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

10章 リストの探索

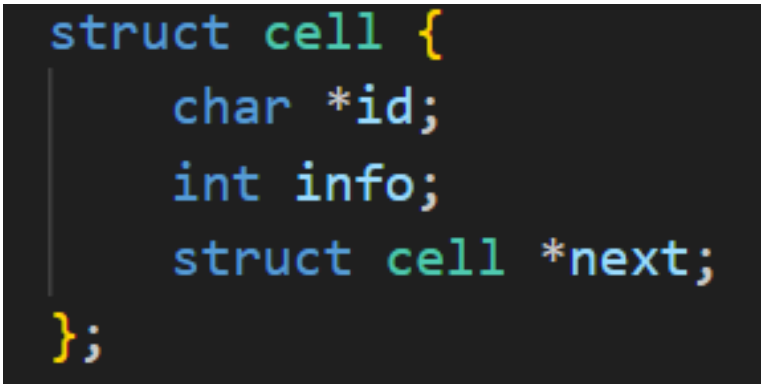
提出日 2025 年 12 月 14 日

3 年 15組 91番 157R257501

SEO BEOMGYO

はじめに

本章では、単方向リストを用いて同様の表を実装し、挿入の方法やリスト構造の工夫によって探索効率がどのように変化するかを調べた。リストの各要素は



という構造体で表し、id にキーとなる文字列、info に整数情報を格納し、next で次のセルを指す。文字列の実体は前回と同様に 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セルに保持する方式とした。

まず、小規模なデータに対してリストの基本的な挿入・探索・表示処理を実装し、その後 Shakespeare テキスト shaks12.txt を用いて、大量データに対する線形探索と自己再構成リスト、sorted list の挙動を比較した。

課題1・2

目的

構造体 struct cell とポインタ head を用いて単方向リストによる表を実装し、基本的な挿入処理 insert_table と探索処理 search_table、およびリスト内容を表示する print_list を完成させることを目的とした。

実装要点

```c
int char_top = 0;
char char_heap[MAX_CHAR];

struct cell {
    char *id;
    int info;
    struct cell *next;
};
struct cell *head;

char *save_string(char *s)
{
    int result = char_top;

    for (;;) {
        if (char_top >= MAX_CHAR) {
            printf("string buffer overflow\n");
            exit(1);
        }
        char_heap[char_top] = *s;
        char_top++;
        if (*s == 0)
            break;
        s++;
    }
    return &char_heap[result];
}
```

文字列管理は前章と同様に、char_heap[MAX_CHAR] とグローバル変数 char_top を用いる。save_string 関数は、終端文字 '¥0' を含めて文字列を char_heap にコピーし、コピー先の先頭アドレスを返す。バッファがあふれる場合は "string buffer overflow¥n" を出力して異常終了とした。

```c
void initialize_table(void)
{
    head = NULL;
}

void insert_table(char *id, int info)
{
    struct cell *p;

    p = new_cell();
    p->id = save_string(id);
    p->info = info;
    p->next = head;
    head = p;
}
```

initialize_table では head = NULL とし、リストが空であることを表す。

insert_table は、new_cell で malloc により新しいセルを確保し、id に save_string(id) の返り値、info に引数の整数を設定し、next に現在の head を代入した後、head 自身を新しいセルに更新する。これにより、新しいデータが常にリストの先頭に挿入される。

```
63  void print_list(void)
64  {
65      struct cell *p;
66
67      p = head;
68      while (p != NULL) {
69          printf("%s %d\n", p->id, p->info);
70          p = p->next;
71      }
72  }
```

print_list は head から順に next をたどりながら、各セルの id と info を一行ずつ出力する簡単なループとした。

```
74  struct cell *search_table(char *id)
75  {
76      struct cell *p;
77
78      p = head;
79      while (p != NULL) {
80          if (strcmp(id, p->id) == 0)
81              return (p);
82          p = p->next;
83      }
84      return (NULL);
85  }
```

search_table は線形探索であり、ポインタ p を head から NULL になるまで進めながら strcmp(id, p->id) で文字列を比較し、一致したセルのポインタを返す。見つからなかった場合は NULL を返す。

