

key = 8

8

arr[3] == 8 なので、3 を戻り値として返す

図 5.5 二分探索の例 4 binary_search(arr, 8, 3, 3) を実行

もし、arr[3] が整数値 8 でなければ、部分配列 arr[3:2] または arr[4:3] に対して binary_search 関数を呼び出すので、5 行目の条件判定式の $r < p$ が True になります。そのため、None が戻り値として返されます。すなわち、部分配列の大きさが 1 のときにキーが見つからなければ、配列内にキーと同じ値をもつ要素が存在しないことを意味します。

■ 二分探索プログラムの実行

ソースコード 5.3 (binary.py) を実行した結果をログ 5.4 に示します。整数値 8 は 4 番目の要素であるインデックス 3 に格納されているので、3 行目で要素が見つかったとの旨が表示されます。

ログ 5.4 binary.py プログラムの実行

```
01 $ python3 binary.py
02 配列: [1, 3, 5, 8, 12, 17, 25, 33, 54, 85]
03 arr[ 3 ]に要素が見つかりました。
```

5.3

ハッシュ探索

ハッシュ探索 (hash search) とは、平均で $O(1)$ の計算量で指定したキーをもつ要素を探索できるアルゴリズムです。ちから技とは無縁ともいえる、素晴らしいアルゴリズムです。何かの問題にぶつかったら、先達者が知恵と工夫、長足の進歩を促したということを思い出してください。まず、ハッシュ探索の核となるハッシュ関数とハッシュ表について解説します。

■ 5.3.1 ハッシュ関数

ハッシュ関数 (hash functions) は任意のビット列を固定長のビット列に変換する関数です。数学

的には、**ハッシュ関数****H**は $H : \{0, 1\}^* \rightarrow \{0, 1\}^k$ となります。すなわち、入力が任意長のビット列で、出力が k ビットの長さをもつ固定長のビット列です。ハッシュ関数への入力を x とすると、出力は $H(x)$ と記述できます。このハッシュ関数の出力を**ハッシュ値 (hashed value)** と呼びます。

■ ハッシュ関数の用途

良いハッシュ関数の定義はアプリケーションによってさまざまですが、ここでは**衝突 (collision)** が少ないハッシュ関数を想定しています。衝突とは、異なる入力値を入力したにもかかわらず、同じハッシュ値が出力されることです。すなわち、入力値 x と y ($x \neq y$) に対して $H(x) = H(y)$ となることです。厳密には**衝突困難性 (collision resistance)** といった専門用語がありますが、本筋から逸れるので割愛します。

ハッシュ関数は、さまざまなアプリケーションで使用されています。たとえば、ソフトウェアをインターネット上で配布する場合は、ソフトウェアのバイナリファイルからダイジェスト (digest) を公開します。第三者がベンダーを名乗って偽のソフトウェアを配布するのを防ぐためです。このときにハッシュ関数を用いて、ダイジェストを生成します。具体的には、ソフトウェアのバイナリファイルを 256 ビットまたは 512 ビットなどの固定長のビット列に変換して、その値を公開します。少しでもファイルの中身が異なれば、異なるハッシュ値が生成されます。そのため、第三者が偽のソフトウェアを公開して配布したとしても、ハッシュ値が公式のものと異なれば、それは偽物だと判断できます。

図 5.6 を見てください。著者が出版した本で使用しているソースコードをオーム社のウェブページで配布しています。そのウェブページの抜粋です。

内容紹介 目次 ダウンロード 正誤表

ここでは、本書を取り上げたサンプルプログラムを、圧縮ファイル（zip形式）で提供しています。圧縮ファイル（ohmv2.zip：約77KB）をダウンロードし、ご解凍してご利用ください。

- 本ファイルは、本書をお買い求めになった方のみご利用いただけます。ご利用にあたっては、本書の注意書き等をよくお読みください。また、本ファイルの著作権は、本書の著作者にあり、配布等は禁止します。
- 本ファイルを利用したことによる直接あるいは間接的な損害に関して、著作者およびオーム社はいっさいの責任を負いかねます。利用は利用者個人の責任において行ってください。また、ソフトウェアの動作・実行環境・操作についての質問には一切お答えすることはできません。

sha256: 28fdd60629a1818122c155b43235a6798657a2cb024f8bfdd0780a66423b3bf

→ ohmv2.zip (約77KB)

ハッシュ値

図 5.6 ハッシュ値の例

画像の下の方に載せてある「ohmv2.zip (約 77KB)」が配布ファイルです。その上に以下に示すランダムな 16 進数の値が記載されています。

28fdd60629a1818122c155b43235a6798657a2cb024f8bfdd0780a66423b3bf

これが SHA256 と呼ばれる手法で計算した ohmv2.zip のハッシュ値です。もし、第三者がミラーサイトだと偽って偽物の配布ファイル（本家のソフトウェアを装ったマルウェアなど）を公開したとしても、偽物のファイルのハッシュ値は上記の値と異なる可能性が極めて高いので、なりすましができません。256 ビットであれば、現実的な時間内に衝突を発見すること（同じハッシュ値をもつ別のファイルの発見）はほとんど不可能です。

他にも応用例はたくさんあります。情報セキュリティや暗号論の授業で学ぶメッセージ認証コードでもハッシュ関数が使用されます。このような分野では、暗号論的ハッシュ関数が用いられます。

■ 5.3.2 ハッシュ表（ハッシュ探索の準備のために）

本章では、ハッシュ関数を応用したデータ構造である**ハッシュ表 (hash table)**について解説します。ハッシュ表の行は、データのハッシュ値と、実際のデータが格納しているオブジェクトへのポインタ 2 つからなります。ハッシュ値はインデックスとしての機能をもちます。つまり、探索するデータのハッシュ値の計算結果は、それはつまり、即インデックスとなり、オブジェクトへのポインタが得られます。つまり、 $O(1)$ の計算量で済むという優れたアルゴリズムです。

