データ構造とアルゴリズム実験レポート
課題：連結リスト, スタック, キュー

202110796 4 クラス 高橋大粋

締切日:2024 年 11 月 7 日
2024 年 11 月 7 日

1　基本課題

この課題では、教科書リスト 3.2(p.54) のオープンアドレス法を用いた辞書の C プログラム open_addressing.c のリストおよび実行結果を示した。

1.1　オープンアドレス法を用いた辞書の実装

1.1.1　実装の方針

まず、オープンアドレス法を用いた辞書の機能としてサイズ len の辞書配列を作成・初期化し、処理が完了したら、DictOpenAddr 型のポインタを返す create_dict(), ハッシュ関数 h(), データ d を辞書に挿入する insert_hash(), データ d が辞書に含まれているかを探索し、該当データが存在するなら d が格納されている配列要素のインデックスを返し、存在しなければ-1 を返す search_hash(), データ d を辞書から削除し、辞書内に該当するデータが存在しなければ何もしない delete_hash(), 辞書配列の要素をすべて表示する display(), 辞書を破棄する delete_dict() を open_addressing.c に実装した。また、main 関数は別ファイル main_open_addressing.c に実装した。

1.1.2　実装コードおよびコードの説明

プログラム 1 に、open_addressing.c の主要部を示す。

DictOpenAddr *create_dict(int len) 関数は入力として int 型のデータ len を受け取り、新しい辞書 dict を作成し、dict のメンバ変数に初期値を代入する。B はバケット数、H はデータ name と状態 state を保持するための配列で初期値は name=0,state=EMPTY としてある。2 行目では DictOpenAddr サイズ分のメモリ領域を動的に確保し、そのアドレスをポインタ dict に格納している。メモリの確保に失敗した際は標準エラーを出力してプログラムを終了する。int h(DictOpenAddr *dict, int d, int count) 関数は衝突が生じた際に再ハッシュするために $h_i(d) = (h(d) + i) mod B (i = 1, 2, ...)$ という関数を実装した。h_0 には $dmodB$ を設定した。int search_hash(DictOpenAddr *dict, int d) 関数は教科書リスト 3.2 のコードを元に条件式をわかりやすくまとめて実装した。探索したいデータを d を受け取り、線形走査法を用いて探索し、辞書のデータの state が OCCUPIED かつ name が d なら探索成功とし、ハッシュ値を返す。もし辞書のデータの state が EMPTY なら探索終了で-1 を返す。最後の return -1 は無限ループを防ぐためのもの。void