テストと実行結果

main では、配列版と同じく 6 人の学者を順に insert_table で挿入し、最後に ”Newton” を search_table で探索して結果を表示する。

print_list を有効にして実行すると、リストの内容が常に先頭への挿入順と逆順になることが確認できる。例えば ”Copernicus” を最初に挿入し、最後に ”Heisenberg” を挿入した場合、リストの先頭には ”Heisenberg” が位置し、末尾に ”Copernicus” が位置する。search_table(”Newton”) の結果として生年 1643 が正しく出力されることから、ポインタを用いたリスト構造が正しく構築されていると判断した。

課題3

目的

課題1・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。これにより、静的なテストデータではなくインタラクティブな利用場面においても、リスト構造が破綻せずに維持されることを確認する。

実装要点

データ構造と save_string、 new_cell、 insert_table、 search_table は課題 1・2 を踏襲しつつ、リストの要素数を保持するグローバル変数 count を導入し、print_list で ”count = %d” およびインデックス付きの出力を行うようにした。

```
49    void initialize_table(void)
50    {
51        head = NULL;
52        char_top = 0;
53        count = 0;
54    }
```

initialize_table では head = NULL に加えて char_top = 0、 count = 0 に初期化し、プログラムを何度実行しても同じ初期状態から開始できるようにした。

```
56    void print_list(void)
57    {
58        struct cell *p;
59        int i = 0;
60
61        printf("count = %d\n", count);
62        p = head;
63        while (p != NULL) {
64            printf("%d: %s %d\n", i, p->id, p->info);
65            p = p->next;
66            i++;
67        }
68        printf("----\n");
69    }
```

print_list では、p を先頭から順にたどりながら、i: id info の形式で各要素を表示し、最後に ”----” を出力する。これにより、表の状態を視覚的に把握しやすくした。

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

        if (command[0] == 'I') {
            if (scanf("%d", &year) != 1) {
                printf("年の入力エラーです。\n");
                break;
            }
            p = search_table(name);
            if (p != NULL) {
                printf("%s はすでに登録されています。\n", name);
            } else {
                insert_table(name, year);
            }
        } else if (command[0] == 'S') {
            p = search_table(name);
            if (p == NULL) {
                printf("%s は登録されていません。\n", name);
            } else {
                printf("%d\n", p->info);
            }
        } else {
            printf("不明なコマンドです: %s\n", command);
        }
    }
```

main 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む。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” を出力する。

I コマンドでは続けて整数 year を読み込み、入力に失敗したときは ”年の入力エラーです。¥n” と表示してループを抜ける。成功した場合は search_table で重複チェックを行い、既に登録されていれば ”〇〇 はすでに登録されています。”
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。”、 見つかった場合は info を出力する。

テストと実行結果

複数の I と S コマンドを混在させた入力を与え、重複挿入が正しく拒否されること、探索に成功した場合のみ生年が出力されることを確認した。挿入のたびに print_list が呼び出されるため、リストの先頭に新しく挿入されたデータが追加され、既存の要素が後ろへずれていく様子が出力から容易に追跡できる。

課題4

目的

課題3では、常にリストの先頭へ新しいデータを挿入する方式を採用した。しかし、先頭挿入では挿入順とリストの順序が一致しないため、「最後に登録したデータがリストの最後に来る」ような仕様を実現しにくい。課題4では、挿入位置をリストの末尾とし、かつ毎回 O(1) で挿入できるようにすることを目的とした。

実装要点

先頭ポインタ head に加えて、リストの最後のセルを指す tail を導入した。initialize_table では両者を NULL に初期化する。

