データ構造とアルゴリズム実験レポート

＜課題 3: ハッシュ法＞

202213025 – 3 クラス – Can Minh Nghia

締切日：2023/11/7

提出日：2023/11/3

1. 必須課題

課題 3-1

3-1-1 実装の仕様

データとして整数値をとるオープンアクセス法を用いた辞書を実現するﾌﾟﾛｸﾞﾗﾑ open_addressing を実装。

それに、以下の条件も満たせる。

・辞書に整数を格納でﾞきること

・格納した整数を探索でﾞきること

・同じﾞハッシュ値をもつ整数を，それぞﾞれ他の配列要素に格納でﾞきること

・同じﾞハッシュ値をもつ複数の整数を探索でﾞきること

・ﾃﾞータを挿入しようとして，配列に空きがﾞなく挿入でﾞきない時，その旨を標準エラー出力すること

3-1-2 実装コードおよび実装コードの説明

open_addressing.c

ここで、7 つの関数があるので、1 つずつ開設する。

・サイズ len の辞書配列を作成・初期化する関数

```
 5 DictOpenAddr *create_dict(int len){
 6       DictOpenAddr *new_dict = (DictOpenAddr*)malloc(sizeof(DictOpenAddr));
 7       new_dict -> H = (DictData*)malloc(sizeof(DictData)*len);
 8       new_dict -> B = len;
 9       //initial value = EMPTY
10       int pos = 0;
11       while (pos < len){
12             new_dict -> H[pos].state = EMPTY;
13             pos ++;
14       }
15       printf("%s\n", "Dictionary created!");
16       return new_dict;
17 }
```

Line 6 と 7: malloc 関数を使って、dictionary と H にメモリを与える。ここで、dict はハッシュテーブルであり H はそのテーブルに指すポインタである。

Line 8: B は配列のサイズである。

配列を作成したら、格位置の状態を初期値 EMPTY を定める。

・ハッシュ関数

```
19 int h(DictOpenAddr *dict, int d, int count){
20       int hash = (d + count) % (dict -> B);
21       return hash;
22 }
```

まず、データ d を d % B 位置に格納する。もしその位置が開いていなければ、count 変数を使って、次の空いている位置に格納する。

。

データ d が辞書内に含まれるかを探索する関数

```
24 int search_hash(DictOpenAddr *dict, int d){ // search if d has already been in the hash table, as well as find
   the position of d
25       int count = 0;
26       int b = h(dict, d, count);
27       int initial_b = b;
28
29       do{
30             if (dict -> H[b].state == OCCUPIED){
31                   if (dict -> H[b].name == d){
32                         return b; // return the positiion of d
33                   }
34             }
35             else if (dict -> H[b].state == EMPTY){
36                   return -1;
37             }
38             count += 1;
39             b = h(dict, d, count);
40       } while (b != initial_b);
41       return -1; // when state == DELETED
42 }
```

該当データが在するなら d が格納されている配列要素のインデックスを返し，存在しないなら -1 を返す。

Do-while ループを使って、ハッシュテーブルの各々位置の状態をチェックする。何かに格納されていたら（state == OCCUPIED)、それをさらにチェックする。もし格納されているものが d だったら、 d の位置(ここには b)を返す。 d ではなければ、次の位置に移動しているまたチェックする。

また、while(b != initial_b)を使うことで、無限ループが避けられる。

データ d を辞書に挿入する関数

```
45 void insert_hash(DictOpenAddr *dict, int d){
46       if (search_hash(dict, d) != -1) { //d has already been in the hash table
47             return; //do nothing
48       }
49
50       int count = 0;
51       int b = h(dict, d, count);
52       int initial_b = b;
53
54       do{
55             if (dict->H[b].state == EMPTY || dict->H[b].state == DELETED){
56                   dict->H[b].name = d;
57                   dict->H[b].state = OCCUPIED;
58                   printf("%d \n", dict->H[b].name);
59                   if (dict->H[b].state == OCCUPIED){
60                   }
61                   return;
62             }
63             count += 1;
64             if (count >= dict -> B){
65                   printf("Dictionary is full, cannot insert! Already inserted %d numbers! \n", count);
66                   exit(EXIT_FAILURE); //when dictionary is full, stop inserting
67             }
68             b = h(dict, d, count);
69       } while (b != initial_b);
70       return;
71 }
```