そして、オブジェクトは、連結リストと同様にデータと値という 2 つのデータから構成されます。たとえば、 $(3, "Alice")$ や $(12, "Bob")$ などです。整数値 3 がハッシュ値であり、要素を識別するキーです。 $"Alice"$ が人名を表す値です。

ハッシュ表は、ハッシュ値と要素へのポインタ 2 つの項目から列で構成されます。ハッシュ値は表の行を指すインデックスそのものです。要素にはデータを格納しているオブジェクトへのポインタが格納されます。

各要素はキーと値の 2 つのデータから構成されます。ここではキーを整数値、値を人の名前（文字列）とします。たとえば、 $(3, "Alice")$ や $(12, "Bob")$ などです。整数値 3 が要素を一意に識別するキーとなり、 $"Alice"$ が人の名前を表すデータです。キーと値が同じ整数値だと紛らわしいので、値を文字列としました。

ハッシュ表では、キーに対してハッシュ関数を適応させ、ハッシュ値に対応するインデックスに要素を格納します。ここではハッシュ関数 H を $H(x) = x \bmod 10$ と定義します。ここで \bmod は剰余算を表すため、キーを整数値 10 で割った値をハッシュ値とします。実際にはもっと複雑なハッシュ関数を用いますが、理解しやすさのために単純なハッシュ関数を定義しました。各要素のキーをハッシュ関数への入力し、出力であるハッシュ値に対応するインデックスに要素を格納します。

具体例として、 $(3, "Alice")$ と $(12, "Bob")$ 、 $(37, "Chris")$ 、といった 3 つの要素をハッシュ表に格納したときの状態を図 5.7 に示します。各要素のハッシュ値はそれぞれ 3 と 2、7 です。そのため、インデックス 3 には $(3, "Alice")$ というオブジェクトへのポインタが格納されます。他の 2 つも同様です。なお、エントリーとはハッシュ表の各々の行のことです。

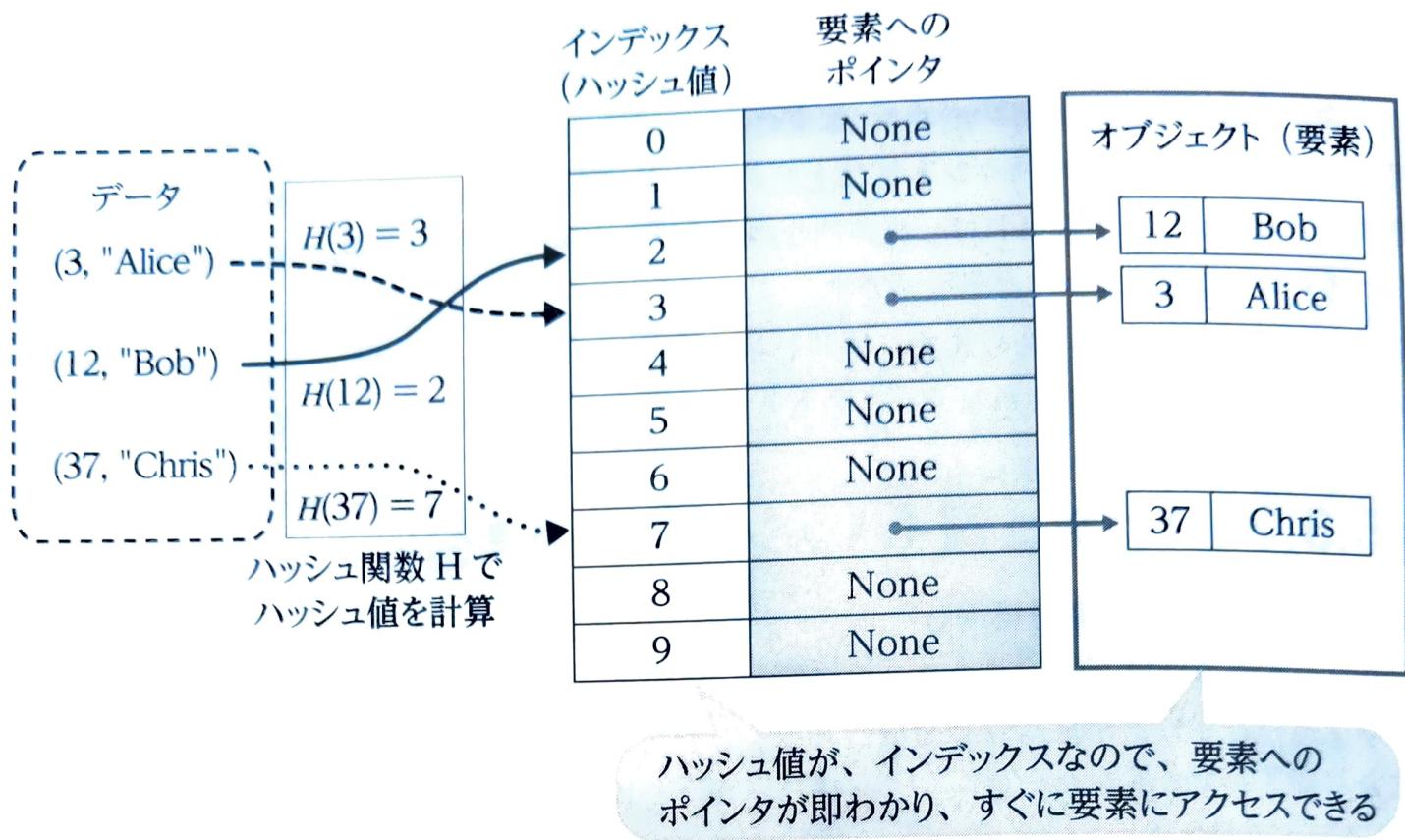


図 5.7 ハッシュ表の例（エントリー数が 10 のハッシュ表）

ハッシュ値の出力範囲は入力範囲より遥かに小さいので、衝突（異なるキーをもつ 2 つの要素が同じハッシュ値になる）が発生する可能性があります。対処方法はいくつかありますが、本書ではハッシュ探索で用いられる**チェイン法 (chain method)** を解説します。チェイン法では、衝突が起こった要素を連結リストで接続します。すなわち、ハッシュ表の各エントリーは連結リストへのポインタとなります。

