

# 第5章

## 探索アルゴリズム

本章では、探索アルゴリズム (search algorithm) について解説します。

たとえば、整数値をデータとしてもつ配列から、整数値  $x$  を探したいとします。整数値  $x$  をキー (key) と呼びます。探索アルゴリズムは、整数値  $x$  が格納されている場所を探索し、そのインデックスを返します。また、連結リストなどのデータ構造であれば、インデックス (配列上の位置) の代わりに指定したキーをもつ要素のオブジェクトを返します。

本章で解説する探索アルゴリズムは、線形探索と二分探索、ハッシュ探索の3つです。最初の2つは名前があり、 $O(n)$  と  $O(\log n)$  の時間でキーを探索します。なお、 $n$  は前章までと同様にデータ数です。ハッシュ探索は、ハッシュ表と呼ばれるデータ構造を用いた手法です。平均計算量が  $O(1)$  となる高速な探索アルゴリズムです。

## 5.1

# 線形探索

探索 (search) とはデータの集合から指定したキーをもつ要素を検索して取り出すことで、広辞苑では「コンピューターで、希望するデータを取り出したり、格納されている場所を決定したりすることの処理のこと」と書かれています。

線形探索 (linear search) は、配列または連結リストの各要素を順番に確認し、指定したキーと同じ値かを比較することによって探索を行います。このためにループ構造を用いて、要素を1つひとつ走査します。

運が良ければ、最初に調べた要素がキーと同じ値をもち、1回の処理で探索が終了します。最後の要素がキーである場合や、そもそもデータ構造内にキーが存在しない場合は、すべての要素を走査することになります。この場合、パフォーマンスが最悪になり、計算量が  $O(n)$  となります。データ構造内にキーが存在する場合の平均探索時間は  $\frac{n}{2}$  となりますが、この場合も計算量は  $O(n)$  です。本書の例では、各要素は異なる値をもつと仮定します。もちろん同じ値をもつ要素が複数あった場合でも動作しますが、この場合は、最初に見つかったキーと同じ値をもつ要素を探索結果とします。線形探索は非常に簡単なので、早速ソースコードを見てていきます。

### 5.1.1 配列を用いた線形探索の実装コード

ソースコード 5.1 (linear.py) に配列の要素を探索する線形探索アルゴリズムの実装例を示します。配列内から指定したキーと同じ値をもつ要素を探索し、当該インデックスを戻り値として返す linear\_search 関数を実装しています。また、キーが配列内に存在しない場合は、戻り値として None を返します。

## ■ main 関数説明

9行目～19行目で main 関数を定義しています。11行目でランダムな要素をもつ配列 arr を宣言し、12行目で arr の中身を表示します。配列 arr は、[5, 1, 33, 25, 85, 12, 3, 8, 54, 17] といった整数値の集合で初期化しています。

15行目で線形探索を実行する linear\_search 関数を実行します。引数として配列 arr と整数値 3 を渡します。配列 arr 内に整数値 3 が存在すれば、そのインデックスが変数 index に格納されます。存在しなければ、None が格納されます。

16行目～19行目では、探索の結果を表示します。整数値 3 が格納されているインデックスの表示、または指定した数値が存在しなかった旨を表示する処理を if 文を用いて制御します。配列 arr を見ると、整数値 3 は 7 番目の要素であるインデックス 6 に格納されているので、この例では 17 行目の print 関数が実行されます。

## ■ linear\_search 関数（線形探索）の説明

2行目～7行目で線形探索を行う linear\_search 関数を定義しています。引数は配列 arr とキーとなる要素を示す変数 key です。for ループを用いて配列 arr の先頭から 1 つずつ変数 key と同じ値であるかどうかを判定し、同じであればインデックスを戻り値として返します。配列の最後尾まで探索しても、変数 key と同じ値をもつ要素がなければ、戻り値として None を返します。

## ■ 線形探索プログラムの実行

ソースコード 5.1 (linear.py) を実行した結果をログ 5.2 に示します。整数値の 3 をキーとして指定して探索を行った結果、3 行目にインデックス 6 に要素が見つかったという情報が表示されます。2 行目に表示されている配列の中身を見ると、7 番目の要素であるインデックス 6 に整数値の 3 があることが確認できます。

```
01 $ python3 linear.py
02 配列: [5, 1, 33, 25, 85, 12, 3, 8, 54, 17]
03 arr[6] に要素が見つかりました。
```

