明治大学理工学部情報科学科

ソフトウェア実習 レポート

11章 ハッシュその1

提出日 2025 年 12 月 17 日

3 年 15組 91番 157R257501

SEO BEOMGYO

はじめに

本章では、配列と線形探索・2分探索を扱った前回までの内容を発展させ、ハッシュ法による表の実装と探索処理を学んだ。

構造体 struct item { char *id; int info; }; を要素とする配列 table にデータを保持し、文字列の実体は前回までと同様に char_heap 配列に連続して保存し、save_string 関数でコピーした位置へのポインタだけを表に持たせる方式とした。

探索にはハッシュ関数 hash で求めた位置を起点として線形走査を行うオープンアドレス法を用いた。ハッシュ値の計算方法やテーブルのサイズを変化させ、大量データに対する処理時間や strcmp 呼び出し回数の違いを比較することで、ハッシュ法の特性とパラメータ選択の重要性について考察した。

課題1・2

目的

課題1では、11.5節の未完成部分を埋めて insert_table 関数を完成させ、ハッシュ関数で得た位置を起点とする線形走査によってデータを挿入できるようにすることを目的とした。課題2では、ハッシュ表が満杯になった場合に登録を行わず、日本語のエラーメッセージを出力する機能を追加し、配列境界のチェックの重要性を確認することを目的とした。

実装要点

```
15  char *save_string(char *s)
16  {
17      int result = char_top;
18
19      while(1) {
20          if (char_top >= MAX_CHAR) {
21              printf("文字列バッファがあふれました。\n");
22              exit(1);
23          }
24          char_heap[char_top] = *s;
25          char_top++;
26          if (*s == 0)
27              break;
28          s++;
29      }
30      return &char_heap[result];
31  }
```

save_string は、終端文字 '¥0' を含めて引数の文字列を char_heap にコピーし、コピー開始位置のポインタを返す。バッファがあふれたときには "文字列バッファがあふれました。¥n" を出力して exit(1) により異常終了させる。

```
33  int hash(char *v)
34  {
35      int x;
36      x = 0;
37      while (*v != 0)
38          x = 256 * x + (*v++);
39      x %= M;
40      if (x < 0)
41          x = (-x);
42      return(x);
43  }
```

hash 関数は、x = 256 * x + *v という形で文字列を基数256の数として畳み込み、M で剰余を取って配列インデックスに変換する。負の値になった場合は符号を反転させて非負にすることで、配列添字として利用できる値に正規化した。

```c
void initialize_table(void)
{
    int i;
    for (i = 0; i < M; i++)
        mark[i] = 0;
}
```

insert_table では、hash(id) を開始位置 x とし、mark[x] == 1 の間は x = (x + 1) % M でlつずつ先に進む線形走査を行う。課題1では、最初に見つかった空きスロットに save_string(id) の返り値と info を格納し、mark[x] = 1 として登録完了とした。

```c
void insert_table(char *id, int info)
{
    int x, ep;

    x  = hash(id);
    ep = x;

    while (mark[x] == 1) {
        x = (x + 1) % M;
        if (x == ep) {
            printf("ハッシュ表が一杯です。これ以上登録できません。\n");
            return;
        }
    }

    table[x].id   = save_string(id);
    table[x].info = info;
    mark[x] = 1;
}
```

課題2では、開始位置を ep に保存し、探索中に再び ep に戻ってしまった場合を「ハッシュ表が一杯」の条件とみなし、”ハッシュ表が一杯です。これ以上登録できません。¥n” を表示して登録処理を中止するように変更した。

```c
int search_table(char *id)
{
    int x, ep;
    x = hash(id);
    ep = x;
    while (mark[x] == 1) {
        if (strcmp(table[x].id, id) == 0)
            return (x);
        x = (x + 1) % M;
        if (x == ep)
            return (-1);
    }
    return (-1);
}
```

search_table は、hash(id) で求めた位置から mark[x] == 1 の要素だけを対象に strcmp による比較を行い、一致した位置を返す。空きスロットに到達するか、一周して開始位置に戻った場合には -1 を返す。

テストと実行結果

main では、Copernicus から Heisenberg まで 6 人の学者名と生年を insert_table を通して登録し、最後に ”Newton” を search_table で探索して結果を表示する。