insert_hash(DictOpenAddr *dict, int d) 関数はまず挿入したいデータ d を search_hash() で探索し、見つかれば格納済みなので何もせず終了。見つからなければ配列の状態が EMPTY または DELETED のものを探してそこに挿入する。挿入した後は配列の状態を OCCUPIED にして終了。もし配列に空きがない場合は標準エラーを出力してプログラムを終了する。void delete_hash(DictOpenAddr *dict, int d) 関数は消したいデータを search_hash() で探索し、見つからなければ削除する必要がないので終了、見つかれば配列の状態を DELETED にして終了。void display(DictOpenAddr *dict) 関数は switch 文を使って配列の状態ごとに出力する内容を分けてある。OCCUPIED ならデータを、EMPTY なら e を、DELETED なら d を出力する。これを配列の要素すべてに適用する。void delete_dict(DictOpenAddr *dict) 関数は malloc で確保した分のメモリを開放している。

```
1   DictOpenAddr *create_dict(int len){
2       DictOpenAddr *dict = (DictOpenAddr *)malloc(sizeof(DictOpenAddr));
3       if (dict == NULL) {
4           fprintf(stderr, "Memory allocation failed for dictionary.\n");
5           exit(EXIT_FAILURE);
6       }
7       dict->H = (DictData *)malloc(len * sizeof(DictData));
8       if (dict->H == NULL) {
9           fprintf(stderr, "Memory allocation failed for hash table.\n");
10          free(dict);
11          exit(EXIT_FAILURE);
12      }
13      dict->B = len;
14      for (int i = 0; i < len; i++) {
15          dict->H[i].name = 0;
16          dict->H[i].state = EMPTY;
17      }
18      return dict;
19  }
20
21  int h(DictOpenAddr *dict, int d, int count){
22      int B = dict->B;
23      int h_0 = d % B;
24      return (h_0 + count) % B;
25  }
26
27  void insert_hash(DictOpenAddr *dict, int d){
28      if (search_hash(dict, d) != -1) {
29          return;
30      }
```

```
31          int count = 0;
32          int b = h(dict, d, count);
33          int init_b = b;
34          do {
35              if (dict->H[b].state == EMPTY || dict->H[b].state == DELETED) {
36                  dict->H[b].name = d;
37                  dict->H[b].state = OCCUPIED;
38                  return;
39              }
40              count++;
41              b = h(dict, d, count);    //rehush
42              if (count >= dict->B) {
43                  fprintf(stderr, "ERROR: Hash table is full. Cannot insert %
                        d.\n", d);
44                  exit(EXIT_FAILURE);
45              }
46          } while (b != init_b);
47          return;
48      }
49
50      int search_hash(DictOpenAddr *dict, int d){
51          int count = 0;
52          int b = h(dict, d, count);
53          int init_b = b;
54          do {
55              if(dict->H[b].state == OCCUPIED && dict->H[b].name == d) {
56                  return b;
57              }
58              else if (dict->H[b].state == EMPTY) {
59                  return -1;
60              }
61              count++;
62              b = h(dict, d, count);    //rehush
63          } while (b != init_b);
64          return -1;
65      }
66
67      void delete_hash(DictOpenAddr *dict, int d){
68          int b = search_hash(dict, d);
69          if (b == -1) {
```

```
70              return;
71          }
72          dict->H[b].state = DELETED;
73      }
74
75      void display(DictOpenAddr *dict){
76          for (int i = 0; i < dict->B; i++) {
77              switch(dict->H[i].state){
78                  case OCCUPIED:
79                      printf("%d ", dict->H[i].name);
80                      break;
81                  case EMPTY:
82                      printf("e ");
83                      break;
84                  case DELETED:
85                      printf("d ");
86                      break;
87              }
88          }
89          printf("\n");
90      }
91
92      void delete_dict(DictOpenAddr *dict){
93          if (dict != NULL) {
94              free(dict->H);
95              free(dict);
96          }
97      }
```

プログラム 1　open_addressing.c の主要部

1.1.3　実行結果

まず、main_open_addressing.c を以下に示す。

```
1       int main(void) {
2       //test1
3       DictOpenAddr *test_dict = create_dict(10);
4
5       insert_hash(test_dict, 1);
6       insert_hash(test_dict, 2);
7       insert_hash(test_dict, 3);
```

```
    insert_hash(test_dict, 11);
    insert_hash(test_dict, 12);
    insert_hash(test_dict, 21);
    display(test_dict);

    printf("Search 1 ...\t%d\n", search_hash(test_dict, 1));
    printf("Search 2 ...\t%d\n", search_hash(test_dict, 2));
    printf("Search 21 ...\t%d\n", search_hash(test_dict, 21));
    printf("Search 5 ...\t%d\n", search_hash(test_dict, 5));

    delete_hash(test_dict, 3);
    display(test_dict);

    delete_hash(test_dict, 11);
    display(test_dict);

    delete_dict(test_dict);

    //test2
    // DictOpenAddr *test_dict = create_dict(5);

    // insert_hash(test_dict, 1);
    // insert_hash(test_dict, 2);
    // insert_hash(test_dict, 3);
    // insert_hash(test_dict, 4);
    // insert_hash(test_dict, 5);
    // display(test_dict);

    // printf("Attempting to insert into a full hash table...\n");
    // insert_hash(test_dict, 6);

    // delete_dict(test_dict);

    // return EXIT_SUCCESS;
}
```