## 5.2

# 二分探索

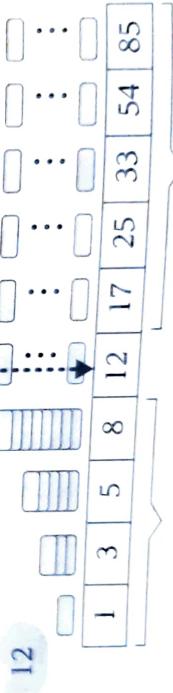
本項の例では配列をデータ構造として用いましたが、連結リストを用いた場合も同様に、連結リストの先頭要素から1つひとつ調べる必要があります。

二分探索 (binary search) は、 $O(\log n)$  の時間で配列内からキーと同じ値をもつ要素を探索します。二分探索では、要素がソートされていること、ランダムアクセスが可能であることが前提条件となっています。そのため、データの集合が連結リストに格納されている場合、二分探索は実行できません。二分探索は、線形探索に比べてずいぶん計算量は少なくてすみますが、事前にソートが必要です。実装に際しては、ソートアルゴリズムにも注意を払ってください。

### 5.2.1 二分探索の概要

二分探索の本質は、探索空間（探索対象の要素数）を半分に減らすことを行なうことです。昇順にソート済みの配列 `arr` 内からキーとなる要素 `key` が格納されているインデックスを探索したいとします。配列 `arr` の大きさが 10、変数 `key` が整数値 8 の場合の例を図 5.1 に示します。配列の中身は `[1, 3, 5, 8, 12, 17, 25, 33, 54, 85]` としています。

$$\text{key} = 8 \text{ のとき } \text{arr}[q] > \text{key} \quad q = 4 \text{ (インデックスの中心)}$$



最初のイテレーションでは、  
部分配列 `arr[0:3]` を探索する  
部分配列 `arr[5:9]` 内に、  
整数値 8 は存在し得ない

図 5.1 二分探索の概要

まず、配列 `arr` の中心をインデックスを表す変数を `q` とします。ここでは、要素数が 8 なので、`q`

は4となります。要素数が奇数の場合には、中心の前後のどちらでもかまいません。まず、 $\text{arr}[q]$  が変数  $\text{key}$  と同じかどうかを調べます。同じであれば、それで探索は終了です。図の例では、 $\text{key}$  が8で  $\text{arr}[4]$  が12なので、 $\text{arr}[4] > \text{key}$  となります。部分配列  $\text{arr}[5:9]$  に含まれるすべての要素は  $\text{arr}[4]$  よりも大きい値をもつため、整数値8が部分配列  $\text{arr}[5:9]$  に含まれる可能性はありません。そのため、探索空間を  $\text{arr}[0:q-1]$  に限定することができます。

一般化すると、データ数  $n$  の配列に対して、 $\text{arr}[q]$  と  $\text{key}$  の大小を比較し、探索空間を  $\text{arr}[0:q-1]$  または  $\text{arr}[q+1:n-1]$  に限定できます。すなわち、探索空間が  $\frac{n}{2}$  になります。厳密には  $\frac{n}{2}$  または  $\frac{n}{2} - 1$  ですが、ここでは気にしないで構いません。

同様の処理を繰り返し、探索空間を半分ずつに減らしていくと、最終的に探索空間が1になります。最後の要素と  $\text{key}$  を比較し、同じであれば探索が終了します。もし、異なる値であれば配列  $\text{arr}$  内に  $\text{key}$  と同じ値をもつ要素は存在しないことが確定します。

それでは、配列を半分に分割する処理は最大で何回実行されるでしょうか？データの数を  $n$  とした場合、探索空間は  $n$ 、 $\frac{n}{2}$ 、 $\frac{n}{4}$ 、 $\dots$ 、 $\frac{n}{2^k}$  と減っていきます。ここで  $k$  は配列を分割する回数です。探索空間が1になるまで繰り返すので、 $\frac{n}{2^k} = 1$  となる  $k$  を調べれば良いわけです。両辺に対数を取れば、 $k = \log n$  となります。そのため、計算量が  $O(\log n)$  となります。

## Column

### 連結リストを用いた二分探索

連結リストを拡張したスキップリスト (skip lists) と呼ばれる、 $O(\log n)$  で探索が可能なデータ構造が提案されています。スキップリストは、第6章で解説する木構造の代替となる確率的データ構造 (probabilistic data structure) という位置づけです。ただ、大学や大学院の授業で学ぶようなレベルではありません。

