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

9章 配列の探索

提出日 2025 年 12 月 7 日

3 年 15組 91番 157R257501

SEO BEOMGYO

はじめに

配列 table に構造体を格納することで簡易な表を実装し，線形探索と2分探索という二つの探索アルゴリズムを比較した。さらに，英語テキストから単語を抽出して出現回数を数えるプログラムを実装し，大量データに対して探索アルゴリズムの計算量の違いが実行時間にどのように現れるかを検証した。文字列の取り扱いに関しては，配列 char_heap に文字列をコピーし，ポインタのみを表に保持する方式を用いることで，入力バッファの寿命に依存しない安全な管理を行った。

課題2

目的

線形探索を用いて配列中に文字列データを保存し，指定されたキーに対応するレコードを探索する基本的なプログラムを完成させる。表の状態を逐次出力することで，挿入処理と探索処理の動作を確認することも目的とした。

実装要点

```
 8   struct item { char *id; int info; };
 9
10   struct item table[TABLE_SIZE];
```

- 構造体 struct item { char *id; int info; }; を定義し，人物名 id と整数情報 info（西暦年）を1件のレコードとして保持する。これを長さ TABLE_SIZE の配列 table に格納する。

```
14   char char_heap[MAX_CHAR];
15
16   char *save_string(char *s)
17   {
18       int result = char_top;
19
20       for (;;) {
21           if (char_top >= MAX_CHAR) {
22               printf("string buffer overflow\n");
23               exit(1);
24           }
25           char_heap[char_top] = *s;
26           char_top++;
27           if (*s == 0)
28               break;
29           s++;
30       }
31       return &char_heap[result];
32   }
```

- 文字列の実体は char_heap[MAX_CHAR] に連続して保存し，save_string 関数でコピーした位置のポインタを返す。insert_table ではこのポインタを id に格納することで，入力バッファの寿命に依存しない形で文字列を管理する。

```c
void insert_table(char *id, int info)
{
    if (count >= TABLE_SIZE) {
        printf("表があふれました。\n");
        exit(1);
    }

    table[count].id   = save_string(id);
    table[count].info = info;
    count++;

    print_table();
}
```

- insert_table では，まず count が TABLE_SIZE に達していないかを確認し，あふれた場合は「表があふれました。」と出力して exit(1) で異常終了させる。空きがある場合は，save_string でコピーした文字列ポインタと info を table[count] に格納し，count をインクリメントする。挿入後には print_table を呼び出して現在の配列内容と count を表示する。

```c
int search_table(char *id)
{
    int i;

    for (i = 0; i < count; i++)
        if (strcmp(id, table[i].id) == 0)
            return i;
    return -1;
}
```

- search_table は線形探索であり，添字 0 から count-1 まで順に strcmp で id を比較し，一致した添字を返す。一致しない場合は -1 を返す。
- Copernicus など6名の学者名と生年を insert_table を通じて登録し，最後に ”Newton” を探索して見つかった場合は生年を，見つからなければ ”not found.” を表示する。

テストと実行結果

プログラムを実行すると，挿入のたびに print_table の出力として count と各要素の（添字，id，info）が表示され，count が 1 から 6 まで順に増加することを確認した。配列の内容は挿入順に並び，それぞれの id と info が期待通り格納されている。最後に ”Newton” を探索した結果，添字 2 が返り，標準出力に 1643 と表示されることを確認した。線形探索は単純であるが，データ件数が増えると比較回数が増加することを意識する必要があると感じた。

課題3

目的

課題2の表を拡張し，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して，動的に登録・探索を行えるようにする。これにより，静的なテストデータではなく任意の順序で操作した場合でも表構造が正しく維持されることを確認する。

実装要点

- データ構造および save_string，initialize_table，insert_table，search_table，print_table は課題2と同一であり，線形探索を用いる。

```
83      while (1) {
84          if (scanf("%s %s", command, name) == EOF)
85              break;
```