実行の結果、”Newton” に対して正しく 1643 年が出力され、ハッシュ法による探索が期待どおりに動作していることを確認した。また、M を小さく設定し、多数のダミーデータを挿入することで、課題 2 で追加した「満杯チェック」が有効に働き、表があふれた時点で登録が打ち切られることを確認した。

課題3

目的

課題1・2で作成したハッシュ表を拡張し、キーボードから S（Search）および I（Insert）コマンドとデータを入力することで、任意の順序で動的に登録・探索を行えるようにすることを目的とした。配列版およびリスト版の課題3と同様に、インタラクティブな利用場面においてもハッシュ表が正しく動作するかどうかを確認した。

実装要点

データ構造と save_string、hash、insert_table、search_table は課題 2 と同一であり、線形走査によるオープンアドレス法を用いる。

```
95      while (1) {
96          if (scanf("%s %s", command, name) == EOF)
97              break;
98
99          if (command[0] == 'I') {
100             if (scanf("%d", &year) != 1) {
101                 printf("年の入力エラーです。\n");
102                 break;
103             }
104
105             pos = search_table(name);
106             if (pos != -1) {
107                 printf("%s はすでに登録されています。\n", name);
108             } else {
109                 insert_table(name, year);
110             }
111
112         } else if (command[0] == 'S') {
113             pos = search_table(name);
114             if (pos == -1) {
115                 printf("%s は登録されていません。\n", name);
116             } else {
117                 printf("%d\n", table[pos].info);
118             }
119
120         } else {
121             printf("不明なコマンドです: %s\n", command);
122         }
123     }
```

main では無限ループ内で scanf(”%s %s”、 command、 name）によりコマンドと名前を読み込み、EOF に達した場合にループを終了する。command[0] == ’I’ のとき挿入、’S’ のとき探索を行い、それ以外の文字が指定された場合は ”不明なコマンドです: %s¥n” を出力する。

I コマンドでは整数 year を読み込み、入力に失敗した場合は ”年の入力エラーです。¥n” と表示してループを抜ける。成功した場合には search_table で重複チェックを行い、既に

登録されている場合は ”〇〇 はすでに登録されています。¥n” を表示し、新規データのみ insert_table を呼び出す。

S コマンドでは、見つからない場合 ”〇〇 は登録されていません。¥n”、見つかった場合は対応する info を出力する。

テストと実行結果

テストデータとして教科書に記載された 1～8 の操作列（未登録データの探索、複数の登録操作、重複登録の試行など）をファイルに記述し、リダイレクトで入力した。

その結果、未登録名の探索時には「登録されていません」というメッセージが表示され、既に登録済みの名前に対して I コマンドを与えた場合には「すでに登録されています」と表示されて登録が行われないことを確認した。これにより、配列版・リスト版と同様のエラーチェックをハッシュ表でも実現できていると判断した。

課題4

目的

課題4では、前回までと同じ Shakespeare テキスト shaks12.txt を入力として、英単語の出現回数一覧を作成するプログラムをハッシュ表で実装し、線形探索や2分探索と比較したときの実行時間と比較回数の違いを調べることを目的とした。

実装要点

```
5    #define M 50000
6    #define MAX_CHAR 10000000
7    #define MAX_WORD 1024
8    #define HASH_A 256
```

M = 50000、MAX_CHAR = 10000000、HASH_A = 256 を用い、単語数より十分大きなテーブルサイズを確保した。

```
61   int search_table(char *id)
62   {
63       int x = hash(id);
64       int ep = x;
65
66       while (mark[x] == 1) {
67           if (my_strcmp(table[x].id, id) == 0)
68               return x;
69           x = (x + 1) % M;
70           if (x == ep)
71               return -1;
72       }
73       return -1;
74   }
```

search_table では、my_strcmp を介して strcmp を呼び出し、そのたびにグローバル変数 n_strcmp をインクリメントすることで、探索中に実際に行われた比較回数を測定できるようにした。

```
76   void insert_table(char *id, int info)
77   {
78       int x = hash(id);
79       int ep = x;
80
81       while (mark[x] == 1) {
82           x = (x + 1) % M;
83           if (x == ep) {
84               printf("ハッシュ表が一杯です。これ以上登録できません。\n");
85               return;
86           }
87       }
88
89       table[x].id   = save_string(id);
90       table[x].info = info;
91       mark[x] = 1;
92   }
```