## 5.2.2 配列の二分探索の実装例

ソースコード5.3 (`binary.py`) に配列内から指定したキーを二分探索するプログラムを示します。配列に含まれる整数値は、あらかじめソートされているものとします。

### ソースコード5.3 配列の二分探索

#### ソースコードの概要

`~/ohm/ch5/binary.py`

4行目～14行目 二分探索を行う `binary_search` 関数の定義

16行目～26行目 `main` 関数の定義

01 import math

```

03 # 指定したキーの要素のインデックスを返す
04 def binary_search(arr, key, p, r):
05     if r < p:
06         return None
07     else:
08         q = math.floor((p + r) / 2)
09         if arr[q] > key:
10             return binary_search(arr, key, p, q - 1)
11         elif arr[q] < key:
12             return binary_search(arr, key, q + 1, r)
13         else:
14             return q
15
16     if __name__ == "__main__":
17         # データの宣言と初期化
18         arr = [1, 3, 5, 8, 12, 17, 25, 33, 54, 85]
19         print("配列:", arr)
20
21     # 数値8の探索
22     index = binary_search(arr, 8, 0, len(arr) - 1)
23
24     if index != None:
25         print("arr[", index, "]に要素が見つかりました。")
26     else:
27         print("指定した数値は配列内に存在しません。")

```

## ■ main 関数の説明

16行目～26行目で main 関数の定義をしています。18行目で配列 arr を宣言し、[1, 3, 5, 8, 12, 17, 25, 33, 54, 85] といったソート済みの整数値の集合で初期化します。19行目で配列 arr の中身を表示しています。

22行目で整数値 8 をキーとして指定し、binary\_search 関数を実行します。関数への引数は、配列 arr と探索したいキーである整数値 8、配列の先頭と最後尾のインデックスである 0 と len(arr)-1 です。ここで len(arr) は、配列 arr の要素数なので整数値の 10 です。探索結果を変数 index に代入します。整数値 8 が配列 arr に存在すれば、インデックスが格納されます。存在しなければ None が格納されます。23行目～26行目で、探索結果に関する情報を表示します。

## ■ binary\_search 関数（二分探索）の説明

4行目～14行目で二分探索を実行する binary\_search 関数を定義しています。引数として、配列 arr と変数 key、インデックスを表す変数 p と r を受け取ります。まず、5行目の if 文で、r < p が

どうかを判定します。もし、 $r$ の値が $p$ よりも小さければ部分配列 $arr[p:r]$ は空です。配列 $arr$ 内に変数 $key$ と同じ値をもつ要素が見つからなかった場合に限り、 $r < p$ がTrueとなります。なお、部分配列の大きさが1なら、 $r$ と $p$ は同じ値になるので、if ブロックを実行します。

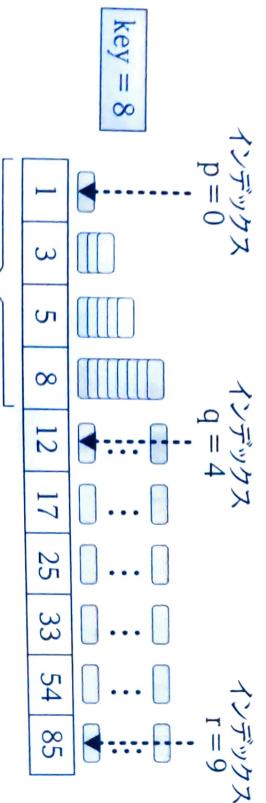
部分配列の大ささが1以上であれば、8行目～14行目のelse ブロックを実行します。まず、8行目で部分配列の中間のインデックスを計算し、変数 $q$ に格納します。インデックスは整数値であるため、冒頭でmath モジュールをインポートしてfloor 関数を適応させます。

7行目～14行目で再度、if 文が登場します。今回は3つのケースに分岐するため、if と elif と else の3つのブロックで制御します。まず、9行目で $arr[q]$ がKeyの値より大きいかどうかを判定します。判定結果がTrue であれば、 $arr[q]$ より後ろにはkey が存在しないことが確定するため、 $arr[p:q-1]$ を探索することとなります。そのため、10行目でbinary\_search 関数を再帰的に呼び出します。部分配列を参照するために、インデックスを $p$ と $q-1$ とします。これによって部分配列 $arr[p:q-1]$ 内から変数 $key$ の値を探索することとなります。

9行目の条件判定がFalse であれば、11行目で $arr[q]$ がkey の値より小さいかどうかを判定します。True であれば、 $arr[q]$ より前には変数key が存在しないことが確定します。同じ要領になりますが、12行目でbinary\_search 関数を呼び出します。今回は探索範囲を $arr[q+1:r]$ にします。

