第2回　プログラミングゼミ

programming semi

川崎 高志　佐野 正樹　奥田 環

ver.2.0　　2001.09.26

1　はじめに

今回のテーマは **探索** (search) です．探索とは，配列などのデータの集合体から特定のデータを取り出すことです．これは何もプログラミングに限ったことではなく，私たちが日常的に行っていることです．例えば，電話帳から病院の電話番号を調べたり，本棚から本を探したり，雑然とした机の上から資料を探したり，などはまさに探索といえます．

私たちが何かを探すとき，散らかったものの中から探すのと整理されたものの中から探すのとでは，速さも疲労も異なります．プログラムでも同様で，対象の整理のされ方によって，いろいろな探索手法があります．以降では，そのうちの代表的なものについて説明します．

2　データ構造

探索の対象とするデータは，Fig. 1 に示す名刺のデータです．ここでは，1 枚の名刺に該当するデータの単位を **レコード** と呼び，レコードの中の各項目を **フィールド** と呼びます．1 つのレコードは,「名前」,「会社」,「電話番号」という 3 つのフィールドからなっています．

名前	会社名	電話番号
Yusuke Tanimura	IBN	262-7743-5933
Yusuke Watanabe	SCI	508-9700-1529
Keiko Hanada	COMPAP	797-2786-0837
Ryuichi Nakamura	NEG	028-2843-7202
Ryuichi Nishimura	COMPAP	255-9116-7209

Fig. 1　探索対象のデータ構造 (名刺)

1 つのレコードは，プログラム上で Fig. 2, Fig. 3 に示すように実装されています．同図において，name は「名前」を，company は「会社」を，phone は「電話番号」を，それぞれ示しています．ここでは便宜上，phone に重複はないものとします．

探索するデータを特定するものを **探索キー** といいます．以降で紹介する探索プログラムでは，phone を探索キーとして用います．そのため，CARD 構造体には，phone を用いてレコードを比較するための operator

を 2 つ定義しています．ひとつは比較演算子 < であり，2 つの phone を辞書順比較した結果を返します．もうひとつは比較演算子 == であり，phone が文字列として完全に一致する場合に真を返します．

本ゼミでは，CARD 構造体の配列を探索対象とします．

```
struct CARD
{
    char name[25];
    char company[24];
    char phone[15];    // key

    bool operator<(const CARD& ob) const;
    bool operator==(const CARD& ob) const;
    bool isEmpty() const {return phone[0] == 0;}
    void clear() {phone[0] = 0;}
};

std::ostream& operator<<(std::ostream& out, const CARD& card);
```

Fig. 2　探索対象のデータ構造 (ヘッダ)

3　線形探索

最も単純な探索方法は，データ列の初めから順番に調べていくというものです (Fig. 4)．これを，**線形探索 (linear search)** といいます．本棚を左上から順番に探す，散らかった部屋の中をしらみつぶしに (しかし系統的に) 探す，といったことがこれにあたります．

線形探索のプログラムを Fig. 5 に示します．このプログラムでは，与えられた探索キー (phone の値) と同じキー値を持つデータがないかを，配列の先頭から順番に比較して探索を行います．

線形探索は，すべてのデータをたどる手段さえ提供されていれば，どのような構造の探索対象に対しても適用できます．その反面，探索速度が遅いという欠点を持っています．線形探索の比較回数は，探索対象のデータが N 個の場合，平均探索回数が $(N+1)/2$，最大探索回数が N です．すなわち，計算量は $O(N)$ [1] となります．

[1] $\lim_{N\to\infty} \frac{x}{N} = C$ のとき, x は $O(N)$ であるという (ただし, C は定数)．直感的には, x は N に比例している．

```
bool CARD::operator<(const CARD& ob) const
{
    return strcmp(this->phone, ob.phone) < 0;
}


bool CARD::operator==(const CARD& ob) const
{
    return strcmp(this->phone, ob.phone) == 0;
}


std::ostream& operator<<(std::ostream& out, const CARD& card)
{
    out << card.name << "\t" << card.company << "\t" << card.phone;
    return out;
}
```

Fig. 3　探索対象のデータ (ソース)

Fig. 4　線形探索

```
int linear_search(const CARD* a, int n, const CARD& val)
{
    for(int i = 0; i < n; ++i)
        if(val == a[i])
            return i;

    return n;
}
```