insert_table は課題2と同様に線形走査で空きスロットを探し、満杯に達した場合には日本語のエラーメッセージを出力して登録を中止する。

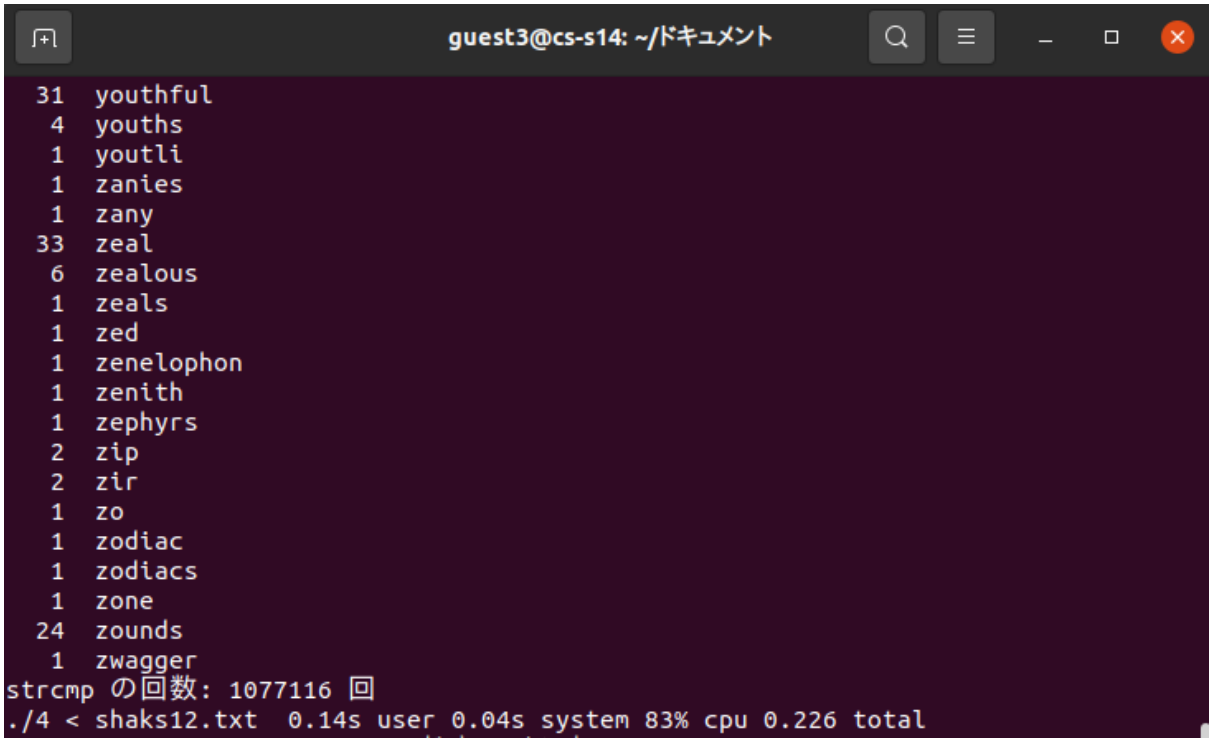
```
113     for (i = 0; i < M; i++) {
114         if (mark[i] == 1) {
115             used[n++] = table[i];
116         }
117     }
118 
119     qsort(used, n, sizeof(struct item), compare_item);
120 
121     for (i = 0; i < n; i++) {
122         printf("%4d  %s\n", used[i].info, used[i].id);
123     }
124 
125     printf("strcmp の回数: %lld 回\n", n_strcmp);
```

