担当：長尾 智晴 (nagao@ynu.ac.jp)
濱上 知樹 (hamagami@ynu.ac.jp)

「アルゴリズムとデータ構造」講義日程

~~1. 基本的データ型~~
~~2. 基本的制御構造~~
~~3. 変数のスコープルール．関数~~
~~4. 配列を扱うアルゴリズムの基礎(1)．最大値，最小値~~
~~5. 配列を扱うアルゴリズムの基礎(2)．重複除去，集合演算，ポインタ~~
~~6. ファイルの扱い~~
~~7. 整列(1)．単純挿入整列・単純選択整列・単純交換整列~~
~~8. 整列(2)．マージ整列・クイック整列~~
~~9. 再帰的アルゴリズムの基礎．再帰におけるスコープ．ハノイの塔など．~~
~~10. バックトラックアルゴリズム．8 王妃問題など．~~
~~11. 線形リストを扱うアルゴリズム~~
~~12. 木構造を扱うアルゴリズム(1) 基礎~~
~~13. 木構造を扱うアルゴリズム(2) 挿入，削除，バランスなど．~~
14. ハッシング（座学最終回） ← 本日の内容
15. その他のアルゴリズム（自習．講義なし）

※ 試験期間中に期末試験が行われるので必ず参加しましょう．欠席の場合は「単位放棄」とみなされます．

第 14 回（座学最終回）「ハッシング 」

「キーを使ったデータ探索」の要約・復習

◎全 n 個のデータの中から、キーを指定して、同じ値が登録されている該当データを探し出す

☆ 静的データ構造でのデータ探索

- 配列の探索
  - ✓ 線形探索：先頭から順番に探索→ O(n)
  - ✓ 2 分探索：現在地より前か後ろか→ O(log n)
    ただし、あらかじめソートされている必要あり
    - 配列のソート
      - ✓ 単純ソート： 単純挿入、単純選択、単純交換 → $O(n^2)$
      - ✓ 高速ソート：クイックソート、マージソート → O(n log n)
- あらかじめ、データ領域のサイズが決められている（静的）

☆ 動的データ構造でのデータ探索

- 線形リストの探索
  - ✓ 順次探索：先頭から順番に探索→ O(n)
- 探索木（2 分木）の探索
  - ✓ 2 分探索：現在ノードより右か左か→ O(log n)
- データの追加・削除に応じてデータ領域のサイズが増減（動的）

「キーを使ったデータ探索」の復習の終わり

第 14 回「ハッシング」のはじまり

☆やりたいこと：データの格納位置を見つける

◎ 全 n 個のデータの中から、キーを指定して、同じ値が登録されている該当データを探し出す

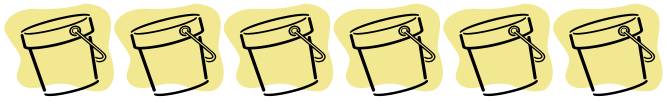
- スペースに十分なゆとりがある
- 上手に配置する
- 探索コストを O(log n)よりも良くできる？

→　データを上手に「散らして」格納する：ハッシング／ハッシュ法

ハッシュ（hash）：
ごちゃまぜにする
取り散らかす

例）ハッシュポテト

- ハッシュ法の探索コスト：
  - ✓ 最良時 O(1) → データの個数によらず一定
  - ✓ 最悪時 O(n)
  - ✓ 平均 O(1 + n/B)
    n<<B なら O(1)
    ただし、B は「バケット数」



☆よくある問題（例題）

◎ 名簿を管理したい

- 名前や ID を指定すると、その人の情報が取り出せるようにしたい
  - ✓ 名簿全部を見なくても、その人の情報を探し出す方法は？　→索引を使う

- 名前や ID は、そのままでは配列の添え字にできない
  例 1）学籍番号（ID）がキー
  学籍番号 9044086 の学生の情報 → data[9044086] に格納する
  ・ data[10000000] の配列が必要：　× 実際はそんなにいない
  例 2）名前がキー
  氏名 “Yokohama Kunihiro”の学生の情報 → data[YOKOHAMAKUNIHIRO] に格納する
  ・ data[26^{16}] の配列が必要（英文字 16 字なので）：　× 領域の確保が不可能（多すぎ）

- あらかじめ決められた数（B 個）の領域にデータを「散らして」格納する
  - ✓ データを**ある種の方法**で B 個の「バケツ」に分散させて格納する
    キー ⇒ インデックス（索引：データ格納位置＝バケット番号）への変換
    → **ハッシュ関数**

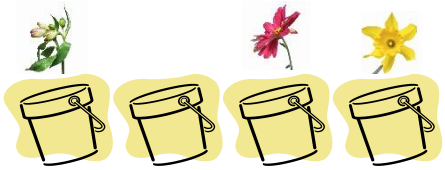
☆ハッシュ関数

◎ キー ⇒ インデックス の変換関数　H(k)

- キー k の値をインデックスの値域に一様に「ばらまく」性質の関数がよい
  例）　H(k) = ORD(k) % B
  関数 ORD(k)：キー k が何番目かを返す（たとえば、文字コード値の総和）
  関数 H(k)：ORD(k)を B で割った余り→ 0～B－1 のどれか
  → キー k を与えると H(k)は 0～B－1 のどれかに「ばらまかれる」
- H(k) の値をキー k の**ハッシュ値**と呼ぶ

☆ハッシュ表

◎ 要素数 B 個の配列：添え字にハッシュ値を用いる



4 個のどのバケットに格納するか
ハッシュ関数 H(k)で決める

- ハッシュ表：要素を B 個格納できる配列 hashtable[B]
  - ✓ **バケット**(bucket＝バケツ)：ハッシュ表の 1 要素
  - ✓ ハッシュ表は B 個のバケットを持つ
  - ✓ ハッシュ値＝バケット番号（データ格納位置）
- キーk を持つデータは hashtable[H(k)]に格納する
  - ✓ データは B 個のバケットのどこかに「ばらまいて」格納される
  - ✓ データ格納位置は、ハッシュ値（ハッシュ関数）によって求めることができる
- 同じハッシュ値をもつキーk_1 と k_2 は「**衝突**」(collision)する
  - ✓ 同じバケットに格納する（外部ハッシュ法）：
    線形リストを構成
  - ✓ 同じバケットに格納しない（内部ハッシュ法）：
    別のハッシュ関数を使って新たなハッシュ値を求める　→　**再ハッシュ**

