ハッシュテーブル

探索アルゴリズム

- 表の中に入っているデータ(表の中のデータの個数をn個とする)の中から，目的のデータを探索するアルゴリズム
  - 線形探索
    - 表中のデータを1つ1つ順に端から調べるだけの単純な探索法
    - 計算量は O(n)
  - 二分探索，木
    - キー値の比較に基づき，途中で探索候補を絞っていくことによって,表全部を調べることなく高速化を図る
    - 計算量は O(log n)
  - ハッシュ法
    - 平均してO(1)の計算量で探索，挿入，削除の全てを行うことができる

配列による探索

- 「キー値の範囲をおおうような配列」を使う方法
  （例）レコード{学籍番号，名前，生年月日}
  - キー値： 学籍番号（1から100までの値をとりうる)
  - 1～100までの整数を添字とする配列を用意
    （例）配列student[1]～student[100] に，それぞれレコードデータを入れる
  - 配列を使って，キー値からレコードを取り出す方法：
    与えられたキー値（この場合，学籍番号）を添字として，その配列を調べる
- 単純に考えて，「データが何個存在するかに関係なく，配列を1つ調べればいい」と考えれば，探索1回当りの計算量はO(1)

ハッシュ法

- 配列による探索
  - 配列の添字として使用できるのは正の整数のみ
  - キーの値が限られた範囲の整数である場合にしか使えない（一般的ではない）
- ハッシュ法
  - キーの値を直接配列の添字とするのではなく，キー値をある関数（ハッシュ関数）によって整数に変換して，この関数値を添字として配列を参照する
  - キー値として，文字列など可能（名前による検索など）

    （例） 名前によって探索する場合

    1．関数Hash に引数として，検索したい名前(文字列)を与える．

    2．関数Hash は，与えられた文字列からなんらかの計算を行い，整数値を返す．

    3．その整数値を配列の添字とする配列を参照する．

ハッシュ法

- ハッシュ関数:
  - キー値を整数に変換する関数
  - 常に，同じハッシュ関数を使うので，同じキーの値を与えた時,結果はいつも同じになる.
    - → データを挿入する時も探索する時も配列の同じ場所を参照する
- ハッシュ値
  - ハッシュ関数の計算によって得られる整数値
- ハッシュテーブル
  - データが格納された配列
- ハッシュ関数の計算，配列の参照にかかる時間は，データ数nに無関係で，O(1)の計算量

衝突

- キーのとり得る値の種類に比べて，配列の添字として使える値の範囲は小さい
  - 配列の添字：

    正の整数で，4バイトの整数 （0から2147483648）
  - キーのとり得る値の種類の例：

    4バイトの浮動小数なら： -3.4e38から3.4e38
- 全てのキー値を，1対1で整数に変換することは不可能
- 違うキーの値でも，ハッシュ値（ハッシュ関数を適用した結果）が同じになるということが起る．

