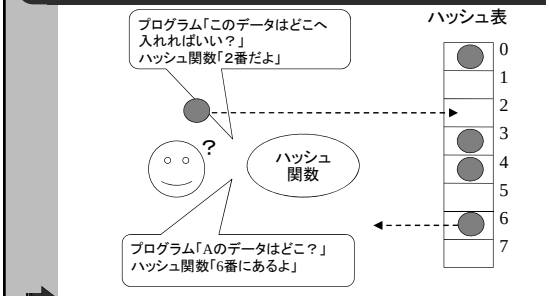
ハッシュ法

高速なデータ探索
アルゴリズム

ハッシュ法（hashing）

- データ探索アルゴリズムの一種
- ハッシュ表と呼ばれる単純な一次元配列にデータを格納
- 探索対象のデータを一定の規則にしたがって格納場所を表す値に変換し、この値を使って格納・検索
  - ハッシュ関数：元データを検索用の数値に変換する関数
  - ハッシュ値：ハッシュ関数によって生成される数値
  - ひとつひとつデータを比較していくのではなく，ハッシュ値から格納場所を決定する
    - データ比較のコストが不要
  - ハッシュ関数が複雑になりすぎる（データ比較のコスト＜ハッシュ関数実行のコスト）と意味がない

ハッシュ法のイメージ



アニメーション

ハッシュ法の特徴

- 利点
  - 一定時間で探索・挿入・削除が可能（概念的にはすべてO（1））
  - キーの大小比較を行わないので、キーに順序関係が不要
    - 任意の文字列やラベルなど、順序関係が明確に定まっていないデータをキーとすることができる
- 問題点
  - 整列は別に実装する必要がある（O(n log n)）
    - 二分探索木は通りがけ順になぞれば整列される
  - 近似探索（目的のものに近いものを探索する）ができない
  - ハッシュ表の拡大・縮小に時間がかかる

ハッシュ法の応用例

- コンパイラやインタプリタの名前管理
  - 変数名や関数名からハッシュ値を生成
  - プログラムで定義されるたくさんの変数・関数の情報検索を高速に
- 連想配列
  - 任意のデータ型（シンボルやオブジェクトなど）を使って参照可能な配列
  - 多くのプログラミング言語がサポート

ハッシュ関数（hash function）

- 与えられた元データから固定サイズの疑似乱数（ハッシュ値）を生成する関数
- 期待される性質
  - 似たデータから近いハッシュ値が生成されない
  - あるデータとハッシュ値が等しい別のデータ（ハッシュ値の衝突(後述)）が容易に生成できない
  - ハッシュ値に値域上の偏りがない
- 通信分野での応用
  - データを送受信する際に、経路の両端でデータのハッシュ値を求めて比較し，通信途中で改ざんされていないか調べる
  - 暗号化の補助
  - 別名：メッセージダイジェスト関数（message digest function）

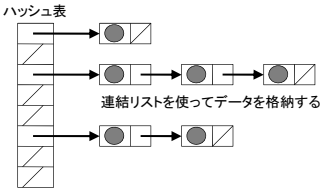
ハッシュ関数の問題

- ハッシュ関数をどう作るのか？
  - 全てのハッシュ値が均等な確率で生成されることが望ましい
- 衝突：異なるデータから同じ格納位置が生成される状況
  - 例：キーの値をハッシュ表の大きさで割り余りを求める
    - ハッシュ表サイズが100の時，bucket = key % 100;
    - keyが10と110の時で衝突
- 衝突してしまうこと自体は仕方がない
- 衝突した時にどういう動作をするのか？
  - チェイン法
  - オープンアドレス法

アニメーション

チェイン法

- 衝突しても同じ位置に複数のデータを格納する
  - 連結リストを使ってデータを連結（チェイン）
- 連結リストの部分があまりに長くなると、検索効率が落ちる
  - ハッシュ関数＋リストを辿る処理



アニメーション

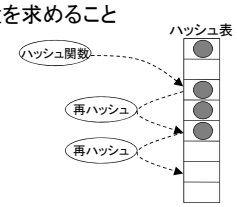
チェイン法のプログラミング例

```
/* ハッシュ表にデータを追加する */
void add(int key, int data) {
  struct cell *p;
  int bucket;
  /* チェインのための領域を確保 */
  p = malloc(sizeof(struct cell));
  if(p == NULL){
    error("メモリ不足");
  }
  bucket = hash(key);  /* 追加位置 */
  pが指し示すセルにキーとデータを代
    入;
  p->next = ハッシュ表[bucket];
  ハッシュ表[bucket] = p;
}
```

```
/* ハッシュ表からデータを検索する */
int search(int key) {
  struct cell* p;
  int bucket;
  bucket = hash(key);  /* 検索位置 */
  /* 連結リストを辿る */
  for(p=ハッシュ表[bucket];p!=NULL;
        p=p->next){
    if(pの先のキーがkeyと等しい){
      return pの先のデータ;
    }
  }
  return 見つからなかった;
}
```

オープンアドレス法

- 衝突した時，新たな格納位置を計算によって求め直す
  - 複数のデータが同じ格納位置を使うことはない
  - ハッシュ表の大きさを超える数のデータは格納できない
    - チェイン法は連結リストを使うためメモリさえ許せば限界なし
- 再ハッシュ：新しい格納位置を求めること
  - ハッシュ関数とは異なる計算
  - 有効な格納位置が発見されるまで繰り返す



アニメーション

オープンアドレス法の注意

- 空いている格納場所の表現
  - 一度も使われていない場所
  - 格納されたあと削除された場所
- データ検索時，再ハッシュで格納場所を辿る過程で，
  - 一度も使われていない場所
    ＝これ以上再ハッシュは不要→データは見つからなかった
  - 格納されたあと削除された場所
    ＝再ハッシュすることで見つかる可能性あり

オープンアドレス法のプログラミング例

```
/* ハッシュ表にデータを追加する */
int add(int key, int data) {
  int bucket, count, k;
  bucket = hash(key);  /* 追加位置 */
  while(ハッシュ表サイズの繰り返し){
    k = ハッシュ表[bucket].key;
    if(kが空を意味する || kが削除済みを意味する){ /* この位置にデータ追加 */
      ハッシュ表[bucket].key = key;
      ハッシュ表[bucket].data = data;
      return 成功;
    }
    bucket = rehash(bucket);  /* 再ハッシュにより次の候補位置へ */
  }
  return 失敗;
}
```

オープンアドレス法のプログラミング例

```
/* ハッシュ表からデータを検索する */
int search(int key) {
  int bucket, count, k;
  bucket = hash(key);  /* 検索位置 */
  while(ハッシュ表サイズの繰り返し){
    k = ハッシュ表[bucket].key;
    if(kが空を意味する) break; /* 空のバケットであった */
    if(kが削除済みを意味するものでない && kが目的としているキー){
      return ハッシュ表[bucket].data;
    }
    bucket = rehash(bucket);  /* 再ハッシュにより次の候補位置へ */
  }
  return 見つからなかった;
}
```

次回予定

- ハッシュ法のプログラミング演習
  - チェイン法とオープンアドレス法
  - データ探索時間の計測
    - 演習2(線形探索、二分探索)との比較