☆ハッシュ表に対する基本操作とハッシュ法が有効な場面

◎ 探索 search：キーkey を与えると、ハッシュ表の該当位置を返す

void printsearch(char key[], struct item hashtable[])

◎ 挿入 insert：キーkey を持つデータ*x をハッシュ表に格納する

void insert(struct record *x, char key[], struct item hashtable[])

◎ 削除 delete：キーkey を持つデータをハッシュ表から削除する

void delete(char key[], struct item *hashtable[])

◎ ハッシュ法が有効な場面

- n も B も大きいとき、高効率な探索が要求される場面
  - ✓ 大容量のメモリや 2 次記憶に多数のデータを格納する際に、探索が速い
  - ✓ 名前を直接のキーとして探索できる
  - ✓ 要素の追加や削除が可能

ハッシュ法が使われている具体例：

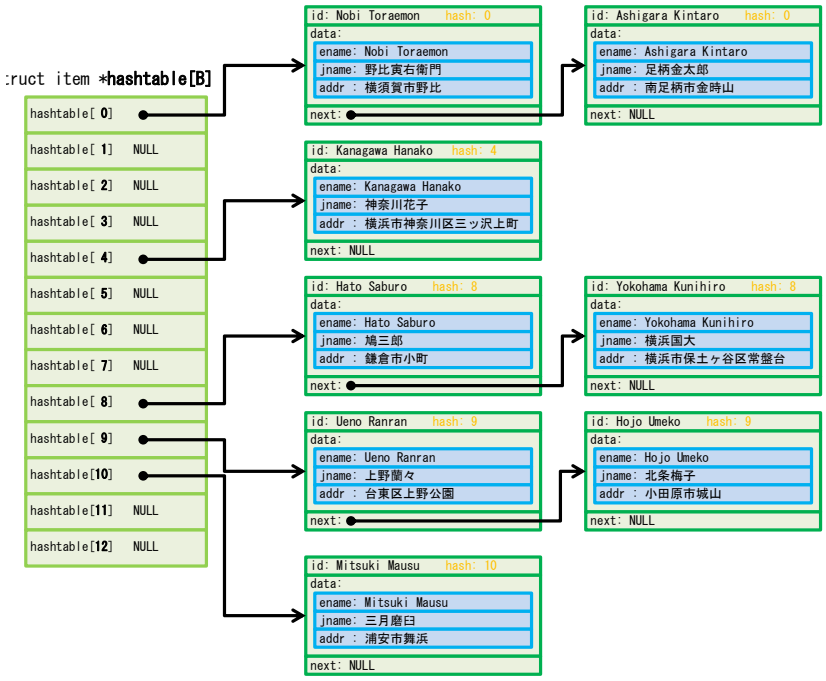
講義で示した例を記入せよ

☆ 衝突処理の違いによる 2 種類のハッシング

◎ 外部ハッシュ法（ダイレクトチェイニング法）：サンプルプログラム directchaining.c

- 要素は要素リスト（線形リスト）に格納される
- ハッシュ表の各バケットには、同じキーを持つ要素リストの先頭アドレスを保持する
- 格納できる長さに制限がない

- 挿入：　ハッシュ値のバケットの要素リストに追加 → 衝突しても差し支えない
- 削除：　ハッシュ値からバケットを特定し、要素リストを探索して、該当要素をリストから削除する
- 探索：　バケットを特定する：O(1)　（ハッシュ表に格納されているデータ件数によらない）
  要素リストの探索：O(n/B)



ダイレクトチェイニング法による
ハッシュ表

同じハッシュ値をもつ要素は線形リスト構造をなす

リストの先頭アドレスは、ハッシュ値を添え字とするバケットに保持される

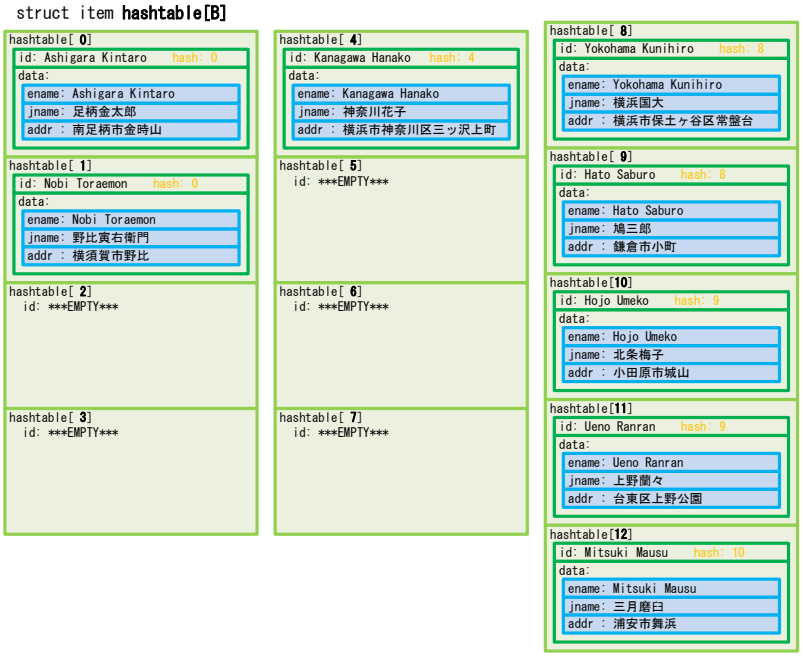
バケット数（ここでは B=13 個）の要素リストを保持することができる

◎ 内部ハッシュ法（オープンアドレッシング法）：サンプルプログラム openaddressing.c

- ハッシュ値から、ハッシュ表の格納位置（アドレス）を特定する
  - ✓　ハッシュ値が衝突する場合には再ハッシュ
- ハッシュ表の各バケットには、要素が直接格納される
- 格納できる長さはバケット数まで（制限がある）