9行目と11行目の条件判定がともにFalse の場合は、 $arr[q]$ とkey が同値であることを意味します。したがってkey はインデックス $q$ に格納されていることが確定します。そのため、14行目で変数 $q$ の値を戻り値として返します。

それでは、配列 $[1, 3, 5, 8, 12, 17, 25, 33, 54, 85]$ から整数値8を探索するときの具体例を示します。main 関数からbinary\_search( $arr, 8, 0, 9$ )を呼び出したときの状態を図5.2に示します。なお、22行目の $len(arr)-1$ は $arr$ の最後のインデックス番号なので、整数値の9です。中間のインデックス $q$ は4となります。 $arr[4]$ の値が12なので $12 > 8$ 、つまり、 $arr[q] > key$ です。そのため、 $arr[0:3]$ に探索範囲を限定できます。ここで再帰的にbinary\_search( $arr, 8, 0, 3$ )を実行します。

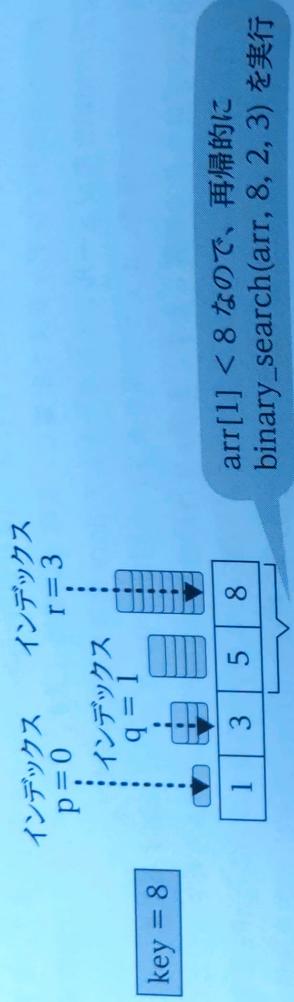


$arr[4] > 8$ なので、再帰的にbinary\_search( $arr, 8, 0, 3$ )を実行

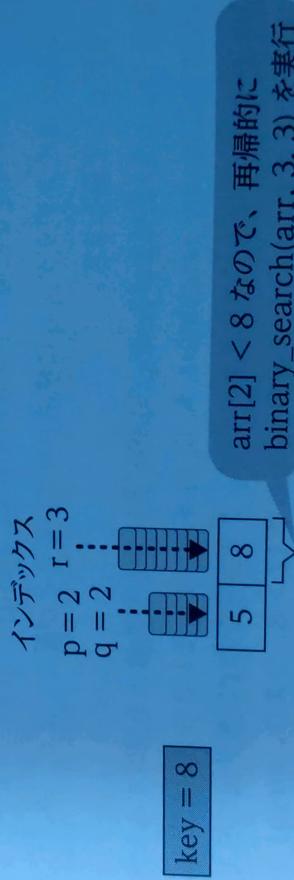
図5.2 二分探索の例1 main 関数からbinary\_search( $arr, 8, 0, 9$ )を実行

binary\_search( $arr, 8, 0, 3$ )を実行したときの状態を図5.3に示します。変数 $p$ と $r$ の値がそれ

ぞれ 0 と 3 なので、変数  $q$  の値は 1 となります。 $\text{arr}[1]$  の値は 3 であるため、 $\text{arr}[1] < 8$  となります。部分配列の後半部分である  $\text{arr}[2:3]$  に探索範囲を限定します。

図 5.3 二分探索の例 2  $\text{binary\_search}(\text{arr}, 8, 0, 3)$  を実行

$\text{binary\_search}(\text{arr}, 8, 2, 3)$  を実行したときの状態を図 5.4 に示します。変数  $p$  と  $r$  は、それぞれ 2 と 3 です。そのため、変数  $q$  の値は 2 となります。 $\text{arr}[2]$  の値は 5 なので、 $\text{arr}[2] < 8$  となります。次は  $\text{arr}[3:3]$  を探索します。