Line 46: 上記の search_hash 関数は、データ d の位置「b」、または「-1」という 2 つの価値しか返さない。ということで、search_hash 関数が返す価値が-1 でなければ、それは b となるに違いない。つまり、d が既にハッシュテーブルの位置 b に存在しているということだ。従って、挿入不要。そのまま関数が終わる。

Line 54 ~ 69: do-while ループを使って、ハッシュテーブルに空いている位置が見つかったら、データ d をそこに挿入する。挿入後、その位置の状態を初期値 EMPTY から OCCUPIED に変更する。

注意してほしいのは、line 64 ~ 66 である。

```
63            count += 1;
64            if (count >= dict -> B){
65                  printf("Dictionary is full, cannot insert! Already inserted %d numbers! \n", count);
66                  exit(EXIT_FAILURE); //when dictionary is full, stop inserting
67            }
```

それは課題の条件

> 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時，その旨を標準エラー出力すること

を満たせるように定めた。

データ d を辞書から削除する関数:

```
73 void delete_hash(DictOpenAddr *dict, int d){
74       int b = search_hash(dict, d);
75       if (b != -1) { //d is in the hash table
76             dict -> H[b].state = DELETED;
77
78       return;
79       }
80       else { //d is not in the hash table
81             return;
82       }
83 }
```

Line 75: search_hash 関数が-1 を返さなければ、ハッシュテーブルにある d の位置を返すこととなる。つまり、line 75 では、 b が d の位置だ。 d を消したので、 b の状態を DELETED と変更する。ここで、本当に d を消すことではなく、ただ状態を変えるだけだ。しかし、状態が DELETED であれば、空の位置とみなせて、ほかのデータを挿入できる。

辞書配列の要素を全て表示する関数:

```
85 void display(DictOpenAddr *dict){
86        int pos = 0;
87
88        while (pos < dict -> B){
89              State s = dict -> H[pos].state;
90              if (dict -> H[pos].state == OCCUPIED){
91                    printf ("%d ", dict -> H[pos].name);
92              }
93
94              else if (dict -> H[pos].state == EMPTY){
95                    printf ("%s ", "e");
96              }
97
98              else { //DELETED
99                    printf ("%s ", "d");
100             }
101
102             pos ++;
103       }
104       printf ("\n");
105 }
```

具体的には，辞書配列の要素の状態が OCCUPIED なら値を表示，EMPTY なら空であることを示す文字'e'を表示，DELETED ならデータが去されたことを示す文字'd'を表示する.

ここで、pos はテーブルの位置である。

辞書配列を破棄関数:

```
107 void delete_dict(DictOpenAddr *dict){
108         free (dict -> H);
109         free (dict);
110         printf("%s\n", "Dictionary deleted!");
111 }
```

Free 関数を使って、与えたメモリを廃棄する。

3-1-3　実行結果

```
azalea02:~ s2213025$ ./open_addressing
Dictionary created!
1
2
3
11
12
21
e 1 2 3 11 12 21 e e e
Search 1 ...    1
Search 2 ...    2
Search 21 ...   6
Search 5 ...    -1
e 1 2 d 11 12 21 e e e
e 1 2 d d 12 21 e e e
Dictionary deleted!
```

授業のスライドの結果とあっているので、実行が成功した。

3-1-4　感想

insert_hash 関数には、最初のところ、dict -> H[b].name を使わず、dict -> H[b] -> name を使った。しかし、実行が失敗した。'->'を使うか '.'を使うか、注意しないといけないことが分かった。

2. 発展課題

課題 3-2

3-2-1 実装の仕様

再ハッシュのためのハッシュ関数として，線形走査法を使用したﾌﾟﾛｸﾞﾗム linear_probing と 2 重ハッシュ法（教科書 p. 56 コラムを参照）を使用したﾌﾟﾛｸﾞﾗム double_hashing を作成し，占有率α(登録したﾃﾞｰﾀ数/全ﾊｹｯﾄ数)ｶﾞ 0，0.1，0.2, , , 1.0 の場合における探索に要する実行時間をそれぞれのプログラム内で調べ，結果をグラフを用いて分析しなさい.