次に、open_addressing.c を以下の make コマンドを実行してコンパイルし、プログラムを実行する。今回実行したのは test1 のみ。

```
    PS C:\Users\daiki\Desktop\DSA\kadai3> make
    gcc    -c -o main_open_addressing.o main_open_addressing.c
```

```
3     gcc   open_addressing.o main_open_addressing.o   -o open_addressing
4     PS C:\Users\daiki\Desktop\DSA\kadai3> ./open_addressing
5     e 1 2 3 11 12 21 e e e
6     Search 1 ...    1
7     Search 2 ...    2
8     Search 21 ...   6
9     Search 5 ...    -1
10    e 1 2 d 11 12 21 e e e
11    e 1 2 d d 12 21 e e e
```

まず、create_dict で要素数 10 の辞書を作成した。insert_hash で 1,2,3,11,12,21 を配列に格納している。1 と 11 と 21,2 と 12 は配列の要素数が 10 なのでハッシュ値が同じになり、衝突が起きる。しかし、insert_hash の中で再ハッシュを行っているので正しく格納されていることが確認できる。次に search_hash で 1,2,21,5 を探索した結果、1,2,21 は正しいインデックスが返され、配列に含まれていない 5 は-1 が返されているので正しく探索できている。delete_hash で 3 を削除した後、続けて 11 を削除した。どちらも d に変わっているため正しく削除できている。最後に delete_dict で辞書を削除した。

次に test2 を以下の make コマンドで実行した。

```
1     PS C:\Users\daiki\Desktop\DSA\kadai3> make
2     gcc    -c -o main_open_addressing.o main_open_addressing.c
3     gcc   open_addressing.o main_open_addressing.o   -o open_addressing
4     PS C:\Users\daiki\Desktop\DSA\kadai3> ./open_addressing
5     5 1 2 3 4
6     Attempting to insert into a full hash table...
7     ERROR: Hash table is full. Cannot insert 6.
```

test2 では配列を満杯にし、その状態で insert_hash を行うとエラーを吐くのかテストした。まず、要素数 5 の辞書を作成し、insert_hash で 1,2,3,4,5 を挿入した。その状態で insert_hash で 6 を挿入しようとすると配列に空きがないため、エラーメッセージが出力され、プログラムが終了した。以上により、辞書に整数が格納できること、格納した整数を探索できること、同じハッシュ値を持つ整数を、それぞれ他の配列要素に格納できること、同じハッシュ値を持つ複数の整数を探索できること、データを挿入しようとして配列に空きがなく挿入できないときは、標準エラー出力にエラーメッセージを表示し、exit 関数を用いてプログラムを異常終了することを満たすことが確認できた。なお、以上のプログラムは 1.1.2 で説明したように動作する。

2 発展課題

この課題では、教科書コラム (p.56) に基づいて、2 重ハッシュ法を用いてオープンアドレス法を用いた辞書を double_hashing.c に実装し、search_hash の実行時間を計測し、グラフにプロットした。

2.1 オープンアドレス法を用いた辞書 (2 重ハッシュ法) の実装

2.1.1 実装の方針

まず、オープンアドレス法を用いた辞書 (2 重ハッシュ法) の機能として、異なる 2 つのハッシュ関数 h(d),g(d) を用いた 2 重ハッシュ法の関数 h(), その他 1.1.1 で実装した関数を double_hashing.c に実装した。また、main 関数は別ファイル main_double_hashing.c に実装した。

