アルゴリズムとデータ構造

- 第12回講義トピック：探索（続き）
  - ハッシュ法 (hashing)
    - ハッシュ関数
    - 分離連鎖法
    - 開番地法
      - 線形探索法
      - 2重ハッシュ法

ハッシュ法はなぜ必要か

- N個のデータをサイズNの配列に格納し、データのキーとして配列のインデックスを持たせれば、キーから直接そのデータレコードにアクセスできるので、最も効率よく探索できる。
- しかし、データサイズNに対して、キーの取りうる範囲は一般的に非常に大きい（例えば、全ての整数値）ので、それに対応する領域を配列として確保することは現実的でない。
- 有限の記憶領域を使って、より広大なキーの範囲をカバーする方法⇒ハッシュ法
- ハッシュ法の特徴は比較操作によって探索するこれまでの方法と違って、速いデータの挿入と探索ができる。

ハッシュ法における衝突処理

- ハッシュ法は小さい格納空間により広大なキーの領域を対応させるため、1つの番地に複数のデータを対応させる可能性が生じ、衝突処理が必要となる。
  - 分離連鎖法(separate chaining)：同じハッシュ番地に対応する複数のデータを連結リストを用いて格納する。
    - データのサイズが前もってわからない動的な状態に対応できる。
  - 開番地法(open addressing)：データが衝突した場合は線形探索法(linear probing) あるいは2重ハッシュ法(double hashing)によって、次の空き番地を探す。
    - 配列の大きさが固定で、多数の空き番地があることが前提となるため、予め、データのサイズが予想できる場合が有効である。探索は高速に行える。

ハッシュ法におけるキーをアドレスの対応

- キーに対して、算術演算を施してハッシュ表のアドレスに変換し、ハッシュ表を直接参照する方法を考える。
  - たとえば、キーの取り得る値が整数で、ハッシュ表のサイズ（データ格納領域）をNとする。キーiをもつレコードをハッシュ表の i%N 番目の位置に格納すれば、キーの値から直ちに（データの衝突がなければ、）レコードを参照できる。
  - 以上の単純な例を一般化し、さらに重複の場合の対策も含めて、キーに関する特別な知識を前提としない普通の探索問題に適用できるようにする。

ハッシュ関数

- ハッシュ関数はキーをハッシュ表の番地に変換する
  - 相異なるキーを相異なる番地に写像できれば理想的であるが、ハッシュ表がキーの領域と同じ大きさの空間が必要となるので、不可能な場合が多い。
  - 現実には2つ以上のキーが同じ番地に写像されることを前提とする。ハッシュ表の大きさはキーの空間よりも小さいので、どのハッシュ関数も完全ではない。
  - ハッシュ関数は計算が簡単で、出力が偏らないで、ハッシュ表のどの番地にも等確率にデータが格納されることが理想的である。

ハッシュ関数の例

- Mは定数で、キーkに対してハッシュ番地を次の式で与える

  $h(k) = k \bmod M$

  ここで、ハッシュ番地への規則性をできるだけなくすため、Mは素数を選ぶ。
  - 例えば、M = 101 とし、4文字からなるキーAKEYに対応する添字を計算する。
    各文字を5ビットのコードで符号化すると、

    00001 01011 00101 11001

    10進では44217となり、h(k)は80となる。従ってキーAKEYはハッシュ表の80番地にハッシュされる。
    衝突可能性：キーBARHもハッシュ番地80に対応する。

Mの値がハッシュ関数に与える影響

- 上述の例ではキーの各文字を5ビットのコードで符号化するので、キーAKEYは、

  $1x32^3 + 11x32^2 + 5x32^1 + 25x32^0$ (= 44217)

  となる。
  例え、M=32を選んだとすると、32の倍数を加えてもk mod 32の値は変わらないので、どのキーもハッシュ関数の値は最後の文字で決まってしまう。
  よいハッシュ関数はキーのすべての文字を利用するべきである。

