

    int size;
    cin >> size;
    for (int i = 0; i < size; i++) {
        int temp;
        cin >> temp;
```

```
        h.push (temp);
    }
    cout << h.top () << "\n";

    int times;
    cin >> times;
    for (int i = 0; i < times; i++) {
        int func_num;
        cin >> func_num;
        if (func_num == 1) {
            int temp;
            cin >> temp;
            h.push (temp);
            cout << h.top () << "\n";
        } else if (func_num == 2) {
            h.pop ();
            cout << h.top () << "\n";
        } else if (func_num == 3) {
            h._clearAndInit ();
            int size;
            cin >> size;
            for (int i = 0; i < size; i++) {
                int temp;
                cin >> temp;
                h.push (temp);
            }
            for (int i = 0; i < size; i++) {
                cout << h.top () << " ";
                h.pop ();
            }
        }
    }

    cout << "\n";

    /*这里是构建哈夫曼树*/
    cout << "输出霍夫曼树结构：\n";
    //首先空出数组的第一个元素
    int ws[10] = {0, 9,9,9,9,9,9,9 };
    char chars[10] = {'-', 'a','b','c','d','e' };

    //构建哈夫曼树
    btree<char>* x = huffmanTree (ws,chars, 5);
    x->levelOut (cout);
    cout << "\n";



    /*这里是 OJ 第二个题目*/
    cout << "输出霍夫曼编码长度：\n";
    char str[10000];
    cin >> str;
    cout << calHlen (str);
    cout << "\n";

    return 0;
}
```

---

文件 2 queue.h

```
#include<iostream>
#include<cstring>
#define max(a,b) (a<b ? b:a)//用于获取左右子树中最大的那个层数
using namespace std;

template<typename T>
class queue {
public:
    enum queue_err { queue_empty };
private:
    typedef struct node {
        T data;
        node* next;
        node () { next = nullptr; }
    }node;
    node* _head;
    node* _end;
    int _length;
public:
    queue () {
        _head = new node;
        _end = _head;
        _length = 0;
```

```
    }
    ~queue () {
        while (_head->next != NULL) {
            node* temp = _head;
            _head = _head->next;
            delete temp;
        }
        delete _head;
    }

    void push (const T& in) {
        _length++;
        node* n_end = new node;
        n_end->data = in;
        n_end->next = NULL;

        _end->next = n_end;
        _end = n_end;
    }
    T front () {
        if (empty ()) {
            throw queue_empty;
        }
        return _head->next->data;
    }
    void pop () {
        if (empty ()) {
            throw queue_empty;
        }
        node* n_head = _head->next;
        delete _head;

        _head = n_head;

        _length--;
        return;
    }
    bool empty ()const { return _head == _end; }
    int size ()const { return _length; }
};
```

文件 3 btree.cpp

```
#pragma once
#include<iostream>
#include"queue.h"

using std::ostream;

template<class T>
class btree {
public:
    typedef enum {} err;
    typedef struct node {
        T data;
        node* left;
        node* right;
        node (T data, node* left, node* right)
            :data (data), left (left), right (right) {}
        node ()
            :data(),left (nullptr), right (nullptr) {}

    } node;
protected:
    node* _root;
    int _size;

    void deleteNodes (node* root) {
        if (root) {
            //cout << "delete" << root->data << "\n";
            if (root->left)deleteNodes (root->left);
            if (root->right)deleteNodes (root->right);
            delete root;
        }
    }

    ostream& _preOut (ostream& out, node* rootin) {
        if (rootin == nullptr) {
            return out;
        } else {
            out << rootin->data << " ";
            _preOut (out, rootin->left);
            _preOut (out, rootin->right);
            return out;
        }
    }
    ostream& _postOut (ostream& out, node* rootin) {
```

```
        if (rootin == nullptr) {
            return out;
        } else {
            _postOut (out, rootin->left);
            _postOut (out, rootin->right);
            out << rootin->data << " ";
            return out;
        }
    }
public:
    btree () {
        _root = nullptr;
    }
    ~btree () {
        //cout << "dis\n";
        if (_root) {
            if (_root->left)deleteNodes (_root->left);
            if (_root->right)deleteNodes (_root->right);
            delete _root;
        }
    }
    void clear () {
        if (_root) {
            if (_root->left)deleteNodes (_root->left);
            if (_root->right)deleteNodes (_root->right);
            delete _root;
        }
    }

    ostream& preOut (ostream& out) {
        if (_root == nullptr) {
            return out;
        } else {
            out << _root->data << " ";
            _preOut (out, _root->left);
            _preOut (out, _root->right);
            return out;
        }

    }
    ostream& postOut (ostream& out) {
        if (_root == nullptr) {
            return out;
        } else {
            _postOut (out, _root->left);
            _postOut (out, _root->right);
            out << _root->data << " ";
            return out;
        }
    }
    ostream& levelOut (ostream& out) {
        queue<node* > q;
        node* t = _root;
        q.push (t);
        //通过队列存储待打印元素，这样一层的数据会相邻在一起
        while (!q.empty ()) {
            t = q.front ();
            q.pop ();
            out << t->data << " ";
            if (t->left != nullptr) {
                q.push (t->left);
            }
            if (t->right != nullptr) {
                q.push (t->right);
            }
        }
        return out;
    }
    void setRoot (node* rootin) {
        clear ();
        _root = rootin;
        return;
    }
    void makeTree (T datai, btree& lefttreei, btree& righttreei) {
        _root = new node;
        _root->data = datai;
        _root->left = lefttreei._root;
        _root->right = righttreei._root;
        lefttreei._root = nullptr;
        righttreei._root = nullptr;
    }
};
```

文件 4 huffMan.cpp