図 5.4 二分探索の例 3  $\text{binary\_search}(\text{arr}, 8, 2, 3)$  を実行

$\text{binary\_search}(\text{arr}, 8, 3, 3)$  を実行したときの状態を図 5.5 に示します。部分配列の大きさが 1 なので、変数  $p$  と  $r$  はともに 3 です。変数  $q$  の値も 3 になります。 $\text{arr}[3]$  の値は 8 なので、9 行目と 11 行目の条件判定 ( $\text{arr}[q] > k$  と  $\text{arr}[q] < k$ ) のいずれも False になります。すなわち、 $\text{arr}[3] == 8$  です。そのため、13 行目の else ブロックに処理が進み、インデックス  $q$  の値である整数値 3 が戻り値として返されます。

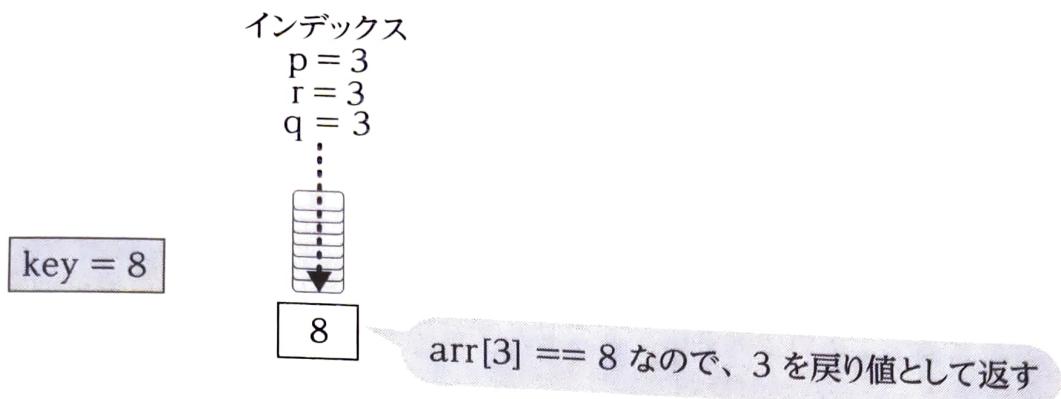


図 5.5 二分探索の例 4 `binary_search(arr, 8, 3, 3)` を実行

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。

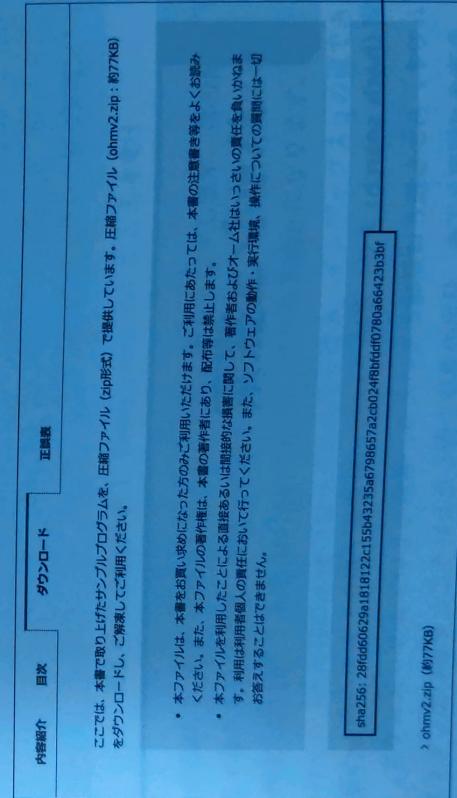


図 5.6 ハッシュ値の例

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。

28fdd60629a1818122c155b43235a6798657a2cb024f8bfddf0780a66423b3bf

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。

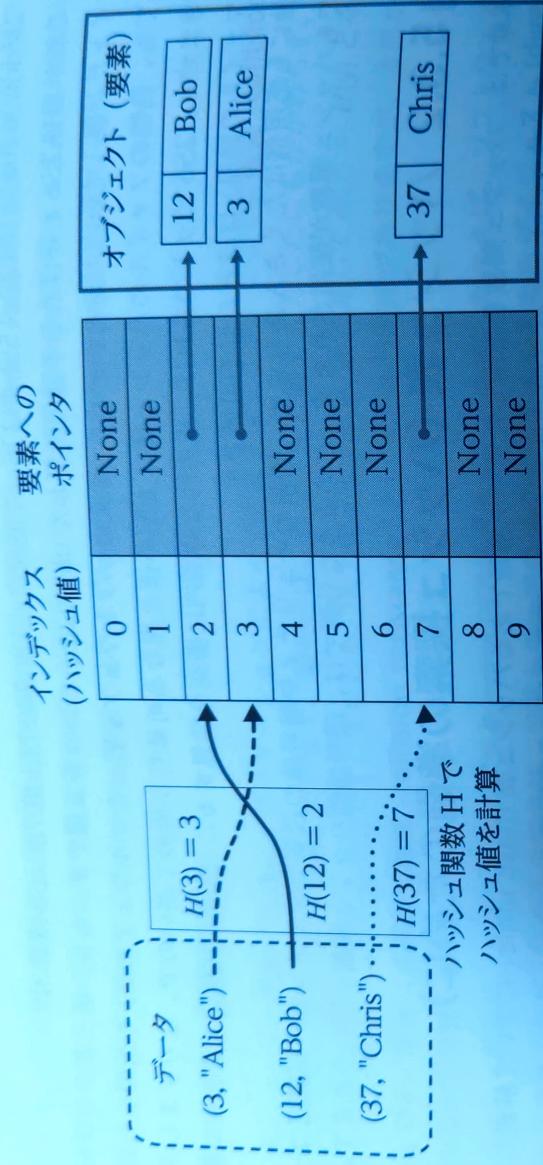


図 5.7 ハッシュ表の例（エントリー数が 10 のハッシュ表）

ハッシュ値の出力範囲は入力範囲より遙かに小さいので、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タリーは連結リストへの pointer 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。