VERYLONGKEYに対する扱い

- 場合によっては、計算機の通常の算術演算能力を超えた大変長いキーを扱う必要がある。例えば、以下の関数では、ホーナー法の原理を利用する。

$$f(x) = a_0 x^n + a_1 x^{n-1} + \ldots a_{n-1} x + a_n$$
$$= (\ldots(((a_0 x + a_1) x + a_2) x + a_3) x + \ldots + a_{n-1}) x + a_n$$

- unsigned hash(char *v) {
  int h;
  for (h = 0; *v != '¥0'; v++)
    h = (64*h + *v) %M;
  return h;
}

分離連鎖法

- 連結リストとハッシュ表を組み合わせる方法
  - 表の各番地毎に、連結リストを用意して、その番地にハッシュされたキーをもつレコードを保持する。
  - 連結リスト内のデータの並ぶに関しては、ソートしないで挿入順の形で保持することもでき、挿入する時にキーの順に並べることもできる。
- 分離連鎖法の実現
  - M個のヘッドheadを使ってM個のリストを保持する
  - 連結リストに関する探索と挿入の手続きはそのまま使用できる。
  - headへの参照をheads[hash(v)]への参照に置き換えて、ハッシュ関数をリストの選択に使うように修正するだけでよい。

分離連鎖法における連結リストの初期設定

```
struct node {
   char key;  int info; struct node *next;
};
struct node *heads[M], *z;
hashlistinitialize(){
   int i; z = (struct node *)  malloc(sizeof *z);
   z->next = z; z->info = -1;
   for (i=0; i < M; i++) {
       heads[i] = (struct node *) malloc(sizeof *z);
       heads[i]->next = z;
   }
}
```

分離連鎖法の例

- 以下の文字列に対して分離連鎖法によるキー挿入の様子

  A S E A R C H I N G E X A M P L E

  - ハッシュ関数 (M=11)の結果は以下とする。

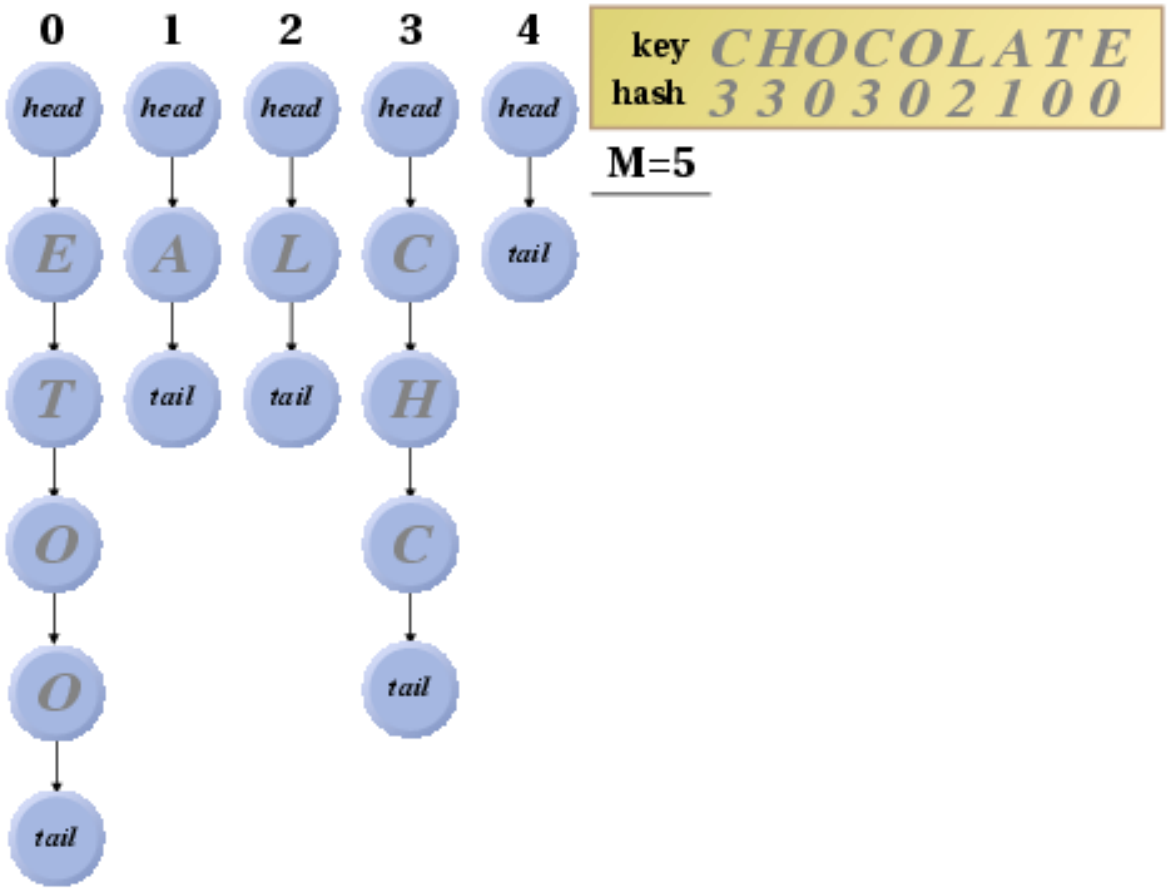
key:	A	S	E	A	R	C	H	I	N	G	E	X	A	M	P	L	E
hash:	1	8	5	1	7	3	8	9	3	7	5	2	1	2	5	1	5