- 挿入：　ハッシュ値のバケットが「未使用」か「削除済」ならそこに格納
  衝突の際は、別のハッシュ関数を用いる再ハッシュによって別の位置を探す
- 削除：　ハッシュ値・再ハッシュ値からバケットを特定し、削除する
  - ✓　探索のために、「未使用バケット」と「削除済バケット」の区別がある
- 探索：　バケットの特定（最良時）：O(１)
  衝突がある場合：見つかるまで再ハッシュ→（最悪時）：O(n)

ハッシュ表の埋まり具合にゆとりを持たせると、O(１)に近くなる



オープンアドレッシング法によるハッシュ表

バケット数（ここでは B=13 個）の要素を格納することができる

バケット番号が直接ハッシュ値になっていない箇所がある
→衝突時の再ハッシュによる

【サンプルプログラム directchaining.c】

```
/*************************************************************
    アルゴリズムとデータ構造
    サンプルプログラム  directchaining.c
    <<ダイレクトチェイニング法/外部ハッシュ法>>
    ・アイテムの ID に対してハッシュ値を作成
    ・アイテムは要素リスト（ハッシュ表の外部）に格納される
    ・衝突（同じハッシュ値を持つアイテムがある場合）
      の際は要素リストの先頭に格納
    ・ハッシュ表は同じハッシュ値をもつ要素リストの先頭を保持
    ・このプログラムの要素リストは実体を伴う
    ・アイテムの ID は同じ値の重複を許さない
    Copyright (c) 2010   T.Tomii <tommy@ynu.ac.jp>
 *************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define B 13 /* バケット数; 7 に変更して実行し比較せよ */

/* 1 件分のデータであるレコードを表す構造体 struct record */
#define STRLEN 30
struct record {
    char ename[STRLEN];
    char jname[STRLEN];
    char addr[STRLEN];
};

/* ハッシュ表に格納する 1 アイテムを表す構造体 struct item */
struct item {
    char id[STRLEN];     /* 探索のキーとするアイテム id, 重複不許可 */
    struct record data; /* 1 アイテムのデータ実体 */
    struct item *next;   /* 各バケットではリスト構造をなす */
};

/* プロトタイプ宣言 */
int getrecord(struct record *x); /* 共通 */
void printitem(struct item *y);   /* 共通 */
int hash(char key[]);              /* 共通 */

void makenull(struct item *hashtable[]);
struct item *search(char key[], struct item *hashtable[]);

void insert(struct record *x, char key[], struct item *hashtable[]);
void delete(char key[], struct item *hashtable[]);
void printsearch(char key[], struct item *hashtable[]);
void printhashtable(struct item *hashtable[]);
-----------------------------------------------------------------------------------------
/* プログラム開始 */
int main(void)
{
    struct item *hashtable[B]; /* ハッシュ表はリストを参照する ** ここだけ異なる ***/
    struct record x; /* レコードから読み出したデータ 1 件 */
    struct record dummy = {"Yokohama Kunihiro", "横濱邦博", "横浜市中区日本大通"};

    /* ハッシュ表初期化 */
    printf("===Initialize===¥n");
    makenull(hashtable);
    printhashtable(hashtable);

    /* 初期データを ename をキーとしてハッシュ表に登録 */
    printf("===Insert===¥n");
    while( getrecord(&x) )
        insert(&x, x.ename, hashtable);
    printhashtable(hashtable);
```

```
65
66      /* 重複データの登録試み */
67      printf("===Insert Dummy Data===¥n");
68      insert(&dummy, dummy.ename, hashtable);
69
70      /* ハッシュ表を対象とした探索 */
71      printf("===Search===¥n");
72      printsearch("Hato Saburo", hashtable);
73      printsearch(dummy.ename, hashtable); /* 同姓同名データの検索 */
74
75      /* ハッシュ表からのデータ削除 */
76      printf("===Delete===¥n");
77      delete("Hato Saburo", hashtable);
78      delete("Ueno Ranran", hashtable);
79      delete("Nobi Toraemon", hashtable);
80      delete("Nanashi Gonbei", hashtable);  /* 未登録データの削除試み */
81      delete(dummy.ename, hashtable); /* 同姓同名データの削除試み */
82      printhashtable(hashtable);
83
84      /* ハッシュ表を対象とした探索 */
85      printf("===Search===¥n");
86      printsearch("Hato Saburo", hashtable);
87      printsearch(dummy.ename, hashtable); /* 同姓同名データの検索 */
88
89      /* 再登録・再探索 */
90      printf("===Re-insert===¥n");
91      insert(&dummy, dummy.ename, hashtable);
92      printsearch(dummy.ename, hashtable);
93      printsearch("Mitsuki Mausu", hashtable);
94      printhashtable(hashtable);
95
96      return 0;
97  }
98  ----------------------------------------------------------------------------------------
99  /* 関数 getrecord: 1 レコード分のデータを x に取り出す
100     End of Data のとき 0 を返す
101     そうでないとき 1 を返す*/
102 /* ファイルから取り出すように変更してもよい */
103 int getrecord(struct record *x)
104 {
105     /* 初期データリスト */
106     struct record datafile[] = {
107         {"Yokohama Kunihiro", "横浜国大", "横浜市保土ヶ谷区常盤台"},
108         {"Kanagawa Hanako", "神奈川花子", "横浜市神奈川区三ツ沢上町"},
109         {"Hato Saburo", "鳩三郎", "鎌倉市小町"},
110         {"Hojo Umeko", "北条梅子", "小田原市城山"},
111         {"Ashigara Kintaro", "足柄金太郎", "南足柄市金時山"},
112         {"Ueno Ranran", "上野蘭々", "台東区上野公園"},
113         {"Mitsuki Mausu", "三月磨臼", "浦安市舞浜"},
114         {"Nobi Toraemon", "野比寅右衛門", "横須賀市野比"},
115         {"", "", ""} /* End of Data */
116     };
117     static int i=0;  /* 内部カウンタ */
118     int flag=1;      /* データ読み出しに成功 */
119
120     *x = datafile[i]; /* 戻り値は内部カウンタで示されるデータレコード */
121     if(datafile[i].ename[0]!='¥0') i++; /* End of Data でなければ内部カウンタを進める */
122     else {
123         i=0; /* End of Data ならば内部カウンタを 0 に戻す */
124         flag=0; /* End of Data であることを返す */
125     }
126     return flag;
127 }
128 ----------------------------------------------------------------------------------------
129 /* 手続 printitem: ハッシュ表のアイテム 1 件を印刷する */
130 void printitem(struct item *y)
```

```
131  {
132      printf("<(%d) %s %s %s>", hash(y->id), y->data.ename, y->data.jname, y->data.addr);
133  }
134  ------------------------------------------------------------------------------------------
135  /* 関数 hash: ハッシュ関数:
136     キー文字列の文字コード総和をバケット数 B で割った余り */
137  int hash(char key[])
138  {
139      int i=0, sum=0;
140      while( key[i]!='¥0' ){
141          sum = sum + (unsigned char)key[i]; /* 8bit-16bit コード対応のため unsigned */
142          i++;
143      }
144      return sum % B;
145  }
146  ------------------------------------------------------------------------------------------
147  /* 手続 makenull: ハッシュ表を初期化する */