出力時には、mark[i] == 1 の要素のみを一時配列にコピーし、strcmp を用いる qsort によって単語の辞書順に整列してから、info と id を一行ずつ表示する。最後に ”strcmp の回数: %lld 回¥n” の形式で比較回数を出力し、探索アルゴリズムの負荷の目安とした。

```
137     while (c != EOF) {
138         if (!((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'))) {
139             c = getchar();
140         } else {
141             i = 0;
142             while ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
143                 if (i >= MAX_WORD) {
144                     printf("単語が長すぎます。\n");
145                     exit(1);
146                 }
147                 if (c >= 'A' && c <= 'Z')
148                     c = c - 'A' + 'a';
149                 word[i] = c;
150                 i++;
151                 c = getchar();
152             }
153             word[i] = 0;
154 
155             p = search_table(word);
156             if (p == -1)
157                 insert_table(word, 1);
158             else
159                 table[p].info++;
160         }
161     }
```

main では1文字ずつ読み込み、英字以外を区切りとして単語を抽出する。大文字は ’a’ - ’A’ を加算して小文字へ変換し、大文字・小文字を区別しない集計とした。単語ごとに search_table で探索し、見つからなければ insert_table で新規登録、見つかった場合には info をインクリメントする。

テストと実行結果


guest3@cs-s14: ~/ドキュメント
31 youthful
4 youths
1 youtli
1 zanies
1 zany
33 zeal
6 zealous
1 zeals
1 zed
1 zenelophon
1 zenith
1 zephyrs
2 zip
2 zir
1 zo
1 zodiac
1 zodiacs
1 zone
24 zounds
1 zwagger
strcmp の回数: 1077116 回
./4 < shaks12.txt 0.14s user 0.04s system 83% cpu 0.226 total