2.1.2 実装コードおよびコードの説明

プログラム 2 に、double_hashing.c の主要部を示す。

int g(int d, int B) 関数は g(d) を $1 + (dmod(B-1))$ としている。これは g(d) が出力するハッシュ値とハッシュ表のサイズ B が互いに素になるように設定した関数である。互いに素でなければすべてのバケットを調べ尽くすことができないので g(d) の値が 0 にならないように +1 をしてある。また、ハッシュ表のサイズ B は素数に設定してある。int h(DictDoubleHashing *dict, int d, int count) 関数は教科書コラムの再ハッシュ関数 $h_i(d) = (h(d) + i * g(d))modB(i = 1, 2, ...)$ を $h(d) = dmodB, g(d) = 1 + (dmod(B-1))$ として実装した。これにより h(d) に対して同じハッシュ値を持つ 2 つのデータが g(d) に対しても同じハッシュ値を持つことはほとんど生じない。

```
1   DictDoubleHashing *create_dict(int len){
2       DictDoubleHashing *dict = (DictDoubleHashing *)malloc(sizeof(
            DictDoubleHashing));
3       if (dict == NULL) {
4           fprintf(stderr, "Memory allocation failed for dictionary.\n");
5           exit(EXIT_FAILURE);
6       }
7       dict->H = (DictData *)malloc(len * sizeof(DictData));
8       if (dict->H == NULL) {
9           fprintf(stderr, "Memory allocation failed for hash table.\n");
10          free(dict);
11          exit(EXIT_FAILURE);
12      }
13      dict->B = len;
14      for (int i = 0; i < len; i++) {
15          dict->H[i].name = 0;
16          dict->H[i].state = EMPTY;
17      }
18      return dict;
19  }
20
21  int g(int d, int B) {
22      return 1 + (d % (B - 1));
```

```
23      }
24
25      int h(DictDoubleHashing *dict, int d, int count) {
26          int B = dict->B;
27          int h_0 = d % B;
28          int hash_value = (h_0 + count * g(d, B)) % B;
29          return hash_value;
30      }
31
32      void insert_hash(DictDoubleHashing *dict, int d) {
33          if (search_hash(dict, d) != -1) {
34              return;
35          }
36          int count = 0;
37          int b = h(dict, d, count);
38          int MAX_REHASH_ATTEMPTS = dict->B;
39
40          while (count < MAX_REHASH_ATTEMPTS) {
41              if (dict->H[b].state == EMPTY || dict->H[b].state == DELETED) {
42                  dict->H[b].name = d;
43                  dict->H[b].state = OCCUPIED;
44                  return;
45              }
46              count++;
47              b = h(dict, d, count);
48          }
49          fprintf(stderr, "ERROR: Hash table is full or rehash limit exceeded
                . Cannot insert %d.\n", d);
50          exit(EXIT_FAILURE);
51      }
52
53      int search_hash(DictDoubleHashing *dict, int d) {
54          int count = 0;
55          int b = h(dict, d, count);
56          int MAX_REHASH_ATTEMPTS = dict->B;
57
58          while (count < MAX_REHASH_ATTEMPTS) {
59              if (dict->H[b].state == OCCUPIED && dict->H[b].name == d) {
60                  return b;
61              } else if (dict->H[b].state == EMPTY) {
```

```
62                  return -1;
63              }
64              count++;
65              b = h(dict, d, count);
66          }
67          return -1;
68      }
69
70      void delete_hash(DictDoubleHashing *dict, int d){
71          int b = search_hash(dict, d);
72          if (b == -1) {
73              return;
74          }
75          dict->H[b].state = DELETED;
76      }
77
78      void display(DictDoubleHashing *dict){
79          for (int i = 0; i < dict->B; i++) {
80              switch(dict->H[i].state){
81                  case OCCUPIED:
82                      printf("%d ", dict->H[i].name);
83                      break;
84                  case EMPTY:
85                      printf("e ");
86                      break;
87                  case DELETED:
88                      printf("d ");
89                      break;
90              }
91          }
92          printf("\n");
93      }
94
95      void delete_dict(DictDoubleHashing *dict){
96          if (dict != NULL) {
97              free(dict->H);
98              free(dict);
99          }
100     }
```