148  void makenull(struct item *hashtable[])
149  {
150      int i;
151      for( i = 0; i < B; i++ )
152          hashtable[i]=NULL; /* 各バケットのポインタを NULL に */
153  }
154  ------------------------------------------------------------------------------------------
155  /* 関数 search: key を ID とするアイテムが hashtable に登録済みかどうか
156      登録済みの場合：そのアイテムへのポインタを返す
157      未登録の場合　：NULL を返す */
158  struct item *search(char key[], struct item *hashtable[])
159  {
160      struct item *p; /* p は探索対象をさすポインタ */
161      int flag = 1; /* 該当アイテムを見つけたら 0 とするフラグ */
162
163      /* 探索対象 p はハッシュ表に格納されたポインタ
164          （リストの先頭）からスタート */
165      p = hashtable[ hash(key) ];
166
167      /* p が NULL なら該当アイテムは無しなのでループを抜ける。
168         flag が 0 なら発見したので、
169         そのときの p を保持したままループを抜ける。 */
170      while( p != NULL && flag ){
171          /* key と探索対象 p の id が一致しない場合 */
172          if( strcmp( p->id, key) )
173              p = p->next; /* p をリストの次候補に進める */
174          /* 一致した場合 */
175          else
176              flag = 0; /* p を保持したままループを抜ける */
177      }
178      return p;
179  }
180  ------------------------------------------------------------------------------------------
181  /* 手続 insert: レコード*x の内容を key をハッシュのキーとして hashtable に登録する */
182  void insert(struct record *x, char key[], struct item *hashtable[])
183  {
184      int bucket;
185      struct item *p, *oldheader;
186
187      /* key でハッシュ表をサーチしその位置を p に保持 */
188      p = search( key, hashtable );
189
190      /* key が登録済みでなかった場合 */
191      if( p == NULL ){
192          bucket = hash( key ); /* ハッシュ関数から、格納先バケットを決定 */
193
194          /* 新しく挿入する場所として、
195             ハッシュ表のバケットがさすリストの先頭に領域を割当てる
196             その前に、割当前のリスト先頭を oldheader に保持 */
```

```
197             oldheader = hashtable[ bucket ];
198             /* バケットに新しい領域を割当 */
199             hashtable[ bucket ] = (struct item *)malloc( sizeof(struct item) );
200             /* 新しい領域にレコード値*x と id を登録 */
201             hashtable[ bucket ]->data = *x;
202             strcpy( hashtable[ bucket ]->id, key );
203             /* 割当前のリストを新しい領域の後につなぎかえ */
204             hashtable[ bucket ]->next = oldheader;
205         }
206         /* key が登録済みだった場合 */
207         else {
208             printf("Insert <%s> is rejected: same id is already used.¥n", key );
209         }
210     }
211     ------------------------------------------------------------------------------------------
212     /* 手続 delete: key を持つアイテムを hashtable から削除する */
213     void delete(char key[], struct item *hashtable[] )
214     {
215         int bucket, flag = 1; /* flag: 該当アイテム未発見なら 1 */
216         struct item *p, *prev; /* p: 探索、削除対象保持, prev: p の前位置保持 */
217
218         /* 対象バケットと対象リストを保持 */
219         bucket = hash( key );
220         p = hashtable[ bucket ];
221
222         /* key を含む可能性があるリストが存在する場合
223            なければ flag=1(未発見)のままで終了 */
224         if( p != NULL ){
225             /* リストの先頭に削除対象がある場合 */
226             if( strcmp( p->id, key ) == 0 ){
227                 /* ハッシュテーブルのバケットの参照先を更新し、削除対象を解放 */
228                 hashtable[ bucket ] = p->next;
229                 free( p );
230                 flag = 0;
231             }
232             /* リストの先頭よりも後に key を含む可能性がある場合 */
233             else {
234                 /* prev に一個前のアイテムを保持し、次要素が対象かどうか調べる
235                    それを、リストの最後まで繰り返す */
236                 do {
237                     prev = p;
238                     p = p->next;
239                     /* 次要素が削除対象の場合 */
240                     if( p != NULL )
241                         if( strcmp( p->id,   key) == 0 ){
242                             /* 削除対象 p を飛ばすようにリストを更新し、削除対象を解放 */
243                             prev->next = p->next;
244                             free( p );
245                             flag = 0; /* 見つけたので探索終了 */
246                         }
247                 } while( p != NULL && flag );
248                 /* 次要素があって、かつ、削除が済んでなければ繰り返し */
249             }
250         }
251         if( flag )
252             printf("Delete <%s> is rejected: not found¥n", key);
253     }
254     ------------------------------------------------------------------------------------------
255     /* 手続 printsearch: key を探して、結果を印刷する */
256     void printsearch(char key[], struct item *hashtable[])
257     {
258         struct item *p;
259
260         p = search(key, hashtable);
261             if( p != NULL ){
262             printf("Search <%s>: found ", key);
```

```
263             printitem( p );
264             printf("¥n");
265         }
266         else
267             printf("Search <%s>: not found¥n", key);
268     }
269     ------------------------------------------------------------------------------
270     /* 手続 printhashtable: ハッシュ表の状態を印刷する */
271     void printhashtable(struct item *hashtable[])
272     {
273         int i;
274         struct item *p;
275
276         printf("*****begin*****¥n");
277         for( i = 0; i < B; i++ ){
278             printf("   hashtable[%d]-", i);
279             /* バケット i のリストを表示 */
280             p = hashtable[i];
281             while( p != NULL ){
282                 printitem( p );
283                 printf("-");
284                 p = p->next;
285             }
286             printf("NULL¥n");
287         }
288         printf("*****end******¥n");
289     }
```