Fig. 5　線形探索のプログラム

4　2 分探索

現実の生活では,「探す」ことの効率を上げるために，あらかじめ整理しておくことが多いと思います．50 音順あるいはアルファベット順に並んでいない辞書や電話帳などがあったらとても不便です．

プログラムにおいても，探索対象となるデータ列をあらかじめ整理しておくことで，効率的な探索を行うことができます．そのようなアルゴリズムの 1 つが，2 分探索です．

2 分探索 (binary search) とは，データ列を中央で 2 分割することを繰り返して，探索キーと同じキー値を持つデータの含まれる範囲を絞り込んでいき，データの探索を行うものです．ただし，探索対象とするデータ列は，あらかじめ昇順（または降順）にソートしておく必要があります．

2 分探索の概念図を，Fig. 6 に示します．2 分探索では，まずデータ列の中央のデータの探索キー値を参照します．その値が探したいキー値よりも大きければ，データ列の前半に探したいデータが存在することになり，逆ならば後半に存在することになります．このように，比較を行うたびに，探索範囲を半分に絞り込んでいき，最後に目的のデータに到達します．

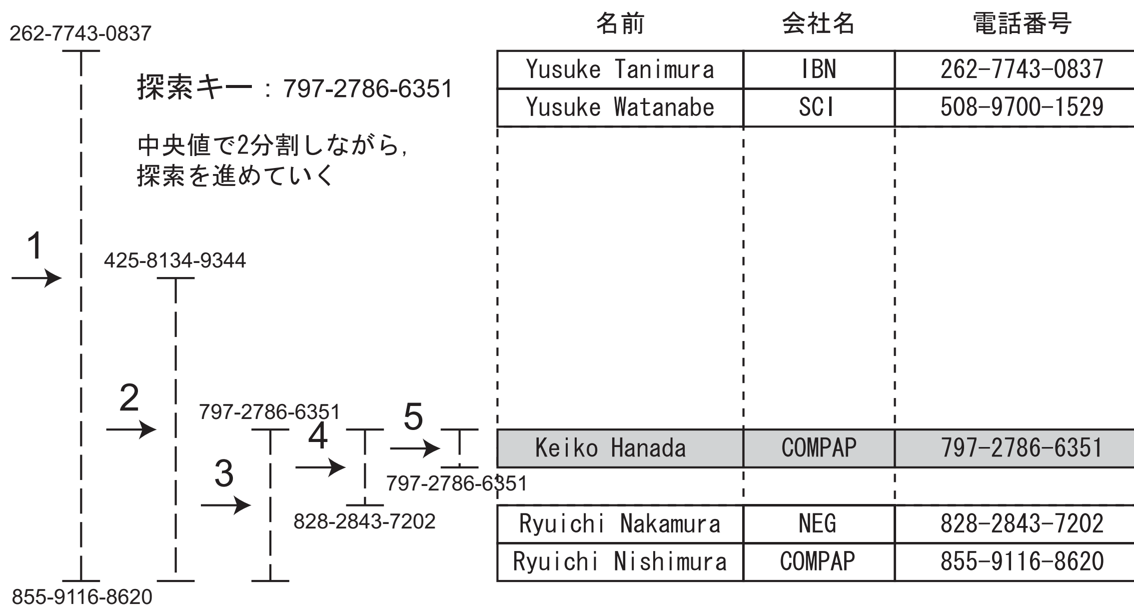

Fig. 6　2 分探索

2 分探索のプログラムを Fig. 7 に示します．このプログラムでは，目的のデータがデータ列に存在しない場合には，配列の最後の添え字に 1 を加えたもの (配列のサイズ) を返します．

2 分探索における比較回数は，キー項目が N 個の場合，平均探索回数が $\log_2 N$，最大探索回数が $\log_2 N + 1$ です．よって計算量は $O(log_2 N)$ となります．

5　ハッシュ法

本の中からある特定の記述を探したい場合，巻末の索引を用いると，目的のページにすぐに辿り着けることがあります．ただ，探索したいデータの全てに索引を用意することは事実上不可能な場合もありますし，索引を格納する領域が必要になったり，索引を作成する手間がかかったりといった欠点もあります．そこで，このような 2 次的なデータを作成せずに高速な探索を実現する方法の 1 つに，ハッシュ法があります．