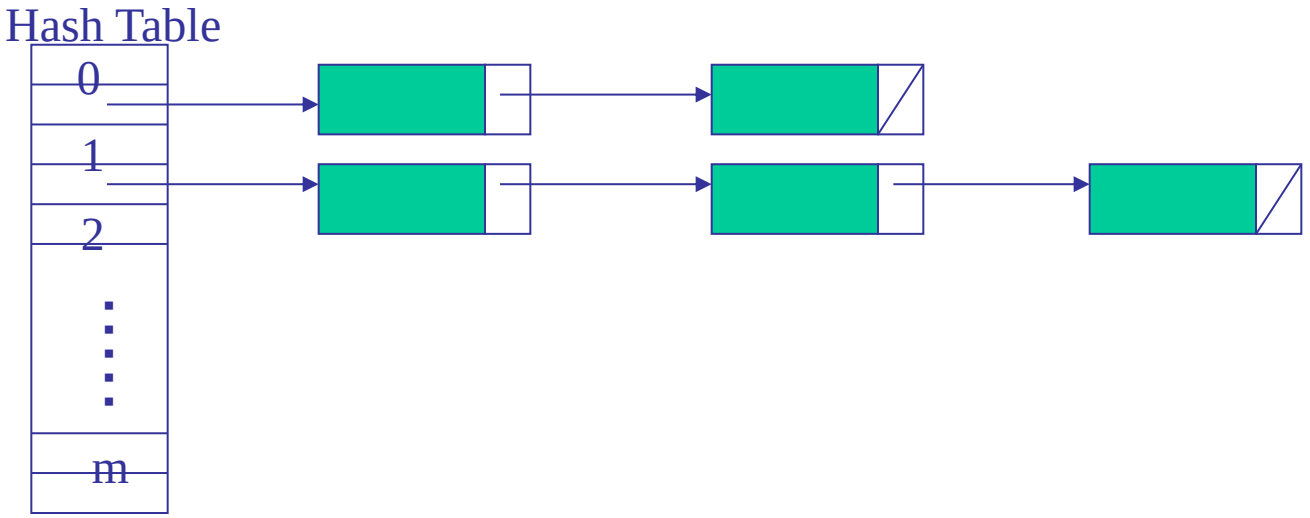
  → このような事態を衝突と呼ぶ

衝突への対処法

- 連鎖法（チェイン法）
  - ハッシュ関数の値（ハッシュ値）が同じであるレコード同士をリストでつないでおく
  - ハッシュテーブルには，リストを指すポインターを入れる
- 開放番地法
  - ハッシュテーブルそのものにレコードを入れる

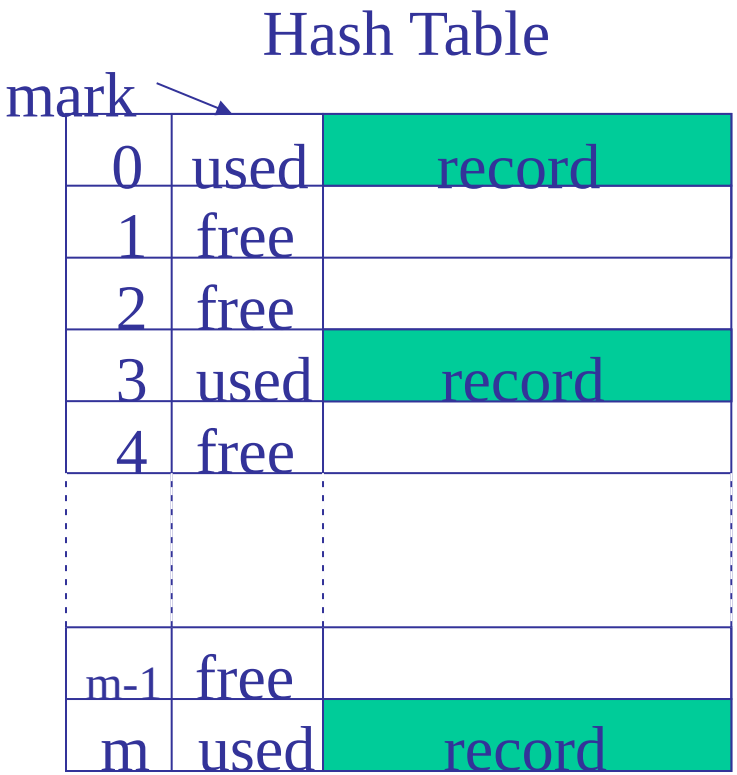
連鎖法(チェイン法)

- ハッシュ関数の値(ハッシュ値)が同じであるレコード同士をリストでつないでおく
- 下の図のようにハッシュテーブルの各要素には，レコードを直接入れるのではなく，レコードをつないだリストの先頭を指すポインターを入れる
- 探索の際は，まずハッシュ関数を計算して，ハッシュテーブルから特定のリストを選んで，その後リスト上を探索するという手順になる



開放番地法のデータ挿入手順

- まず，ハッシュテーブルからハッシュ値が指す要素を調べる.
- もしそこにデータが入っていなかったら，そこにレコードを入れる．別のレコードがすでにそこに入っていれば，ハッシュテーブルの別の場所を調べ，そこが未使用であればそこへデータを入れ，使用済であればまた別の場所を探すという方法をとる.
- この方法では,ハッシュ表の各要素が使用中であるか否かを表す印が必要となる