課題5

目的

課題5では、ハッシュ関数に用いる定数 256 が必ずしも良い値ではないことを確認するため、係数を 37 に変更した場合の挙動を調べることを目的とした。

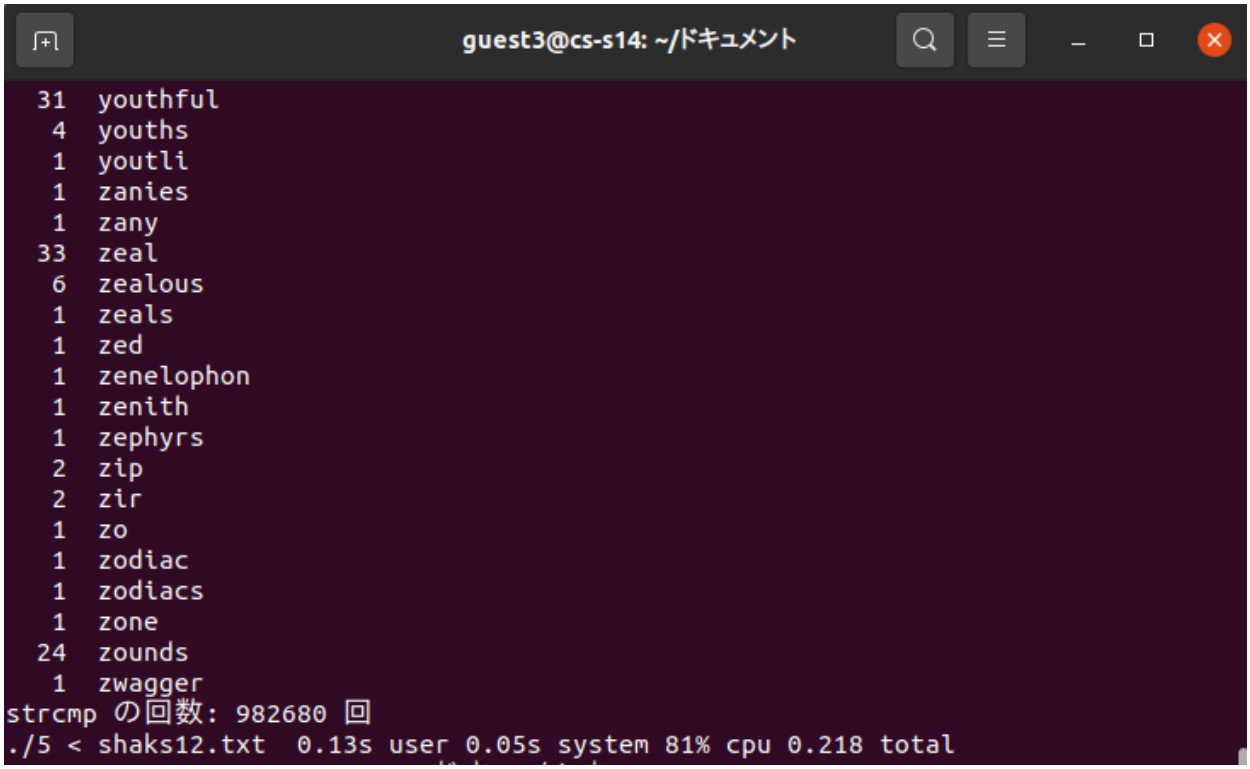
実装要点

プログラム全体は課題4と同一であり、HASH_A の値のみ 256 から 37 に変更した。これにより、ハッシュ値の計算は

$$x = 37 \times x + (\text{文字コード})$$

という形になり、2 の冪乗ではない係数を用いることでハッシュ値の分布がよりばらけることが期待される。

テストと実行結果



課題6

課題6では、ハッシュ定数を 0 や 1、2、3 といった小さな値に変更した場合の挙動を調べ、ハッシュ関数の選び方が分布と性能にどのような影響を与えるかを考察することを目的とした。

実装要点

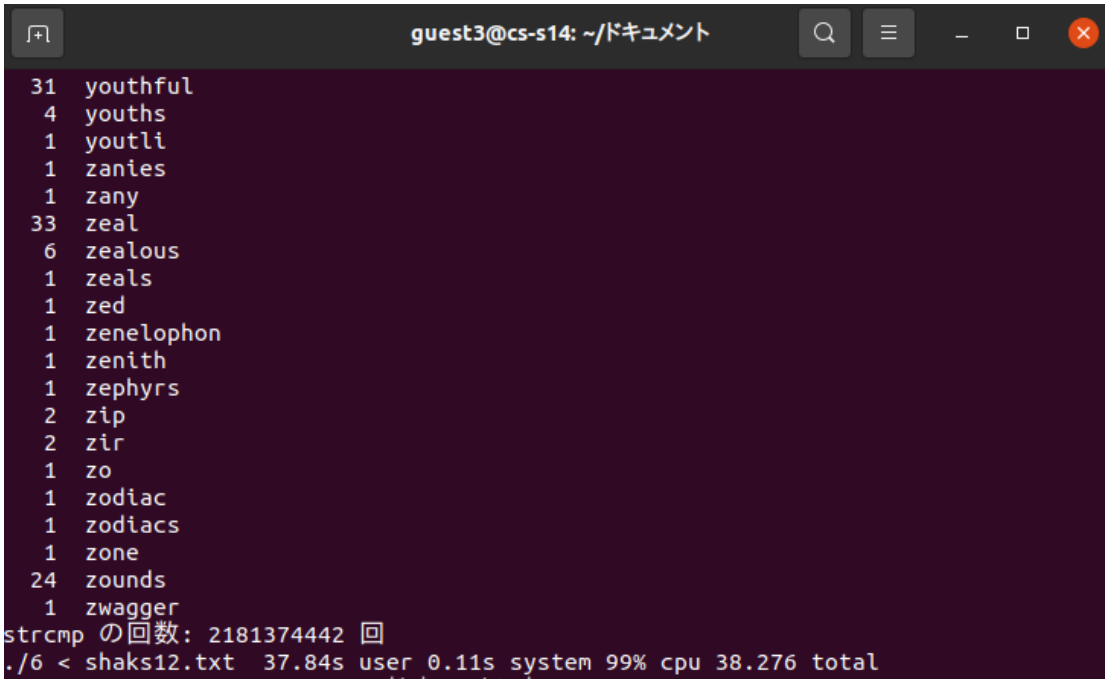
HASH_A = 0、1、2、3 とした以外は課題4の実装と同一である。すなわち、hash の計算において