プログラム 2　double_hashing.c の主要部

2.1.3　実行結果

まず、main_double_hashing.c を以下に示す。main 関数では実験ページのサンプルの timer を windows 環境のため、QueryPerformanceCounter を用いて実装した。分解能は 1 マイクロ秒である。ハッシュ表のサイズは 5003 で、データの値の範囲は 1 から 10000 である。また、search_hash の実行時間を計測する際のゆらぎをなくすために、MAX_SEARCH を 10000 に設定している。これにより、実行時間の精度が上がる。計測結果は csv ファイル search_time.csv に保存する。

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4   #include <windows.h>
5   #include "double_hashing.h"
6
7   #define B  5003
8   #define MAX_SEARCH 10000
9
10  typedef struct timer {
11      double seconds;
12      LARGE_INTEGER start_time;
13      LARGE_INTEGER frequency;
14      void (*reset)(struct timer *this_timer);
15      void (*start)(struct timer *this_timer);
16      void (*stop)(struct timer *this_timer);
17      double (*result)(struct timer *this_timer);
18  } Timer;
19
20  void timer_reset(Timer *this_timer) {
21      this_timer->seconds = 0.0;
22      QueryPerformanceFrequency(&this_timer->frequency);
23  }
24  void timer_start(Timer *this_timer) { QueryPerformanceCounter(&
        this_timer->start_time); }
25  void timer_stop(Timer *this_timer) {
26      LARGE_INTEGER end_time;
27      QueryPerformanceCounter(&end_time);
28      this_timer->seconds += (double)(end_time.QuadPart - this_timer->
            start_time.QuadPart) / this_timer->frequency.QuadPart;
```

```
29      }
30      double timer_result(Timer *this_timer) { return this_timer->seconds; }
31
32      int main() {
33          Timer stop_watch = {0.0, {0}, {0}, timer_reset, timer_start,
                timer_stop, timer_result};
34          stop_watch.reset(&stop_watch);
35
36          FILE *fp = fopen("search_time.csv", "w");
37          if (fp == NULL) {
38              fprintf(stderr, "Error: Unable to create file.\n");
39              return EXIT_FAILURE;
40          }
41
42          DictDoubleHashing *dict = create_dict(B);
43          if (dict == NULL) {
44              fprintf(stderr, "Error: Unable to create dictionary.\n");
45              fclose(fp);
46              return EXIT_FAILURE;
47          }
48
49          int *data = (int *)malloc(B * sizeof(int));
50          if (data == NULL) {
51              fprintf(stderr, "Error: Memory allocation failed.\n");
52              delete_dict(dict);
53              fclose(fp);
54              return EXIT_FAILURE;
55          }
56          srand((unsigned)time(NULL));
57
58          for (int i = 0; i < B; i++) {
59              data[i] = ((rand() << 15) | rand()) % (B*10000) + 1;
60          }
61          fprintf(fp, "Occupancy,Time(sec)\n");
62
63          for (double occ = 0.0; occ <= 1.0; occ += 0.1) {
64              int insert_count = (int)(occ * B);
65              if(insert_count != 0){
66
67                  delete_dict(dict);
```