たとえば、前述の 3 つの要素が、(3, "Alice") と (12, "Bob")、(33, "Chris") だったとしましょう。Alice と Chris のキーが 3 と 33 なので、 $H(x) = x \bmod 10$ より、 $H(3) = H(33) = 3$ となり、衝突します。この場合は、連結リストを使って 2 つの要素を接続します。連結リスト内での順番は、ハッシュ表へ要素を挿入した順番になります。

(3, "Alice") と (12, "Bob")、(33, "Chris") という順番で要素をハッシュ表へ挿入したときの状態を図 5.8 に示します。Alice と Chris のオブジェクトが連結リストで繋がっていることが確認できます。このように鎖 (chain) で繋げたように見えるので、チェイン法と呼ばれるわけです。

エントリー数が 10 のハッシュ表

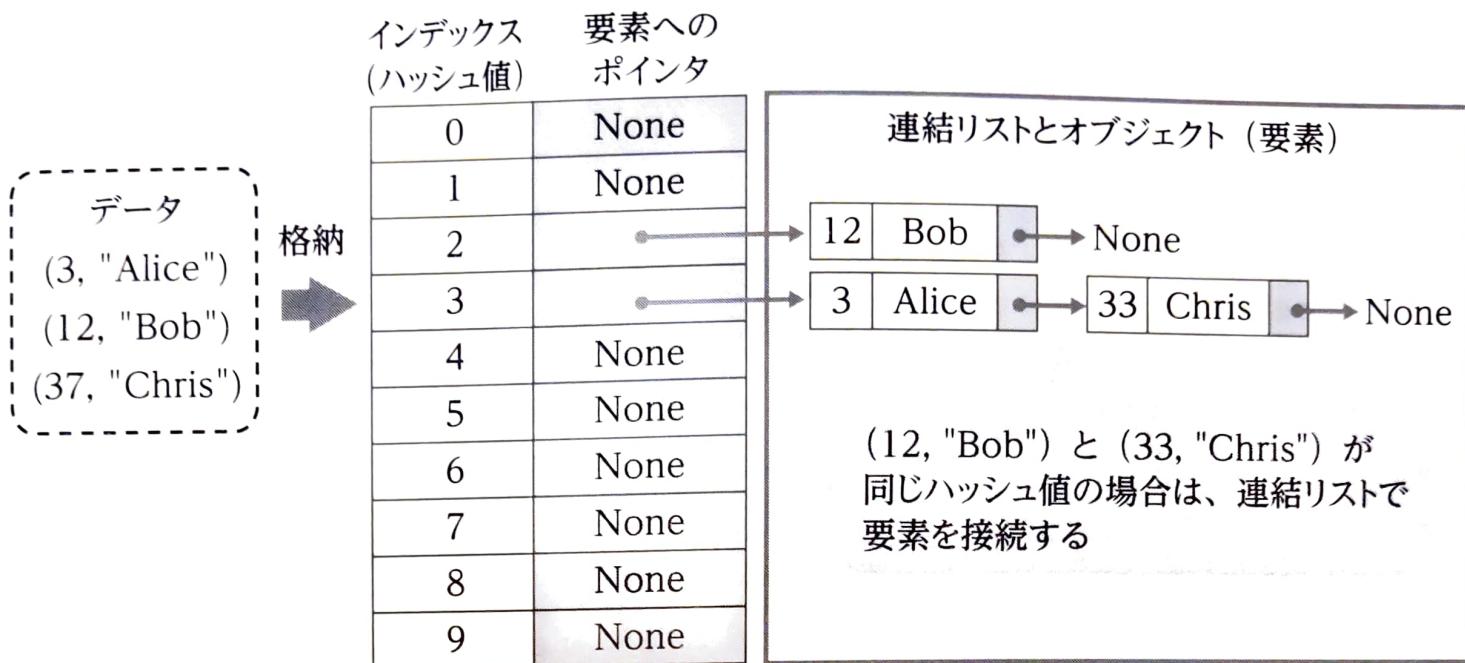


図 5.8 チェイン法の例

データ構造で用いるハッシュ関数の出力ビット数はハッシュ表の大きさに依存します。前述のソフトウェアのダイジェストのように 256 ビットという大きさのハッシュ値は使用できません。32 ビットでもハッシュ表のエントリー数が 2^{32} 個になるので、現実的なアプリケーションではもっと小さなビット数を使用します。そのため、衝突が起こる可能性があります。衝突が発生する頻度を小さくするには、データ数 n に対して適切なハッシュ表のエントリー数を設定する必要がありますが、アルゴリズムの話題からは逸れるので割愛します。

■ ハッシュ探索の概要

ハッシュ探索では、指定したキーの要素を探索します。データとなる各要素はハッシュ表の中に連結リストの要素として格納されています。そのため、ハッシュ探索アルゴリズムを、整数値として指定したキーに対応する要素のオブジェクトを返す関数として定義します。