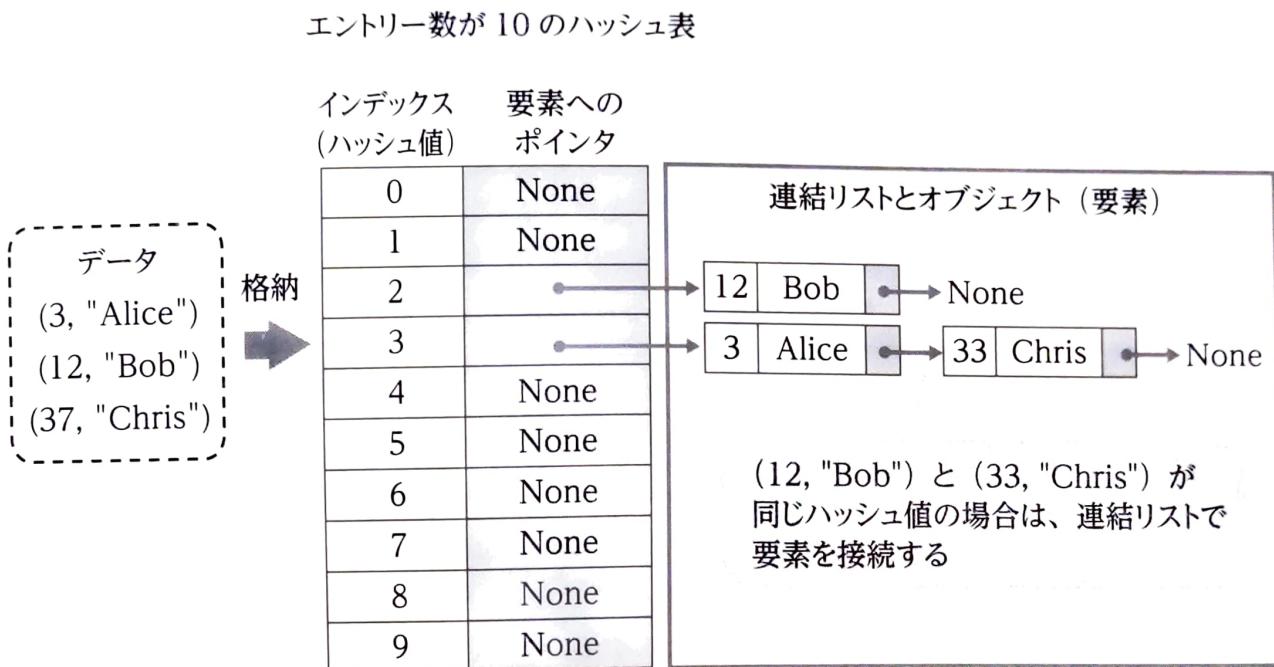


図 5.8 チェイン法の例

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査します。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
```

要素を格納するときは、MyNode クラスのインスタンスを生成して、ハッシュ値  $H(\text{key})$  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[ $H(\text{key})$ ] が MyNode オブジェクトを格納する場所となります。もし、self.tbl[ $H(\text{key})$ ] 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  $H(\text{key})$  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タリーが 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タ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つたとしましょう。もちろん現実問題として、そ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が必要があります。この場合、すべての要素が同じエンタリーに格納され、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。

## 5.3.3 チェイン法を用いたハッシュ探索の実装例

ソースコード5.5 (my\_hash.py) にチェイン法を用いたハッシュ探索の実装例を示します。少し長いソースコードですが、1つひとつ解説していきます。

### ソースコード5.5

ハッシュ表を用いた探索

~/ohm/ch3/my\_hash.py

#### ソースコードの概要

- 1行目～8行目 連結リストの要素を表す MyNode クラスの定義
- 10行目～91行目 ハッシュ表を表す MyHashTable クラスの定義
- 15行目と 16行目 ハッシュ値を計算する \_get\_hash メソッドの定義
- 19行目～31行目 要素をハッシュ表へ挿入する insert メソッドの定義
- 34行目～58行目 指定したキーをもつ要素をハッシュ表から削除する delete メソッドの定義
- 61行目～77行目 指定したキーをもつ要素を探索する search メソッドの定義
- 93行目～117行目 main 関数の定義

```

01 class MyNode:
02     def __init__(self, key, val):
03         self.key = key
04         self.val = val
05         self.next = None
06
07     def to_string(self):
08         return "(" + str(self.key) + ", " + str(self.val) + ")"
09
10 class MyHashTable:
11     def __init__(self, size):