```
73  void insert_table(char *id, int info)
74  {
75      struct cell *p;
76
77      p = new_cell();
78      p->id = save_string(id);
79      p->info = info;
80      p->next = NULL;
81
82      if (head == NULL) {
83          head = p;
84          tail = p;
85      } else {
86          tail->next = p;
87          tail = p;
88      }
89
90      count++;
91
92      print_list();
93  }
```

insert_table では、新しく確保したセル p の next を常に NULL とし、リストが空（head == NULL）の場合は head と tail をともに p に設定する。そうでない場合は tail->next = p として現在の末尾の後ろにつなぎ、tail = p に更新することで、リスト末尾への挿入を O(1) で実現した。

search_table と main の処理・メッセージは課題3と同一であり、利用者から見たインタフェースは変えていない。

テストと実行結果

挿入を複数回行った後に print_list の出力を確認すると、配列版と同様に、リストの順序が常に挿入順と一致することが分かる。先頭挿入版と比較して、末尾挿入版では新しいデータがリストの最後に追加されるため、履歴として見たときの直感的な理解がしやすい構造になっていると感じた。

課題5

目的

リストを用いた線形探索に対して自己再構成の考え方を適用し、探索成功時に見つかった要素をリストの先頭に移動することで、頻出データに対する平均探索回数を減らすことを目的とした。これは self-organizing list と呼ばれる典型的な手法である。

実装要点

データ構造と insert_table は、課題4の tail を持つ実装をもとに、挿入位置のみ先頭に戻した。リストが空のときに head と tail の両方を新しいセルに設定する点は維持した。

```
while (p != NULL) {
    if (strcmp(id, p->id) == 0) {
        if (prev != NULL) {
            prev->next = p->next;

            if (p == tail) {
                tail = prev;
            }
            p->next = head;
            head = p;

            print_list();
        }
        return (p);
    }
    prev = p;
    p = p->next;
}
```

search_table では、p とともに一つ前のセルを指す prev を保持しながら線形探索を行う。目的の id と一致した場合、prev が NULL（すでに先頭）のときは何もせずにポインタを返し、そうでないときは prev->next = p->next で p をリストから一度外し、

p->next = head、head = p で p をリストの先頭に移動する。

さらに、p が元々末尾だった場合は tail = prev へ更新する。

見つかったセルが先頭以外から移動した場合には print_list を呼び出し、自己再構成によりリストの順序が変化する様子を確認できるようにした。

テストと実行結果

同じ名前に対して繰り返し S コマンドで探索を行うと、最初の 1 回でその名前のセルが先頭に移動し、それ以降の print_list 出力でも常にインデックス 0 に現れることを確認した。

課題6

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し、リストを用いた三種類の探索方式の挙動を比較した。具体的には、

(1) 先頭挿入の線形探索版（6_head.c）、

(2) 末尾挿入の線形探索版（6_tail.c）、

(3) 自己再構成リスト版（6_self.c）

の 3 種類を実装した。いずれも文字列管理には char_heap と save_string を用い、struct cell に単語 id と出現回数 info を格納する。

探索アルゴリズムの負荷をより具体的に把握するため、各プログラムでは strcmp の呼び出し回数を long long int strcmp_count で数え、最後に ”strcmp = %lld” の形式で出力するようにした。

6.1 先頭挿入版（6_head.c）

実装要点

search_table は課題3と同様の純粋な線形探索であり、head から末尾まで順に strcmp を行う。

insert_table は新しい単語を常にリストの先頭に挿入し、count をインクリメントするだけの単純な実装である。リストの順序は出現順とは一致しない。

出力時には、いったん全セルを一時配列 struct item にコピーし、id をキーとする qsort で辞書順にソートしてから（info、 id）を一行ずつ表示する。

main では getchar で 1 文字ずつ読み込み、英字以外を区切りとして単語を切り出す。大文字は ’a’ - ’A’ を加算することで小文字に変換し、大文字・小文字を区別しない集計とした。

6.2 末尾挿入版（6_tail.c）

実装要点

insert_table では課題4と同じく head と tail を用い、新しいセルを常にリスト末尾に追加する。

search_table は 6_head.c と同じ線形探索であり、リスト構造の違いによって探索順序のみが変化する。

print_table は 6_head.c と同様に配列にコピーしてから qsort を用いてソートを行う。

この版では、実行時間を測るために clock_t start、 end を用い、単語処理全体を計測対象として time = %f を出力するようにした。strcmp_count も併せて表示する。

6.3 自己再構成リスト版（6_self.c）

実装要点

search_table は課題5と同様の self-organizing list を用いており、探索に成功したときに見つかったセルをリストの先頭に移動する。これにより、頻繁に出現する単語ほど先頭付近へ集まり、その後の探索で必要となる比較回数が減少することが期待される。

insert_table は新しい単語を常に先頭に挿入する。すなわち、挿入時点では頻度情報を用いず、探索成功時の自己再構成によって順序が変化する。

print_table は他の 2 版と同様に qsort を用いて id の辞書順で出力する。

テストと実行結果

（6_head.c）= 123.83s