ハッシュ表における要素を表現するために以下のようない名前のクラスを定義します。キーと値を表すためのクラスメンバとして変数 `key` と `val` を定義します。また、片方向連結リストに必要な次の要素へのポインタとして、クラスメンバ `next` を定義します。

```

class MyNode:
    def __init__(self, key, val):
        self.key = key
        self.val = val
        self.next = None
  
```

上記の書式でハッシュ表にデータとなるオブジェクトを格納します。ハッシュ表を表すクラスとして、MyHashTable を以下のように定義します。コンストラクタの引数である size はハッシュ表のエントリー数です。引数 size で指定した大きさの配列 tbl を生成して、ハッシュ表のエントリーを実装します。配列 tbl の各要素は、None で初期化します。

```
class MyHashTable:  
    def __init__(self, size):  
        self.tbl = [None] * size
```

要素を格納すときは、MyNode クラスのインスタンスを生成して、ハッシュ値 $H(key)$ を計算します。ハッシュ値が要素を格納するインデックスなので、self.tbl [$H(key)$] が MyNode オブジェクトを格納する場所となります。もし、self.tbl [$H(key)$] にその他のオブジェクトが格納されていれば、連結リストの最後尾にオブジェクトを挿入します。

探索方法は単純です。指定したキーを key とします。まず、ハッシュ関数を適応させて、ハッシュ値 $H(key)$ を計算します。次にハッシュ値と同じインデックス（表の行）に要素があるかどうかを確認します。当該インデックスのエントリーが None であれば、ハッシュ表に指定したキーの要素が存在しないことを意味します。要素が存在すれば、当該インデックスのエントリーには連結リストへのポインタが格納されています。そのポインタから当該キーに対応する要素のオブジェクトを取得します。

衝突が発生していないければ、連結リストの要素数は 1 つだけです。すなわち、ハッシュ値を計算し、当該インデックスの要素を確認するだけで探索が終了します。そのため、計算量が $O(1)$ となります。衝突が頻繁に発生しないようにハッシュ表を設計するので、通常は $O(1)$ で探索が終了しますが、運が悪いと $O(n)$ の時間がかかります。

たとえば、すべての要素のハッシュ値が同じになったとしましょう。もちろん現実問題として、のような事態は起こらないでしょうが、最悪のケースとして考える必要があります。この場合、すべての要素が同じエントリーに格納され、連結リストの大きさが n になります。そのため、指定したキーの要素を探索するためには、連結リストを先頭から順にたどる必要があるので $O(n)$ となります。

■ ハッシュ表の操作

また、ハッシュ表はデータ構造なので、第 3 章で解説したように要素の挿入 (insert) と削除 (delete) を定義する必要があります。チェイン法を使用する場合、ハッシュ値を計算する手続きが必要であること以外は、連結リストへの挿入操作や削除操作と同様です。なお、ハッシュ表においてインデックスを指定した要素の取得 (get) は定義しません。各要素はキーと値をもつことを前提としているため、インデックスを指定した要素の取得に意味がないからです。

本書での実装例では、上記の MyHashTable クラスに探索を行う search メソッド、ハッシュ値を計算する __get_hash メソッド、要素の挿入を行う insert メソッド、要素の削除を行う delete メソッドを定義します。

```

109     if x != None:
110         print("取得した要素", x.to_string())
111     else:
112         print("指定したキーに対応する要素が存在しません。")
113
114     # キー233の要素を削除
115     my_hash.delete(233)
116     print("削除後のハッシュ表の状態:")
117     print(my_hash.to_string())

```

MyNode オブジェクトの情報と MyHashTable オブジェクトがもつハッシュ表の情報を文字列として返す `to_string` メソッドをそれぞれのクラス内で定義しています。アルゴリズムに直接関係ないので、詳細は割愛します。

■ main 関数の説明

93 行目～117 行目で `main` 関数を定義しています。まず、95 行目で、エントリー数が 10 のハッシュ表を生成します。生成した `MyHashTable` クラスのインスタンスを変数 `my_hash` に格納します。

98 行目～103 行目で 6 つの要素を `my_hash` に `insert` メソッドを使用して挿入します。挿入する要素は、それぞれ (3, "Alice") と (12, "Bob")、(233, "Chris")、(95, "David")、(183, "Eav")、(25, "George") です。104 行目と 105 行目の命令で、ハッシュ表の中身を表示します。

108 行目からが本題です。整数値 233 をキーとしてもつ要素をハッシュ表の中から探索します。`my_hash.search(233)` と記述して `search` メソッドを実行し、探索結果を変数 `x` に格納します。233 をキーとしてもつ要素が存在すれば、変数 `x` に `MyNode` オブジェクトが格納されます。存在しなければ `x` の値が `None` になります。109 行目～112 行目で、変数 `x` の情報を表示します。

削除操作の例として、115 行目では、整数値 233 をキーとしてもつ要素をハッシュ表から削除します。このために `my_hash.delete(233)` と記述して、`delete` メソッドを呼び出します。要素を削除したあとのハッシュ表の状態を 116 行目と 117 行目の命令で表示します。

■ get_hash メソッド (ハッシュ値の計算) の説明

15 行目と 16 行目でハッシュ値を計算する `_get_hash` メソッドを定義しています。`MyHashTable` クラス内だけで使用するメソッドなので、メソッド名の前にアンダーバー (_) を 2 つ付けています。前述の説明では、 $H(x) = x \bmod 10$ という式でハッシュ値を計算していましたが、ここでは整数値 10 の代わりにハッシュ表のエントリー数を用います。`main` 関数の 93 行目で、割った余りがハッシュ値になります。

■ insert メソッド (ハッシュ表への新しい要素の挿入) の説明

19行目～31行目で新しい要素をハッシュ表へ挿入するinsert メソッドを定義しています。メソッドへの入力はキーと値なので、変数key と val を引数とします。まず、新しい要素を挿入するハッシュ表のインデックスを決めるために、21行目で__get_hash メソッドを適応して、key に対応するハッシュ値を計算します。その結果を変数hash_val に格納します。