12         self.tbl = [None] * size
13
14     # ハッシュ値の計算
15     def __get_hash(self, key):
16         return key % len(self.tbl)
17
18     # 要素の追加
19     def insert(self, key, val):
20         # ハッシュ値を計算
21         hash_val = self.__get_hash(key)
22
23         # 要素を追加
24         n = MyNode(key, val)

```

```

109 if x != None:
110     print("取得了要素", 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, "Eve")、(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身から探し、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。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。要素を削除了あと、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、MyHashTable のインスタンスを生成するときに、大きさを 10 と指定しているので、実質 10 で割った余りの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。

エントリーナ数が10のハッシュ表

インデックス (ハッシュ値)	要素への ポインタ	連結リストヒオブジェクト (要素)
0	None	
1	None	
2	12	Bob
3	12	Bob
4	None	
5	95	David
6	None	
7	None	
8	None	
9	None	

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 節で解説した双向連接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 要素への  
(ハッシュ値) ポインタ

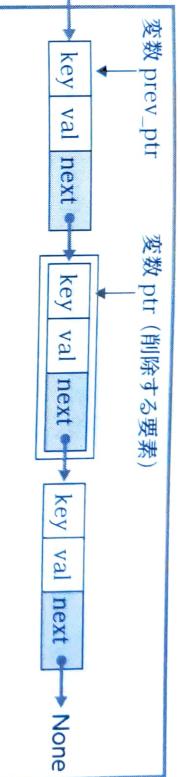
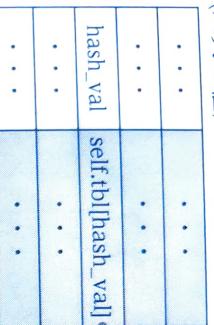


図 5.10 要素の削除ケース 1-1

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 = 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。

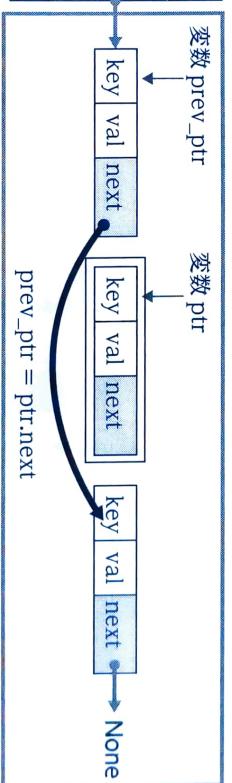


図 5.11 要素の削除ケース 1-2

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] = ptr.next` が `ptr.next` を参照するように修正します。

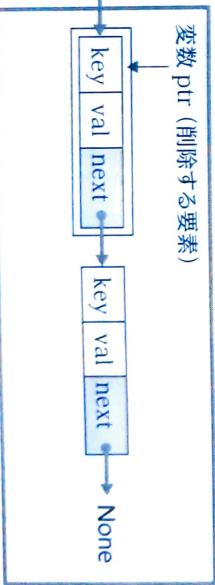
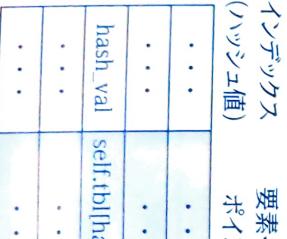


