数据结构与算法第五次作业

陈梓乐 19336035
部分代码可在 gitee.com/Czile/homework 中找到

---

1. 证明：满二叉树的分支数为 B = 2(n_0 - 1). n_0 为叶子节点数

由性质5-3[1]， n_2 = n_0 - 1， n_2 是度数为2的节点数。于是题设要证 B = 2n_2.

由满二叉树的性质[2]， 满二叉树只有度数为0和度数为2的节点，于是没有度数为1的节点，每一个度数为2的节点有两个子节点，

因此分支数为度数为2的节点的两倍。[3]

> [1] 王红梅.数据结构与算法(第二版).清华大学出版社.P110.性质5-3

> [2] 王红梅.数据结构与算法(第二版).清华大学出版社.P108.满二叉树

> [3] 本题也可用图论中边数等于节点度数和来证明

2. 已知度为 m 的树中，有 n_i 个度数为 i 的节点，求树中有多少个叶子节点。

由图论中总度数等于总边数[3]的结论，总度数为： sum($i \cdot n_i$)

除了根节点外，每个节点有且仅有一个父节点，于是得到结论：节点数 - 1 = 分支数。

于是得到等式：n_0 = sum($i \cdot n_i$) - sum(n_i) + 1 [4]

> [4] 本题限定只存在度数为0和度数为2的节点，即可得到性质5-3

3. 已知二叉树的中序遍历和后序遍历分别为 `CBEDAFIGH` 和 `CEDBIFHGA`，试构造二叉树。

使用顺序表来按层表示二叉树如下（空位置用0表示）

`ABGCDFH00E00I00`

4. 给定一组键值 `W = (5, 2, 9, 11, 8, 3, 7)` 试构造haffman-tree，并计算带权路径长度。

构造haffman-tree如下，用haffman编码给出haffman-tree的构造：

```
2 - 0000
3 - 0001
5 - 001
11 - 01
7 - 10
8 - 11
```

5. 字符串S中各字符出现次数分别为 `(1, 2, 3, 4, 4, 5, 7, 9)`，试问haffman编码。

```
1 - 00000
2 - 00001
3 - 0001
4 - 100
4 - 101
5 - 001
7 - 11
9 - 01
```

至少两位，至多5位。

6. 设计算法求二叉树深度

不妨假定二叉树是使用顺序储存结构，则代码如下：

```
#include <cmath>
template <class T>
int depth(const vector<T> &p) noexcept{
    return p.size() ? int(log2(p.size())) + 1 : 0;
}
```

7. 以二叉链表为储存结构，设计算法求节点的双亲。

```
// Find the parent of x
// Return a pointer that points to x's parent if it exists, else return nullptr
// p is the root pointer of the tree that remains to search.
template <class T>
node<T> * getParent(node<T> *p, node<T> *x) {
    if (p -> left && p -> left == x)
        return p;
    if (p -> right && p -> right == x)
        return p;
    node<T> * left, right;
    if (p -> left)
        left = getParent(p -> left, x);
    if (left)
        return left;
    if (p -> right)
        right = getParent(p -> right, x);
    if (right)
        return right;
    return nullptr;
}
```

8. 以二叉链表为存储结构，删除以值为x的二叉树的子树

```
template <class T>
void erase(node<T>* p) {
    if (p && p -> left)
        erase(p -> left);
    if (p && p -> right)
        erase(p -> right);
    if (p)
        delete p;
}

template <class T>
void erase(node<T>* p, T & val) {
    erase(p -> left, val);
    erase(p -> right, val);
    if (p -> val == val)
        erase(p);
}
```