24行目で引数として与えられたkey と val からMyNodeオブジェクトを生成し、変数n に格納します。挿入処理は25行目～31行目で行います。25行目のif文で、self.tbl[hash_val] がNoneかどうかを判定します。Trueであれば、当該インデックスに何もない状態です。この場合、26行目のself.tbl[hash_val] = n という命令で、新しい要素のオブジェクトを格納します。変数n のn.next はNoneで初期化されているので、特に何もする必要がありません。self.tbl[hash_val] が連結リストの先頭要素を参照し、連結リストには要素n が1つだけ含まれている状態です。

条件判定式のself.tbl[hash_val] == None がFalseだった場合、28行目～31行目のelseブロックを実行します。新たに挿入する要素のkey と同じハッシュ値になる要素がすでにハッシュ表に存在する状態なので、衝突が発生したことを意味します。この場合、self.tbl[hash_val] にある連結リストの最後尾に要素n を挿入します。挿入処理は連結リストと同様に、先頭から各要素のnext に格納されている次の要素へのポインタをたどっていきます。

main関数内の98行目～103行目の挿入命令を実行した後のmy_hash.tblの状態は図5.9に示すとおりです。

エントリー数が10のハッシュ表

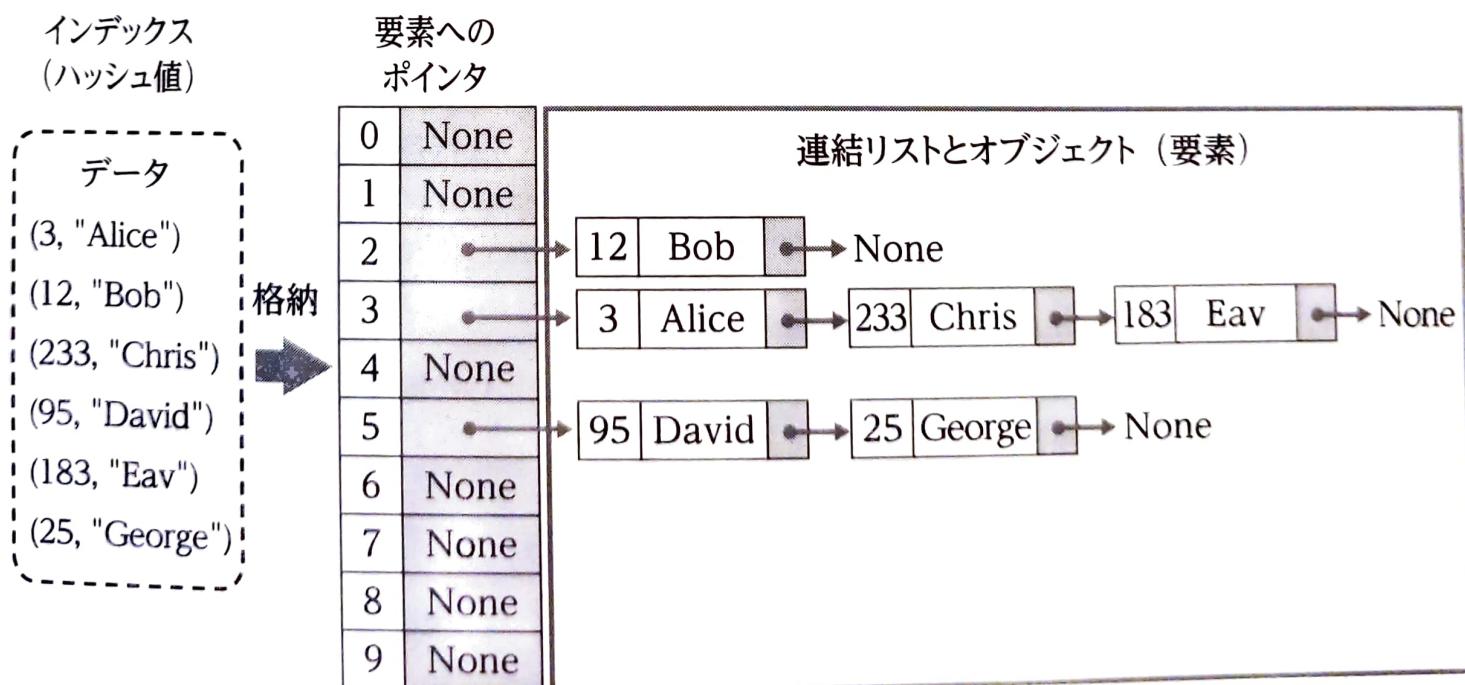


図5.9 要素の入力後のハッシュ表の状態

■ delete メソッド (ハッシュ表からの要素の削除) の説明

34 行目～58 行目で指定したキーをもつ要素をハッシュ表から削除する delete メソッドを定義しています。引数としてキーを変数 key で受け取ります、まず、36 行目で削除したい要素の場所を調べる必要があるので、`_get_hash` メソッドを適応させて変数 key のハッシュ値を計算します。計算結果を変数 hash_val に格納します。

削除処理は片方向連結リスト内の要素を削除する処理と同じです。片方向なので、第 3.2 節で解説した双方向連結リストからの要素の削除とは少し異なります。各要素は次の要素を参照するポインタをクラスメンバ next に保存していますが、1 つ前の要素へのポインタは保持していません。そのため、連結リストをたどっていくときに、前の要素を記録しておく必要があります。39 行目で、変数 prev_ptr を宣言し None で初期化し、40 行目で変数 ptr を `self.tbl[hash_val]` で初期化します。変数名のとおり prev_ptr が 1 つ前の要素のポインタ、ptr が現在参照中の要素のポインタとなります。