```cpp
#pragma once
#include"btree.h"
#include"minHeap.h"

template<class T>
struct huffmanNode {
    btree<T>* tree;
    int weight;

    operator int () const { return weight; }
    bool operator <(huffmanNode& b) const { return weight < b.weight; }
    bool operator >(huffmanNode& b) const { return weight > b.weight; }
};

template <class T>
btree<T>* huffmanTree (int weight[],T* datas,int n) {
    huffmanNode<T>* hNode = new huffmanNode<T>[n + 1];
    btree<T> emptyTree;
    for (int i = 1; i <= n; i++) {
        hNode[i].weight = weight[i];
        hNode[i].tree = new btree<T>;
        hNode[i].tree->makeTree (datas[i], emptyTree, emptyTree);
    }

    // make node array into a min heap
    minHeap<huffmanNode<T> > heap (1);
    heap.initialize (hNode + 1, n);

    // repeatedly combine trees from min heap
    // until only one tree remains
    huffmanNode<T> w, x, y;
    btree<T>* z;
    for (int i = 1; i < n; i++) {
        // remove two lightest trees from the min heap
        x = heap.top (); heap.pop ();
        y = heap.top (); heap.pop ();

        // combine into a single tree
        z = new btree<T>;
        z->makeTree (0, *x.tree, *y.tree);
        w.weight = x.weight + y.weight;
        w.tree = z;
        heap.push (w);
        delete x.tree;
        delete y.tree;
    }

    // destructor for min heap deletes hNode
    return heap.top ().tree;
}


int calHlen (char* si) {
    //统计字符串中各字符的长度
    int char_count[128] = { 0 };
    int type_length = 0;
    int str_length = 0;
    for (int i = 0; si[i] != '\0'; i++) {
        str_length++;
        if (char_count[si[i]] == 0) {
            type_length++;
        }
        char_count[si[i]]++;
    }
    minHeap<int>q;
    for (int i = 0; i < 128; i++) {
        int val = char_count[i];
        if (val > 0) {
            q.push (val);
        }
    }

    if (q.size () == 1)//只有 1 个节点情况
        return q.top ();

    int sum = 0;//初值 0，涵盖了空串情况

    //不断选择集合中权值最小的两个点，计算和 再插入
    while (q.size () > 1) {
        int v1 = q.top (); q.pop ();
        v1 += q.top (); q.pop ();
        sum += v1;
        q.push (v1);
    }
    return sum;
```

```
}
```

文件 5 minHeap.cpp

```
#pragma once
template<class T>
/*分布式排序*/
class minHeap {
public:
    typedef enum { min_head_empty }err;
private:
    int _size;
    int _length;
    T* _head;
    void _extLength () {
        T* temp = new T[_length * 2];
        copy (_head, _head + _length, temp);
        delete[] _head;
        _length *= 2;
        _head = temp;
    }
    void _clear () {
        /*delete[] _head;*/
    }
public:
    minHeap (int lengthi = 10) {
        _length = lengthi + 1;
        _head = new T[_length];
        _size = 0;
    }
    void initialize (T* arri, int sizei) {
        _clearAndInit ();
        for (int i = 0; i < sizei; i++) {
            push (arri[i]);
        }
    }
    ~minHeap () { _clear (); }
    void _clearAndInit () {
        _clear ();
        _length = 11;
        _head = new T[11];
        _size = 0;
    }
    const T& top () {
        if (_size == 0)
            throw min_head_empty;
        return _head[1];
    }
    void pop () {
        if (_size == 0) {
            throw min_head_empty;
        }
        _head[1].~T ();
        T to_be_insert = _head[_size--];
        int insert_index = 1,
            child_index = 2;     // child_index of current_node

        //将新的头部元素逐层向下移动，向下移动到左子还是右子？这里需要判断
        //起码有一个左子树，所以要<=
        while (child_index <= _size) {
            //如果左子比右子大，则根应当与右子交换，使新根小，这样可以保持最小堆特性
            //如果左子树卡到了 size 位置，说明没有右子树，不必寻找左右中最小的元素
            if (child_index < _size && _head[child_index] > _head[child_index + 1]) {
                child_index++;
            }
            //如果根比两个子都小，那直接退出就行了，不必再交换
            if (to_be_insert <= _head[child_index]) {
                break;
            }
            _head[insert_index] = _head[child_index];
            insert_index = child_index;
            child_index *= 2;
        }
        _head[insert_index] = to_be_insert;
    }
    void push (const T& datai) {
        //进行越界检查
        if (_size == _length - 1) {
            _extLength ();
        }

        int insert_index = ++_size;

        while (insert_index != 1 && _head[insert_index / 2] > datai) {//插入元素的父元素不小于插
```

```
入元素，说明需要调整
            _head[insert_index] = _head[insert_index / 2]; //该父元素放到子节点位置
            insert_index /= 2;//子节点位置指向原父节点那里去，也即发生父子交换，只不过子元素还没有插
入

            //继续循环查看新的父节点
        }

        _head[insert_index] = datai;
    }

    bool empty () const { return _size == 0; }
    int size () const { return _size; }
};
```