ハッシュ関数の選び方

　ハッシングアルゴリズムを効率的に実行するためには、キーの値をインデックスの値域に一様にばらまく性質のある関数を用いなければなりません。そのために様々な関数が提案さています。

静的ハッシュ法と動的ハッシュ法

　この資料では、ハッシュ表の大きさがあらかじめ決められた B 個に制限されたハッシュ法を紹介しています。このように、ハッシュ表の大きさが変化しないハッシュ法は「**静的ハッシュ法**」と呼ばれ、主記憶内にあるデータ管理などに用いられます。

　一方、要素数の増減に伴ってハッシュ表の大きさも変化させる「**動的ハッシュ法**（dynamic hashing)」も考案されており、大容量の 2 次記憶（ハードディスクやデータベースなど）にあるデータ管理などに用いられます。

【directchaining.c 実行結果】

```
===Initialize===
*****begin*****
  hashtable[0]-NULL
  hashtable[1]-NULL
  hashtable[2]-NULL
  hashtable[3]-NULL
  hashtable[4]-NULL
  hashtable[5]-NULL
  hashtable[6]-NULL
  hashtable[7]-NULL
  hashtable[8]-NULL
  hashtable[9]-NULL
  hashtable[10]-NULL
  hashtable[11]-NULL
  hashtable[12]-NULL
*****end******
===Insert===
*****begin*****
  hashtable[0]-<(0) Nobi Toraemon 野比寅右衛門 横須賀市野比>-<(0) Ashigara Kintaro 足柄金太郎 南足柄市金時山>-NULL
  hashtable[1]-NULL
  hashtable[2]-NULL
  hashtable[3]-NULL
  hashtable[4]-<(4) Kanagawa Hanako 神奈川花子 横浜市神奈川区三ツ沢上町>-NULL
  hashtable[5]-NULL
  hashtable[6]-NULL
  hashtable[7]-NULL
  hashtable[8]-<(8) Hato Saburo 鳩三郎 鎌倉市小町>-<(8) Yokohama Kunihiro 横浜国大 横浜市保土ヶ谷区常盤台>-NULL
  hashtable[9]-<(9) Ueno Ranran 上野蘭々 台東区上野公園>-<(9) Hojo Umeko 北条梅子 小田原市城山>-NULL
  hashtable[10]-<(10) Mitsuki Mausu 三月磨臼 浦安市舞浜>-NULL
  hashtable[11]-NULL
  hashtable[12]-NULL
*****end******
===Insert Dummy Data===
Insert <Yokohama Kunihiro> is rejected: same id is already used.
===Search===
Search <Hato Saburo>: found <(8) Hato Saburo 鳩三郎 鎌倉市小町>
Search <Yokohama Kunihiro>: found <(8) Yokohama Kunihiro 横浜国大 横浜市保土ヶ谷区常盤台>
===Delete===
Delete <Nanashi Gonbei> is rejected: not found
*****begin*****
  hashtable[0]-<(0) Ashigara Kintaro 足柄金太郎 南足柄市金時山>-NULL
  hashtable[1]-NULL
  hashtable[2]-NULL
  hashtable[3]-NULL
  hashtable[4]-<(4) Kanagawa Hanako 神奈川花子 横浜市神奈川区三ツ沢上町>-NULL
  hashtable[5]-NULL
  hashtable[6]-NULL
  hashtable[7]-NULL
  hashtable[8]-NULL
  hashtable[9]-<(9) Hojo Umeko 北条梅子 小田原市城山>-NULL
  hashtable[10]-<(10) Mitsuki Mausu 三月磨臼 浦安市舞浜>-NULL
  hashtable[11]-NULL
  hashtable[12]-NULL
*****end******
===Search===
Search <Hato Saburo>: not found
Search <Yokohama Kunihiro>: not found
===Re-insert===
Search <Yokohama Kunihiro>: found <(8) Yokohama Kunihiro 横濱邦博 横浜市中区日本大通>
Search <Mitsuki Mausu>: found <(10) Mitsuki Mausu 三月磨臼 浦安市舞浜>
*****begin*****
```

```
65      hashtable[0]-<(0) Ashigara Kintaro  足柄金太郎  南足柄市金時山>-NULL
66      hashtable[1]-NULL
67      hashtable[2]-NULL
68      hashtable[3]-NULL
69      hashtable[4]-<(4) Kanagawa Hanako  神奈川花子  横浜市神奈川区三ツ沢上町>-NULL
70      hashtable[5]-NULL
71      hashtable[6]-NULL
72      hashtable[7]-NULL
73      hashtable[8]-<(8) Yokohama Kunihiro  横濱邦博  横浜市中区日本大通>-NULL
74      hashtable[9]-<(9) Hojo Umeko  北条梅子  小田原市城山>-NULL
75      hashtable[10]-<(10) Mitsuki Mausu  三月磨臼  浦安市舞浜>-NULL
76      hashtable[11]-NULL
77      hashtable[12]-NULL
78    *****end******
```