```
   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
time = 124.077477
strcmp = 13431945410
./6_head < shaks12.txt  123.83s user 0.33s system 98% cpu 2:06.16 total
```

(6_tail.c) = 19.39s

```
   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
time = 19.369946
strcmp = 2193749775
./6_tail < shaks12.txt  19.39s user 0.05s system 99% cpu 19.551 total
```

(6_self.c) = 21.99s

```
   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
time = 22.019812
strcmp = 1065963611
./6_self < shaks12.txt  21.99s user 0.10s system 97% cpu 22.727 total
```

課題7 sorted list による探索

目的

最後に、リスト中の要素を常にキーの昇順に保つ sorted list を実装し、探索時には「探している値より大きい値が現れたらその時点で探索を打ち切る」という工夫を適用することで、単純な線形探索リストおよび自己再構成リストと比較したときの速度差を検討することを目的とした。

実装要点

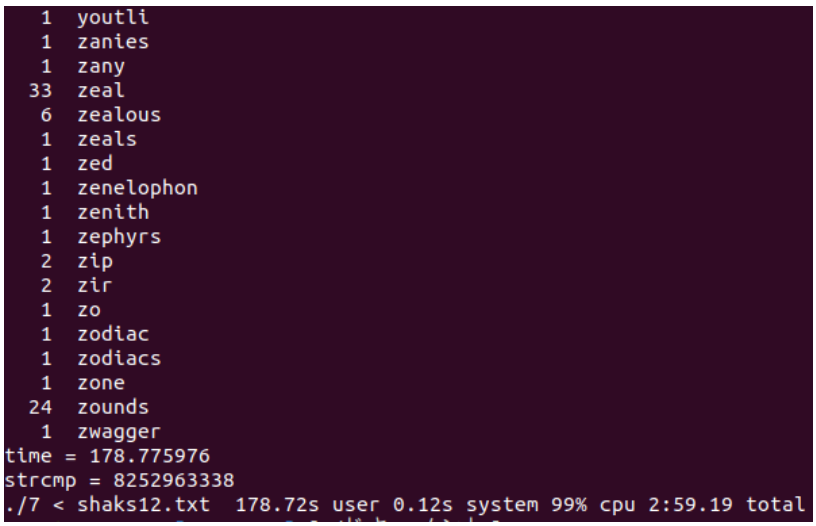
search_table は head から順に走査しながら cmp = strcmp(id、 p->id) を計算する。cmp == 0 の場合にポインタを返す点は通常の線形探索と同じであるが、cmp < 0（探している単語よりも大きい単語に達した場合）には、それ以降に目的の単語が現れないことが分かるのでその場で NULL を返して探索を打ち切る。この早期終了によって平均比較回数の削減を狙った。

insert_table では、挿入すべき位置を線形に探索し、id が昇順になるように新しいセルを挿入する。具体的には、先頭が空か、先頭の id より小さい場合は先頭に挿入し、それ以外の場合は prev と p を進めながら id > p->id の間ループし、prev と p の間に新しいセルを挿入する。

print_table は他の版と同様に一度配列にコピーしてから qsort によって辞書順に整列し、(出現回数、 単語）を表示する。リスト自体も昇順で維持されているため、本来は qsort を省略することも可能であるが、今回は他のプログラムと形式を揃えるために同じ手順を用いた。

テストと実行結果

（7.c）= 178.72s



考察

本演習では，配列による表実装をリストへ置き換え，挿入位置やリスト再構成の戦略を変えることで探索アルゴリズムの挙動がどのように変化するかを実験的に検討した。前章の配列版では，メモリ上で連続した領域を利用する代わりに，要素の挿入や削除を行う際に大規模なシフトが必要となるという欠点があった。リスト構造では，malloc とポインタの付け替えによって局所的な変更だけで要素の追加・移動を行うことができ，特に自己再構成や sorted list のような手法を自然に実装できる利点がある。