2019

アルゴリズムとデータ構造

第二回レポート課題

2019/07/31

US162039：梶田悠

目次

課題内容 1

処理手順 2

ソースコード 4

実行結果 6

考察 7

課題内容

1 行が「氏名,携帯電話番号」で構成されるデータファイル（data.txt）を読み込み、氏名をアルファベット順にソートし、氏名に対応する電話番号を探索して表示するプログラムを作成した（sample.cpp）。

sample.cpp では、氏名と携帯電話番号との対応のデータを格納するデータ構造に線形リスト、ソートアルゴリズムにバブルソートを採用している。 しかし、データ量は 630126 件と膨大であり、現在のプログラムでは処理が終了するまで約 4 時間もの時間を要する。 そこで、データ構造にハッシュ、ソートアルゴリズムにクイックソートを

採用することで処理を高速化させ、短時間（約１分）で処理が終了できるようにプログラムを改良せよ。 実行結果としては、氏名と携帯電話番号の間に ＞ を示し、また、レポートに示す実行結果としてはアルファベット順の上位５件、下位５件を示せ。（作成するプログラムは全件表示のままでよい。） また、レポートの課題内容として sample.cpp、および、data.txt を記載する必要はない。

処理手順

data.txt 内の名前データのみを格納した配列と、名前と番号をデータとして持つノードから構成されるハッシュ構造を構築し、配列をクイックソートでソートししたのちに、配列内の名前を先頭から順に参照しハッシュ構造から同じ名前のノードを検索し番号を出力する事で課題を解決した。

課題の重要な部分であるクイックソートと、ハッシュ構造に関する部分についてのみ処理手順を解説する。

クイックソート

quick_sort()関数は引数は、string 配列の先頭アドレスとソート区間の両端のインデックスの３つである。配列の先頭要素の値を基準に処理を進める。先頭要素から末尾に向かって基準より大きい要素を探し、インデックス(=index1)を記録する。同様に末尾から、先頭に向かって基準より小さい要素を探しインデックス(=index2)を記録す

る。Index1<=index2 であれば二つの要素を入れ替える。index1>index2 になるまでこの処理を繰り返すと、配列が基準より小さい値が先頭側、大きい値が末尾側に集まる。この時、基準として使用した値は先頭要素の値なのでこの要素の位置を適切に調整する必要がある。先頭要素を基準より小さい値を持つ要素群の最後の要素と交換する事で、配列の大小関係の分割を完了する。基準の要素の前後の要素群をそれぞれ同じアルゴリズムで処理する事で書く要素が前後の要素と昇順の関係を持つようになり配列全体のソートが完了する。

ハッシュ構造

　前述したように名前から番号をしたい状況なので、名前の文字列からハッシュ値を生成しこれをインデックスに table 配列にノードを格納した。別ノードとハッシュ値が競合した場合はチェイン法によって線形リストで同じハッシュ値のノードを管理するようにデータ構造を構築した。名前から番号を検索する search()関数は引数で入力された名前からハッシュ値を生成し該当する線形リストを順に同じ名前を持つノードが見つかるまで探索し、該当要素があればそのノードの番号のデータを出力するように実装した。

ソースコード

作成したソースコードは github にアップロードしています。見ずらい場合や情報に不足がある場合は適宜参照して下さい。

コード 1 quick_sort()関数

```
void exchange_pVal(string* a, string* b)
{
    string tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}

void quick_sort(string a[], int left, int right)
{
    int index1 = left ;
    int index2 = right;
    string x = a[left];

    do
    {
        while (a[index1] < x)
        {
            index1++;
        }

        while (a[index2] > x)
        {
            index2--;
        }

        if (index1 <= index2)
        {
            exchange_pVal(&a[index1], &a[index2]);
            index1++;
            index2--;
        }
    } while (index1 < index2);

    if (left < index2) {
        exchange_pVal(&a[left], &a[index2]);
```

```
    }

    if (left < index2) {
        quick_sort(a, left, index2);
    }

    if (index1 < right) {
        quick_sort(a, index1, right);
    }
}
```

コード 2 customHash クラス

```
int hashfunction(string key, int size) {
    unsigned int v = 0;
    for (int i = 0; i < key.length(); i++) {
        v += key[i];
    }
    return v % size;
}

struct node {
    string key;
    string value;
    struct node* next;
};

struct customHash
{
    int size;
    struct node** table;

    int initialize(customHash* h, int size) {
        h->table = (node * *) new struct node[size];
        h->size = size;
        for (int i = 0; i < size; i++) {
            h->table[i] = NULL;
        }
        return 1;
    }

    int add(customHash* h, string key, string value) {
        int hash_value = hashfunction(key, h->size);
        node* temp;
        temp = new node;
        temp->key = key;
```

```
        temp->value = value;

        temp->next = h->table[hash_value];
        h->table[hash_value] = temp;
    }

    string search(customHash* h, string key) {
        int hash_value = hashfunction(key, h->size);
        node* p = h->table[hash_value];
        while (p != NULL) {
            if (p->key == key) {
                return p->value;
            }
            p = p->next;
        }
        return "none";
    }
};
```

実行結果

実行結果 1　report2.cpp の実行結果の抜粋

```
ABERU_ANDORE -> 090-9855-2390
ABERU_ANZYARI -> 090-6353-5820
ABERU_ARUTOXURO -> 090-3985-2411
ABERU_BAIZYAKU -> 090-1166-7929
ABERU_BANSON -> 090-3290-2835
ZYURI_ZINNOSUKE -> 090-7213-5425
ZYURI_YUMINOSIN -> 090-8635-8566
ZYURI_YUKITADA -> 090-5051-6946
ZYURI_YOSITORA -> 090-1623-3215
ZYURI_YOSISADA -> 090-7102-2011
```

考察

今回作成したハッシュ構造で使用したハッシュ関数について考察する。まず、作成したハッシュ関数を以下に再掲する。

```
int hashfunction(string key, int size) {
    unsigned int v = 0;
    for (int i = 0; i < key.length(); i++) {
        v += key[i];
    }
    return v % size;
}
```

このアルゴリズムだと、人の名前には母音の文字が共通して頻繁に出てきたり、名前の文字列長に偏りがある為にある程度は偏りが発生するはずなので更なる改善が望まれると考えた。疑似乱数のような偶然性を持たせてハッシュ値を生成する手法が考えられるが、ハッシュ値の生成が実際の問題の解決において本質的な部分ではない場合の方が多いと考えられるのでハッシュ値の生成のどこまで計算リソースを割くかという議論が生まれる。汎用ライブラリ作成などの使用目的が具体的でないコードを作る場合は各文字や文字列長に依存するアルゴリズムで作成しても妥当かもしれないが、用途が明確な場合はハッシュ値の生成を何にどう依存させるかはハッシュ構造の構築に重要である事が分かった。