ハッシュ法の種類と呼び方

この資料では 2 種類のハッシングアルゴリズムを紹介しましたが、同じアルゴリズムでも書籍によってその呼び方が異なります。その例を紹介します。

データの格納場所による分類	[書籍 1]	[書籍 2]
外部ハッシュ法	直接連鎖（direct chaining）	オープンハッシュ法
内部ハッシュ法	解放番地方式（open addressing）	クローズドハッシュ法

[書籍 1] N.ヴィルト著，浦昭二，國府方久史訳，「アルゴリズムとデータ構造」，近代科学社, 1990

[書籍 2] A.V.エイホ, J.E.ホップクロフト, J.D.ウルマン著，大野義夫訳,「データ構造とアルゴリズム」，培風館, 1987

【サンプルプログラム openaddressing.c】

```
/*************************************************************
    アルゴリズムとデータ構造
    サンプルプログラム  openaddressing.c
    <<オープンアドレッシング法/内部ハッシュ法>>
    ・アイテムの ID に対してハッシュ値を作成
    ・アイテムはハッシュ表内に直接格納される
    ・衝突（同じハッシュ値を持つアイテムがある場合）
      の際は再ハッシュ
    ・ハッシュ表がいっぱいのときには挿入できない
    ・このプログラムのハッシュ表は実体を伴う
    ・アイテムの ID は同じ値の重複を許さない
    Copyright (c) 2010   T.Tomii <tommy@ynu.ac.jp>
 *************************************************************/
#include <stdio.h>
#include <string.h>


#define B 13 /* バケット数; 7 に変更して実行し比較せよ */

/* 1 件分のデータであるレコードを表す構造体 struct record */
#define STRLEN 30
struct record {
    char ename[STRLEN];
    char jname[STRLEN];
    char addr[STRLEN];
};

/* ハッシュ表に格納する 1 アイテムを表す構造体 struct item */
struct item {
    char id[STRLEN];     /* 探索のキーとするアイテム id, 重複不許可 */
    struct record data; /* 1 アイテムのデータ実体 */

};

/* プロトタイプ宣言 */
int getrecord(struct record *x); /* 共通 */
void printitem(struct item *y);  /* 共通 */
int hash(char key[]);            /* 共通 */
int rehash(char key[], int n);
void makenull(struct item hashtable[]);
int search(char key[], struct item hashtable[]);
int locate(char key[], struct item hashtable[]);
void insert(struct record *x, char key[], struct item hashtable[]);
void delete(char key[], struct item hashtable[]);
void printsearch(char key[], struct item hashtable[]);
void printhashtable(struct item hashtable[]);
------------------------------------------------------------------------------------------
/* プログラム開始 */
int main(void)
{
    struct item hashtable[B]; /* ハッシュ表は実体を伴う ** ここだけ異なる ***/
    struct record x; /* レコードから読み出したデータ 1 件 */
    struct record dummy = {"Yokohama Kunihiro", "横濱邦博", "横浜市中区日本大通"};

    /* ハッシュ表初期化 */
    printf("===Initialize===¥n");
    makenull(hashtable);
    printhashtable(hashtable);

    /* 初期データを ename をキーとしてハッシュ表に登録 */
    printf("===Insert===¥n");
    while( getrecord(&x) )
        insert(&x, x.ename, hashtable);
    printhashtable(hashtable);
```

```
65
66      /* 重複データの登録試み */
67      printf("===Insert Dummy Data===¥n");
68      insert(&dummy, dummy.ename, hashtable);
69
70      /* ハッシュ表を対象とした探索 */
71      printf("===Search===¥n");
72      printsearch("Hato Saburo", hashtable);
73      printsearch(dummy.ename, hashtable); /* 同姓同名データの検索 */
74
75      /* ハッシュ表からのデータ削除 */
76      printf("===Delete===¥n");
77      delete("Hato Saburo", hashtable);
78      delete("Ueno Ranran", hashtable);
79      delete("Nobi Toraemon", hashtable);
80      delete("Nanashi Gonbei", hashtable);  /* 未登録データの削除試み */
81      delete(dummy.ename, hashtable); /* 同姓同名データの削除試み */
82      printhashtable(hashtable);
83
84      /* ハッシュ表を対象とした探索 */
85      printf("===Search===¥n");
86      printsearch("Hato Saburo", hashtable);
87      printsearch(dummy.ename, hashtable); /* 同姓同名データの検索 */
88
89      /* 再登録・再探索 */
90      printf("===Re-insert===¥n");
91      insert(&dummy, dummy.ename, hashtable);
92      printsearch(dummy.ename, hashtable);
93      printsearch("Mitsuki Mausu", hashtable);
94      printhashtable(hashtable);
95
96      return 0;
97  }
98  ------------------------------------------------------------------------------
99  /* 関数 getrecord: 1 レコード分のデータを x に取り出す
100     End of Data のとき 0 を返す
101     そうでないとき 1 を返す*/
102 /* ファイルから取り出すように変更してもよい */
103 int getrecord(struct record *x)
104 {
105   /* 初期データリスト */
106     struct record datafile[] = {
107         {"Yokohama Kunihiro", "横浜国大", "横浜市保土ヶ谷区常盤台"},
108         {"Kanagawa Hanako", "神奈川花子", "横浜市神奈川区三ツ沢上町"},
109         {"Hato Saburo", "鳩三郎", "鎌倉市小町"},
110         {"Hojo Umeko", "北条梅子", "小田原市城山"},
111         {"Ashigara Kintaro", "足柄金太郎", "南足柄市金時山"},
112         {"Ueno Ranran", "上野蘭々", "台東区上野公園"},
113         {"Mitsuki Mausu", "三月磨臼", "浦安市舞浜"},
114         {"Nobi Toraemon", "野比寅右衛門", "横須賀市野比"},
115         {"", "", ""} /* End of Data */
116     };
117     static int i=0;  /* 内部カウンタ */
118     int flag=1;       /* データ読み出しに成功 */
119
120     *x = datafile[i]; /* 戻り値は内部カウンタで示されるデータレコード */
121     if(datafile[i].ename[0]!='¥0') i++; /* End of Data でなければ内部カウンタを進める */
122     else {
123         i=0; /* End of Data ならば内部カウンタを 0 に戻す */
124         flag=0; /* End of Data であることを返す */
125     }
126     return flag;
127 }
128 ------------------------------------------------------------------------------
129 /* 手続 printitem: ハッシュ表のアイテム 1 件を印刷する */
130 void printitem(struct item *y)
```

```
131  {
132      printf("<(%d) %s %s %s>", hash(y->id), y->data.ename, y->data.jname, y->data.addr);
133  }
134  ------------------------------------------------------------------------------------------
135  /* 関数 hash: ハッシュ関数:
136     キー文字列の文字コード総和をバケット数 B で割った余り */
137  int hash(char key[])
138  {
139      int i=0, sum=0;
140      while( key[i] != '¥0' ){
141          sum = sum + (unsigned char)key[i]; /* 8bit-16bit コード対応のため unsigned */
142          i++;
143      }
144      return sum % B;
145  }
146  ------------------------------------------------------------------------------------------
147  /* 関数 rehash: 再ハッシュ関数: 再ハッシュ n 回目は (hash(key)+n)%B とする */
148  int rehash(char key[], int n)
149  {
150      return (hash(key)+n)%B;
151  }