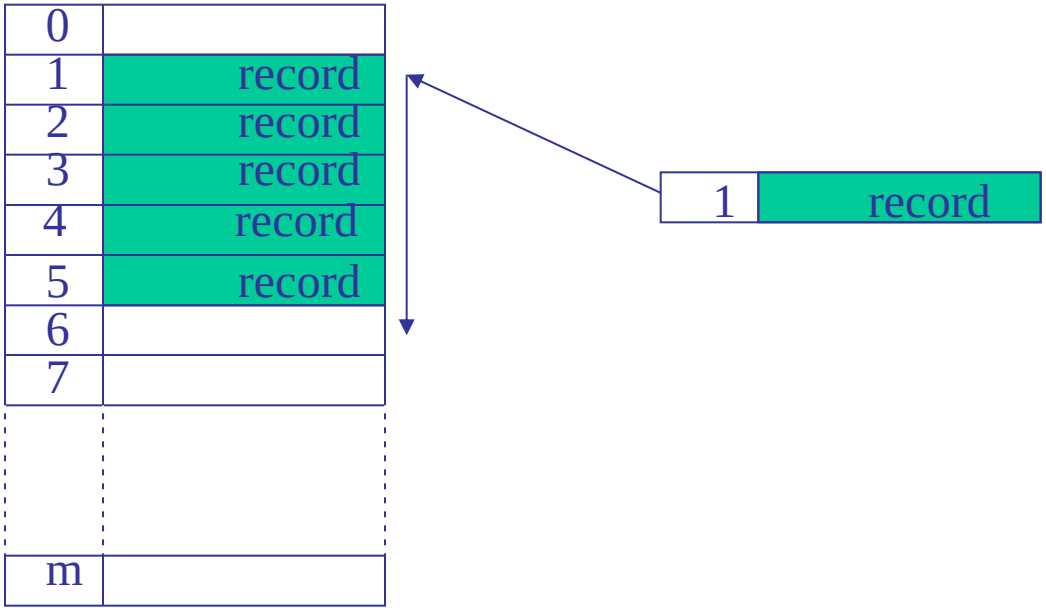


開放番地法

- 開放番地法では，要素を調べたときに使用済であった場合に，次にどこの場所を調べるかということを前もって方針を決めておく
  - 線形走査法
    - 使用済であった場合に，次の要素，その次の要素．．．と添字を1つずつ増やしながら調べていく方法
  - 均一ハッシュ法
    - 違うハッシュ値を持つデータ同士がひとかたまりにならないようにする方法

線形走査法

- 使用済であった場合に，次の要素，その次の要素．．．と添字を1つずつ増やしながら調べていく方法
- クラスターによる性能低下が起こる
  - 下図では，「1から5」に固まり（クラスター）ができている
  - ハッシュ値が1のレコードを挿入しようとするとき，1から5までは塞がっているので，6の場所に入れる．
  - その付近のハッシュ値を持つデータは次々に固まりに加わっていく．
  - このように．同じ値のハッシュ値だけでなく，違う値のハッシュ値の影響を受けて固まりができていく現象をクラスターという



均一ハッシュ法

- 関数をm個（関数h1,h2,....hmとし，これらはある同じ値を入力しても全て違う値を返すとする）用意して，最初に関数h1により，ハッシュテーブルを調べ,塞がっていれば次に関数h2を適用して，その場所を調べる．さらに塞がっている場合，h3,h4,...と空いているまで関数によってその場所をみつけていく．
- これにより，データがバラバラに配置されてクラスターを避ける
- この方法では,ハッシュ関数の計算の手間が増えて効率が悪くなる

チェイン法と開放番地法の比較

- 開放番地法
  - 衝突が起ったときに，ハッシュテーブル内の別の場所に格納していく

    → 扱えるデータの総数はハッシュテーブルの大きさに制限される（欠点）
  - データの削除では，単純にその場所の印を未使用にしてしまうと，探索の時にその場所で止まってしまう．

    → そこで，新たな印として，削除済ということを表す印を加え，（使用中，未使用，削除済）の3種類の印を用いると，削除済というのは探索に関しては，使用中と同じことになる．削除が多くなると，ハッシュテーブルに無駄な部分が増えて，効率が悪くなる（欠点）
- チェイン法
  - 大きな欠点は無い
  - 線形リストを使うため若干メモリ使用量が増える