9. 编写算法实现顺序储存的二叉树的前序遍历

```
// Find all element in the tree.
// p: The root of the tree
// n: the size of the tree if the tree is full
// nT: the element if it's not an element
template <class T>
vector<T> NLP(vector<T>::iterator p, int n, T&nT) {
    if (*p == nT)
        return vector<T>();
    vector<T> ans;
    ans.push_back(*p);
    if (n == 1)
        return ans;
    auto tmp = NLP(p + 1, n / 2, nT);
    ans.push_back(ans.end(), tmp.begin(), tmp.end());
    tmp = NLP(p + n / 2 + 1, n / 2, nT);
    ans.push_back(ans.end(), tmp.begin(), tmp.end());
    return ans;
}

template <class T>
vector<T> NLP(vector<T> & p) {
    return NLP(p.begin(), p.size(), T());
}
```

10. 编写算法交换所有节点的左右子树

```
#include <algorithm>
using namespace std;
void swap(node<T> * p) {
    if (p) {
        swap(p -> left, p -> right);
        swap(p -> left);
        swap(p -> right);
    }
}
```

11. 实现树的各种遍历

前序遍历

```
template <class T>
vector<T> NLR(node<T> *p) {
    if (p == nullptr)
        return;
    auto ans = p -> val;
    auto left = NLP(p -> left);
    auto right = NLP(p -> right);
    ans.insert(ans.end(), left.begin(), left.end());
    ans.insert(ans.end(), right.begin(), right.end());
    return ans;
}
```

中序遍历

```
template <class T>
vector<T> LNR(node<T> *p) {
    if (p == nullptr)
        return vector<T>();
    auto ans = NLP(p -> left);
    auto right = NLP(p -> right);
    ans.push_back(p -> val);
    ans.insert(ans.end(), right.begin(), right.end());
    return ans;
}
```

后序遍历

```
template <class T>
vector<T> LRN(node<T> *p) {
    if (p == nullptr)
        return vector<T>();
    auto ans = NLP(p -> left);
    auto right = NLP(p -> right);
    ans.insert(ans.end(), right.begin(), right.end());
    ans.push_back(p -> val);
    return ans;
}
```

层序遍历

```
template <class T>
vector<T> BFS(node <T> *p) {
    if (p == nullptr)
        return vector<T>();
    queue<node <T> *> q;
    vector<T> ans;
    q.push(p);
    while (q.size()) {
        auto top = q.top();
        q.pop();
        ans.push_back(top -> val);
        if (tmp -> left)
            q.push(tmp -> left);
        if (tmp -> right)
            q.push(tmp -> right);
    }
    return ans;
}
```

12. 编写程序，实现各种线索二叉树

前序线索二叉树

```
template <typename T> void Binarytree<T>::_preorder(
    const Binarytreenode * p,
    thrnode *& y, thrnode *& pre
) noexcept {
    if (p == nullptr) return;
    y = new thrnode(p -> val);
    if (pre && pre -> rtag) pre -> right = y;
    y -> rtag = !(p -> right);
    y -> ltag = !(p -> left);
    if (y -> ltag) y -> left = pre;
    pre = y;
    _preorder(p -> left, y -> left, pre);
    _preorder(p -> right, y -> right, pre);
}
```

中序线索二叉树

```
template <typename T> void Binarytree<T>::_inorder(
    const Binarytreenode * p,
    thrnode *& y, thrnode *& pre
) noexcept {
    if (p == nullptr) return;
    y = new thrnode(p -> val);
    _inorder(p -> left, y -> left, pre);
    if (pre && pre -> rtag) pre -> right = y;
    y -> rtag = !(p -> right);
    y -> ltag = !(p -> left);
    if (y -> ltag) y -> left = pre;
    pre = y;
    _inorder(p -> right, y -> right, pre);
}
```

后序线索二叉树

```
template <typename T> void Binarytree<T>::_postorder(
    const Binarytreenode * p,
    thrnode *& y, thrnode *& pre
) noexcept {
    if (p == nullptr) return;
    y = new thrnode(p -> val);
    _postorder(p -> left, y -> left, pre);
    _postorder(p -> right, y -> right, pre);
    if (pre && pre -> rtag) pre -> right = y;
    y -> rtag = !(p -> right);
    y -> ltag = !(p -> left);
    if (y -> ltag) y -> left = pre;
    pre = y;
}
```