$$X = (0、1、2、3) \times x + (文字コード)$$

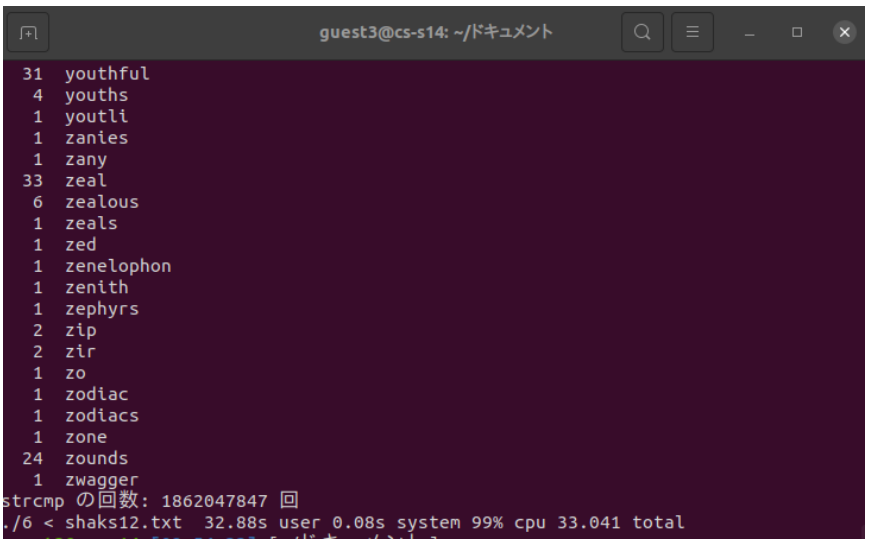
と更新し、残りの処理はそのままとした。

テストと実行結果

(ハッシュ定数 = 0の時)