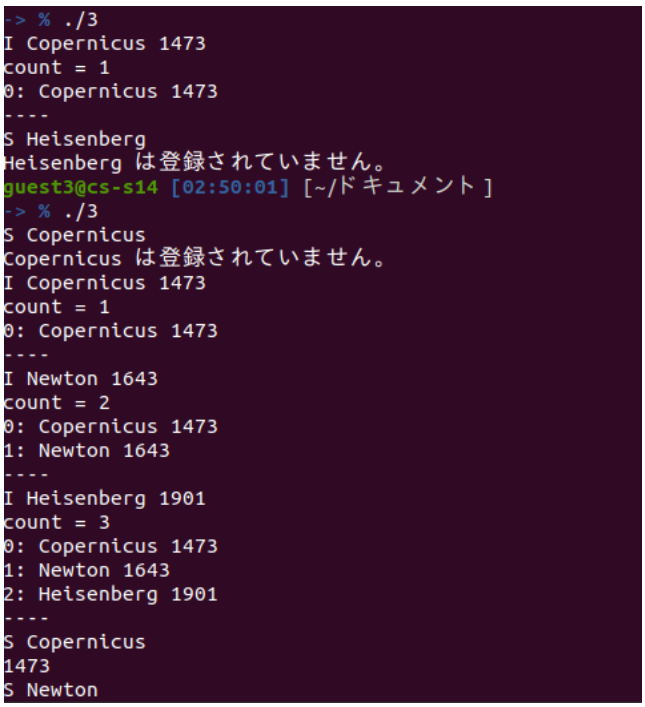
- main では while（1）ループ内で scanf(”%s %s”, command, name）を用いてコマンド文字列と名前を読み込む。EOF に達した場合にループを抜けて終了する。

```
87          if (command[0] == 'I') {
88              if (scanf("%d", &year) != 1) {
89                  printf("年の入力エラーです。\n");
90                  break;
91              }
92              pos = search_table(name);
93              if (pos != -1) {
94                  printf("%s はすでに登録されています。\n", name);
95              } else {
96                  insert_table(name, year);
97              }
98          } else if (command[0] == 'S') {
99              pos = search_table(name);
100             if (pos == -1) {
101                 printf("%s は登録されていません。\n", name);
102             } else {
103                 printf("%d\n", table[pos].info);
104             }
105         } else {
106             printf("不明なコマンドです: %s\n", command);
107         }
108     }
```

- command[0] == ’I’ の場合は挿入処理とし，整数 year を scanf で読み込む。入力が失敗した場合は「年の入力エラーです。」と表示して異常終了させる。成功した場合は名前で search_table を呼び出し，既に登録済みなら「○○ はすでに登録されています。」と表示し，新規の場合のみ insert_table を呼び出す。
- command[0] == ’S’ の場合は探索処理とし，search_table の返り値が -1 なら「○○ は登録されていません。」と表示し，見つかった場合は対応する info を出力する。

- その他の文字がコマンドとして入力された場合は，「不明なコマンドです：%s」と表示してユーザに誤りを通知する。

テストと実行結果



これらの結果から，線形探索を用いた表の登録・探索が，インタラクティブな利用場面でも正しく動作していると判断した。

課題5

目的

表の要素を名前順に常にソートされた状態で保持し，探索には2分探索を用いるプログラムを作成する。これにより，探索の計算量を O(log n) まで削減できることを確認し，そのために必要となる挿入処理の工夫を理解する。

実装要点

- データ構造や文字列管理の方法は課題3と同じであるが，insert_table と search_table の実装を変更し，表を常に昇順に保つ。

```
51  void insert_table(char *id, int info)
52  {
53      int pos;
54
55      if (count >= TABLE_SIZE) {
56          printf("表があふれました。\n");
57          exit(1);
58      }
59
60      pos = count;
61      while (pos > 0 && strcmp(id, table[pos - 1].id) < 0) {
62          table[pos] = table[pos - 1];
63          pos--;
64      }
65
66      table[pos].id   = save_string(id);
67      table[pos].info = info;
68      count++;
69
70      print_table();
71  }
```

- insert_table では，配列末尾から前方へ向かって strcmp(id, table[pos-1].id) < 0 で比較し，挿入すべき位置まで既存要素を1つずつ右にシフトする。挿入位置が決まったら save_string の返り値と info を格納し，count を増やす。配列のコピーは構造体ごと行うことで，ポインタと整数を一括して移動させている。挿入後には課題2同様 print_table を呼び出す。

```c
int search_table(char *id)
{
    int lo = 0, hi = count - 1;
    int mid, cmp;

    while (lo <= hi) {
        mid = (lo + hi) / 2;
        cmp = strcmp(id, table[mid].id);
        if (cmp < 0)
            hi = mid - 1;
        else if (cmp > 0)
            lo = mid + 1;
        else
            return mid;
    }
    return -1;
}
```