分離連鎖法の例

- 分離連鎖法

0	1	2	3	4	5	6	7	8	9	10
	A	M	N		E		G	H	I	
	A	X	C		E		R	S		
	A				E					
	L				P					

もう1つの例


0
1
2
3
4
head
head
head
head
head
E
A
L
C
tail
T
tail
tail
H
O
C
O
tail
tail
key CHOCOLATE
hash 3 3 0 3 0 2 1 0 0
M=5

分離連鎖法の計算量

- ハッシュ関数における計算
  - データのサイズによらないほぼ一定である。
- 連結リストにおける計算
  - データのサイズを平均1/Mに減らす。
  - ただし、M個のリストへのヘッド分のための記憶領域を余分に使う。

開番地法

- N個のレコードをサイズM(＞N)のハッシュ表に格納する。α=N/Mを占有率といい、1より小さい値をとる。
- 空き領域は衝突処理のために必要であるため、ハッシュ法を効率よくするためには占有率は小さく抑える必要がある。
- 開番地方を使うための条件
  - ハッシュ表におかれる要素の数が前もって予測できる。
  - 余裕を持ってすべての要素を保持できる連続した記憶領域が確保できる。

開番地法における線形探査法

1. ハッシュ関数によって、キーに対応する番地を計算する。
2. ハッシュ番地が空いている場合は探索不成功で終了する。
3. ハッシュ番地に探索キーと同じデータがある場合は、探索成功でそのデータを返す。
4. ハッシュ番地が探索キーと異なるデータによってすでに使われている場合は、ハッシュ表の次の番地を探す。
5. 探索キーと同じデータあるいは空きの場所が見つかる1から4を繰り返す。ハッシュ表はデータの数よりも大きな領域を持っているので、どこも空いていない飽和状態は起こらないとする。

線形探査法によるデータ挿入

1. ハッシュ関数によって、キーに対応する番地を計算する。
2. ハッシュ番地が空いている場合は探索不成功で新しいデータを挿入する。
3. ハッシュ番地に探索キーと同じデータがある場合は、探索成功でデータの挿入は行わないで終了する。
4. ハッシュ番地が探索キーと異なるデータによってすでに使われている場合は、ハッシュ表の次の番地を探す。
5. 探索キーと同じデータあるいは空きの場所が見つかるまで手順1から4を繰り返す。ハッシュ表は実際のデータの数よりも大きな領域を持っているので、どこも空いていない飽和状態は起こらないとする。