层序线索二叉树

```cpp
template <typename T> typename Binarytree<T>::thrnode *
Binarytree<T>::getBFSThrBiTree() noexcept {
    erase(t);
    if (p == nullptr) return t;
    std::vector<thrnode *> ans;
    std::queue<Binarytreenode *> tmp;
    auto i = 0;
    ans.push_back(new thrnode(p -> val));
    tmp.push(p);
    while (i < ans.size()) {
        auto _p = tmp.front();
        if (_p -> left) {
            ans.push_back(
                ans[i] -> left = new thrnode(_p -> left -> val)
            );
            tmp.push(_p -> left);
        } else {
            ans[i] -> ltag = 1;
            if (i) ans[i] -> left = ans[i - 1];
        }
        if (_p -> right) {
            ans.push_back(
                ans[i] -> right = new thrnode(_p -> right -> val)
            );
            tmp.push(_p -> right);
        } else {
            ans[i] -> rtag = 1;
            if (i != ans.size() - 1) ans[i] -> right = ans[i + 1];
        }
        ++i;
        tmp.pop();
    }
    return t = ans[0];
}
```

13. 使用非递归算法求叶子节点数

```cpp
template <typename T> size_t Binarytree<T>::sizeOfLeaf() const noexcept {
    std::queue<Binarytreenode*> q;
    int ans = 0;
    if (p) q.push(p);
    while (q.size()) {
        auto tmp = q.front();
        if (tmp -> left) q.push(tmp -> left);
        if (tmp -> right) q.push(tmp -> right);
        if (!(tmp -> right || tmp -> left)) ++ans;
        q.pop();
    }
    return ans;
}
```

14. 求字符串的最短编码长度

本代码模拟haffman-tree建立与应用过程，具体的类定义与实现请参见第六次上机作业第二题

```
// @brief 统计所有字符出现的个数
map<char, unsigned long long> readfiles(string &s) {
    map<char, unsigned long long> charmap;
    for (auto &i : s)
        charmap[i] += 1;
    return charmap;
}

// @brief 根据词频建立小根堆
heap make_heap(map<char, unsigned long long> & charmap) {
    heap h;
    for (auto i: charmap) {
        haffman_tree * tmp = new haffman_tree(i.first, i.second);
        h.push(tmp);
    }
    return h;
}

// @brief 根据小根堆建立haffman - tree
haffman_tree * make_tree(heap h) {
    if (h.empty())
        return NULL;

    while (h.size()>1) {
        haffman_tree * tmp1 = h.front();
        h.pop();
        haffman_tree * tmp2 = h.front();
        h.pop();
        haffman_tree * tmp = new haffman_tree(tmp1, tmp2);
        h.push(tmp);
    }

    return h.front();
}

// @brief 根据haffman - tree 获得对应字符的haffman编码
map<char, string> translate(haffman_tree * t) {
    queue<pair<haffman_tree *, string>> lst;
    map<char, string> ans;
    lst.push(make_pair(t, ""));
    while (lst.empty() == false) {
        haffman_tree * tmp = lst.front().first;
        if (tmp->left())
            lst.push(make_pair(
                tmp->getleft(), lst.front().second + "0"
            ));
        if (tmp->right())
            lst.push(make_pair(
                tmp->getright(), lst.front().second + "1"
            ));
        if (tmp->isleaf())
            ans[tmp->val()]=lst.front().second;
        lst.pop();
    }
    return ans;
}

int main(int argc, char *argv[]) {
    map<char, unsigned long long> frequency = readfiles(argv[0]);
    heap tmp_heap = make_heap(frequency);
    haffman_tree * tree = make_tree(tmp_heap);
    map<char, string> codetable = translate(tree);
    auto ans = 0;
    for (auto &i: frequency)
        ans += codetable[i.first].size();
    cout << ans;
    return 0;
}
```