3-1-2 実装コードおよび実装コードの説明

ここで、基本課題で定めた **open_addressing プログラム**を使って、linear_probing と double_hashing を作成する。後 **main_open_addressing プログラム**も変更する。課題が与えた dict-sample.txt も使って、number.txt に命名する。

・**linear_probing** は、**open_addressing プログラム**である。
open_addressing プログラムを命名したら、linear_probing が得られる。

・**double_hashing** は、linear_probing とほとんど同じだ。ただし、ハッシュ関数をこのように変更する。

```
19 int h(DictOpenAddr *dict, int d, int count){
20        int hash = (d % (dict -> B) + count*(7 - d % 7)) % (dict -> B);
21        return hash;
22 }
```

ここで注意してほしいのは、教科書に書いてある h(d) + i*g(d)のように関数を定めては**いけない**ことだ。なぜかというと、d % dict -> B の結果は 0 から B – 1 まで、count*(7 – d % 7)も count の値によって B より大きい可能性もある。そのため、h(d) + i*g(d)と後必ずもう一度 '%(dict -> B)'を書く。そうしたら、ハッシュ関数が返す値が B より小さい。

・**main_open_addressing**

実は、課題が与えた main_open_addressing が変更されたものだ。Makefile の操作が詳しくないので、プログラムの名前はそのままにする。

変更されたところ：

・number.txt に 10000 個の整数があるから create_dict 関数の len を 10000 と定める

```
7 int main(void) {
8   DictOpenAddr *test_dict = create_dict(10000);
```

・ファイルから数と読み取り、録したデータ数/全バケット数)が 0.1, 0.2, , , 1.0 の場合の実行時間を計算して返す。

```
10  const char *filename = "number.txt";
11  char line[100]; // Assuming each line has at most 100 characters
12  FILE *file = fopen(filename, "r");
13    if (file == NULL) {
14        perror("Error opening file");
15        return 1;
16    }
17
18    struct timeval start, end;
19    gettimeofday(&start, NULL);
20    int linesRead = 0;
21
22     while (fgets(line, sizeof(line), file) != NULL) {
23        int num = atoi(line); // Convert the line to an integer
24        insert_hash(test_dict, num);
25
26        linesRead++;
27
28          if (linesRead % 1000 == 0) { //to get the time when 10%, 20%, 30%, etc amount of work is done
29            gettimeofday(&end, NULL);
30            double elapsed_time = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6;
31            printf("\nAfter %d: ,time %lf[sec]\n", linesRead, elapsed_time);
32         }
33    }
34
35     fclose(file);
```

Line 10 ~ 35:　FILE 型のポインタを使って、fopen 関数、fgets 関数、fclose 関数を使って number.txt の内容を読み取ってから、line 24 の insert_hash 関数で数をハッシュテーブルに挿入する。

Line 18, 19, 29: gettimeofday 関数を使って、実行が始まる時点と終わる時点を定める。

Line 20：number.txt には、1 行にあたって 1 つの整数があるから、いくら行が読み取れたかというとは、いくら整数が読み取れたかと同じだ。ここで、読み取れた整数（行）を linesRead 変数に格納する。

