2019

アルゴリズムとデータ構造

第一回レポート課題

2019/05/11

US162039：梶田悠

目次

レポートの課題内容

１行が「数値,文字列」で構成されるファイル（data_long.txt）を読み込み，ファイルの全てのデータを線形リストと木構造（２分探索木）に格納するプログラムを作成し、各データ構造の構築時間を比較せよ。 また、構築された各データ構造に対して「数値」で探索し、対応する「文字列」を表示する機能を実装し、「数値」で１００００回、探索する場合の探索時間を比較せよ。

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。

線形リスト

処理手順

データ構造構築の処理手順を以下に示す。

1. コマンドライン引数でデータが記述されたファイルを入力。

2. データを一行ずつ読み込み、セルにキーと値を格納。

3. 全データを読むまで２を繰り返す。この時一つ前のセルの参照を各セルが保持することで線形リストとしてのデータ構造を構築している。（末尾は NULL）

探索処理の手順を以下に示す。

for ループで 0~999 までの整数をキーとして持つデータを探索するプログラムになっているが、以下はその for 文内での処理の手順である。

1. リストの先頭要素とキーの値を引数に、渡されたキーを持つデータを探し該当するデータがあればその要素のポインタを返す search_cell()関数を呼び出す。

2. 1 の返り値が NULL の場合、探索がリストの末尾の要素まで到達した事を示すので、該当データーが存在しない旨の文を出力。

3. 1 の返り値が NULL でなかった場合は、該当するデータが存在した事になるのでデータのキーと値を出力。

コード

コード 1 list_cell.cpp

```
// 線形探索（配列の動的確保）
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <string.h>
//時間計測用
#include <chrono>

using namespace std;

// セルを表わす構造体の定義
struct cell
```

```
{
    int key;          // キー
    char data[256];    // キーに対応する値（文字列）

    struct cell* next;    // 次のデータへのポインタ
};

// 機能：リストの要素生成
// 引数: int key キー
//       char *data 対応するデータとなる文字列
// 戻値: 生成された要素へのポインタ.
struct cell* make_cell(int key, const char* data)
{
    struct cell* mk_cell;

    // cellの領域を確保する
    mk_cell = new struct cell;

    /* 文字列 data を cell->data へコピーする */
    strcpy_s(mk_cell->data, data);

    // 整数へ変換
    mk_cell->key = key;

    // nextを初期化
    mk_cell->next = NULL;

    return mk_cell;
}


// 機能：リストを検索
// 引数：head -- リストの先頭要素のアドレス
//       key -- 検索したいキー
// 戻値：キーが存在した要素のポインタ．ない場合は NULL
struct cell* search_cell(struct cell* head, int key)
{
    struct cell* buf;
    buf = head;

    while (buf != NULL)
    {
        if (buf->key == key)
        {
            return buf;
        }

        buf = buf->next;
```

```
    }

    return NULL;
}


int main(int argc, char* argv[])
{
    if (argc == 1)  // 引数がない場合
    {
        cout << "input file name¥n";
        return -1;
    }

    // ファイルオープン
    ifstream fp(argv[1]);

    // リストのヘッドを定義し初期化
    struct cell* head;
    head = NULL;

    // データのリストへの格納
    cout << "Loading " << argv[1] << endl;
    string line;

  //構築時間の計測開始
    chrono::system_clock::time_point start, end;
    start = chrono::system_clock::now();

    while (fp >> line)
    {
        // ファイルからキーと対応する値（ファイル名）を取得
        int index = line.find(",");    // 区切り文字までの文字数
        string key_str = line.substr(0, index);
        string value = line.substr(index + 1);

        // キーを整数変換 （本当は危険）
        int key = atoi(key_str.data());

        if (value.length() > 255)
        {
            cout << "length of value is too long¥n";
            return -1;
        }

        // リスト要素の生成
        struct cell* e_cell;
        e_cell = make_cell(key, value.data());
```

```
        // 生成した要素をリストの先頭に追加
        e_cell->next = head;

        // 先頭アドレスを更新
        head = e_cell;
    }
    //構築時間の計測終了
    end = chrono::system_clock::now();
    double elapse = chrono::duration_cast<chrono::microseconds>(end -
start).count();
    cout << "構築時間：" << elapse << " (micro sec)" << endl;

    //0~1000の整数をkeyにして1000回検索し、探索時間を計測
    start = chrono::system_clock::now();

    for (int i = 0; i < 1000; i++)
    {
        int ky = i;
        // 配列 data から値がkyである要素番号を探索
        struct cell* s_cell;
        s_cell = search_cell(head, ky);

        if (s_cell == NULL)
        {
            cout << ky << "に対応する値は存在していません。" << endl;
        }
        else
        {
            cout << s_cell->key << " -> " << s_cell->data << endl;
        }
    }
    //探索時間の計測終了
    end = chrono::system_clock::now();
    elapse = chrono::duration_cast<chrono::microseconds>(end - start).count();
    cout << "探索時間：" << elapse << " (micro sec)" << endl;
    fp.close();

    return 0;
}
```