線形探索法のプログラム

```
struct node { char *key; int info; };
struct node a[M+1];
hashinitialize() { int i;
   for (i = 0; i <= M; i++) {
       a[i].key = " "; a[i].info = -1;
} }
hashinsert(char *v, int info) { int x = hash(v);
   while (strcmp(" ", a[x].key)) x = (x+1) % M;
   a[x].key = v; a[x].info = info;
}
```

線形探査法の例

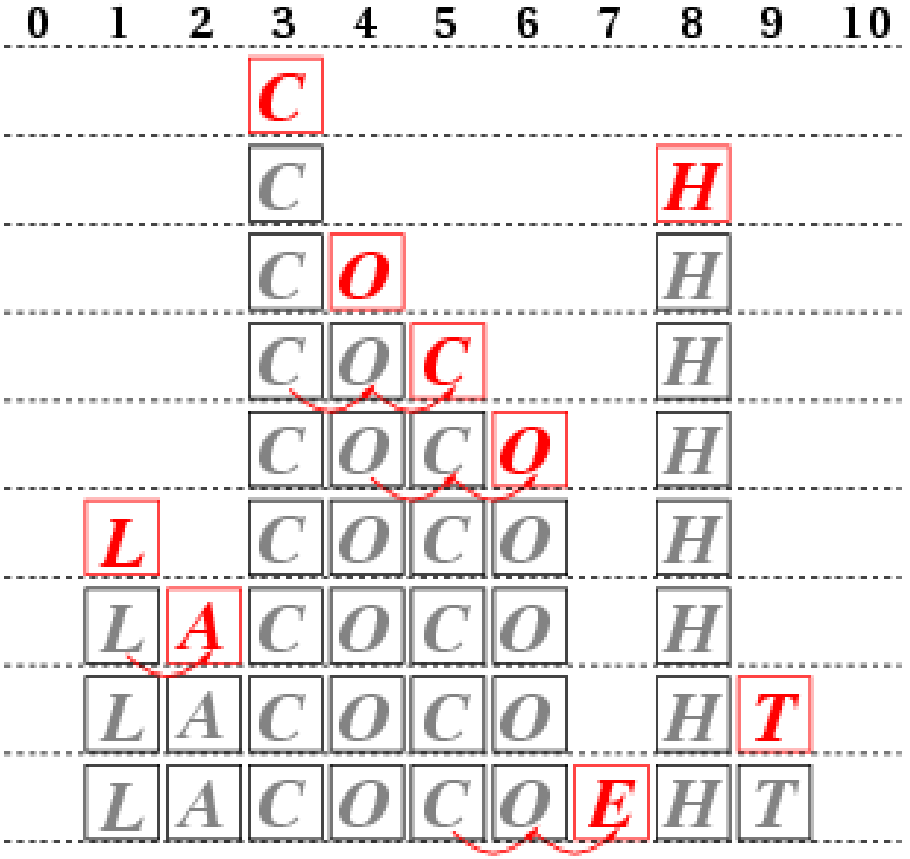
- 以下の文字列に対して線形探査法によりキー挿入の様子

  A S E A R C H I N G E X A M P L E

  - ハッシュ関数（M=19)の結果は以下とする。

key:	A	S	E	A	R	C	H	I	N	G	E	X	A	M	P	L	E
hash:	1	0	5	1	18	3	8	9	14	7	5	5	1	13	16	12	5

もう1つの例



key C H O C O L A T E
hash 3 8 4 3 4 1 1 9 5

M=11

線形探査法の計算量

- 線形探索法の計算量はハッシュ表のデータ占有率によって変化する。
- ハッシュ表の占有率が2/3以下の場合、ハッシュ探査は、平均5回未満の比較が必要となる。

２重ハッシュ法

- ２重ハッシュ法では衝突発生箇所からすぐ後に続く要素を１つ1つ調べる代わりに、もう1つのハッシュ関数によって一定の増分を求め、それを使って探索を行う。
- h2をセカンドハッシュ関数とする

```
hashinsert(char *v, int info) {
int x = hash(v), u = h2(v);
  while (strcmp(" ", a[x].key))
    x = (x+u) % M;
  a[x].key = v; a[x].info = info;
}
```