チェイン法のハッシュ関数

- ハッシュ関数は，キー値を，一様に分散させるようなものが望ましい

  ハッシュテーブルの大きさがmである場合：

  – チェイン法の効率をあげるには，データをm本のリストにできるだけ均等に振り分けて，個々のリストをできるだけ短くすることが必要

- キーの値に対して，できるだけランダムな値を返すハッシュ関数が望ましい
- ハッシュ関数の作り方は，キーの種類，とりうる値の範囲に影響される
- 任意のデータに対して常に一様なばらつきを持ったハッシュ関数を作ることは不可能

チェイン法のハッシュ関数

- キー値が正の整数の場合：
  - キーの値をmで割った余りをハッシュ関数とする
  - キー値は0～m-1の範囲の整数に分けられる
  - 同じハッシュ値となるのは，mの倍数だけ離れた値を持つキー
- ハッシュ関数をできるだけランダムなものにするためには，キーの全てのビットの値が結果に影響することが望ましい
  - mの大きさは，素数をとるのが良いとされる
  - m = 2のK乗とすると，最下位kビットのみでハッシュ値が決まる．データ自体の傾向やハッシュ関数の癖が強調されてばらつきが悪くなってしまうようと言われる
- キーが文字列の時：
  - 何らかの方法で文字列を整数値に変換して，それをmで割った余りをとるようにすることができる

ハッシュ法の欠点

- ハッシュ関数をうまく作って，ハッシュテーブルのサイズmを十分大きくとれば，探索,その他の操作は高速になる
- 対象とするデータの数があらかじめ分かってないと，表の大きさmを決めるのは難しい
  - あまりmを大きくとると記憶域の無駄が大きくなってしまう
- 表の中のデータの順序がでたらめになるので，表の中の値を大きさの順に取り出す場合は，新たに整列を行う必要がある.
  - このような場合，整列を必要としない2分探索木などと比べて劣っている

サンプルプログラム

```
#include<stdio.h>
#include<stdlib.h>
        struct RECOAD{                          /*レコードを構造体で定義*/
                char *value;                    /*レコードの中身*/
                struct RECOAD *next;            /*次のレコードを指すポインタ*/
        }

main(){
        struct RECOAD *table[10];
        int i;
        char in[20];
        for (i=0;i<10;i++)table[i]=NULL;                /*テーブルを初期化*/
        for (i=0;i<100;i++){
                printf("Input Word (end: %%%%%%) :");           /*語を入力*/
                scanf("%s",in);
```

```
            if (strcmp(in,"%%%")==0)break;
            else Chain(in,table);                    /*チェイン法で挿入*/

        }

        while (1){
            printf("Search Word (end: %%%%%%) :");      /*探す語を入力*/
            scanf("%s",in);
            if (strcmp(in,"%%%")==0)break;
            else Search(in,table);                   /*探索*/
        }
}

Chain(char *in,int *table){
        int k,h;
        char *in2;
```

```
        struct RECOAD *rec;
        rec=malloc(sizeof(rec));        /*レコードを作製*/
        in2=malloc(20);
        strcpy(in2,in);                         /*文字列を格納する領域を作成*/
        rec->value=in2;
        h=0;
        for (k=0;k<20;k++){                     /*ハッシュ値を計算(h=0,1,2,...,9)*/
                if (in[k]==NULL)break;
                h=h+in[k];
        }
        h=h % 10;
        rec->next=table[h];             /*レコード挿入*/
        table[h]=rec;
}
```

```
Search(char *in,int *table){
        struct RECOAD *rec;
        int k,h;
        h=0;
        for (k=0;k<20;k++){                    /*ハッシュ値を計算(h=0,1,2,...,9)*/
                if (in[k]==NULL)break;
                h=h+in[k];
        }
        h=h % 10;
        rec=table[h];
        while (1){
        if (rec==NULL){
                printf ("Not Found %s¥n",in);
                break;
                }
```

```
        if (strcmp(rec->value,in)==0){
                printf ("Found %s¥n",in);
                break;
                }
        rec=rec->next;                          /*次のレコード参照*/
        }
}
```