図 5.12 要素の削除ケース 2-1

47行目の `self.tbl[hash_val] = ptr.next` という命令を実行したあとと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。

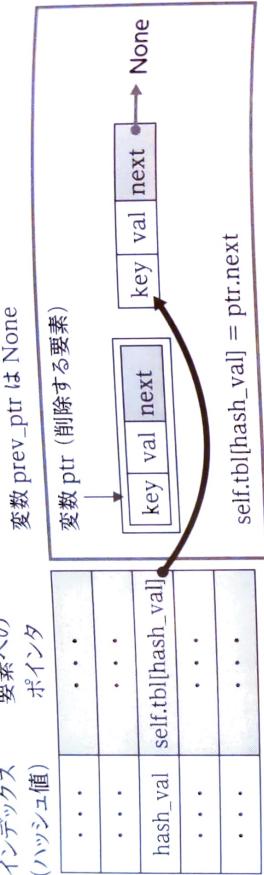


図 5.13 要素の削除ケース 2-2

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、一つ前の要素の `next` を `None` に変更するだけです。

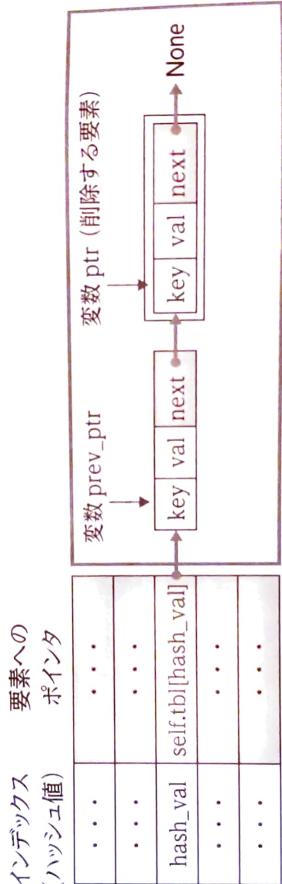


図 5.14 要素の削除ケース 3-1

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

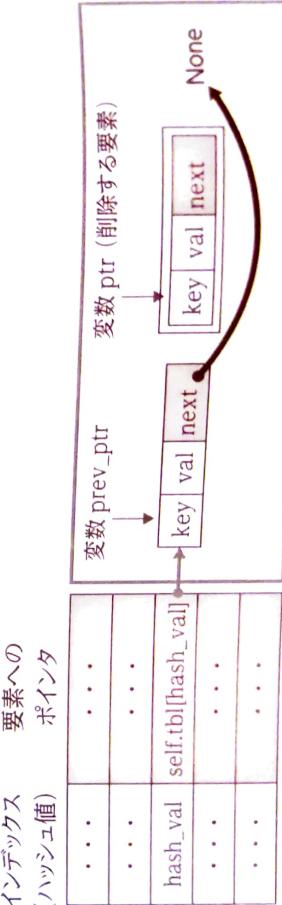


図 5.15 要素の削除ケース 3-2

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。観察化すると図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。

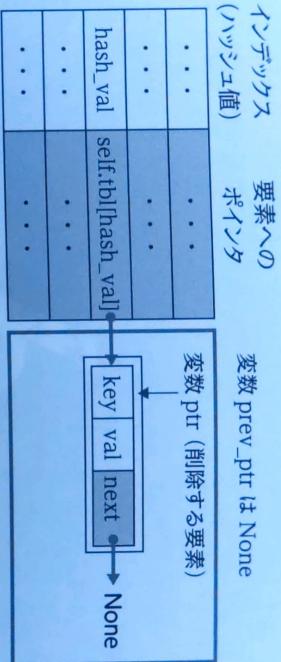


図5.16 要素の削除ケース4-1

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。

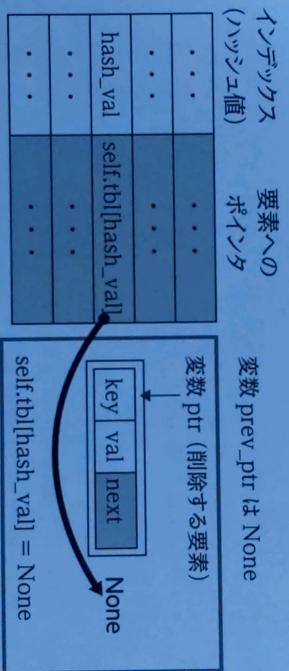


図5.17 要素の削除ケース4-2

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 ループに入っています。

67 行目でいたん 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 番目に格納されました。また、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)
```