ハッシュ法 (hashing) では，**ハッシュ関数** を用いてデータ (キー値) を整数値 (**ハッシュ値**) に変換し，その値によって登録場所および探索場所を決定します．ハッシュ関数では，異なるデータがなるべく同じ値に写像されないことが望ましいといえます．また，同一のデータは必ず同一のハッシュ値に写像されなければなりません．

```
int binary_search(const CARD* a, int n, const CARD& val)
{
    int mid, first = 0, last = n - 1;

    while(first < last){
        mid = (first + last) / 2;
        if(a[mid] < val) first = mid + 1;
        else             last  = mid;
    }

    if(a[first] == val) return first;
    else                return n;
}
```

Fig. 7　2 分探索のプログラム

ハッシュ法を用いたデータの挿入方法を，Fig. 8 に示します．まず，挿入したいデータのキー値にハッシュ関数を適用します．そして，得られたハッシュ値に対応した場所に，データを格納します．なお，ハッシュ法では，データの格納先を **ハッシュ表** といいます．

挿入の際に問題となるのは，同一のハッシュ値を持つ複数のデータが発生した場合です．これをハッシュ値の衝突といいます．衝突が起きたときの対処法としては，第 2 のハッシュ関数を適用して他の場所に格納する方法や，同一のハッシュ値を持つものをリストの形にしてハッシュ表に登録する方法などがあります．

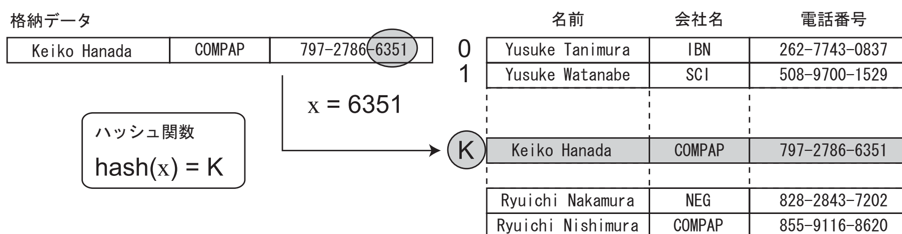

Fig. 8　ハッシュ表への挿入

探索は Fig. 9 のようになります．ハッシュ法による探索は，ハッシュ表に対して適用されます．挿入の時と同様にハッシュ関数を用いてハッシュ値を算出し，その値に該当する場所を探索します．

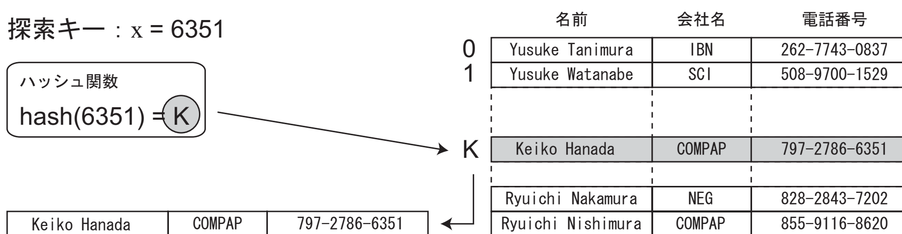

Fig. 9　ハッシュ表の探索

ハッシュ法のプログラムを，Fig. 10, Fig. 11, Fig. 12 に示します．このプログラムでは，phone の下 4 桁をハッシュ値として返すハッシュ関数を使用しています．ハッシュ表のサイズ M が 10000 よりも小さい場合に

は，phone の下 4 桁を M で割った余りをハッシュ値としています．

挿入 (insert() 関数) の際に衝突が起きた場合，隣のエントリを参照します．そこが空ならばデータを挿入し，そうでなければさらに隣を参照する，という操作を繰り返しています．

探索 (search() 関数) では，該当するデータが存在しない場合にはハッシュ表のサイズを返します．

```
class Hash
{
public:
    Hash(unsigned n) : sz(n) {ar = new CARD[n]; clear();}
    ~Hash() {delete[] ar;}

    void clear();
    bool insert(const CARD& val);
    void insert(const CARD* a, int n);
    int search(const CARD& val) const;
    int size() const {return sz;}
    CARD& operator[](int n) {return ar[n];}
    const CARD& operator[](int n) const {return ar[n];}

private:
    int hash(const CARD& val) const;

    CARD* ar;
    int sz;
};
```