2番目のハッシュ関数の選び方

- 2番目のハッシュ関数は、1番目の関数と異なるように選ぶ。
- uは0にはならないようにする。
- Mとuが互いに素であるようにする。
  - 例えばMを素数とし、u<Mのようにする。
- 計算が簡単なものにする。
  - たとえば、h2(k)=8-(k mod 8)

２重ハッシュ法の例

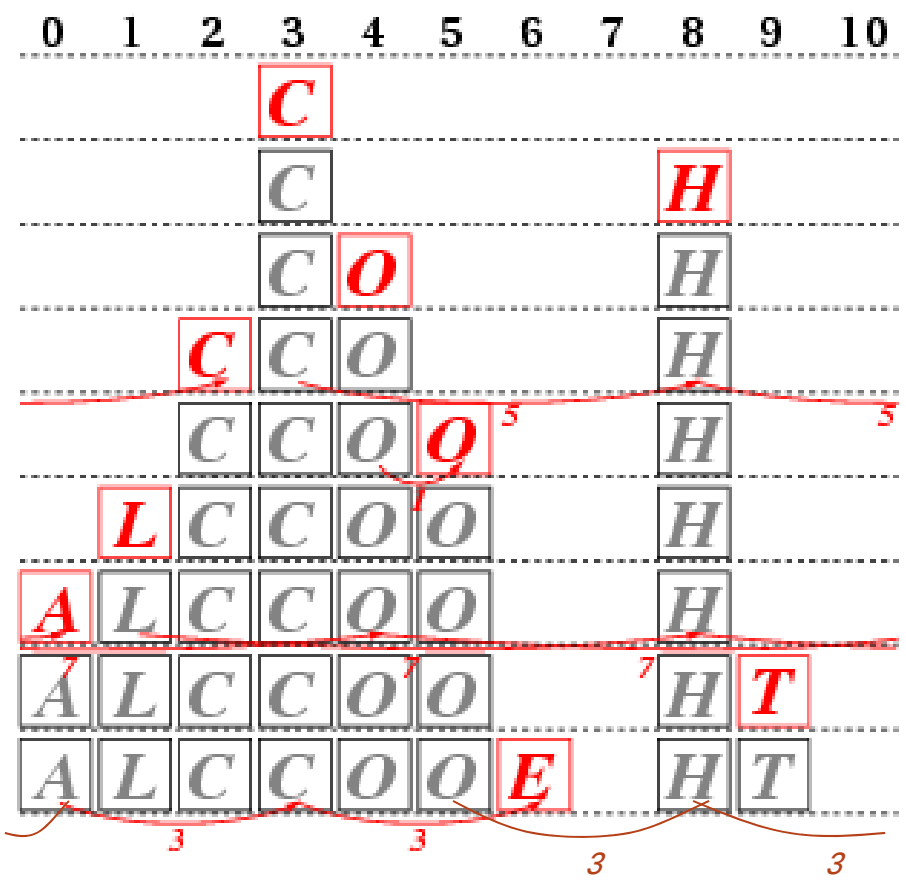
- 以下の文字列に対して、２重ハッシュ法によるキー挿入の様子

  A S E A R C H I N G E X A M P L E

  - ハッシュ関数（M=19)の結果は以下とする。

key:	A	S	E	A	R	C	H	I	N	G	E	X	A	M	P	L	E
hash1:	1	0	5	1	18	3	8	9	14	7	5	5	1	13	16	12	5
hash2:	7	5	3	7	6	5	8	7	2	1	3	8	7	3	8	4	3

もう1つの例



key	C	H	O	C	O	L	A	T	E
hash1	3	8	4	3	4	1	1	9	5
hash2	5	8	1	5	1	4	**7**	4	3

M=11

２重ハッシュ法の計算量

- ２重ハッシュ法の計算量解析は複雑であるが、
- 一般的に2重ハッシュ法での探索回数は、平均として線形探査法よりも少ない。