2019

上級プログラミング

第五回レポート課題

2019/07/12

US162039　梶田悠

目次

レポートの課題内容

指定されたファイルからすべての英単語（すべて小文字と考えて良い）を読み込み、アルファベット昇順に表示するプログラムを、二分木を用いて作成せよ。ただし同じ単語が複数ある場合には、二分木を構成するとき、ノードに単語のつづり以外にその単語の出現回数をデータメンバとして追加し、単語を表示するときには、つづりと出現回数を以下の実行例のように表示すること。また、アルファベット順の表示のあと、指定した出現回数の単語を削除できるようにせよ。

補足

コードは github にアップロードしてあるので、レポートのコードが読みづらい場合は適宜参照して下さい。

コード

ソースコード全文を以下に掲載する。

コード 1 jpro_report05.cpp

```
#include <iostream>
#include <cstdlib>
#include <fstream>
using namespace std;


class BinTree {
private:
    class BinNode {
    public:
        string data;
        int appearanceNum;
        BinNode* left;
        BinNode* right;
        BinNode(string a = 0, BinNode * b = NULL, BinNode * c = NULL) {
            data = a; left = b; right = c; appearanceNum = 1;
        }
        void printNode() {
            cout << data << "[" << appearanceNum << "] ";
#ifdef _DEBUG
        cout << "this = " << this << endl;
        cout << "left = " << left << endl;
        cout << "right = " << right << endl;
#endif // DEBUG

        }
    };

    BinNode* root;
    void traverse(BinNode* rp);
    BinNode* addNode(BinNode* rp, BinNode* node);
```

```
    BinNode* delNode(BinNode* rp, int  appearanceNum);
public:
    BinTree() { root = NULL; }
    void printTree() { traverse(root); }
    void insert(string str) {
        BinNode* np = new BinNode(str);
        root = addNode(root, np);
    }
    void remove(int apppearanceNum) {
        root = delNode(root, apppearanceNum);
    }
};

void BinTree::traverse(BinNode* rp) {
    if (rp == NULL) return;
    traverse(rp->left);
    rp->printNode();
    traverse(rp->right);
}

BinTree::BinNode* BinTree::addNode(BinNode* rp, BinNode* node) {
    if (node == NULL) return rp;
    if (rp == NULL) return node;
    else {
        if (node->data == rp->data) {
            rp->appearanceNum++;
        }
        else if (node->data < rp->data) {
            rp->left = addNode(rp->left, node);
        }
        else {
            rp->right = addNode(rp->right, node);
        }
        return rp;
    }
}

BinTree::BinNode* BinTree::delNode(BinNode* rp, int appearanceNum) {

    if (rp == NULL) return NULL;
    rp->left = delNode(rp->left, appearanceNum);
#ifdef _DEBUG
    cout << rp->data << " : " << rp->appearanceNum << endl;
#endif // _DEBUG

    if (appearanceNum == rp->appearanceNum) {
#ifdef _DEBUG
        cout << "削除します" << endl;
```

```
        cout << "right = " << right << endl;
        cout << "left = " << left << endl;
#endif // _DEBUG

        BinNode* left = rp->left;
        BinNode* right = rp->right;

        delete rp;
        rp = addNode(right, left);//rp下のツリーの再構成
        //再構成後のツリーに対して削除処理
        rp = delNode(rp, appearanceNum);
        return rp;
    }
    else rp->right = delNode(rp->right, appearanceNum);

    return rp;
}

int main() {
    BinTree bt;  // 空の二進木を作成
    string str;
    ifstream fin("words.txt");
    if (!fin) {
        cerr << "ファイルを開けませんでした。" << endl;
        exit(EXIT_FAILURE);
    }

    while (fin >> str) {
        bt.insert(str);   //単語をツリーに入力
    }

    bt.printTree();   // bt の木全体を表示する
    cout << endl;

    int appearanceNum = 0;
    while (cout << "何回出現した単語を削除しますか --> " && cin >> appearanceNum) {
        bt.remove(appearanceNum);
        bt.printTree();
        cout << endl;
    }

    return 0;
}
```