- search_table は 2分探索を実装し，lo と hi で探索範囲の両端を管理する。中央 mid を計算し，strcmp の結果が負なら探索範囲を左半分，正なら右半分に縮小する。一致した場合には mid を返し，最後まで見つからない場合は -1 を返す。
- main は課題3と同じコマンド形式を持ち，利用者から見たインターフェースは変えずに内部実装のみを2分探索対応に置き換えた。

テストと実行結果

```
guest3@cs-s14 [02:51:52] [~/ドキュメント]
-> % ./5
S Copernicus
Copernicus は登録されていません。
I Copernicus 1473
count = 1
0: Copernicus 1473
----
I Newton 1643
count = 2
0: Copernicus 1473
1: Newton 1643
----
I Heisenberg 1901
count = 3
0: Copernicus 1473
1: Heisenberg 1901
2: Newton 1643
----
S Copernicus
1473
S Newton
1643
S Heisenberg
1901
I Newton 1643
Newton はすでに登録されています。
```

このことから，2分探索用の表では，挿入時に要素のシフトという追加コストを負う代わりに，探索を効率化できるという設計上のトレードオフが実感できた。

課題6　大量データに対する探索速度の比較

課題6では，英語テキスト shaks12.txt を入力として，単語ごとの出現回数一覧を作成するプログラムを線形探索版と2分探索版の2種類実装し，実行時間と挙動を比較した。

6.1 線形探索版（6_linear.c）

実装要点

- search_table は課題3と同様の線形探索であり，配列先頭から順に strcmp を行う。
- insert_table は配列末尾に新しい単語を追加するだけであり，表の順序は特に保持しない。
- 出力時に qsort と比較関数 compare_item を用いて id の辞書順にソートし，その後（出現回数，単語）を一行ずつ表示する。
- main では getchar で1文字ずつ読み込み，英字のみを単語として扱う。大文字は c += 'a' - 'A' によって小文字に変換してから word 配列に格納し，大文字・小文字を区別しない集計とした。単語境界に達すると終端文字 '¥0' を付加し，search_table で既存レコードを探索して，見つかれば info をインクリメント，見つからなければ insert_table により新規追加する。

6.2 二分探索版（6_binary.c）

実装要点

- search_table は課題5と同様の2分探索を用いる。表は常に id の昇順で保持されるため，lo，hi，mid を更新しながら strcmp の結果に応じて探索範囲を半分ずつに縮小する。
- insert_table では，まず2分探索と同様のループで挿入位置 pos を求める。その後，memmove を用いて pos 以降の要素を一括で1つ後ろに移動し，新しい単語を table[pos] に格納する。これにより，配列全体が常にソート済みである状態を維持する。
- print_table は，既に表が昇順であることを前提に，単純なループで（info，id）を出力するのみとした。ソートは不要である。
- main における単語抽出ロジックは線形探索版と共通であり，英字以外を区切りとし，大文字を小文字に変換してから word 配列に格納する。

テストと実行結果

time ./linear < shaks12.txt と time ./binary < shaks12.txt を実行し，両者の user 時間を比較したところ，2分探索版の方が線形探索版よりも短い時間で処理を完了することを確認した。特に，単語数が多い後半の入力では，線形探索では既存の全単語と順に比較する必要があるのに対し，2分探索では比較回数が O(log n) に抑えられるため，探索に要す

る時間が大きく削減されていると考えられる。

一方で，新しい単語を挿入する際には，2分探索版では要素のシフトに O(n) のコストがかかる。しかし，実際のテキストでは同じ単語が繰り返し登場し，新規単語の追加回数は全単語数に比べて少ない。そのため，総合的には2分探索版が有利になるという結果になった。

考察

本演習では，配列とポインタを用いて簡易な表を実装し，線形探索と2分探索という二つの探索アルゴリズムを比較した。線形探索は実装が単純である一方，データ件数に比例して比較回数が増加するため，大量データに対しては実行時間が問題となることが確認できた。

他方，2分探索はデータをソート済みで保持する前提のもとで，高速な探索を実現できる。しかしその代償として，挿入時に要素の移動が必要となり，特に新規要素が頻繁に追加される場合には負荷が大きくなる。単語出現回数カウントの例では，新規単語の追加よりも既存単語のカウント更新が支配的であるため，2分探索版の優位性が現れた。