152  ------------------------------------------------------------------------------------------
153  #define EMPTY    "***EMPTY***" /* マジック定数: 空バケットの id */
154  #define DELETED "**DELETED**" /* マジック定数: 削除済バケットの id */
155  ------------------------------------------------------------------------------------------
156  /* 手続 makenull: ハッシュ表を初期化する */
157  void makenull(struct item hashtable[])
158  {
159      int i;
160      for( i = 0; i < B; i++ )
161          strcpy( hashtable[i].id, EMPTY ); /* 各バケットの id を EMPTY に */
162  }
163  ------------------------------------------------------------------------------------------
164  /* 関数 search: key をキーとしてハッシュ表を探索し、
165     該当要素バケットか、または、最初の空きバケットを返す
166       key が登録済みの場合：該当要素のバケットを返す
167       key が未登録の場合　：最初の EMPTY バケットを返す
168    ※ ハッシュ表がいっぱいのときの返り値は規定しない
169    ※ DELETED は、そのバケットが使われていたことを示すので、
170       その先も探索を続けなければならない */
171  int search(char key[], struct item hashtable[])
172  {
173      int bucket, i=0; /* i: チャレンジ回数 */
174
175      /* まずは通常のハッシュ値からチャンレンジを始める */
176      bucket = hash(key);
177      while( i < B
178              && strcmp( hashtable[ bucket ].id, key )
179              && strcmp( hashtable[ bucket ].id, EMPTY ) ){
180          i++; /* 該当バケットでなければ次のチャレンジ */
181          bucket = rehash(key, i); /* 該当バケットでなければ再ハッシュ */
182      }
183      return bucket;
184  }
185  ------------------------------------------------------------------------------------------
186  /* 関数 locate: key をキーとするアイテムを登録すべきバケットを返す
187       key が登録済みの場合：該当要素のバケットを返す
188       key が未登録の場合　：最初の EMPTY または DELETED のバケットを返す
189    ※ ハッシュ表がいっぱいのときの返り値は規定しない
190    ※ 挿入位置は DELETED のバケットでもよい */
191  int locate(char key[], struct item hashtable[])
192  {
193      int bucket, i;
194
195      i=0;
196      bucket = hash(key);
```

```
197         while( i < B
198                 && strcmp( hashtable[ bucket ].id, key )
199                 && strcmp( hashtable[ bucket ].id, EMPTY )
200                 && strcmp( hashtable[ bucket ].id, DELETED ) ){
201             i++;
202             bucket = rehash(key, i); /* 該当バケットでなければ再ハッシュ */
203         }
204         return bucket;
205     }
206     ------------------------------------------------------------------------------------------
207     /* 手続 insert: レコード*x の内容を key をハッシュのキーとして hashtable に登録する */
208     void insert(struct record *x, char key[], struct item hashtable[])
209     {
210         int bucket;
211
212         /* key でハッシュ表をサーチしその位置を bucket に保持 */
213         bucket = search( key, hashtable );
214
215         /* key が登録済みでなかった場合 */
216         if( strcmp( hashtable[ bucket ].id, key ) ){
217             /* まずは DELETED も含めて新たな挿入先候補バケットを決める */
218             bucket = locate( key, hashtable );
219
220             /* 挿入先候補バケットが空または削除済みバケットならば */
221             if( strcmp( hashtable[ bucket ].id, EMPTY ) == 0
222                 || strcmp( hashtable[ bucket ].id, DELETED ) == 0 ){
223                 /* そのバケットにレコード値*x と id を登録 */
224                 hashtable[ bucket ].data = *x;
225                 strcpy( hashtable[ bucket ].id, key );
226             }
227             /* 挿入先候補バケットが空または削除済みでない場合には
228                hashtable が Full */
229             else {
230               printf("Insert <%s> is rejected: hashtable is full.¥n", key);
231             }
232         }
233         /* key が登録済みだった場合 */
234         else{
235             printf("Insert <%s> is rejected: same id is already used.¥n", key);
236         }
237     }
238     ------------------------------------------------------------------------------------------
239     /* 手続 delete: key を含むバケットを hashtable から削除（DELETED に）する */
240     void delete(char key[], struct item hashtable[] )
241     {
242         int bucket;
243
244         /* まずは対象のバケットを探索する */
245         bucket = search( key, hashtable );
246
247         /* そのバケットに削除対象があったら、id を DELETED にする */
248         if( strcmp( hashtable[ bucket ].id, key ) == 0 )
249             strcpy( hashtable[ bucket ].id, DELETED );
250         else
251             printf("Delete <%s> is rejected: not found¥n", key);
252     }
253     ------------------------------------------------------------------------------------------
254     /* 手続 printsearch: key を探して、結果を印刷する */
255     void printsearch(char key[], struct item hashtable[])
256     {
257         int bucket;
258
259         bucket = search(key, hashtable);
260         if( strcmp( hashtable[ bucket ].id, key ) == 0 ){
261             printf("Search <%s>: found hashtable[%d]=", key, bucket);
262             printitem( &hashtable[ bucket ] );
```

```
263             printf("¥n");
264         }
265         else
266             printf("Search <%s>: not found¥n", key);
267     }
268     ------------------------------------------------------------------------------------------
269     /* 手続 printhashtable: ハッシュ表の状態を印刷する */
270     void printhashtable(struct item hashtable[])
271     {
272         int i;
273
274         printf("*****begin*****¥n");
275         for( i = 0; i < B; i++ ){
276             printf("   hashtable[%d]=", i);
277             if( strcmp( hashtable[i].id, EMPTY ) == 0 )
278                 printf("EMPTY¥n");
279             else if( strcmp( hashtable[i].id, DELETED ) == 0 )
280                 printf("DELETED¥n");
281             else {
282                 printitem( &hashtable[i] );
283                 printf("¥n");
284             }
285         }
286         printf("*****end******¥n");
287     }
  1
```