Fig. 10　ハッシュのプログラム (ヘッダ)

ハッシュ法では，先に述べた線形探索や 2 分探索と異なり，あらかじめハッシュ関数を用いてデータを格納しておかなければなりません．しかし，非常に高速で，計算量は O(1) です[2]．

6　速度比較実験

これまでに紹介した 3 つの探索方法の速度比較を行います．比較に用いたプログラムを，付録 A に載せておきます．

同プログラムでは，900 件のデータに対する探索を 40000 回行うのに要した時間を計測しています．その結果 (Fig. 13) より，線形探索が最も遅く，ハッシュ法が最も速いことがわかります．1 度しか探索を行わないのならばどの探索方法でも効率は変わらないかもしれませんが，同じ探索対象に対して繰り返し探索を行う場合には，2 分探索やハッシュ法が有効です．

7　おわりに

本ゼミでは，代表的な 3 つの探索手法である，線形探索，2 分探索，ハッシュ法について説明しました．これらを通していえることは,「あらかじめきちんと整理していたデータは，後から探すのが簡単である」という，

[2]衝突の際のアルゴリズムによっては，ハッシュ表のサイズに対するデータ量の割合が大きくなると，効率が悪化します．

ごく当たり前のことです．

また，実際のプログラミングにおいては，探索のコードを自分で書くよりも，言語が提供するライブラリを用いた方が望ましいといえます．例えば，C++ には 線形探索の find() や，2 分探索の binary_search() という関数がありますし，perl や php にはハッシュが用意されています．自分で書く量が増えれば，それだけ誤ったコードが混入する可能性も増えますし，何より手と頭が疲れます．このような正しく動作することが保証されているルーチンを利用することは，正しく短く見やすいプログラムを書く秘訣の 1 つといえます．

```
bool Hash::insert(const CARD& val)
{
    int idx = hash(val);
    int start = idx;
    while(idx != start - 1){
        if(ar[idx] == val)
            return false;

        if(ar[idx].isEmpty()){
            ar[idx] = val;
            return true;
        }
        idx = (idx + 1) % sz;
    }

    return false;
}


void Hash::insert(const CARD* a, int n)
{
    for(int i = 0; i < n; ++i)
        insert(a[i]);
}


void Hash::clear()
{
    for(int i = 0; i < sz; ++i)
        ar[i].clear();
}
```

Fig. 11　ハッシュのプログラム (ソース)

```
int Hash::search(const CARD& val) const
{
    int idx = hash(val);
    int start = idx;

    while(!ar[idx].isEmpty() && idx != start - 1){
        if(ar[idx] == val) return idx;
        idx = (idx + 1) % sz;
    }

    return sz;
}


int Hash::hash(const CARD& val) const
{
    const char* last4digits = val.phone + strlen(val.phone) - 4;
    return atoi(last4digits) % sz;
}
```

Fig. 12　ハッシュのプログラム (つづき)

```
linear search:  5422000
sort:           0
binary search:  297000
insert hash:    0
hash search:    156000
```

Fig. 13　計測結果 単位 [us]

A　プログラムリスト

補足説明　timer() 関数は，start_timer() 関数が呼ばれてからの経過時間をマイクロ秒単位で返す関数.

```
//=======================================================
// search_test.h
//=======================================================
#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <algorithm>

#include "search.h"
using namespace std;

enum {NUM = 900, HASH_SIZE = 10000, TRIALS = 400000};

int main()
{
    CARD a[NUM];
    FILE *fp = fopen("people.dat", "r");
    for(int i = 0; i < NUM; ++i){
        char buf1[255], buf2[255];
        fscanf(fp, "%s %s %s %s", buf1, buf2, a[i].company, a[i].phone);
        sprintf(a[i].name, "%s %s", buf1, buf2);
    }
    fclose(fp);

    const CARD target = {"Keiko Hanada", "COMPAP", "797-2786-0837"};
    cout << "target:\t" << target << endl << endl;

    start_timer();
    int t = timer();
    for(int i = 0; i < TRIALS; ++i)
        linear_search(a, NUM, target);
    cout << "linear search:\t" << timer() - t << endl;
```