（ハッシュ定数 = 1の時）



（ハッシュ定数 = 2の時）

（ハッシュ定数 = 3の時）

課題7

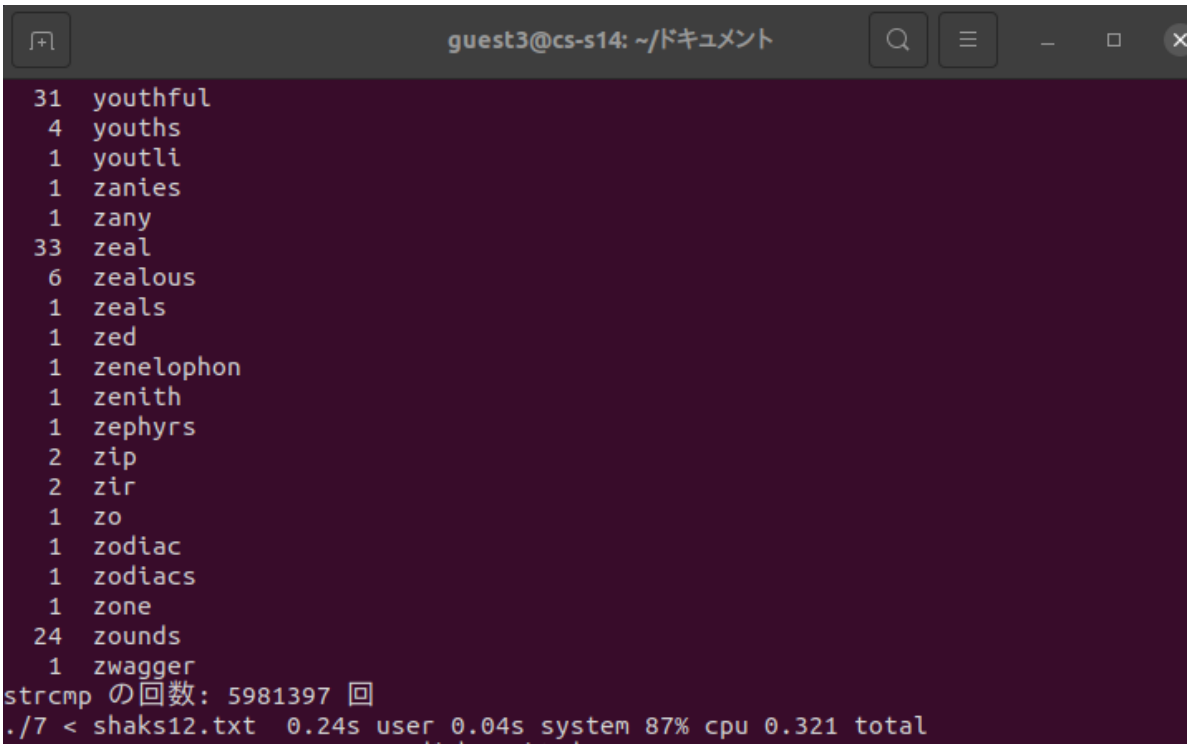
目的

課題 7 では、テーブルサイズ M を変化させたときの処理時間と strcmp 回数の変化を調べ、ハッシュ表における負荷率の影響を確認することを目的とした。

実装要点

ハッシュ定数としては課題 5 と同様に HASH_A = 37 を用い、M のサイズを複数回縮小した結果、24000 が結果への影響を及ぼさず最小の数であると判断した。 その他の構造は課題 5 と同一であり、search_table で my_strcmp を用いて比較回数を数える。

テストと実行結果



課題 8

目的

課題 8 では、探索・挿入に加えてデータの削除も行えるようにハッシュ表を拡張することを目的とした。単純に table の要素を消して mark を 0 に戻すだけでは、線形走査の途中で探索が打ち切られてしまい、その後ろにあるデータを見つけられなくなるという問題がある。そのため、削除済みスロットを区別するマーキング手法を実装した。

実装要点

mark の値として 0（未使用）、1（使用中）に加えて 2（削除済み）を導入し、削除された位置を「墓石」として記録するようにした。

```
52  int search_table(char *id)
53  {
54      int x = hash(id);
55      int ep = x;
56
57      while (mark[x] != 0) {
58          if (mark[x] == 1 && strcmp(table[x].id, id) == 0)
59              return x;
60          x = (x + 1) % M;
61          if (x == ep)
62              return -1;
63      }
64      return -1;
65  }
```

search_table では、mark[x] != 0 の間ループを続け、mark[x] == 1 のときだけ strcmp による比較を行う。mark[x] == 2（削除済み）のスロットは、探索対象のチェイン上には存在するがデータは無効という扱いになり、探索を打ち切らずに通過する。これにより、削除済みスロットの後ろ側にあるデータも正しく探索できる。

```
85      if (first_deleted != -1) {
86          x = first_deleted;
87      } else if (mark[x] == 1) {
88          printf("ハッシュ表が一杯です。これ以上登録できません。\n");
89          return;
90      }
```

insert_table では、線形走査中に最初に見つけた削除済みスロットの位置を first_deleted に記録しておき、完全な空きスロットに到達しなかった場合には first_deleted に新しいデータを挿入する。これにより、削除済みの領域を再利用しつつ、チェインを途切れさせない構造を実現した。ループを一周しても空きスロットが見つからない場合には、ハッシュ表が満杯と判断してエラーメッセージを出力する。

```
97  void delete_table(char *id)
98  {
99      int pos = search_table(id);
100     if (pos == -1) {
101         printf("%s は登録されていません。\n", id);
102         return;
103     }
104     mark[pos] = 2;
105     printf("%s を削除しました。\n", id);
106 }
```

delete_table は search_table で位置を特定し、見つかった場合には mark[pos] = 2 として削除済み状態にするだけである。未登録名に対して削除を行おうとした場合には ”〇〇 は登録されていません。¥n” を表示する。

```
116     while (1) {
117         if (scanf("%s", command) == EOF)
118             break;
119 
120         if (command[0] == 'I') {
121             if (scanf("%s %d", name, &year) != 2) {
122                 printf("入力形式が正しくありません。\n");
123                 break;
124             }
125             insert_table(name, year);
126 
127         } else if (command[0] == 'S') {
128             if (scanf("%s", name) != 1) {
129                 printf("入力形式が正しくありません。\n");
130                 break;
131             }
132             pos = search_table(name);
133             if (pos == -1)
134                 printf("%s は登録されていません。\n", name);
135             else
136                 printf("%d\n", table[pos].info);
137 
138         } else if (command[0] == 'D') {
139             if (scanf("%s", name) != 1) {
140                 printf("入力形式が正しくありません。\n");
141                 break;
142             }
143             delete_table(name);
144 
145         } else {
146             printf("不明なコマンドです: %s\n", command);
147         }
148     }
```

main では、I name year による挿入、S name による探索に加えて、D name による削除コマンドを受け付けるようにした。不正な入力形式に対しては ”入力形式が正しくありません。¥n” を出力し、未知のコマンドについては ”不明なコマンドです: %s¥n” を表示する。