実行結果

実行結果 1　list_cell.cpp の実行結果の抜粋

```
Loading data_long.txt
構築時間：782510 (micro sec)
//--中略--
探索時間：2.16433e+07 (micro sec)
```

木構造

処理手順

データ構造構築の処理手順を以下に示す。

1. コマンドライン引数でデータが記述されたファイルを入力。

2. データを一行ずつ読み込み、ノードにキーと値を格納。

3. 全データを読むまで 2 を繰り返す。この時、既に木構造がノードを持っている場合は、ツリーのルートから大小関係に基づいてノードを末端まで辿りノードを追加する。木構造がノードを持たない場合は、ルートのノードとして扱いツリーを構築する。

探索処理の手順を以下に示す。

for ループで 0~999 までの整数をキーとして持つデータを探索するプログラムになっているが、以下はその for 文内での処理の手順である。

1. ツリーのルートとキーの値を引数に、渡されたキーを持つデータを探し該当するデータがあればその要素のポインタを返す search_tree()関数を呼び出す。（該当要素が存在しない場合 NULL を返す。）

2. 1 の返り値が NULL の場合、該当データーが存在しない旨の文を出力。

3. 1 の返り値が NULL でなかった場合は、該当するデータが存在した事になるのでデータのキーと値を出力。

コード

コード 2　mk_tree.cpp

```
// 木構造の構築と検索
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <string.h>
//時間計測用
#include <chrono>

using namespace std;
```

```
// ノードを表わす構造体の定義
struct node
{
    int key;          // キー
    char data[256];    // キーに対応する値（文字列）

    struct node* r_next;   // 次の右ノードへのポインタ
    struct node* l_next;  // 次の左ノードへのポインタ
};


// 機能：ツリーのノードを生成
// 引数: int key キー
//       char *data 対応するデータとなる文字列
// 戻値: 生成されたノードへのポインタ.
struct node* make_node(int key, const char* data)
{
    struct node* e_node;

    // nodeの領域を確保する
    e_node = new struct node;

    /* 文字列 data を cell->data へコピーする */
    strcpy_s(e_node->data, data);

    // 整数へ変換
    e_node->key = key;

    // nextを初期化
    e_node->r_next = NULL;
    e_node->l_next = NULL;

    return e_node;
}



// ノードをツリーに追加する
// 引数:  root -- ツリーのルートのポインタ
//        e_node -- ツリーに加えたいノードのポインタ
// 戻り値: なし
void insert_node(struct node* root, struct node* e_node)
{
    // 大小を比較し、ノード値よりも小さい場合
    if (root->key > e_node->key)
    {
        if (root->l_next != NULL)
        {
```

```
            // 左側にデータがある場合，再帰処理
            insert_node(root->l_next, e_node);
        }
        else
        {
            // 左側に追加する
            root->l_next = e_node;
        }
    }
    else  // 大小を比較し、ノード値よりも大きい場合
    {
        if (root->r_next != NULL)
        {
            // 右側にデータがある場合，再帰処理
            insert_node(root->r_next, e_node);
        }
        else
        {
            // 右側に追加する
            root->r_next = e_node;
        }
    }
}


// 機能：ツリーを検索
// 引数：root -- ツリーのルートのポインタ
//       key -- 検索したいキー
// 戻値：キーが存在したノードへのポインタ．ない場合は NULL
struct node* search_tree(struct node* root, int key)
{
    // ノードの値より小さい値ならば，左側
    if (root->key > key)
    {
        // もし左側になければ，key はない
        if (root->l_next == NULL)
        {
            return NULL;
        }

        return search_tree(root->l_next, key);
    }

    // ノードの値より大きい値ならば，右側
    if (root->key < key)
    {
        // もし右側になければ，key はない
        if (root->r_next == NULL)
```

```
        {
            return NULL;
        }

        return search_tree(root->r_next, key);
    }

    // 見付かった場合
    if (root->key == key)
    {
        return root;
    }
}

int main(int argc, char* argv[])
{
    if (argc == 1)  // 引数がない場合
    {
        cout << "input file name¥n";
        return -1;
    }

    // ファイルオープン
    ifstream fp(argv[1]);

    // ツリーのrootを定義し初期化
    struct node* root;
    root = NULL;

    // データのツリーへの格納
    cout << "Loading " << argv[1] << endl;
    string line;

    //構築時間の計測開始
    chrono::system_clock::time_point start, end;
    start = chrono::system_clock::now();

    while (fp >> line)
    {
        // ファイルからキーと対応する値（ファイル名）を取得
        int index = line.find(",");    // 区切り文字までの文字数
        string key_str = line.substr(0, index);
        string value = line.substr(index + 1);

        // キーを整数変換 （本当は危険）
        int key = atoi(key_str.data());

        if (value.length() > 255)
```

```
        {
            cout << "length of value is too long¥n";
            return -1;
        }

        // cout << key << endl;

        // ノードの生成
        struct node* e_node;
        e_node = make_node(key, value.data());

        // １つもノードが存在しない場合
        if (root == NULL)
        {
            // 生成したノードをrootとしてツリーを開始
            root = e_node;
        }
        else
        {
            // ノードを追加
            insert_node(root, e_node);
        }
    }
    //構築時間の計測終了
    end = chrono::system_clock::now();
    double elapse = chrono::duration_cast<chrono::microseconds>(end -
start).count();
    cout << "構築時間：" << elapse << " (micro sec)" << endl;
    fp.close();

    //0~1000の整数をkeyにして1000回検索し、探索時間を計測
    start = chrono::system_clock::now();
    for (int i = 0; i < 1000; i++)
    {
        int ky = i;
        // 配列 data から値がkyである要素番号を探索
        struct node* s_node = NULL;
        s_node = search_tree(root, ky);

        if (s_node == NULL)
        {
            cout << ky << "に対応する値は存在しません" << endl;
        }
        else
        {
            cout << s_node->key << " -> " << s_node->data << endl;
        }
    }
```

```
    //探索時間の計測終了
    end = chrono::system_clock::now();
    elapse = chrono::duration_cast<chrono::microseconds>(end - start).count();
    cout << "探索時間：" << elapse << " (micro sec)" << endl;

    return 0;
}
```