Line 28 ~ 31: 読み取れた整数が 1000 個(占有率 0.1)、2000 個(占有率 0.2)などの処理時間を printf 関数で表示する。

3-2-3　実行結果

・double_hashing：

```
After 1000: ,time 0.000092[sec]
```

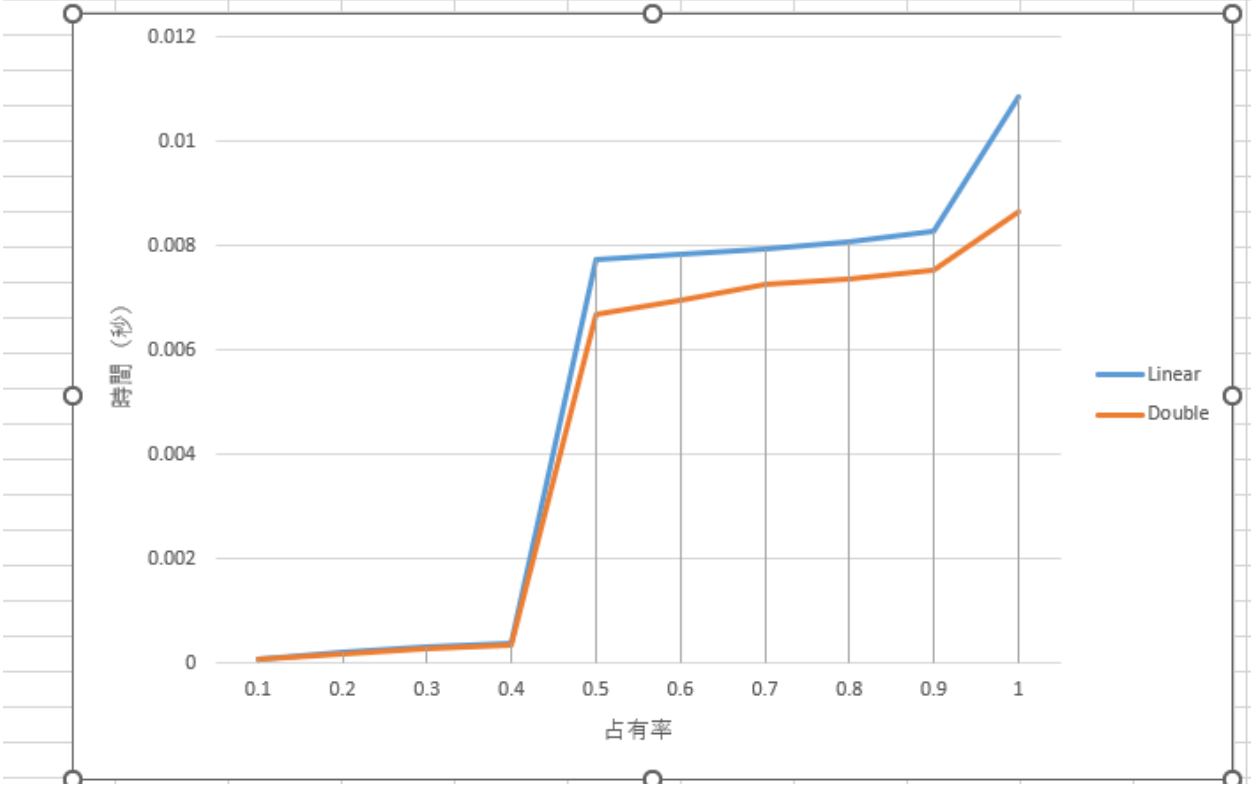
```
After 2000: ,time 0.000188[sec]

After 3000: ,time 0.000274[sec]

After 4000: ,time 0.000362[sec]

After 5000: ,time 0.006701[sec]

After 6000: ,time 0.006961[sec]

After 7000: ,time 0.007255[sec]

After 8000: ,time 0.007376[sec]

After 9000: ,time 0.007545[sec]

After 10000: ,time 0.008658[sec]
Dictionary deleted!
```

・linear_probing：

```
After 1000: ,time 0.000088[sec]

After 2000: ,time 0.000217[sec]

After 3000: ,time 0.000300[sec]

After 4000: ,time 0.000385[sec]

After 5000: ,time 0.007733[sec]

After 6000: ,time 0.007836[sec]

After 7000: ,time 0.007947[sec]

After 8000: ,time 0.008073[sec]

After 9000: ,time 0.008264[sec]
```

```
After 10000: ,time 0.010837[sec]
Dictionary deleted!
```

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Linear	0.000088	0.000217	0.0003	0.000385	0.007733	0.007836	0.007947	0.008073	0.008264	0.010837
Double	0.000092	0.000188	0.000274	0.000362	0.006701	0.006961	0.007255	0.007376	0.007545	0.008658



グラフから見ると、占有率が 0.1 から 0.4 までのときに、Linear と Double がほとんど同じだ。一方、0.4 から 0.5 までの差が大きくなる。占有率が大きくなるにつれて、Linear の方は時間がかかることがわかる。

3-2-4　感想

難しい課題だったが、ハッシュ関数が面白いと思う。