Fig. 14　速度比較のプログラム (search_test.cpp)

```
    t = timer();
    sort(&a[0], &a[NUM]);
    cout << "sort:\t" << timer() - t << endl;

    t = timer();
    for(int i = 0; i < TRIALS; ++i){
        binary_search(a, NUM, target);
    }
    cout << "binary search:\t" << timer() - t << endl;

    t = timer();
    Hash hash(HASH_SIZE);
    hash.insert(a, NUM);
    cout << "insert hash:\t" << timer() - t << endl;

    t = timer();
    for(int i = 0; i < TRIALS; ++i)
        hash.search(target);
    cout << "hash search:\t" << timer() - t << endl;
}
```

Fig. 15 速度比較のプログラム (search_test.cpp) (つづき)

```
//===================================================
// search.h
//===================================================

#include <cstdlib>
#include <iostream>

//--------------------------------------------
// bussiness card
//--------------------------------------------
struct CARD
{
char name[25];
char company[24];
char phone[15];    // key

bool operator<(const CARD& ob) const;
bool operator==(const CARD& ob) const;
bool isEmpty() const {return phone[0] == 0;}
void clear() {phone[0] = 0;}
};

std::ostream& operator<<(std::ostream& out, const CARD& card);


//-----------------------------------------------------
// linear search, binary search
//-----------------------------------------------------
int linear_search(const CARD* a, int n, const CARD& val);
int binary_search(const CARD* a, int n, const CARD& val);
```

Fig. 16　search.h

```
//------------------------------------------------------
// Hash search
//------------------------------------------------------
class Hash
{
public:
Hash(unsigned n) : sz(n) {ar = new CARD[n]; clear();}
~Hash() {delete[] ar;}

void clear();
bool insert(const CARD& val);
void insert(const CARD* a, int n);
int search(const CARD& val) const;
int size() const {return sz;}
CARD& operator[](int n) {return ar[n];}
const CARD& operator[](int n) const {return ar[n];}

private:
int hash(const CARD& val) const;

CARD* ar;
int sz;
};
```

Fig. 17 search.h (つづき)

```
//====================================================
// search.cpp
//====================================================

#include <cstring>
#include "search.h"
using namespace std;

//--------------------------------------------
// bussiness card
//--------------------------------------------
bool CARD::operator<(const CARD& ob) const
{
    return strcmp(this->phone, ob.phone) < 0;
}


bool CARD::operator==(const CARD& ob) const
{
    return strcmp(this->phone, ob.phone) == 0;
}


std::ostream& operator<<(std::ostream& out, const CARD& card)
{
    out << card.name << "\t" << card.company << "\t" << card.phone;
    return out;
}




//-----------------------------------------------------
// linear search
//-----------------------------------------------------
int linear_search(const CARD* a, int n, const CARD& val)
{
    for(int i = 0; i < n; ++i)
        if(val == a[i])
            return i;

    return n;
}
```

Fig. 18　search.cpp

```
//------------------------------------------------------
// binary search
//------------------------------------------------------
int binary_search(const CARD* a, int n, const CARD& val)
{
    int mid, first = 0, last = n - 1;

    while(first < last){
        mid = (first + last) / 2;
        if(a[mid] < val) first = mid + 1;
        else             last  = mid;
    }

    if(a[first] == val) return first;
    else                return n;
}


//------------------------------------------------------
// hash
//------------------------------------------------------
bool Hash::insert(const CARD& val)
{
    int idx = hash(val);
    int start = idx;
    while(idx != start - 1){
        if(ar[idx] == val)
            return false;

        if(ar[idx].isEmpty()){
            ar[idx] = val;
            return true;
        }
        idx = (idx + 1) % sz;
    }

    return false;
}
```

Fig. 19 search.cpp (つづき 1)

```
void Hash::insert(const CARD* a, int n)
{
    for(int i = 0; i < n; ++i)
        insert(a[i]);
}



void Hash::clear()
{
    for(int i = 0; i < sz; ++i)
        ar[i].clear();
}



int Hash::search(const CARD& val) const
{
    int idx = hash(val);
    int start = idx;

    while(!ar[idx].isEmpty() && idx != start - 1){
        if(ar[idx] == val) return idx;
        idx = (idx + 1) % sz;
    }

    return sz;
}



int Hash::hash(const CARD& val) const
{
    const char* last4digits = val.phone + strlen(val.phone) - 4;
    return atoi(last4digits) % sz;
}
```

Fig. 20　search.cpp (つづき 2)