実行結果

実行結果 2　mk_tree.cpp の実行結果の抜粋

```
Loading data_long.txt
構築時間：2.09494e+06 (micro sec)
//--中略--
探索時間：463457 (micro sec)
```

考察

以下に各構造での実行結果を再掲する。

実行結果 3　list_cell.cpp の実行結果の抜粋

```
Loading data_long.txt
構築時間：782510 (micro sec)
//--中略--
探索時間：2.16433e+07 (micro sec)
```

実行結果 4　mk_tree.cpp の実行結果の抜粋

```
Loading data_long.txt
構築時間：2.09494e+06 (micro sec)
//--中略--
探索時間：463457 (micro sec)
```

構築時間は線形リストの方が速く、探索時間は木構造の方が速いという結果になった。線形リストの構築処理はセルの参照先を変えるだけだが、木構造では要素の追加の為にツリーの末端まで辿るコストがかかる為に構築時間が長くなる。データの件数が増えるほどにツリーは大きくなり一要素の追加にかかる時間は伸びる事になるので、データの追加が頻繁に発生するシステムにおいては不適切に構造である状況が考えられる。

一方で探索時間に注目すると、規則性を持って構築された木構造の方が探索時間は速い。線形リストは単に入力された順にセルを繋いでいるだけなので、データの並びに規則性が無くデータの特定の為に工夫し得る余地が無い。

未検証だが、上記を踏まえてより良いデータ構造を考察してみた。線形リストをベースに要素の並びに規則性を持たせて探索時間を短縮しようとしたものである。具体的には、線形リストに要素を追加する際に既存の要素との大小関係を判定し昇順（もしくは降順）にセルが繋がるようにしたものである。この構造であれば探索処理で二分探索をする事が出来るので探索時間の短縮を期待できる。一方構築処理でも二分探索によって適切に挿入位置を探索する事になるので構築時間は今回実装した線形リストより長くなってしまう。

二分探索木に似た特徴を持ったデータ構造になるわけだが、データの削除処理について考えると昇順に並んだ線形リストの方が良いと考えた。二分探索木の場合、特定の要素を削除しようとした場合その要素から末端へ連なる要素をそれぞれずらし整合性を保つ必要がありその分のコストが発生する。線形リストをベースに構築すれば前後の要素の参照先情報を修正するのみで済み、コストは一定であり実装も簡素に済ませる事ができるのでこの点において二分探索木より利点のあるデータ構造だと考えた。