サンプルプログラムで用いた文字列操作に関するライブラリ関数　（string.h で定義）

◎　int strcmp(cs, ct)
文字列 cs と文字列 ct を比較。cs<ct なら<0 を、cs==ct なら 0 を、cs>ct なら>0 を返す。

◎　char *strcpy(s, ct)
'¥0'を含めて文字列 ct を s にコピーし、s を返す。

B.W.カーニハン, D.M.リッチー著，石田晴久訳「プログラミング言語 C 第 2 版」（共立出版）

【openaddressing.c 実行結果】

```
===Initialize===
*****begin*****
  hashtable[0]=EMPTY
  hashtable[1]=EMPTY
  hashtable[2]=EMPTY
  hashtable[3]=EMPTY
  hashtable[4]=EMPTY
  hashtable[5]=EMPTY
  hashtable[6]=EMPTY
  hashtable[7]=EMPTY
  hashtable[8]=EMPTY
  hashtable[9]=EMPTY
  hashtable[10]=EMPTY
  hashtable[11]=EMPTY
  hashtable[12]=EMPTY
*****end******
===Insert===
*****begin*****
  hashtable[0]=<(0) Ashigara Kintaro 足柄金太郎 南足柄市金時山>
  hashtable[1]=<(0) Nobi Toraemon 野比寅右衛門 横須賀市野比>
  hashtable[2]=EMPTY
  hashtable[3]=EMPTY
  hashtable[4]=<(4) Kanagawa Hanako 神奈川花子 横浜市神奈川区三ツ沢上町>
  hashtable[5]=EMPTY
  hashtable[6]=EMPTY
  hashtable[7]=EMPTY
  hashtable[8]=<(8) Yokohama Kunihiro 横浜国大 横浜市保土ヶ谷区常盤台>
  hashtable[9]=<(8) Hato Saburo 鳩三郎 鎌倉市小町>
  hashtable[10]=<(9) Hojo Umeko 北条梅子 小田原市城山>
  hashtable[11]=<(9) Ueno Ranran 上野蘭々 台東区上野公園>
  hashtable[12]=<(10) Mitsuki Mausu 三月磨臼 浦安市舞浜>
*****end******
===Insert Dummy Data===
Insert <Yokohama Kunihiro> is rejected: same id is already used.
===Search===
Search <Hato Saburo>: found hashtable[9]=<(8) Hato Saburo 鳩三郎 鎌倉市小町>
Search <Yokohama Kunihiro>: found hashtable[8]=<(8) Yokohama Kunihiro 横浜国大 横浜市保
土ヶ谷区常盤台>
===Delete===
Delete <Nanashi Gonbei> is rejected: not found
*****begin*****
  hashtable[0]=<(0) Ashigara Kintaro 足柄金太郎 南足柄市金時山>
  hashtable[1]=DELETED
  hashtable[2]=EMPTY
  hashtable[3]=EMPTY
  hashtable[4]=<(4) Kanagawa Hanako 神奈川花子 横浜市神奈川区三ツ沢上町>
  hashtable[5]=EMPTY
  hashtable[6]=EMPTY
  hashtable[7]=EMPTY
  hashtable[8]=DELETED
  hashtable[9]=DELETED
  hashtable[10]=<(9) Hojo Umeko 北条梅子 小田原市城山>
  hashtable[11]=DELETED
  hashtable[12]=<(10) Mitsuki Mausu 三月磨臼 浦安市舞浜>
*****end******
===Search===
Search <Hato Saburo>: not found
Search <Yokohama Kunihiro>: not found
===Re-insert===
Search <Yokohama Kunihiro>: found hashtable[8]=<(8) Yokohama Kunihiro 横濱邦博 横浜市中
区日本大通>
Search <Mitsuki Mausu>: found hashtable[12]=<(10) Mitsuki Mausu 三月磨臼 浦安市舞浜>
*****begin*****
  hashtable[0]=<(0) Ashigara Kintaro 足柄金太郎 南足柄市金時山>
```

```
65      hashtable[1]=DELETED
66      hashtable[2]=EMPTY
67      hashtable[3]=EMPTY
68      hashtable[4]=<(4) Kanagawa Hanako 神奈川花子 横浜市神奈川区三ツ沢上町>
69      hashtable[5]=EMPTY
70      hashtable[6]=EMPTY
71      hashtable[7]=EMPTY
72      hashtable[8]=<(8) Yokohama Kunihiro 横濱邦博 横浜市中区日本大通>
73      hashtable[9]=DELETED
74      hashtable[10]=<(9) Hojo Umeko 北条梅子 小田原市城山>
75      hashtable[11]=DELETED
76      hashtable[12]=<(10) Mitsuki Mausu 三月磨臼 浦安市舞浜>
77    *****end******
```

これで「アルゴリズムとデータ構造」は終わりです．どうでしたか？ 講義内容は充分理解できましたか？ 本講義の内容は電子情報システムＥＰ・情報工学ＥＰを問わず重要です．実験などで，アルゴリズムとデータ構造を考えて，それをすぐにプログラミングすることができるようになりましょう．皆さんの今後の発展を祈念します．ではまた！