41 行目～54 行目の while ループで、ptr が None でない限り繰り返し処理を行います。そもそも `self.tbl[hash_val]` が None の場合、ptr が None で初期化されるので、ループ内に 1 度も入らずに処理が終了します。また、`self.tbl[hash_val]` が参照する連結リスト内に変数 key の値をもつ要素が存在しない場合は、ループを繰り返したあとに ptr が None となり、while ループを抜けて、そのまま処理が終わります。すなわち、キーに対応する要素がハッシュ表にない場合は、何もせずにメソッドの処理が終了します。

while ループに入ると、42 行目の if 文で `ptr.key == key` かどうかを判定します。False であれば、57 行目と 58 行目まで進み、prev_ptr と ptr を更新します。ここでは、連結リストの次の要素へポインタを移動させる処理をしています。

True であれば、if ブロックの中に入って 43 行目～52 行目の命令を実行します。このとき変数 ptr が参照している要素が連結リストから削除する要素となります。if 文がネストされていることがわかります。これは削除したい要素が連結リストの先頭または最後尾かどうかを確認して個別に処理する必要があるため、複数の if 文で分岐処理を行っています。43 行目で ptr が参照する要素が最後尾の要素かどうかを判定し、44 行目または 49 行目で先頭の要素かどうかを判定します。そのため、合計で 4 つのケースがあります。

ケース 1) ptr は最後尾の要素ではなく、先頭の要素でもない。

ケース 2) ptr は最後尾の要素ではないが、先頭の要素である。

ケース 3) ptr は最後尾の要素であるが、先頭の要素ではない。

ケース 4) ptr は最後尾の要素かつ先頭の要素である (連結リストの要素が ptr だけ)。

ptr が参照する要素が最後尾にあるかどうかの判定は、`ptr != None` でわかります。True であれば、最後尾の要素ではありません。先頭の要素であるかどうかの判定は、`prev_ptr != None` を調べます。True であれば、先頭の要素ではありません。

要素の削除はハッシュ表のエントリー (`self.tbl[hash_val]`) または ptr が参照している 1 つ前の要素の情報 (`prev_ptr.next`) を変更するだけです。ケース 1 は図 5.10 に示す状態です。連結リストには 3 つの要素が含まれていますが、ptr が参照する要素の前後に 2 つ以上の要素がある場合も同様です。

インデックス
(ハッシュ値) 要素への
ポインタ

...	...
...	...
hash_val	self.tbl[hash_val]
...	...
...	...

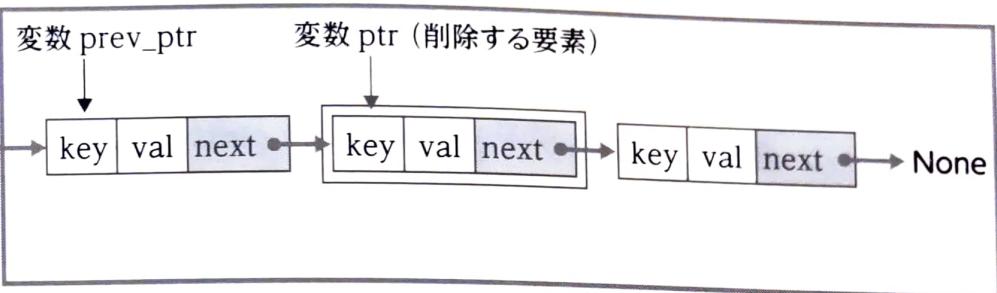


図 5.10 要素の削除ケース 1-1

図を見れば一目瞭然ですが、ptr が参照する要素を連結リストから削除する場合は、prev_ptr.next を ptr.next が参照するオブジェクトに設定すれば良いことがわかります。そのため、45 行目の prev_ptr.next = ptr.next という命令します。その後のハッシュ表の状態は図 5.11 に示すとおりです。

インデックス
(ハッシュ値) 要素への
ポインタ

...	...
...	...
hash_val	self.tbl[hash_val]
...	...
...	...

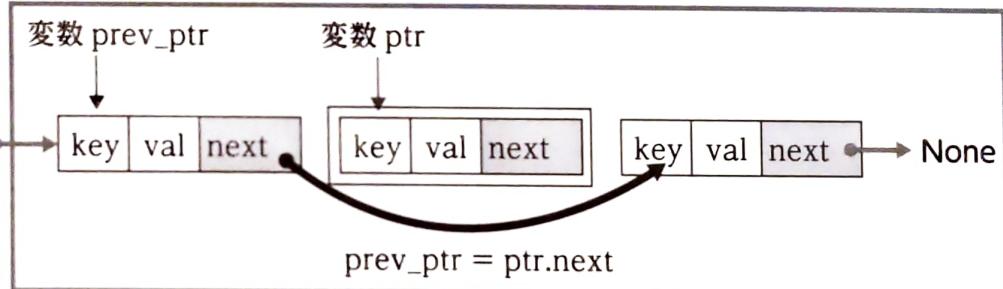


図 5.11 要素の削除ケース 1-2

ケース 2 の場合、ハッシュ表の状態は図 5.12 に示すとおりです。ptr は連結リストの先頭要素なので、self.tbl[hash_val] が ptr.next を参照するように修正します。

インデックス
(ハッシュ値) 要素への
ポインタ

変数 prev_ptr は None

...	...
...	...
hash_val	self.tbl[hash_val]
...	...
...	...