```
68                  dict = create_dict(B);
69
70                  for (int i = 0; i < insert_count; i++) {
71                      insert_hash(dict, data[i]);
72                  }
73
74                  stop_watch.reset(&stop_watch);
75                  stop_watch.start(&stop_watch);
76                  for (int i = 0; i < MAX_SEARCH; i++) {
77                      int target = data[rand() % insert_count];
78                      search_hash(dict, target);
79                  }
80                  stop_watch.stop(&stop_watch);
81
82                  double avg_time = stop_watch.result(&stop_watch) /
                        MAX_SEARCH;
83                  printf("Occupancy: %.1f, Average Search Time: %.11f sec\n",
                         occ, avg_time);
84                  fprintf(fp, "%.1f,%.10f\n", occ, avg_time);
85              } else {
86
87                  double avg_time = 0;
88                  printf("Occupancy: %.1f, Average Search Time: %.11f sec\n",
                         occ, avg_time);
89                  fprintf(fp, "%.1f,%.10f\n", occ, avg_time);
90              }
91          }
92
93          free(data);
94          delete_dict(dict);
95          fclose(fp);
96          printf("Program finished. Results saved to search_time.csv\n");
97          return EXIT_SUCCESS;
98  }
```

次に、double_hashing.c を以下の make コマンドを実行してコンパイルするために Makefile に以下の行を追加した。

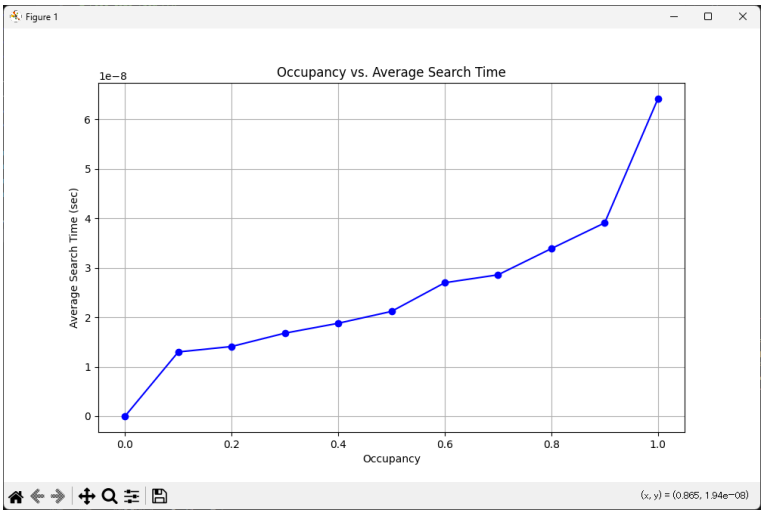
```
1      CC:=gcc
2      open_addressing: open_addressing.o main_open_addressing.o
3      double_hashing: double_hashing.o main_double_hashing.o          #add
```

```
4         gcc double_hashing.o main_double_hashing.o -o double_hashing -
              lwinmm
```

make コマンドでコンパイルし、プログラムを実行した結果は以下のようになった。

```
1     PS C:\Users\daiki\Desktop\DSA\kadai3> make double_hashing
2     gcc    -c -o double_hashing.o double_hashing.c
3     gcc    -c -o main_double_hashing.o main_double_hashing.c
4     gcc double_hashing.o main_double_hashing.o -o double_hashing -lwinmm
5     PS C:\Users\daiki\Desktop\DSA\kadai3> ./double_hashing
6     Occupancy: 0.0, Average Search Time: 0.00000000000 sec
7     Occupancy: 0.1, Average Search Time: 0.00000001299 sec
8     Occupancy: 0.2, Average Search Time: 0.00000001415 sec
9     Occupancy: 0.3, Average Search Time: 0.00000001679 sec
10    Occupancy: 0.4, Average Search Time: 0.00000001881 sec
11    Occupancy: 0.5, Average Search Time: 0.00000002122 sec
12    Occupancy: 0.6, Average Search Time: 0.00000002703 sec
13    Occupancy: 0.7, Average Search Time: 0.00000002859 sec
14    Occupancy: 0.8, Average Search Time: 0.00000003391 sec
15    Occupancy: 0.9, Average Search Time: 0.00000003909 sec
16    Occupancy: 1.0, Average Search Time: 0.00000006425 sec
17    Program finished. Results saved to search_time.csv
```

各占有率に対して search_hash を 10000 回繰り返し、実行時間の平均を小数以下第 11 位まで表示させた。占有率 0.9 までは徐々に増加していき、占有率 1.0 のときは急激に増加していることが読み取れる。以下にこれらのデータをグラフにプロットしたものを示す。グラフを見てもやはり占有率がある程度小さい状況では時間



計算量は O(1) で、占有率 1.0 付近で急激に増加していることが読み取れる。これは占有率が高くなるほど再ハッシュにより探索回数が急激に増加するためだと考えられる。