実行結果

```
advantage[1] advice[1] advise[1] apple[3] approach[1] arrange[1] ask[1] base[1]
bear[1] beautiful[1] behave[1] bottle[3] carry[1] case[4] chance[2] character[1]
close[1] common[2] computer[1] consider[1] continue[1] conversation[1] count[1]
culture[1] danger[1] decide[1] desk[1] develop[1] different[1] difficult[1] easy[1]
eat[1] enter[1] expect[1] explain[2] follow[1] form[1] future[1] gather[1] get[1]
go[5] good[1] grade[2] handle[1] have[1] health[2] healthy[1] in[1] include[3]
increase[1] instance[1] interesting[1] it[1] item[1] language[1] lecture[2]
light[1] like[1] local[1] look[1] lower[1] main[1] method[1] mysterious[2]
natural[2] need[1] neighbor[1] on[4] once[1] original[1] pencil[1] produce[1]
protect[1] require[1] search[1] sigh[1] since[1] skip[2] strongly[1] suppose[1]
table[1] the[1] towards[1] useful[1] usual[1] various[1] whole[1] window[1] wish[2]
with[2] wonder[1] worth[1] wrong[1] zoo[1]
何回出現した単語を削除しますか --> 1
apple[3] bottle[3] case[4] chance[2] common[2] explain[2] go[5] grade[2] health[2]
include[3] lecture[2] mysterious[2] natural[2] on[4] skip[2] wish[2] with[2]
何回出現した単語を削除しますか --> 2
apple[3] bottle[3] case[4] go[5] include[3] on[4]
何回出現した単語を削除しますか --> 3
case[4] go[5] on[4]
何回出現した単語を削除しますか --> 4
go[5]
何回出現した単語を削除しますか --> 5

何回出現した単語を削除しますか --> ^D
```

考察

デバッグの難しさ（もしくは複雑性）

プログラムを作成する過程で木構造の特徴としてデバッグの複雑性が挙げられると感じたので考察として記述する。delNode 関数の作成時に、削除されるべきノードが削除されないバグが発生した。バグ修正前後のコードを以下に掲載する。

コード 2 修正前の delNode 関数

```
BinTree::BinNode* BinTree::delNode(BinNode* rp, int appearanceNum) {

    if (rp == NULL) return NULL;
    rp->left = delNode(rp->left, appearanceNum);

    if (appearanceNum == rp->appearanceNum) {
        BinNode* left = rp->left;
        BinNode* right = rp->right;
        delete rp;
        rp = addNode(right, left);
    }
    rp->right = delNode(rp->right, appearanceNum);

    return rp;
}
```

コード 3 修正後の delNode 関数

```
BinTree::BinNode* BinTree::delNode(BinNode* rp, int appearanceNum) {

    if (rp == NULL) return NULL;
    rp->left = delNode(rp->left, appearanceNum);
    if (appearanceNum == rp->appearanceNum) {
        BinNode* left = rp->left;
        BinNode* right = rp->right;
        delete rp;
        rp = addNode(right, left);//rp下のツリーの再構成
        //再構成後のツリーに対して削除処理
        rp = delNode(rp, appearanceNum);
        return rp;
    }
    else rp->right = delNode(rp->right, appearanceNum);

    return rp;
}
```

修正前は `rp = addNode(right, left);` によってツリーの再構成を行い元々 `rp->right` の位置にあったノードが `rp` の位置に移動しているのに、その後 `rp->right = delNode(rp->right, appearanceNum);` を実行しているので元々 `rp->right` の位置にあったノードに対して削除対象であるかのチェックが出来ていない事が原因であった。このバグの修正

にあたって処理の工程でツリーがどう再構成されているか確認したいと思ったが、ここでツリーの構成を確認する事の難しさを感じた。配列や線形リストのような一方向に要素が並ぶデータ構造であれば順番に要素を出力する事でどのデータがどのデータと隣接しているかを比較的容易に確認できる。木構造の場合は一方向に要素が並んでいるわけではない為コンソール上で直感的にデータの並びを確認できない。今回のデバッグでは自ノードと right,left のノードのポインタを出力するコードを書き足し、出力された情報を元にペンとタブレットで木構造を書いて確認した。このデバッグ難しさは木構造のデメリットとして挙げられると考えた。