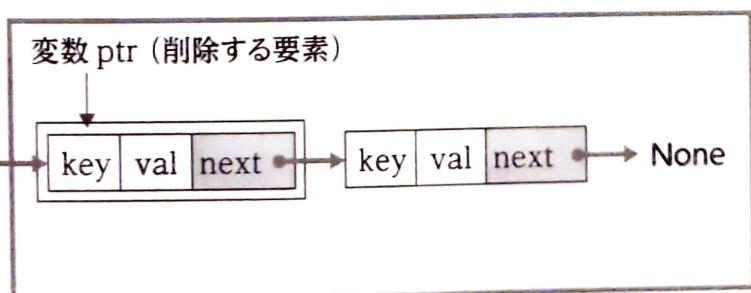


図 5.12 要素の削除ケース 2-1

47行目の `self.tbl[hash_val] = ptr.next` という命令を実行した後のハッシュ表の状態を図 5.13 に示します。ptr が参照する要素が連結リストから外れたことが確認できます。

5.13

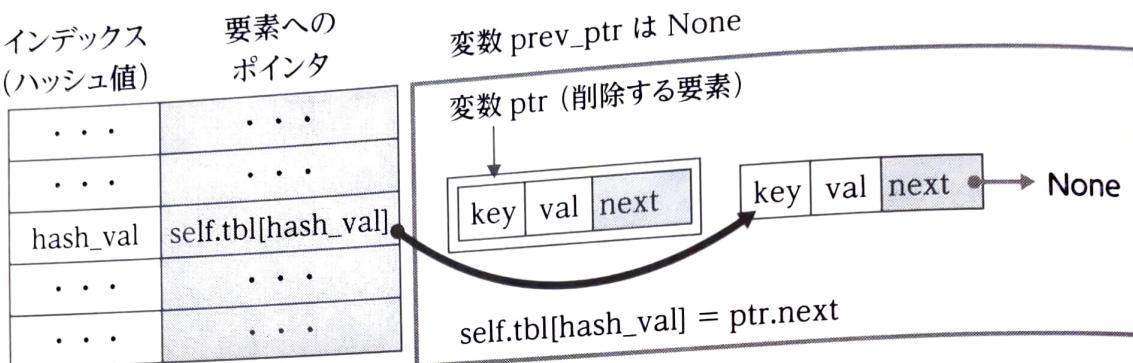


図 5.13 要素の削除ケース 2-2

ケース 3 の場合は図 5.14 に示す状態になります。ptr が参照する要素は最後尾にあるので、1つ前の要素の `next` を `None` に変更するだけです。

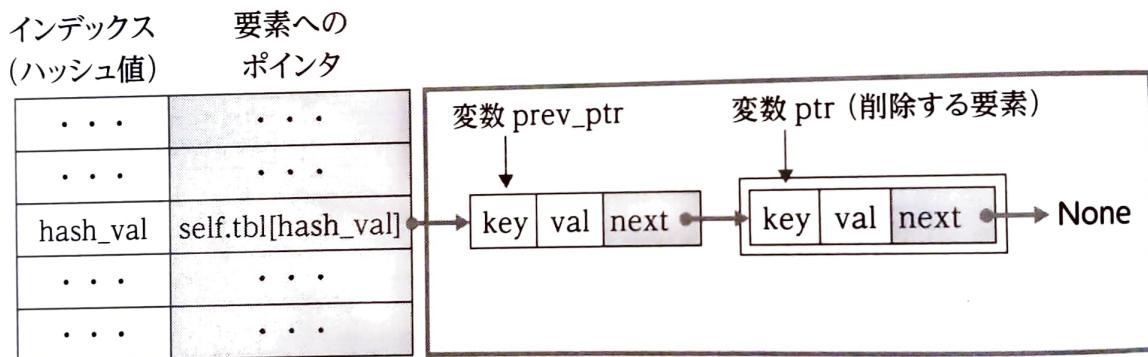


図 5.14 要素の削除ケース 3-1

50行目の `prev_ptr.next = None` を実行した後のハッシュ表の状態を図 5.15 に示します。

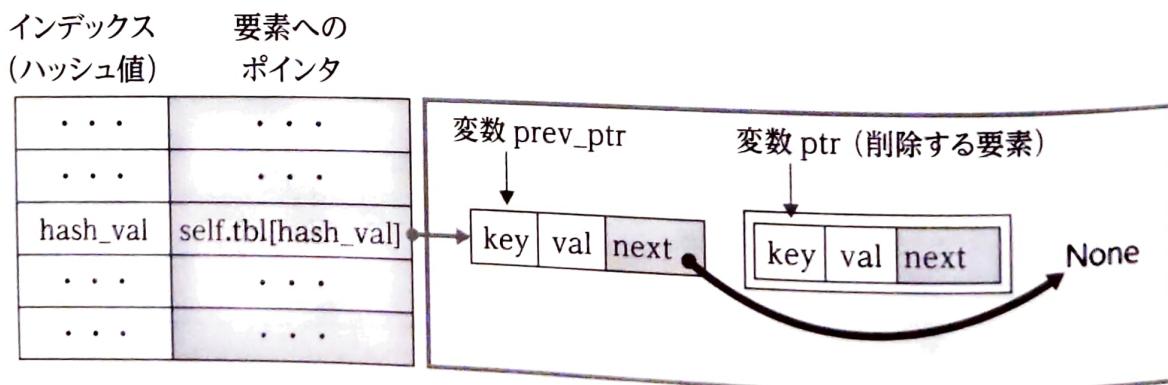


図 5.15 要素の削除ケース 3-2

ケース 4 では、連結リストには `ptr` が参照する要素しか含まれていません。視覚化すると図 5.16 のようになります。ハッシュ表の当該エントリーを `None` にするだけです。