テストと実行結果

同じ名前に対して I → S → D → S → I → S のような操作列を与えたところ、削除後の探索では「登録されていません」というメッセージが表示され、再度挿入した後の探索で正しい info が出力される構造になっていることを確認した。削除済みスロットを 2 で区別することで、探索チェインを保ちながら領域を再利用できることが理解できた。

考察

本演習では、線形探索や 2 分探索と比較しながら、ハッシュ表の性能がハッシュ関数の設計と表サイズに大きく依存することを、実測結果を通して確認できた。とくに、shaks12.txt を用いて strcmp の呼び出し回数と実行時間を測定したところ、パラメータの違いにより、探索回数が数十倍から数千倍まで変化する極端な差が現れた。

まず課題 4 と 5 では、テーブルサイズ M = 50000、HASH_A = 256 と HASH_A = 37 を比較した。HASH_A = 256 の場合、strcmp の回数は 1、077、116 回、実行時間は約 0.23 秒であったのに対し、HASH_A = 37 の場合は 982、680 回、約 0.22 秒であった。比較回数は約 9 % 減少しており、実行時間もわずかながら短くなっている。両者ともハッシュ表としては十分高速であるが、2 の冪乗である 256 よりも、37 のような素数を係数に用いた方が、ハッシュ値の分布がわずかに均一になり、線形走査の平均長が短くなる傾向が確認できたと言える。

次に課題 6 では、ハッシュ定数を 0、 1、 2、 3 と極端に小さくした場合の挙動を調べた。HASH_A = 0 のとき、strcmp の回数は 2、181、374、442 回、実行時間は約 38 秒と、ほぼ最悪に近い結果になった。これは x = 0 * x + c という形になり、実質的に「最後の文字しか見ていない」ハッシュ関数になってしまうためである。同じ語尾を持つ単語がすべて同じ領域に集中し、線形走査の列が非常に長くなってしまう。

HASH_A = 1 の場合も 1、862、047、847 回・約 33 秒と依然として極めて遅く、文字コードの単純な総和だけでは十分な分散が得られないことが分かる。HASH_A = 2 では 304、464、864 回・約 6.5 秒、HASH_A = 3 では 39、073、946 回・約 1.0 秒まで改善するが、それでも 256 や 37 を用いた場合と比べると桁違いに遅い。小さな係数ではハッシュ値の周期性が強くなり、特定のパターンの文字列が同じ位置に集まりやすいことが数値としてはっきり現れている。

課題 7 では、ハッシュ定数を HASH_A = 37 に固定したまま、テーブルサイズ M を 50000 から 24000 に縮小した。M = 50000 のとき strcmp の回数は 982、680 回・約 0.22 秒であったのに対し、M = 24000 の場合は 5、981、397 回・約 0.32 秒となった。比較回数はおよそ 6 倍に増加しているにもかかわらず、実行時間は約 1.5 倍程度にとどまっており、実行時間と比較回数が単純比例しない点も興味深い。しかし、負荷率が高くなると、線形走査でたどるクラスタの長さが急激に伸びるという理論的な性質は、strcmp 回数の増加として明確に確認できた。

以上の結果から、オープンアドレス法によるハッシュ表の性能は、(1) ハッシュ関数の係数の選び方、(2) テーブルサイズと負荷率の管理、という二つの要因に強く依存することが分かった。係数としては 0 や 1 のような極端な値を避け、テーブルサイズとの相性を考慮した「変な周期を持たない」数を選ぶ必要がある。また、負荷率が高くなりすぎると、探索時間が急激に悪化するため、実際のシステムでは一定以上の負荷率で再ハッシュやテーブルの拡張を行うのが妥当であると考えられる。今回の実験は、小さなプログラムであってもパラメータ設定次第で性能が何桁も変わり得ることを示しており、アルゴリズムだけでなく実装上の定数の選択が重要であることを改めて実感する機会となった。