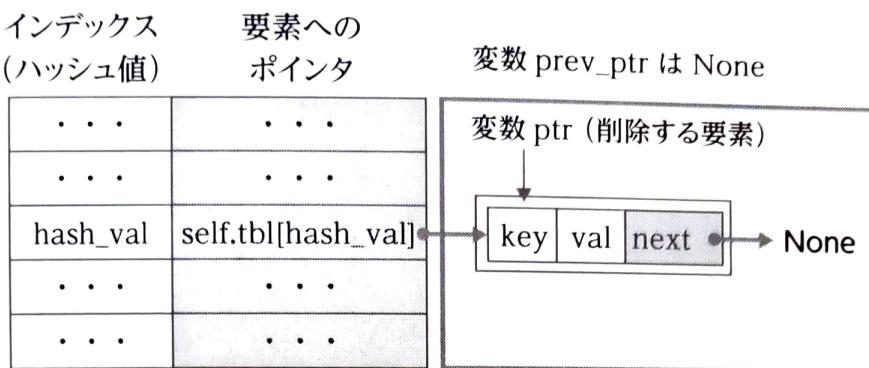


図 5.16 要素の削除ケース 4-1

52 行目の `self.tbl[hash_val] = None` という命令を実行した後のハッシュ表の状態を図 5.17 に示します。当該エントリーには何もない状態になります。

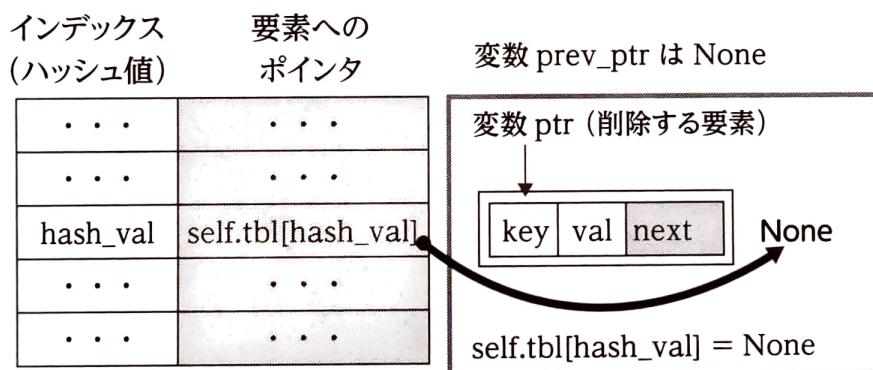


図 5.17 要素の削除ケース 4-2

要素の削除操作が終わると 54 行目の `return None` を実行します。これ以上 while ループを続行する必要がないので、ここでメソッドの処理を終了させるための命令です。

■ search メソッド (指定したキーの探索) の説明

61 行目～77 行目で指定したキーを探索する `search` メソッドを定義しています。引数は探索したい要素のキーなので、変数 `key` で受け取ります。まず、63 行目で、`_get_hash` メソッドを呼び出して `key` のハッシュ値を計算します。計算結果を変数 `hash_val` に格納します。

66 行目でハッシュ値に対応するハッシュ表のインデックスを調べます。`hash_val` がインデックスになるので、if 文を用いて `self.tbl[hash_val]` が `None` かどうかを判定します。`None` でなければ、連結リストの先頭要素へのポインタが `self.tbl[hash_val]` に格納されていますので、68 行目～74 行

目の if ブロック内に入り、条件によっては次の while ループに入っていきます。

67 行目でいったん self.tbl[hash_val] の値を変数 ptr に格納します。連結リストに 2 つ以上の要素があるケースに対応するためです。68 行目の if 文で、連結リストの先頭要素が指定したキーをもつ要素かどうかを確認します。条件判定式の ptr.key == key が True であれば、69 行目で ptr を戻り値として返します。この時点で、ptr は key と同じ値をもつ要素を参照しているので、MyNode オブジェクトが戻り値として返されます。

68 行目の ptr.key == key が False であれば、71 行目～74 行目の while ループで連結リストを前からたどりながら要素を探します。ptr.next が None にならない限り、72 行目の ptr = ptr.next を実行し、73 行目の if 文で、ptr が参照する要素が探索中の要素であるかどうか調べます。key と同じ値をもつ要素が存在すれば、74 行目で当該オブジェクトを戻り値として返します。

66 行目の if 文で、self.tbl[hash_val] != None: が False であれば、77 行目の命令が実行され、None が戻り値として返されます。また、True であったとしても、66 行目～74 行目の if ブロック内の処理でキーに対応する要素が見つからない可能性もあります。この場合も 77 行目の命令が実行されます。

■ ハッシュ探索プログラムの実行

ソースコード 5.5 (my_hash.py) を実行した結果をログ 5.6 に示します。6 つの要素を挿入したあとのハッシュ表の状態が 3 行目～5 行目に表示されています。簡略化のため None のインデックスは表示していません。ハッシュ値の衝突が発生したエントリーでは、挿入した順番に要素が連結リストで接続されていることが確認できます。

7 行目では、整数値 233 をキーとしてもつ要素の探索結果が表示されています。その後、main 関数で当該要素を delete メソッドで削除しています。削除後のハッシュ表の状態は 9 行目～11 行目に表示されています。要素 (233, Chris) は tbl[3] が参照する連結リストの先頭から 2 番目に格納されていましたが、10 行目に示すとおり正しく削除されていることが確認できます。

ログ 5.6 my_hash.py プログラムの実行

```
01 $ python3 my_hash.py
02 ハッシュ表の状態:
03 tbl[2] -> (12, Bob)
04 tbl[3] -> (3, Alice) -> (233, Chris) -> (183, Eav)
05 tbl[5] -> (95, David) -> (25, George)
06
07 取得した要素 (233, Chris)
08 削除後のハッシュ表の状態:
09 tbl[2] -> (12, Bob)
10 tbl[3] -> (3, Alice) -> (183, Eav)
11 tbl[5] -> (95, David) -> (25, George)
```